

译者序

《Java 编程思想》已经成为了广大 Java 程序员和爱好者们手中必备的宝典，它在广大读者心目中的地位已经无可替代。其作者 Bruce Eckel 也已经成为了众多 Java 程序员顶礼膜拜的大师。随着 Java 技术的发展以及对 Java 认识的不断深入，Bruce Eckel 推出了《Java 编程思想》第三版，在这里我们应该向他致敬，他不断精益求精、锐意进取的精神正是我们应该努力学习的榜样。

随着软件开发技术，尤其是 Web 应用的开发技术的发展，Java 越来越受到人们的青睐，越来越多的企业都选择使用 Java 及其相关技术，例如 J2EE 技术来构建它们的应用系统。但是，掌握好 Java 语言并不是一件可以轻松完成的任务，如何真正掌握 Java 语言，从而编写出健壮的、高效的、灵活的程序是 Java 程序员们面临的重大挑战。

《Java 编程思想》就是一本能够让 Java 程序员轻松面对这一挑战，并最终取得胜利的经典书籍。本书深入浅出、循序渐进地把我们领入 Java 的世界，让我们在不知不觉中就学会了用 Java 的思想去考虑问题、解决问题。本书不仅适合 Java 的初学者，对于有经验的 Java 程序员来说，每次阅读本书也总是能够有新的体会，这正是本书的魅力所在。

本书的第二版由侯捷先生翻译，他已经把翻译原著这项工作做到了极致，我们在翻译过程中总是诚惶诚恐，一方面害怕曲解了原著的意思，另一方面也害怕破坏了第二版译著已经做出的让广大中国读者认可本书的种种努力。

我们在翻译本书的过程中力求终于原著。对于本书中出现的大量的专业术语尽量遵循标准的译法，并在有可能引起歧义之处著上了英文原文，以方便读者的对照理解。

全书的翻译由陈昊鹏和饶若楠合作完成，薛翔、郭嘉和方小丽也参与了全书的翻译工作。由于我们水平有限，书中出现错误与不妥之处在所难免，恳请读者批评指正。

前言

我的兄弟托德正准备从硬件工业转移到程序设计领域，我曾向他建议，下一次技术革命的重点将是基因工程。

我们将使用微生物来制造食品、燃料和塑料；这些微生物不仅能清除污染，还能让我们付出比现有少得多的代价就能主宰整个物质世界。我曾以为，相比之下计算机革命将显得微不足道。

后来我意识到自己犯了科幻小说家常犯的错误：迷信技术的力量（当然，这在科幻小说里司空见惯）。有经验的作家都知道，故事的重点不是技术，而在于人。基因工程将对我们的生活造成深远的影响，但它未必就会使计算机革命（或者至少是信息革命）黯然失色，因为正是计算机技术推动了基因工程的发展。信息指的是人与人之间的沟通。当然，汽车、鞋子、包括基因疗法，这些都很重要，但它们最终都只是表面现象。人类如何与世界相处才是问题的本质。这个相处的过程就是沟通。

本书恰好就是一个例子。很多人认为我很大胆、或者有点狂妄，因为我把所有资料都放在网络上。“还有谁会购买呢？”他们这样问。如果我的性格保守谨慎一些，我就不会这么做。但是我真的不想再用传统方式来编写一本新的计算机书籍。我不知道这么做会有什么后果，但结果表明，这是我在写书经历中做过的最明智的一件事。

首先，人们开始把改正后的意见反馈给我。这是个令人惊叹的过程，因为人们仔细检查每个角落、每个缝隙，找出技术上和语法上的种种问题，让我得以改正所有错误，而这些错误单凭我个人是很难觉察到的。人们对这种作法还有些顾虑，他们常常说“我并无冒犯之意...”，然后扔给我一大堆错误。无疑我自己从未察觉过这些错误。我很喜欢这种集体参与的过程，这也使这本书更加特别。这些反馈很有价值，所以我使用称为“BackTalk”的系统创建了一些应用，来对这些建议进行收集和分类。

但是，很快我就开始听到“嗯，很好。把书做成电子版放到网络上是个好主意，可是我希望购买印刷出版并装订成册的书籍”。我曾经作出努力，让每个人都能很容易地打印出美观的书籍，但这样仍然无法满足人们对印刷书籍的需求。大多数人们都不习惯在计算机屏幕上阅读整本书，也不喜欢总是带着一捆捆的纸，所以即使这些打印出来的书籍再美观，也吸引不了他们（而且激光打印机的碳粉并不便宜）。看来即使是计算机革命，也难以抢走出版商的生意。不过，有个学生提出这也许会在未来成为一种出版模式：先在网络上出版书籍，当它引起足够的关注以后，才考虑制作纸质版本。目前，绝大多数书籍都不赚钱，这种新方式或许可以给整个出版业带来更多的利润。

在另一方面，这本书也带给我一些启迪。开始，我认为 Java“只不过是另一种程序设计语言”。从许多方面看的确如此。但随着时间流逝，以及对 Java 学习的深入，我才开始明白，这个编程语言的目的，与我所见过的其它语言根本不同。

程序设计就是对复杂度的管理。它包括：待解问题的复杂度和所依赖的底层机器的复杂度。正是因为这种复杂度的存在，导致了大多数项目的失败。并且据我所知，还没有哪个程序设

计语言全力将主要设计目标放在“征服程序开发与维护过程中的种种复杂度”上¹。当然，许多编程语言设计时也确实考虑到了复杂度问题，但它总是与被视为更本质的问题混在一起。当然，那些也都是会让程序员感到困扰的问题。比如，C++必须向上兼容于C（为使C程序员容易过渡），并具有执行效率高的优点。这两点都很有用，并且帮助C++取得了成功。不过，两者也引入了额外的复杂度，使得某些项目无法完成。（当然，你可以归咎于开发或管理人员，但如果有某种语言可以帮助我们找到错误，何乐而不为呢？）Visual Basic (VB) 是另一个例子，它被局限于BASIC语言（它并不把可扩展能力作为设计目标），这就使得所有堆积于VB 之上的扩充功能，都造成了可怕且难以维护的语法。Perl 也向上兼容于Awk、Sed、Grep、以及其它Unix工具，这就导致了被诟病为“能写不能读”的程序代码（也就是说，一段时间之后，你就无法看懂这些代码）。另一方面，C++、VB、Perl、Smalltalk之类的编程语言，都为复杂度问题付出了相当大的努力，在解决特定类型问题的时候非常成功。

当我开始理解 Java 之后，印象最深的莫过在 Java 上体现出的 Sun 公司的设计目标：为程序员降低复杂度。就好象说：“我们关注的是降低编写健壮代码的困难程度和需要的时间”。以前，编写健壮代码将导致性能降低（尽管有许多承诺，Java 总有一天能够执行得足够快），不过 Java 的确大大缩短了开发时间；相比同等功能的 C++ 程序，它只需一半甚至更少的时间。只凭这一点，就足以省下大量的时间与金钱。不过，Java 并不仅仅如此。它又持续引入了一些日益重要的复杂任务，比如多线程和网络编程。并且通过语言本身的特性和程序库，使得这些工作变得轻而易举。最后，Java 还着眼于某些有着极高复杂度的问题：跨平台程序、动态程序代码联编、甚至安全问题，它们都属于复杂度问题的重要方面。所以尽管存在已知的效率问题，Java 带来的许诺却极其诱人：它能使我们程序员的生产率大大提高。

根据我的观察，Web 是 Java 影响最远的地方之一。网络程序设计总是非常困难，但 Java 使它得以简化（Java 的设计者仍在努力使它变得更简单）。网络程序设计所关注的，就是研究如何让我们用比使用电话更有效率、更廉价的方式进行沟通（单是电子邮件就已经使许多业务领域发生了革命性的变化）。当我们能更有效地进行沟通时，奇妙的事情就开始发生，这也许要比基因工程所作出的许诺更让人感到神奇。

通过所有方式：编写程序、团队开发、设计用户界面（让程序可以和用户交互）、跨平台执行、轻松编写跨互联网的通讯程序等，Java 扩展了人与人之间的通讯带宽。我认为，通讯革命的成果也许不应以海量数据的传输来衡量；我们将体会到真正的革命，因为我们能更容易和他人沟通：可以是一对一的形式、可以是分组形式、也可以是全球通讯的形式。我曾经听人主张，随着足够多的人之间产生了足够多的相互联系，下一次革命将会是一种全球化思维的形成。Java 可能是、也可能不是这场革命的引线，但至少这种可能性使我觉得，教授这门语言是一件非常有意义的事情。

第 3 版前言

这一版的主要目的和大量工作都用在了与Java 1.4 版保持同步上。不过，本书的主旨也更加清楚：使大多数读者通过本书牢牢抓住语言的基本概念，以便深入学习更复杂的主题。因为Java还在不断地演化，所以重新定义“基础知识”的涵义就很有必要，再说本书也不应过

¹ 在本书第二版我收回这句话：我认为Python语言非常接近这个目标。请参考www.Python.org。

于面面俱到。这就意味着，比如，完全重写了“并发”这一章（以前称为“多线程”），它能使你对线程的核心概念有一个基本的了解。没有这一点，你就很难理解线程中那些更复杂的概念。

我还认识到了代码测试的重要性。如果没有内置的测试框架及相应测试用例，并在每次构造系统的时候都进行测试，那么你就无法知道代码是否可靠。在本书中，为了做到这一点，专门编写了单元测试框架，用来演示和验证每个程序的输出。这些内容在第 15 章中有介绍，里面还解释了ant（Java构建系统的事实标准，与make类似），JUnit（Java单元测试框架的事实标准），日志和断言机制（是JDK1.4 新引入的），以及对调试和性能分析的介绍。为了涵盖所有这些概念，我把这一章命名为“发现问题”，里面介绍的内容都是我认为所有的Java程序员都应该具有的基本技能。

此外，我复查了书中所有的示例，并同时问自己：“为什么要用这种方法呢？”。多数情况下我会作出一些修改和润色，这样会使例子更贴切，同时还有助于演示一些我认为比较好的Java编程实践（至少在比较基础的范围内）。此外，我还删除了一些不再有意义的示例，并加入了一些新的示例，许多已有示例也被重新设计和修改过。

全书分为 16 章，涵盖了对Java语言的基本介绍。它可以用作基础课程的教材，但那些更高级的主题又该如何处理呢？

我原计划为本书加入一个新部分，专门介绍“Java 2 企业版”（J2EE）的基础知识。其中许多章节由我的朋友、以及一同授课或开发项目的同事编写，他们有：Andrea Provaglio, Bill Venners, Chuck Allison, Dave Bartlett,和Jeremy Meyer。当我把这些新章节的进度和出版日期相对照的时候，我就有些担心。并且我注意到，前 16 章的篇幅就已经与本书第二版的全部篇幅一样大了。而即使是这个篇幅，读者也会经常抱怨。

对于本书的前两版，读者给予了极高评价，当然我也十分欣慰。但有时他们也会抱怨。其中常被提及的就是“这本书太厚了”。在我看来，如果这就是你挑出的唯一毛病，那真是令人哭笑不得。（这会让人联想到奥地利国王对莫扎特作品的抱怨：“音符太多了”。我没有任何与莫扎特相比的意思）此外，我只能认为发出这种抱怨的人尚未了解Java语言的博大精深，也未见识过其它Java书籍。尽管如此，我还是在这一版中尽量删减掉那些已经过时，或是不那么关键的内容。总之，我仔细检查了所有地方，在第三版中删除了那些不必要的内容，并尽可能作出了修改和润色。这么做我很放心，因为本书的前两个版本还可以从网站

（www.BruceEckel.com）上免费下载，附在书后的光碟中也有。如果你还需要那些老资料，这些地方都能找到。对于作者，这样可以减轻很多负担。比如，“设计模式”这一章太大了，已经可以独立成书：《Thinking in Patterns (with Java)》（也可以从网站上下载）。

在Java的下一个版本（JDK 1.5）中，预计Sun公司会效法C++引入泛型这个新功能。我本来已经决定，到时候把本书分为两册，以加入新的内容。但有个声音悄悄在问：“为什么要等呢？”。于是，我决定在这一版中就这么做，于是一切问题迎刃而解。我以前往一本介绍性的书籍塞入了太多内容。

这本新书并不是第二卷，而是包含了一些高级主题。书名叫《Thinking in Enterprise Java》，它现在可以从www.BruceEckel.com免费下载。由于是一本单独的书，因此它的

篇幅可以随着内容的需要而扩展。与《Thinking in Java》一样，它的目标是向读者提供一本容易理解，涵盖J2EE技术基础知识的介绍。并为读者能学习更深入的内容做准备。你能在附录C中找到更多说明。

对于那些仍然不能忍受本书篇幅的读者，我只能说声抱歉。不管你信不信，为了让它尽可能薄，我已经作了很多努力。先不管书有多厚，我认为还有许多替代方式可以令你满意。比如，本书有电子版，如果你带着便携式电脑的话，你可以把电子版放进电脑，这样也不会给日常生活带来额外的负担。如果你还想更轻巧些，可以使用本书的掌上电脑版本。（有人对我说，他喜欢躺在床上，打开屏幕的背光看书，这样就不会打扰他的妻子。但愿这能帮助他进入梦乡。）如果你一定用纸才能阅读，我知道有人一次打印一章，然后放在公文包里，在火车上阅读。

Java 2, JDK 1.4

JDK的发布版本以 1.0, 1.1, 1.2, 1.3 表示，本书针对 1.4 版。尽管这些版本号还是“各自独立”的，但JDK 1.2 或更高版本的标准称呼是“Java 2”。这表明“旧式Java”（我在本书的第一版中讨论了其中的许多缺陷。）和Java的改进版本之间存在巨大差异，后者的缺陷要少得多，而且引入了很多优秀的设计。

本书针对Java 2 编写，尤其是JDK 1.4（很多代码不能在以前版本的JDK下编译，如果你试图这么做的话，构建系统将给出出错信息并终止。）。本书大刀阔斧地删除了一些过时的内容，并且重写了语言新引入和改进的部分。那些过时的内容可以在本书的以前版本中找到（可以通过Web或者本书光碟）。此外，任何人都可以从java.sun.com免费下载JDK，也就是说，本书针对JDK1.4，不会给任何人带来因为升级而造成的经济负担。

Java的以前版本在Linux系统上发布的速度稍慢（参见www.Linux.org），这个问题正在得到改进，针对Linux的新版本与针对其它平台的版本将同时发布，现在甚至是Macintosh也开始能跟上Java的新版本。与Java相互配合，Linux现在正处于非常重要的发展阶段，它正迅速成为市场上最重要的服务器平台，因为它快速、可靠、健壮、安全、易于维护，并且是免费的。这是计算机历史上的一场真正的革命，我认为以前的任何工具都没能具备所有这些特征。Java在服务器端编程中占据了重要位置，它采用了Servlet和Java 服务器页面（JSP）技术，这与传统的通用网关接口（CGI）技术相比是一个巨大的进步（相关主题请参考《Thinking in Enterprise Java》）。

简介

“上帝赋予人类说话的能力，而说话又创造了人类对宇宙的量度——思想” —*Prometheus Unbound*, Shelley

人类极其受那些已经成为用来表达他们所处社会的媒介的特定语言的支配。想象一下，如果一个人可以不使用语言就能够从本质上适应现实世界，语言仅仅是为了解决具体的交流和反映问题时偶尔被使用到的方式，那么我们会发现这只能是一种幻想。事实上“真实世界”在很大程度上是不知不觉地创建于群体的语言习惯之上的。

摘自 “*The Status of Linguistics As A Science*”，1929, Edward Sapir。

如同任何人类语言一样，**Java** 提供了一种表达概念的方式。如果使用得当，随着问题变得更庞大更复杂，这种表达媒介将会比别的可供选择的语言要更为简单更为灵活。

我们不应该将 **Java** 仅仅看作是一些特性的集合——有一些特性在孤立状态下没有任何意义。如果我们需要考虑设计，而不仅仅只是编码，那么我们可以将 **Java** 的各部分作为一个整体来使用。而且如果要按照这种方式去理解 **Java**，我们通常必须理解有关它的问题以及在程序设计时伴随的问题。这本书讨论的是编程问题、它们为什么成为问题，以及 **Java** 已经采取的用于解决它们的方案。因此，我在每章所阐述的特性集都是基于我所看到的这一语言在解决特定类型问题时的方式。按照这种方式，我希望能够每次引导你向前前进一点，直到 **Java** 思想意识成为你自然不过的语言。

自始至终，我一直持这样的观点：你需要在头脑中创建一个模型，用于加强对这种语言的深入理解；如果你遇到了疑问，你就能够将它反馈给你的模型并推断出答案。

前提条件

本书假定你对程序设计具有一定程度的熟悉：你已经知道程序是一些语句的集合，知道子程序/函数/宏的思想，知道像“if”这样的控制语句和像“while”这样的循环结构，等等。不过，你可能在许多地方已经学到过这些，例如使用宏语言进行程序设计或者使用像 Perl 这样的工具工作。只要你的程序设计已经到达能够自如地运用程序设计基本思想的程度，你就能够顺利阅读本书。当然，本书对 C 程序员来说更容易，对于 C++ 程序员更是如此，因此，即使你没有实践过这两种语言，也不要否定自己——而应该乐于努力学习（并且，伴随本书的多媒体光盘能够带领你快速学习所必需的 java 基础知识）。不过，我还是会介绍面向对象（OOP）的概念和 Java 的基本控制机制的。

尽管经常引用参考 C 和 C++ 语言的特性，但这并不是打算让它们成为内部注释，而是要帮助所有的程序员正确看待这些语言，毕竟 Java 是从这些语言衍生而来的。我将会努力简化这些引用参考，并且解释我认为一个非 C/C++ 程序员可能不太熟悉的任何事情。

学习 Java

大概在我的第一本书《Using C++》（Osborne/McGraw-Hill 于 1989 年出版）出版发行的同一时候，我就开始教授这种语言了。讲授程序设计语言已经成为我的职业了；自 1987 年以来，我在世界各地的听众中看到有的昏昏欲睡、有的面无表情、有的表情迷茫。当我开始给较小的团体进行室内培训时，在这些实践期间我发现了一些事情。即使那些面带微笑频频点头的人也困惑于对很多问题。我发现，多年来在软件开发会议上由我主持的 C++ 分组讨论会（后来变成 Java 分组讨论会）中，我和其他的演讲者往往是在极短的时间内告诉听众许多的话题。因此，最后由于听众的水平不同和讲授教材的方式这两方面原因，我可能最终流失了一部分听众。可能这样要求得太多了，但因为我是传统演讲的反对者之一（而且对于大多数人来说，我相信这种抵制是因为厌倦），因此我想尽力让每个人都可以跟得上演讲的进度。

我曾经一度在相当短的时间内做了一系列不同的演讲。因此，我结束了实践和迭代（一项在 Java 程序设计中也能很好运行的技术）的学习。最后，我使用自己在教学实践中学到的所有东西发展出一门课程。它以离散的、易消化的步骤以及参与讨论班的形式（最理想的学习形式）解决学习问题，并且每一小节课之后都有一些练习。我公司 **MindView, Inc** 现在提供公开的和内部的 *Thinking in Java* 培训课程；这是我们主要的介绍性培训课程，为以后更高级的培训课程提供基础。你可以到网站 www.MindView.net 上详细了解。（培训课程的介绍也可以在附带的 Java 多媒体光盘中得到。在同样的网站上也可以得到这些消息。）

从每个讨论班获得的反馈信息都可以帮助我去修改和重信制订课程教材，直到我认为它能够成为一个良性运转的教学工具为止。不过这本书并不能当作一般的培训课程笔记；我试着在这些书页中放入尽可能多的信息，并且将它构造成能够引导你顺利进入下一课题的结构。最重要的是，这本书是为那些正深入一门新的程序设计语言的单个读者而服务的。

目标

就像我前一本书《Thinking in C++》那样，这本书是围绕着程序设计语言的教学过程而构建的。特别地，我的目的是要建立一套机制，提供一种在自己课程培训班中进行程序语言教学的方式。当我思索书中的一章时，我思索的是如何在培训班上教好一堂课。我的目标是，切割出可以在合理学时内讲完的篇章，随后是适合在课堂上完成的练习作业。

在这本书中我想达到的目标是：

1. 每一次只演示一个简单步骤的材料，以便你在继续后面的学习之前可以很容易地消化吸收每一个概念。
2. 使用的示例尽可能的简单、简略。这样做有时会妨碍我们解决“真实世界”的问题，但是，我发现对于初学者，通常能够理解例子的每一个细节，而不是理解它能够解决的问题范畴会为他们带来更多的愉悦。同样，对于在教室内吸引读者学习的代码数量也有严格限制。正因为这些因素，我将毫无疑问地会遭到使用“玩具般的示例”的批评，但是我乐意接受那些有利于为教育带来益处的种种事物。

3. 谨慎安排呈现特性的先后顺序，以便使你在看到使用某一主题之前已经接触过它。当然，不可能总是这样；在这种情况下，将会给出简洁的介绍性描述。

4. 向你提供那些我认为对理解这种程序设计语言来说是很重要的部分，而不是提供我所知道的任何事情。我相信存在一个信息重要性的层次结构，有一些事实对于 95% 程序员来说永远不必知道——那些只会困扰人们并且凭添他们对程序复杂性感触的细节。举一个 C 语言的例子，如果你能够记住操作符优先表（我从未能记住），那么你可以写出灵巧的代码。但是你需要再想一想，这样做会给读者/维护者带来困惑。因此忘掉优先权，在不是很清楚的时候使用圆括号就行了。

5. 使每部分的重点足够明确，以便缩短教学时间和练习时段之间的时间。这样做不仅使听众在参与讨论班时的思维更为活跃和集中，而且还可以让读者更具有成就感。

6. 给你提供坚实的基础，使你能够充分理解问题，以便转入更困难的课程和书籍中。

JDK 的 HTML 文档

来自于 Sun Microsystems 公司的 Java 语言及其类库（可以从 java.sun.com 免费下载），配套提供了电子版文档，可使用 Web 浏览器阅读。并且实际上，每个厂商开发的 Java 编译器都有这份文档或一套等价的文档系统。大部分出版的 Java 书籍也都有这份文档的备份。所以你或者可能已经拥有了它，或者需要下载；所以除非需要，本书不会再重复那份文档。因为一般来说，你用 Web 浏览器查找类的描述比你在书中查找要快得多（并且在线文档更可能保持更新）。你仅需要参考“JDK 文档”。只有当需要对文档进行补充，以便你能够理解特定实例时，本书才会提供有关类的一些附加说明。

章节

本书设计时在思想中贯穿了一件事：人们学习 Java 语言的方式。讨论班听众的反馈帮助我了解哪些困难部分需要解释清楚。对于这个领域，在我突然雄心勃勃并且想涵盖如此多的特性之处，我渐渐明白——贯穿讲述材料的过程中——如果涉及到许多新特性，你就需要对它们全部解释清楚，不然这会很容易使学生感到困惑。因此，每次我就费尽大量心思介绍尽可能少的特性。

因此，本书的目标是每一章只讲述一个单一特性，或者是一小组相关特性，而不必依赖于其他还没有讲述过的特性。这样一来你在进入下一特性之前就可以消化当前知识背景中的每个部分。

下面是对本书所含章节的简单描述，它们与我在 Thinking in Java 讨论班上的授课和练习时段相对应的。

第 1 章：对象引论

(相应的讲座在光盘上)。这一章是对面向对象的程序设计 (OOP) 的一个综述, 包括对“什么是对象”这种基本问题的回答, 接口与实现、抽象与封装、消息与函数、继承与组合以及非常重要的多态概念。你也可以概要了解对象生成的问题, 例如构造器, 对象存在于什么地方, 一旦创建好放在什么地方, 以及神奇的垃圾回收器 (清除那些不再需要的对象)。还会介绍其他一些问题, 包括异常的错误处理, 响应用户接口的多线程以及网络和 Internet。你将会知道是什么使 Java 如此特别以及它为什么如此成功。

第 2 章: 一切都是对象

(相应的讲座在光盘上)。本章将引导你编写自己的第一个 Java 程序。本章开始先综述一些基本要素: 对象引用的概念; 基本数据类型和数组的简介; 对象的生存空间以及垃圾回收器清除对象的方式; 怎样将 Java 中的所有东西归为一种新的数据类型 (类); 创建自己类的基本要素; 方法、参数以及返回值; 名字可见性以及从其他类库使用组件; static 关键字; 以及注释和内嵌文档。

第 3 章: 控制程序流

(相应的一组讲座在 Thinking in C 的光盘上)。本章以讲述 Java 引自 C / C++ 的所有运算符为开始。另外, 你会看到运算符的共同缺点、转型、类型升级以及优先权。接着介绍基本的控制流程图以及选择操作, 这实际上是任何程序设计语言都具有的特性: if-else 选择结构, for 和 while 循环结构, 用 break 和 continue 退出循环以及 Java 的标注式 break 和标注式 continue (这说明在 Java 中没有“goto”), 以及 switch 分支选择。尽管材料大部分具有 C 和 C++ 代码的相同思路, 但还是存在一些不同之处。

第 4 章: 初始化和清除

(相应的讲座在光盘上)。本章首先介绍构造器, 它用来确保正确的初始化。构造器的定义还涉及方法重载的概念 (因为你可能同时需要几个构造器)。随后讨论的是清除过程, 它并非总是如想象般的那么简单。通常地, 当你不再使用一个对象时, 可以不必管它, 垃圾回收器会最终跟随介入, 释放对象占据的内存。这部分详细探讨了垃圾回收器以及它的一些特性。本章最后将更近地观察初始化过程: 自动成员初始化、指定成员初始化、初始化的顺序、static (静态) 初始化以及数组初始化等等。

第 5 章: 隐藏实现细节

(相应的讲座在光盘上)。本章探讨程序代码被封装到一起的方式, 以及为什么类库的某些部分是暴露的, 而有一部分则处于隐藏状态。首先要讨论的是 package 和 import 关键字, 它们执行文件级别上的封装操作, 并允许你构造类库。然后探讨目录路径和文件名的问题。本章最后部分讨论 public, private 以及 protected 关键字、包内访问的概念以及在不同场合下使用不同访问控制级别的意义。

第 6 章: 复用类

(相应的讲座在光盘上)。复用类的最简单方法是通过组合 (composition) 将对象嵌入到你的新类中。不过, 组合不是从已有的类产生新类的唯一方式。继承这个概念几乎是所有

OOP 语言的标准。它是对已有的类加以利用，并为其添加新功能的一种方式（也可以是修改它，这是第 7 章的主题）。继承常常用于代码的复用，它通过保留相同的“基类”，并且只是将这儿或那儿的東西补缀起来以产生你所期望的类型。在这一章中，大家将学习在 Java 中组合和继承是怎样重用代码的，以及具体如何运用它们。

第 7 章：多态

（相应的讲座在光盘上）。如果靠你自己，你可能要花上 9 个月的时间才能发现和理解多态，这是 OOP 的基础。通过一些小的、简单的例子，你将会看到如何用继承来创建一族类型，并通过它们共有的基类对该族类型中的对象进行操作。Java 的多态可以让你同等地对待同一族中的所有对象，这意味着你编写的大部分代码不必依赖特定的类型信息。这使你的代码更具灵活性，因此，程序的构建和源代码的维护可以变得更为简单，花费也更少。

第 8 章：接口和内部类

Java 提供了专门的工具来建立设计和重用关系：接口，它是对象接口的纯粹抽象。Interface 不仅仅只是达到极致的抽象类，由于它允许你通过创建某个可以向上转型到多个基类的类，因此它也实现了类似于 C++ “多重继承”的变体。

首先，内部类看起来似乎是一种简单的程序代码隐藏机制；你只需将类放置到其他类中。不过，你将会获悉内部类不仅仅只是这些；它可以知晓外围类并能与之通信。你用内部类编写的这种代码更优雅、更清晰。不过，它是一个全新的概念，需要花费一些时间才能习惯于用内部类进行设计。

第 9 章：异常与错误处理

Java 的基本设计哲学是结构不佳的代码将不能运行。编译器会尽可能地去捕获问题，但有时某一问题——或者是程序员错误，或者作为正常执行程序一部分的情形下自然发生的错误——只能在运行时被监测到和被处理。Java 具有异常处理机制用来处理在程序运行时产生的任何问题。本章将解释 try、catch、throw、throws 以及 finally 等关键字在 Java 中是怎样运行的，什么时刻你应当“抛”出异常，以及在捕获到它们时应该做些什么。另外，你还会看到 Java 的标准异常，如何创建自己的异常，在构造器中异常会发生什么，以及如何在异常期间发现异常句柄。

第 10 章：类型检查

当你仅持有一个对某对象基类的引用时，Java 运行时类型识别（RTTI）能让你找出这一对象的确切类型。通常地，你会需要有意地忽略对象的确切类型，以便让 Java 的动态绑定机制（多态）能够为那一类型实现恰当的行为。但有时候，当你仅有对某一对象的基类引用时，能够知道该对象的确切类型则会很有帮助。通常这些信息可以让你更有效地执行某些特殊情况下的操作。本章还将介绍 Java 的反射（reflection）机制。你将会知道 RTTI 和反射是什么，它们是如何使用的，以及当不再需要 RTTI 时，如何避免使用。

第 11 章：对象的集合

一个程序如果只拥有固定数量的对象，并且这些对象的存在时间已知，那么这个程序只会是一个非常简单的程序。但是通常情况下，你的程序总会在不同的时刻创建出一些新的对象，而这些时刻只有在程序运行时才可以知道。此外，除非进入运行期，否则你无法知道所需要的对象数量，以及它们的确切类型。为了解决这个常见的程序设计问题，我们需要在任何时间、任何地点创建任何数量的对象。本章深入地探讨 Java 所提供的集合库：从简单的数组到复杂的容器（数据结构），如 Vector 和 Hashtable，以便你在使用它们时可以持有自己所需的一些对象。

第 12 章：Java I/O 系统

理论上，你可以将任何程序都分成三部分：输入、处理和输出。这意味着 I/O（输入 / 输出）是程序非常重要的一部分。在这一章，你将学到 Java 所提供的各种 I/O 类，用于读写文件、内存块以及控制台等。Java I/O 的演变以及 JDK 1.4 的新 I/O (nio) 也会给予阐明。此外，本节还展示了如何获取一个对象、如何对其进行“流化”操作（使对象可以写入磁盘或通过网络传送）以及如何将其重新构造，Java 的对象序列化将会为你实现这一切。另外，还将讨论 Java 的压缩库，用于 Java 归档文件格式（JAR）。最后，阐述新的优化应用程序接口（API）和正则表达式。

第 13 章：并发

Java 提供了一套内置机制，用以支持多个被称为“线程”的并发子任务。这些线程均在单一的程序内运行。（除非你的机器里安装了多个处理器，否则这将是多子任务的唯一形式）尽管任何地方都可以应用线程，但它大多是被应用于打算创建一个反应灵敏的用户界面的情况，举例来说，虽然有一些别的任务正在执行，但用户仍然可以毫无阻碍地按下按钮或者输入数据。本章会让你在并程序序设计原则中打下坚实的基础。

第 14 章：创建窗口和 Applet 程序

Java 配套提供了 Swing GUI 类库，它是一系列类的集合，能以一种轻便的形式处理窗口。窗口化程序既可以是万维网 applet 也可以是独立的应用程序。本章将介绍用 Swing 来创建程序。还展示了 Applet 签名和 Java Web Start。同时还将介绍重要的“Java Beans”技术，它是创建“快速应用软件开发”（RAD）工具的基础。

第 15 章：问题发现

程序设计语言检测机制只能让你尽量开发出正确运行的程序。本章介绍一些工具用于解决编译器不能解决的问题。其中向前迈出的最大一步是自动单元测试的合并。对于本书，开发了一个自定义的测试系统确保程序输出的正确性，而且还介绍了一个实际中的标准——JUnit 测试系统。开放源码的工具 Ant 实现了自动构建；并且对于团队开发，阐述说明了 CVS 的基本要素。对于运行时的问题报送，本章介绍了 Java 断言机制（这里演示使用的是 Design by Contract）、日志记录 API、调试器、剖析器、以及 doclets（这些用于帮助发现源代码中的问题）。

第 16 章：分析和设计

面向对象的模式是一种用于程序设计的崭新的不同以往的思想方式，而且很多人在开始学习怎样开发 OOP 项目时都会遇到问题。一旦你理解了对象这个概念，而且随着你学会按照面向对象的方式更深入地思考，你就能够开始创建“好的”设计以充分利用 OOP 提供的好处。本章介绍了分析和设计的思想，以及一些解决问题的方法，用于在合理时间内开发出好的面向对象程序。本章的主题包括：UML 图及相关方法、用例、类—职责—合作（CRC）卡片、迭代开发、极限编程（XP）、开发和发展可重用的代码的方式，以及用于向面向对象程序设计过渡（转化）的策略。

附录 A：对象的传递和返回

尽管在 Java 中我们和对象会话的唯一方式是通过引用，但是将对象传递给方法以及将对象从方法返回的概念还是会有一些有趣的结果。此附录说明当你正进入和离开方法时你需要知道哪些是用来操纵对象的，并且还演示了 **String** 类的做法，它使用的是另外一种解决问题的方法。

附录 B：Java 编程指南

这个附录收集了一些我这几年发现和收集到的建议，当你进行低层的程序设计和编写代码时，能够帮助引导你。

附录 C：补充

以下是一些从 MindView 处可得到的附加学习材料的说明：

1. 书后面的光盘，包含 Foundations for Java, seminar-on-CD，为你学习这本书做好了准备。
2. Hands-On Java 光盘的第 3 版本，可以从 www.MindView.net 网站上获取。它是基于本书资料的一张 seminar-on-CD。
3. Think in Java Seminar。MindView 公司——一个主要的基于本书资料的介绍性研讨会。其日程安排和注册页面请参见 www.MindView.net。
4. Thinking in Enterprise Java，一本介绍了更先进的 Java 主题的书，适用于企业版程序设计。可以从 www.MindView.net 网站上获得。
5. J2EE Seminar，向你介绍真实世界 Web 使能的以及 Java 的分布式应用这些实际开发。见 www.MindView.net。
6. 对象设计及系统研讨会。面向对象的分析、设计及技术实现。见 www.MindView.net。
7. Thinking in Patterns（Java 版），介绍一些更先进的、关于设计模式和问题解决技术的 Java 主题。可在 www.MindView.net 网站上获得。
8. Thinking in Patterns Seminar。一个基于上述书籍的充满活力的研讨会。日程安排和注册页面可见 www.MindView.net。
9. Design Consulting and Reviews。协助你的项目处于良好的状态。

附录 D：资源

列出我发现特别有用的一系列 Java 书籍。

练习

在培训班上，我发现一些极其简单的例子对学生的完全理解很是有用，因此在每一章的最后你都会看到一些习题集。

大多数练习设计得都很简单，可以让学生在课堂上在合理的时间内完成这些作业，以便指导老师在查看时，可以确保所有的学生都吸收了教材的内容。有一些题目具有挑战性，但并没有难度很高的题目。（我想，你应该自己找出这些题目——或者更可能的是它们会自动找上门来）。

一些经过挑选的练习答案可以在 *The Thinking in Java Annotated Solution Guide* 的电子文档中找到，或者仅需少许费用便可以从 www.BruceEckel.com 下载得到。

CD-ROM

本书后面配套提供的还有一张光盘。过去我一直反对将光盘附在书的后面，因为我感觉支付了一些额外费用去获取大容量 CD 上的几千字节的源代码不太明智，相反更喜欢让人们到我的网站上下载这些东西。不过，你将会发现这张 CD 还有一些不同之处。

这张 CD 不包含本书的源代码，而是提供了到 www.MindView.net 网站上的一个超链接（你并不需要链接 CD 上的地址以获取代码，只要直接到那个网站去找寻即可）。这样做的原因是：在 CD 送往打印时，源代码还不完整，并且这种方式可以使代码在出现问题的任何时候都能及时得到完善和修正。

因为本书的三个版本之间有非常显著的变化，所以这张 CD 包含了 HTML 格式的本书的第一版和第二版，包括因前述理由而在较新的版本中被移除的，但是可能在某些情况下对你还是有用的章节。另外，你可以从 www.MindView.net 下载本书的当前（第三版）的 HTML 版本，而且它还包含了被发现并被订正的修订。HTML 版本的一个好处是它的索引是超链接的，所以对它的内容进行导航要简单得多。

不过 400+兆字节的 CD 大部分是一个完整的被称为 *Foundations for Java* 的多媒体教程。它包括了 *Thinking in C* 培训班讲座，介绍了一些 Java 所沿用的 C 语言的语法、操作及函数。此外，还包括源自我所开创并讲授的 *Hands-On Java seminar-on-CD* 第二版的前七章的演讲内容。尽管完整的 *Hands-On Java CD* 曾经只单独出售（第 3 版本的 *Hands-On Java CD* 也是这样，你到 www.MindView.net 网站上可以获得），我之所以决定仅包含第二版的前七章是因为它们和该书的第三版相比没有太大变化，因此它不仅可以向提供这本书的基础，而且我还希望它还能让你感受到 *Hands-On Java CD*（第 3 版）的质量和價值。

我原本打算让 **Chuck Allison** 把作为 *seminar-on-CD ROM* 一部分的 *Thinking in C* 创建一个单独产品，不过我还是决定将它和第二版本的 *Thinking in C++* 和 *Thinking in Java* 包含在一起，这样做是为了让参加培训班的、没有太多 C 语言基本语法背景的人们具有连贯一致的体验。应该抛开这种思想“我是一个聪明的程序员，我不想学习 C，而想学习 C++ 或 Java，因此我会跳过 C 直接到 C++/Java。”在到了培训班以后，这些人渐渐明白对

C 语言语法很好的理解这个先决条件很有必要。通过本书配套提供的光盘，我们相信每个参加培训班的人都能够事先做好充分准备。

这张 CD 也让这本书获得了更多的读者。即使书中第三章(程序控制流)没有介绍 Java 继承自 C 的基本部分，但是这张 CD 却进行了很好的介绍，而且它要求的学生应该具备的程序设计背景比这本书要求的还要少。并且通过 *Hands-On Java CD* 第二版中相应的讲稿来贯通前七章的资料，应该可以帮助你进入 Java 的学习之前打下更加扎实的基础。这张 CD 也期望更多的人能够加入 Java 程序设计的大军。只有在网站 www.BruceEckel.com 上直接订购后才能获得 *Hands-On Java CD ROM* 第三版本。

源代码

本书的所有源代码都可以以保留版权的免费软件的形式得到，它们是以一个单一包的形式发布的，访问 www.BruceEckel.com 网站便可获取。为了确保你获得的是最新版本，这个发布这些源代码和本书电子版的网站是一个官方网站。你也可以在其他站点上找到这本电子书籍和这些代码的镜像版本（有一些站点已经在 www.BruceEckel.com 列出），不过你应该检查这个官方网站以确保镜像版本确实是最新的版本。你可以在课堂或其他的教育场所发布这些代码。

保留版权的主要目的是为了确保源代码能够被正确地引用，并且防止在未经许可的情况下，在打印媒体中重新发布这些代码。（只要说明是引用了这些代码，那么在大多数媒介中使用本书中的示例通常不是问题。）

在每个源码文件中，都可以发现下述版本声明文字：

```
This computer source code is Copyright ©2003 MindView, Inc. All Rights Reserved.  
Permission to use, copy, modify, and distribute this computer source code  
(Source Code) and its documentation without fee and without a written agreement  
for the purposes set forth below is hereby granted, provided that the above  
copyright notice, this paragraph and the following five numbered paragraphs  
appear in all copies.
```

```
1. Permission is granted to compile the Source Code and to include the compiled  
code, in executable format only, in personal and commercial software programs.
```

```
2. Permission is granted to use the Source Code without modification in classroom  
situations, including in presentation materials, provided that the book  
"Thinking in Java" is cited as the origin.
```

```
3. Permission to incorporate the Source Code into printed  
media may be obtained by contacting  
MindView, Inc. 5343 Valle Vista La Mesa, California 91941  
Wayne@MindView.net
```

4. The Source Code and documentation are copyrighted by MindView, Inc. The Source code is provided without express or implied warranty of any kind, including any implied warranty of merchantability, fitness for a particular purpose or non-infringement. MindView, Inc. does not warrant that the operation of any program that includes the Source Code will be uninterrupted or error-free. MindView, Inc. makes no representation about the suitability of the Source Code or of any software that includes the Source Code for any purpose. The entire risk as to the quality and performance of any program that includes the Source code is with the user of the Source Code. The user understands that the Source Code was developed for research and instructional purposes and is advised not to rely

exclusively for any reason on the Source Code or any program that includes the Source Code. Should the Source Code or any resulting software prove defective, the user assumes the cost of all necessary servicing, repair, or correction.

5. IN NO EVENT SHALL MINDVIEW, INC., OR ITS PUBLISHER BE LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW, INC., OR ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. MINDVIEW, INC. SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING

SERVICES FROM MINDVIEW, INC., AND MINDVIEW, INC. HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView, Inc. maintains a web site which is the sole distribution point for electronic copies of the Source Code, <http://www.BruceEckel.com> (and official mirror sites), where it is freely available under the terms stated above.

If you think you've found an error in the Source Code, please submit a correction using the URL marked "feedback" in the electronic version of the book, nearest the error you've found.

你可以在自己的项目中引用这些代码，也可以在课堂上引用它们（包括你的演示材料），只要保留每个源文件中出现的保留版权声明即可。

编码标准

在本书的正文中，标识符（方法、变量和类名）被设为粗体。大多数关键字也被设为粗体，但是不包括那些频繁使用的关键字，例如“**class**”，因为如果将它们也设为粗体会令人十分厌烦。

对于本书中的示例，我使用了一种特定的编码格式。此格式遵循 Sun 自己在所有代码中实际使用的格式，在它的网站上你会发现这些代码（见 java.sun.com/docs/codeconv/index.html），并且似乎大多数 Java 开发环境都支持这种格式。如果你已经读过我的其他著作，你会注意到 Sun 的编码格式与我的一致——尽管这与我没有什么关系，但我还是很高兴。对代码进行格式化这个议题常常会招致几个小时的热烈争论，因此我不会试图通过自己的示例来规定正确的格式；我对我自己使用的格式有着我自己的动机。因为 Java 是一种自由形式的程序设计语言，所以你可以继续使用任何自己感觉舒服的格式。

本书中的程序都直接来自于编译过的文件，通过文本处理器，以文本形式呈现。因此，本书打印出的代码文件全部都能够运行，而且无编译错误。那些会引起编译错误消息的错误已经用 `//!` 标注出来了，以便可以使用自动方法来很容易地发现并测试它们。那些被发现并被报送给作者的错误将首先出现在发布的源代码中，并随后出现在本书的更新材料中（还会在网站 www.BruceEckel.com 上出现）。

Java 版本

在判断代码行为是否正确时，我通常以 Sun 公司的 Java 实现为参考。

这本书聚焦在 Java 2 与 JDK 1.4 上，并使用它们进行测试。如果你需要学习在本书的本版本中没有讨论过的 Java 语言的先前版本，可以从网站 www.BruceEckel.com 自由下载本书的第一版和第二版，它们也被包含在本书所附的 CD 中。

错误

无论作者使用多少技巧去发现错误，有些错误总还是悄悄地潜藏了起来，并且经常对新读者造成困扰。

由于机敏的读者所提供的反馈对我是如此的有价值，因此我开发了名为 BackTalk 的几个版本的反馈系统（该系统使用几种不同的技术，得到了 Bill Venners 的辅助，在其他很多人的帮助下得以实现）。在本书的电子版本（可以从 www.BruceEckel.com 自由下载）中，文中的每个段落都有自己唯一的 URL，点击它可以为该特定段落产生一封将你的意见记录到 BackTalk 系统的邮件。这种方式使其很容易追踪和更新修正。如果你发现了任何你确信是错误的东西，请使用 BackTalk 系统提交错误以及你建议的修正。对你的帮助不胜感激。

封面故事

《Thinking in Java》的封面创作灵感来自于美国的 Arts & Crafts 运动，该运动始于世纪之交，并在 1900 到 1920 年期间达到顶峰。它起源于英格兰，是对工业革命带来的机器产品和维多利亚时代高度装饰化风格的回应。Arts & Crafts 强调简洁设计，而回归自然是其整个运动的核心，注重手工制造以及推崇个人手工业者，可是它并不回避使用现代工具。这和我们现今的情形有很多相似之处：世纪之交，从计算机革命的最初起源到对个人来说更精简更意味深长的事物的演变，以及对软件开发技能而不仅是生产程序代码的强调。

我以同样的眼光看待 Java：尝试将程序员从操作系统的机制中解放出来，朝着“软件艺师”的方向发展。

本书的作者和封面设计者（他们自孩提时代就是朋友）从这次运动中获得灵感，并且都拥有源自那个时期的或受那个时期启发而创作的家具、台灯和其他作品。

这个封面暗示的另一主题是一个收集盒，自然学家可以用它来展示他或她保存的昆虫标本。这些昆虫被看作是对象，被放置到“盒”这个对象当中。而盒对象本身放置到“封面对象”当中，这阐释说明了面向对象程序设计中最为基本的“集合”概念。当然，程序员可能不会从中得到任何帮助，只会联想到“程序臭虫（Bug）”；这些虫子被捕获并假设在标本罐中被杀死，最后禁闭于一个展示盒中，似乎暗示 Java 有能力发现、显示和制服程序臭虫（事实上，这也是它最为强大的属性之一）

致谢

首先感谢和我一起授课，提供咨询和开发教学计划的这些合作者：**Andrea Provaglio, Dave Bartlett, Bill Venners, Chuck Allison, Jeremy Meyer, 和 Larry O'Brien**。在我转而继续去竭力为那些像我们一样的独立人群开发在一起协同工作的最佳模式的时候，你们的耐心让我感激不已。

最近，无疑是因为有了Internet，我可以和极其众多的人们一起合作，协助我一起努力，他们通常是在自己的家庭办公室（home office）中工作。过去，我可能必须为这些人们提供相当大的办公空间，不过由于现在有了网络、传真以及偶尔打打的电话，我不需要额外的开销就可以从他们的帮助中受益。在我尽力学习更好地与其他人相处的过程中，你们对我全都很有帮助，并且我希望继续学习怎样使我的工作能够通过他人的努力变得更出色。

Paula Steuer在接管我偶尔的商务实践时发挥了不可估量的价值，他使它们变得井井有条了（Paula，感谢你在我懈怠时对我的鞭策）。Jonathan Wilcox, Esq详细审视了我公司的组织结构，推翻了每一块可能隐藏祸害的石头，并且使所有事情都条理化和合法化了，这让我们心服口服。感谢你的细心和耐心。Sharlynn Cobaugh (他发现了Paula)使自己成为声音处理的专家，是创建多媒体培训CD ROM和解决其他问题的精英成员之一。感谢你在面临难于处理的计算机问题时的坚定不移。Evan Cofsky (Evan@TheUnixMan.com)已经成为了我开发过程中的重要一员，像一只鸭子那样沉迷于Python程序设计语言（嗯，这样一种混杂的隐喻可能会产生一个臃肿的Python脚本），而且解决了多种的难题，包括将BackTalk（最终？）再构造到一个email驱动的XML数据库中。在布拉格Amaio的人们

也提出了一些方案来帮助我。Daniel Will-Harris最先受到利用Internet进行工作的启发，因此他当然是我的所有设计方案的主要人物。

对于这项工程，我采用了另一个曾在我的头脑中翻腾过一段时间的措施。2002 年夏天，我在科罗拉多州的Crested Butte创建了一个实习项目，最初找到了两个实习生，而最后项目结束时有 5 个（有 2 个志愿者）。他们不仅为本书做出了贡献，而且帮助我专心致志地搞这个项目。感谢他们：JJ Badri, Ben Hindman, Mihajlo Jovanovic, Mark Welsh。Chintan Thakker能够留下并呆到第二个实习期（贯穿本书编写的最后处理过程及其他的工作），因此我必须在Mount Crested Butte租赁实习公寓，我们广招志愿者，最后招到了Mike Levin, Mike Shea, 和 Ian Phillips，他们都做出了贡献。以后我可能还要再进行其它的实习项目，请访问网站www.MindView.net查看相关消息。

感谢 Doyle Street Cohousing Community（道尔街住房社区）容忍我花了两年的时间来写这本书（并且一直在容忍我所做的一切）。非常感谢 Kevin 和 Sonda Donovan，在我编写本书第一版的夏季里，他们把位于科罗拉多州宜人的 Crested Butte 市里面的住处租给了我（也感谢 Kevin 为我在 CB 的住处所做的重新装修）。也感谢 Crested Butte 友好的居民；以及 Rocky Mountain Biological Laboratory（岩石山生物实验室），让我有宾至如归的感觉。我在 CB 的瑜伽老师 Maria 和 Brenda，在我编写第三版期间帮助我保持了健康的体魄。

当老师们来提供培训时，科罗拉多州 Crested Butte 的 Camp4 Coffee，已经成为了标准住所，并且在培训班中间休息期间，它是我所遇到的最好最便宜的饮食场所。感谢我的密友 Al Smith，是他使这里成为如此好的一个地方，成为 Crested Butte 培训期间一个如此有趣和愉快的场所。

感谢 Moore Literary Agency 的 Claudette Moore，因为她无比的耐心和毅力，我才能得到我真正想要的。感谢 Prentice Hall 的 Paul Petralia 不断地为我提供我所需要的一切，而且不厌其烦地帮我把所有事情都搞定（并容忍我所有的特殊需求）。

我的前两本书在 Osborne/McGraw-Hill 出版时，Jeff Pepper 是编辑。Jeff 总是在 Prentice Hall 恰当的地点和恰当的时间出现，他将责任转交给 Paul 之前，为这些书奠定了最初的根基。感谢你，Jeff。

感谢 Rolf André Klaedtke (瑞士); Martin Vlcek, Vlada & Pavel Lahoda, (布拉格); 和 Marco Cantu (意大利)在我第一次自行组织的欧洲研讨会巡展中对我的热情款待。

感谢 Gen Kiyooka 和他的同事 Digigami，他慷慨地为我前几年的网上授课提供了 Web 服务器。这是无价的辅助学习手段。

特别感谢Larry 和Tina O'Brien，他们帮助我把我的培训课程制作成了第一版的 Hands-On Java光盘。（你可以到网站www.BruceEckel.com查看更多消息。）

在我开发期间，某些开放源码的工具已经被证明是无价的；并且每次使用它们时都会非常感激它们的创建者。Cygwin (<http://www.cygwin.com>)为我解决了无数 Windows 不能解决的问题，并且每天我都会变得更加依赖它（如果在 15 年前当我的头脑因为使用 Gnu

Emacs 而搞得发懵的时候，能有这些该多好啊）。CVS 和 Ant 已经成为了在我的 Java 开发过程必不可少的部分，现在我已经无法再返回不用它们的时代了。我现在甚至已经变得喜欢 JUnit 了（<http://www.junit.org>），因为他们实际上已经使它成为了“可以运转的最简单的事物。”IBM 的 Eclipse（<http://www.eclipse.org>）对开发社区做出了真正杰出的贡献，并且随着它的不断升级，我期望能看到它的更伟大之处（IBM 是怎样成为潮流所向的？我肯定错过了一份备忘录）。Linux 在开发过程中每天都要用到，特别对实习生来说尤为如此。当然，如果我在其他地方强调得还不够的话，我得再次重申，我经常使用 Python（www.Python.org）解决问题，在我的密友 Guido Van Rossum 和 PythonLabs 那些身材臃肿愚笨的天才人物的智慧结晶的基础上，我花费了好几天的时间在 Zope 3 上进行极限编程（XP）（Tim，我现在已经把你借的鼠标加了个框，正式命名为“TimMouse”）。你们这伙人必须到更健康的地方去吃午餐。（还要感谢整个 Python 社区，他们是一帮令人吃惊的人们）。

很多人向我发送修正意见，我感激所有这些人，第一版特别要感谢：Kevin Raulerson（发现无数的程序臭虫），Bob Resendes（简直难以置信），John Pinto, Joe Dante, Joe Sharp（三位都难于置信），David Combs（校正了许多语法和声明），Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson, 以及许许多多的人。本书第一版本在欧洲发行时，Marc Meurrens 在电子版宣传和制作方面都做出了巨大的努力。

感谢在第 2 版本中，那些使用 Swing 类库帮助我重新编写示例的人，以及其他助手：Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis，以及所有表示支持的人。

曾经有许多技术人员走进我的生活，他们后来都和我成了朋友。他们对影响了我，并对我来说他们是不寻常的，因为他们平时练习瑜伽功，以及另一些形式的精神训练，我发现这些很具有启发性和指导意义。他们是 Kraig Brockschmidt, GenKiyooka 和 Andrea provaglio（他帮助我了解了 Java 和程序设计在意大利的概况，现在他在美国，是 MindView 团队的一员）。

对 Delphi 的一些理解使我更容易理解 Java，这一点都不奇怪，因为它们有许多概念和语言设计决策是相通的。我的 Delphi 朋友提供了许多帮助，使我能够洞察一些非凡的编程环境。他们是 Marco Cantu（另一个意大利人——难道会说拉丁语的人在学习 Java 时有得天独厚的优势？）、Neil Rubenking（直到发现喜欢计算机之前，他一直都在做瑜伽 / 素食 / 禅道），当然还有 Zack Urlocker（Delphi 产品经理），他是我游历世界时的好伙伴。

我的朋友 Richard Hale Shaw（以及 Kim）的洞察力和支持都很有帮助。Richard 和我花了数月的时间将教学内容合并到一起，并为参加学习的学生设计出一套完美的学习体验。

书籍设计、封面设计以及封面照片是由我的朋友 Daniel Will-Harris 制作的。他是一位著名的作家和设计家（<http://www.WillHarris.com>），在计算机和桌面排版发明之前，他在初中的时候就常常摆弄刮擦信（rub-on letter）。他总是抱怨我的代数含糊不清。然而，要声明的是，是我自己制作的照排好的（camera-ready）页面，所以所有排字错误都应该

算到我这里。我是用 Microsoft® Word XP for Windows 来编写这本书的，并使用 Adobe Acrobat 制作照排页面的。本书是直接从 Acrobat PDF 文件而创建的。电子时代给我们带来了厚礼，我恰巧是在海外创作了本书第一版和第二版的最终稿——第一版是在南非的开普敦送出的，而第二版却是在布拉格寄出的。第三版则来自科罗拉多州的 Crested Butte。本书中正文字体是 Georgia，标题是 Verdana。封面字体是 ITC Rennie Mackintosh。

特别感谢我的所有老师和我的所有学生（他们也是我的老师），其中最有趣的一位写作老师是 Gabrielle Rico（《Writing the Natural Way》一书的作者，Putnam 于 1983 年出版）。我将一直珍藏对在 Esalen 所经历的非凡的一周的记忆。

书后面的照片是我爱人 Dawn McGee 照的，并且是她让我那样地笑的。

曾向我提供过支持的朋友包括（当然还不止他们）：Andrew Binstock, SteveSinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, 《Midnight Engineering》杂志社的 Bill Gates, Larry Constantine 和 LucyLockwood, Greg Perry, Dan Putterman, Christi Westphal, Gene Wang, DaveMayer, David Intersimone, Andrea Rosenfield, Claire Sawyers, 另一些意大利朋友（Laura Fallai, Corrado, Ilisa 和 Cristina Giustozzi), Chris 和 Laura Strand, Almquists, Brad Jerbic, Marilyn Cvitanic, Mabrys, Haflingers, Pollocks, Peter Vinci, Robbins Families, Moelter Families（和 McMillans）, Michael Wilk, Dave Stoner, Laurie Adams, Cranstons, Larry Fogg, Mike 和 Karen Sequeira, Gary Entsminger 和 Allison Brody, KevinDonovan 和 Sonda Eastlack, Chester 和 Shannon Andersen, Joe Lordi, Dave 和 Brenda Bartlett, David Lee, Rentschlers, Sudeks, Dick, Patty 和 Lee Eckel, Lynn 和 Todd 以及他们的家人。当然还有我的爸爸和妈妈。

前言

第3版前言

Java 2, JDK 1.4

简介

前提条件

学习Java

目标

JDK的HTML文档

章节

练习

CD-ROM

源代码

编码标准

Java版本

错误

封面故事

致谢

第一章 对象引论

抽象过程

每个对象都有一个接口

每个对象都提供服务

被隐藏的具体实现

复用具体实现

继承：复用接口

是一个（is-a）与像是一个（is-like-a）关系

伴随多态的可互换对象

抽象基类和接口

对象的创建、使用和生命周期

集合（collection）与迭代器（iterator）

单根继承结构

向下转型（downcasting）与模板/泛型（template/generic）

确保正确清除

垃圾回收与效率和灵活性

异常处理：处理错误

并发（concurrency）

持久性

Java与Internet

Web是什么？

客户端编程

服务器端编程

Java为什么成功

系统易于表达、易于理解

通过类库得到最大的支持

错误处理

大型程序设计

Java与C++

总结

第二章 一切都是对象

用引用（reference）操纵对象

必须由你创建所有对象

存储到什么地方

特例：基本类型（primitive type）

Java中的数组（Array）

永远不需要销毁对象

作用域（scoping）

对象作用域（scope of object）

创建新的数据类型：类

域（field）和方法（method）

方法（Method）、参数（argument）和返回值（return value）

参数列表（argument list）

构建一个Java程序

名字可视性（Name visibility）

运用其它构件

Static 关键字

你的第一个Java程序

编译运行

注释和嵌入式文档

注释文档

语法

嵌入式HTML

一些标签实例

文档示例

代码风格

总结

练习

第三章 控制程序流

使用Java操作符

优先级

赋值

算术操作符（Mathematical operator）

正则表达式（regular expression）

自动递增（increment）和递减（decrement）

关系操作符（relational operator）

逻辑运算符（logical operator）

位操作符（bitwise operator）

移位操作符（shift operator）

三元操作符 if-else

逗号操作符（comma operator）

- 字符串操作符 +
- 使用操作符时常犯的错误
- 类型转换操作符 (casting operator)
- Java没有 “sizeof”
- 再论优先级 (precedence)
- 操作符总结

流程控制

- true和false
- if-else
- 迭代 (Iteration)
- break和 continue
- 开关 (switch)

总结

练习

第四章 初始化与清除

- 以构造器确保初始化
- 方法重载 (method overloading)
 - 区分重载方法
 - 涉及基本类型的重载
 - 以返回值区分重载方法
- 缺省构造器
- this关键字

清除 (cleanup): 终结 (finalization) 和垃圾回收 (garbage collection)

- finalize()用途何在?
- 你必须执行清除
- 终结条件
- 垃圾回收器如何工作

成员初始化

- 指定初始化
- 构造器初始化
- 初始化顺序
- 静态数据的初始化
- 明确进行的静态初始化
- 非静态实例初始化

数组初始化

- 多维数组

总结

练习

第五章 隐藏具体实现

- 包 (package): 程序库单元
 - 创建独一无二的包名
 - 定制工具库
 - 用 imports改变行为
- 对使用包 (Package) 的忠告

Java访问权限修饰词（access specifier）

包访问权限

public: 接口访问权限

private: 你不可以去碰！

Protected: 继承访问权

接口（Interface）和实现（implementation）

类的访问权限

总结

练习

第六章 复用类

组合（composition）语法

继承（inheritance）语法

初始化基类

带参数的构造器

捕捉基类构造器异常

结合使用组合（composition）和继承（inheritance）

确保正确清除

名称屏蔽（Name hiding）

组合与继承之间选择

受保护的（protected）

增量开发（incremental development）

向上转型（upcasting）

为什么要使用“向上转型”？

再次探究组合与继承

关键字final

Final 数据

空白final

final 参数

final 方法

final和**private**

final 类

有关final的忠告

初始化及类的加载

继承与初始化

总结

练习

第七章 多态

向上转型

忘记对象类型

曲解

方法调用绑定

产生正确的行为

扩展性

缺陷：“重载”私有方法

抽象类和抽象方法

构造器和多态

构造器的调用顺序

继承与清除

构造器内部的多态方法的行为

用继承进行设计

纯继承与扩展

向下转型与运行期类型标识

总结

练习

第八章 接口与内部类

接口

Java中的多重继承

组合接口时的名字冲突

通过继承来扩展接口

群组常量

初始化接口中的数据成员

嵌套接口

内部类

内部类与向上转型

在方法和作用域内的内部类

匿名内部类

链接到外部类

嵌套类

引用外围类的对象

从多层嵌套类中访问外部

内部类的继承

内部类可以被重载吗？

局部内部类（Local inner classes）

内部类标识符

为什么需要内部类？

闭包与回调

内部类与控制框架

总结

练习

第九章 异常与错误处理

基本异常

异常形式参数

捕获异常

Try块

异常处理程序（Exception handler）

创建自定义异常

异常说明

捕获所有异常

重新抛出异常

异常链

Java标准异常

运行期异常（`RuntimeException`）的特例

使用`finally`进行清理

`finally`用来做什么？

缺憾：异常丢失

异常的限制

构造器（`Constructor`）

异常匹配

其它可选方式

历史

观点

把异常传递给控制台

把“被检查的异常”转换为“不检查的异常”

异常使用指南

总结

练习

第十章 类型检查

为什么需要RTTI

Class对象

类型转换前先作检查

等价性: `instanceof` vs. `Class`

RTTI语法

反射（`Reflection`）：运行期的类信息

类方法提取器

总结

练习

第十一章 对象的集合

数组

数组是第一级对象

返回一个数组

`Arrays`类

填充数组

复制数组

数组的比较

数组元素的比较

数组排序

在已排序的数组中查找

对数组的小结

容器简介

容器的打印

填充容器

容器的缺点：未知类型

- 有时候它也能工作
 - 制作一个类型明确的ArrayList
- 迭代器
- 容器的分类法
- Collection的功能方法
- List的功能方法
 - 使用LinkedList制作一个栈
 - 使用LinkedList制作一个队列
- Set的功能方法
 - SortedSet
- Map的功能方法
 - SortedMap
 - LinkedHashMap
 - 散列算法与散列码
 - 重载hashCode()
- 持有引用
 - WeakHashMap
- 重访迭代器
- 选择接口的不同实现
 - 对List的选择
 - 对Set的选择
 - 对Map的选择
- List的排序和查询
- 实用方法
 - 设定Collection或Map为不可修改
 - Collection或Map的同步控制
- 未获支持的操作
- Java 1.0/1.1 的容器
 - Vector & Enumeration
 - Hashtable
 - Stack
 - BitSet
- 总结
- 练习

第十二章 Java I/O系统

- 文件类
 - 目录列表器
 - 匿名内部类
 - 目录的检查及创建
- 输入和输出
 - InputStream类型
 - OutputStream类型
- 添加属性和有用的接口
 - 通过FilterInputStream从InputStream中读入数据

通过FilterOutputStream向OutputStream写入
读和写

数据的来源和去处

更改“流”的行为

未发生变化的类

自我独立的类：RandomAccessFile

I/O流的典型使用方式

输入流

输出流

文件读写的实用工具

标准I/O

从标准输入读取

将System.out转换成PrintWriter

标准I/O重定向

新I/O

转换数据

获取原始类型

视图缓冲器

用缓冲器操纵数据

缓冲器的细节

存储器映射文件

性能

文件加锁

对映射文件的部分加锁

压缩

用GZIP进行简单压缩

用Zip进行多文件保存

Java档案文件（JAR）

对象序列化

寻找类

序列化的控制

transient（瞬时）关键字

Externalizable的替代方法

版本

使用“持久性”

Preferences

正则表达式

创建正则表达式

量词

模式和匹配器

split()

替换操作

reset（）

正则表达式和Java I/O

需要StringTokenizer吗？

总结

练习

第十三章 并发

动机

基本线程

让步

休眠

优先权

后台线程

加入到某个线程

编码的变体

建立有响应的用户界面

共享受限资源

不正确地访问资源

资源冲突

解决共享资源竞争

临界区

线程状态

进入阻塞状态

线程之间协作

等待与通知

线程间使用管道进行输入/输出

更高级的协作

死锁

正确的停止方法

中断阻塞线程

线程组

总结

练习

第十四章 创建Windows与Applet程序

applet基础

Applet的限制

Applet的优势

应用框架

在Web浏览器中运行applet

使用Appletviewer工具

测试applet

从命令行运行applet

一个显示框架

创建按钮

捕获事件

文本区域

控制布局

BorderLayout

FlowLayout

GridLayout

GridBagLayout

绝对定位

BoxLayout

最好的方式？

Swing事件模型

事件与监听器的类型

跟踪多个事件

Swing组件一览

按钮（Button）

图标（icon）

工具提示（Tool tip）

文本域（Text Field）

边框（Border）

滚动面板（JScrollPane）

一个迷你编辑器

检查框（Check box）

单选按钮（Radio button）

组合框（下拉列表）（Combo box（drop-down list））

列表框（list box）

页签面板

消息框（Message box）

菜单（Menu）

弹出式菜单（Pop-up menu）

绘图

对话框（Dialog box）

文件对话框（File dialog）

Swing组件上的HTML

滑块与进度条（Slider & progress bar）

树（Tree）

表格（Table）

选择外观（Look & Feel）

剪贴板（Clipboard）

把applet打包进JAR文件

为applet签名

JNLP与Java Web Start

编程技巧

动态绑定事件

将业务逻辑与用户界面逻辑分离

典型方式

Swing与并发

Runnable回顾

- 管理并发

- 可视化编程与JavaBean

- JavaBean是什么？

- 使用内省器（Introspector）来抽取出BeanInfo

- 一个更复杂的Bean

- JavaBean与同步

- 把Bean打包

- 对Bean更高级的支持

- Bean的更多信息

- 总结

- 练习

第十五章 发现问题

- 单元测试

- 一个简单的测试框架

- 利用断言提高可靠性

- 断言语法

- 为DBC使用断言

- 实例：DBC+白盒单元测试

- 用Ant构建

- 自动化所有事物

- make的问题

- Ant：事实上的标准

- 用CVS进行版本控制

- 每日构建

- 记录日志

- 记录日志级别

- 日志记录

- Handler

- 例子：发送email来报告日志消息

- 通过名字空间控制记录日志级别

- 大型工程的记录日志实践

- 小结

- 调试

- 使用JDB调试

- 图形化调试器

- 剖析和优化

- 追踪内存消费

- 追踪CPU的使用

- 覆盖测试

- JVM剖析接口

- 使用HPROF

- 线程性能

- 最优化指南

- Doclets

总结

练习

第十六章 分析与设计

方法学

阶段 0：制定计划

任务描述

阶段 1：做什么？

阶段 2：如何构建？

对象设计的五个阶段

对象开发指南

阶段 3：构建系统核心

阶段 4：迭代用例

阶段 5：演化

成功的计划

极限编程

优先编写测试

结对编程

过渡策略

指南

管理上的障碍

总结

附录A: 对象的传递与返回

传引用

别名效应

制作局部拷贝

传值

克隆对象

使类具有克隆能力

成功的克隆

`Object.clone()`的效果

克隆一个组合对象

深层拷贝ArrayList

通过序列化（`serialization`）进行深层拷贝

向继承体系的更下层增加克隆能力

为何采用此奇怪的设计？

控制克隆能力

拷贝构造器

只读类

创建只读类

恒常性（`immutability`）的缺点

恒常的String

String和StringBuffer类

String是特殊的

总结

练习

附录B: Java编程指南

设计

实现

附录C: 补充材料

“Java基础”多媒体讲座

Thinking in Java技术讲座

“Hand-on Java”多媒体讲座第三版

“对象与系统设计”技术讲座

Thinking in Enterprise Java

J2EE技术讲座

Thinking in Patterns (with Java)

Thinking in Patterns技术讲座

设计咨询与评审

附录D: 资源

软件

书籍

分析与设计

Python

我的作品

第一章 对象引论

“我们之所以将自然界分解，组织成为各种概念，并总结出其重要性，主要是因为我们知道我们的语言社区所共同持有的，并以我们的语言的形式所固定下来的一种约定...除非赞成这个约定中所颁布的有关数据的组织和分类的内容，否则我们根本无法交谈。” Benjamin Lee Whorf(1897-1941)

计算机革命起源于机器，因此，编程语言的起源也始于对机器的模仿趋近。

但是，计算机并非只是机器那么简单。计算机是头脑延伸的工具（就象 Steven Jobs 常喜欢说的“头脑的自行车”一样），同时还是一种不同类型的表达媒体。因此，这种工具看起来已经越来越不像机器，而更像我们头脑的一部分，以及一种诸如写作、绘画、雕刻、动画、电影等的表达形式一样。面向对象程序设计（Object-oriented Programming, OOP）便是这种以计算机作为表达媒体的大潮中的一波。

本章将向您介绍包括开发方法概述在内的 OOP 的基本概念。本章，乃至本书中，都假设您在过程型编程语言（Procedural Programming Language）方面已经具备了某些经验，当然不一定必须是 C。如果您认为您在阅读本书之前还需要在编程以及 C 语法方面多做些准备，您可以研读本书所附的培训光盘“Java 基础（Foundations for Java）”。

本章介绍的是背景性的和补充性的材料。许多人在没有了解面向对象程序设计的全貌之前，感觉无法轻松自在地从事此类编程。因此，此处将引入众多的概念，以期助您建立对 OOP 的扎实全面的见解。然而，还有些人可能直到在看到运行机制的某些实例之前，都无法了解面向对象程序设计的全貌，这些人如果没有代码在手，就会陷于困境并最终迷失方向。如果您属于后面的这个群体，并且渴望尽快获取 Java 语言的细节，那么您可以越过本章——在此处越过本章并不会妨碍您编写程序和学习语言。但是，您最终还是会回到本章来填补您的知识，这样您才能够了解到为什么对象如此重要，以及怎样使用对象来进行设计。

抽象过程

所有编程语言都提供抽象（abstraction）机制。可以认为，你所能够解决的问题的复杂性直接取决于抽象的类型和质量。我所谓的“类型”是指“你所抽象的是什么？”汇编语言是对底层机器的小型抽象。接着出现的许多所谓“命令式（Imperative）”语言（诸如 FORTRAN、BASIC、C 等）都是对汇编语言的抽象。这些语言在汇编语言之上有了大幅的改进，但是它们所作的主要抽象仍要求你在解决问题时要基于计算机的结构，而不是基于你试图要解决的问题的结构来考量。程序员必须建立在机器模型（Machine Model）（位于你对问题建模所在的解空间（Solution Space）内，例如计算机）和实际待解决问题模型（Problem Model）（位于问题所在的问题空间（Problem Space）内）之间的关联。建立这种映射（Mapping）是费力的，而且它不属于编程语言的内在性质，这使得程序难以编写，并且维护代价高昂。由此，产生了完整的“编程方法（Programming Method）”产业。

另一种对机器建模的方式就是对待解决问题建模。早期的编程语言，诸如 LISP 和 APL 都选择世界的某种特定视图（分别对应于“所有问题最终都是列表（List）”或者“所有问题都是算法形式的（algorithmic）”）。PROLOG 则将所有问题都转换成为决策链（Chain of decisions）。此外还产生了基于约束条件（constraint-based）编程的语言和专门通过对图形符号操作来实现编程的语言（后者被证明限制性过强）。这些方式对于它们被设计时所瞄准要解决的特定类型的问题都是不错的解决方案，但是一旦超出其特定领域，它们就力不从心了。

面向对象方式（Object-oriented approach）通过向程序员提供用来表示在问题空间中的元素的工具而更进一步。这种表示方式具有足够的概括性，使得程序员不会受限于任何特定类型的问题。我们将问题空间中的元素及其在解空间中的表示成为“对象（Object）”。（你还需要一些无法类比为问题空间元素的对象）。这种思想的实质是：程序可以通过添加新类型的对象使自身适用于某个特定问题。因此，当你在阅读描述解决方案的代码的同时，也是在阅读问题的表述。相比以前我们所拥有的所有语言，这是一种更灵活和更强有力的语言抽象。¹所以，OOP允许以问题的形式来描述问题，而不是以执行解决方案的计算机的形式来描述问题。但是它仍然与计算机有联系：每个对象看起来都有点像一台微型计算机——它具有状态，并且能够执行你赋予它的各种操作。如果要在现实世界中对象作类比，那么说它们都具有特性（Characteristic）和行为（Behavior）似乎不错。

Alan Kay 曾经总结了第一个成功的面向对象语言，同时也是 Java 赖以根基的语言之一的 Smalltalk 的五个基本特性，这些特性表现了一种纯粹的面向对象程序设计方式：

1. **万物皆为对象。**将对象视为奇特的变量，它可以存储数据，除此之外，你还可以要求它在自身上执行操作。理论上讲，你可以抽取待解问题的任何概念化构件（狗、建筑物、服务等），将其表示为程序中的对象。
2. **程序是对象的集合，它们彼此通过发送消息来调用对方。**要想产生一个对对象的请求，就必须对该对象发送一条消息。更具体地说，你可以把消息想象为对某个特定对象的方法的调用请求。
3. **每个对象都拥有由其它对象所构成的存储。**你可以通过创建包含现有对象集合的包的方式来创建新类型的对象。因此，你可以在程序中构建复杂的体系，同时将其复杂性通过对象的质朴性得以屏蔽。
4. **每个对象都拥有其类型（Type）。**按照通用的说法，“每个对象都是某个类（Class）的一个实例（Instance）”，其中“类”就是“类型”的同义词。每个类中最重要的区别于其它类的特性就是“你可以发送什么消息给它？”
5. **某一特定类型的所有对象都可以接收(Receive)同样的消息。**这是一句意味深长的表述，你在稍后便会看到。因为“圆形（circle）”类型的对象同时也是“几何形（shape）”类型的对象，所以一个“圆形”对象必定能够接受（accept）发送给“几何形”对象的消息。这意味着你可以编写与“几何形”交互并自动处理所有与几何形性质相关的事物的代码。这种“可替代性（substitutability）”是 OOP 中最强有力的概念之一。

Booch 提出了一个对对象的更加简洁的描述：对象拥有状态（State）、行为(Behaviour)和标识(Identity)。这意味着每一个对象都可以拥有内部数据（它们给出了该对象的状态）和方法（它们产生行为），并且每一个对象都可以唯一地与其他对象相区分开，具体说来，就是每一个对象在内存中都有一个唯一的地址。²

¹ 某些编程语言的设计者认为面向对象编程本身不足以轻松地解决所有编程问题，所以他们提倡将不同的方式结合到多重范式和编程语言中（multipleparadigm programming language）。您可以查阅Timothy Budd的 Multipleparadigm Programming in Leda一书（Addison-Wesley 1995）

² 这确实显得有一点过于受限了，因为对象可以存在于不同的机器和地址空间中，它们还可以被存储在硬盘上。在这些情况下，对象的标识就必须由内存地址之外的某些东西来确定了。

每个对象都有一个接口

亚里士多德大概是第一个深入研究类型（Type）的哲学家，他曾提出过鱼类和鸟类（the class of fishes and the class of birds）这样的概念。所有的对象都是唯一的，但同时也是具有相同的特性和行为的对象所归属的类的一部分，这种思想被直接应用于第一个面向对象语言 Simula-67，它在程序中使用基本关键词 `class` 来引入新的类型。

Simula，就像其名字一样，是为了开发诸如经典的“银行出纳员问题（Bank teller problem）”这样的仿真程序而创建的。在银行出纳员问题中，有出纳员、客户、账户、交易和货币单位等许多“对象”。在程序执行期间具有不同的状态而其他方面都相似的对象会被分组到对象的类中，这就是关键词 `class` 的由来。创建抽象数据类型（类）是面向对象程序设计的基本概念之一。抽象数据类型的运行方式与内置（built-in）类型几乎完全一致：你可以创建某一类型的变量（按照面向对象的说法，称其为对象或实例），然后操作这些变量（称其为发送消息或请求；你发送消息，对象就能够知道需要做什么）。每个类的成员（member）或元素（element）都共享相同的性质：每个账户都有结余金额，每个出纳都可以处理存款请求等。同时，每个成员都有其自身的状态：每个账户都有不同的结余金额，每个出纳都有自己的名称。因此，出纳、客户、账户、交易等都可以在计算机程序中被表示成为唯一的实体（entity）。这些实体就是对象，每一个对象都属于定义了特性和行为的某个特定的类。

所以，尽管我们在面向对象程序设计中实际所作的是创建新的数据类型，但事实上所有的面向对象程序设计语言都使用 `Class` 关键词来表示数据类型。当你看到类型（Type）一词时，请将其作为类（Class）来考虑，反之亦然。³

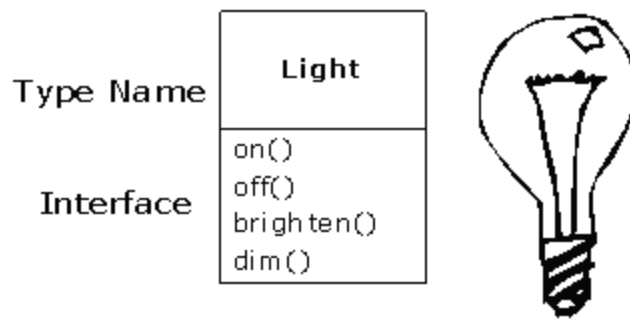
既然类被描述成了具有相同特性（数据元素）和行为（功能）的对象集合，那么一个类就确实是一个数据类型，就像所有浮点型数字具有相同的特性和行为集合一样。二者的差异在于，程序员通过定义类来适应问题，而不再被强制只能使用现有的被设计用来表示在机器中的存储单元的数据类型。你可以根据需求，通过添加新的数据类型来扩展编程语言。编程系统欣然接受新的类，并且给予它们与内置类型相同的管护和类型检查（Type-checking）。

面向对象方法并不是仅局限于构件仿真程序。无论你是否同意任何程序都是你所设计的系统的一个仿真的观念，面向对象技术确实可以将大量的问题降解为一个简单的解决方案。

一旦类被建立，你想要创建该类的多少个对象，就可以创建多少个了，然后去操作它们，就像它们是存在于你的待解问题中的元素一样。事实上，面向对象程序设计的挑战之一，就是在问题空间的元素和解空间的对象之间创建一对一的映射。

但是，你怎样才能获得对你有用的对象呢？必须有某种方式产生对对象的请求，使对象完成诸如完成一笔交易、在屏幕上画图、打开开关之类的任务。每个对象都只能满足某些请求，这些请求由对象的接口（Interface）所定义，决定接口的便是类型（Type）。以电灯泡为例来做一个简单的比喻：

³ 有些人对此还是去别对待的，他们声称类型决定了接口，而类是该接口的一个特定实现。



```
Light lt = new Light();  
lt.on();
```

接口定义了你能够对某一特定对象发出的请求。但是，在程序中必须有满足这些请求的代码。这些代码与隐藏的数据一起构成了实现（implementation）。从过程型编程的观点来看，这并不太复杂。在类型中，每一个可能的请求都有一个方法与之相关联，当你向对象发送请求时，与之相关联的方法就会被调用。此过程通常被总结为：你向某个对象发送消息（产生请求），这个对象便知道此消息的目的，然后执行对应的程序代码。

上例中，类型/类的名称是 **Light**，特定的 **Light** 对象的名称是 **lt**，你可以向 **Light** 对象发出的请求是：打开它、关闭它、将它调亮、将它调暗。你以这种方式创建了一个 **Light** 对象：定义这个对象的“引用（reference）”（**lt**），然后调用 **new** 方法来创建该类型的新对象。为了向对象发送消息，你需要声明对象的名称，并以圆点符号连接一个消息请求。从预定义类的用户观点来看，这些差不多就是用对象来进行设计的全部。

前面的图是 UML(Unified Modelling Language)形式的图，每个类都用一个方框表示，类名在方框的顶部，你所关心的任何数据成员（data member）都描述在方框的中间部分，方法（隶属于此对象的，用来接收你发给此对象的消息的函数）在方框的底部。通常，只有类名和公共方法（Public Method）被示于 UML 设计图中，因此，方框的中部并不绘出。如果你只对类型感兴趣，那么方框的底部甚至也不需要被绘出。

每个对象都提供服务

当你正是如开发或理解一个程序设计时，最好的方法之一就是将对对象想象为“服务提供者（Service Provider）”。你的程序本身将向用户提供服务，它将通过调用其它对象提供的服务来实现这一目的。你的目标就是去创建（或者最好是在现有代码库中寻找）能够提供理想的服务来解决问题的对象集合。

着手从事这件事的方式之一是询问“如果我可以将问题从表象中抽取出来，那么什么样的对象可以马上解决我的问题呢？”例如，假设你正在创建一个簿记（Bookkeeping）系统，你可以想象系统应该具有某些包括了预定义的簿记输入屏幕的对象，一个执行簿记计算的对象集合，以及一个处理在不同的打印机上打印支票和开发票的对象。也许上述对象中的某些已经存在了，但是对于那些并不存在的对象，它们看起来什么样？它们能够提供哪些服务？它

们需要哪些对象才能履行它们的义务？如果你持续这样做，你最终会发现你将到达这样一个节点：你会说“那个对象看起来很简单，以至可以坐下来写代码了”，或者说“我肯定那个对象已经存在了”。这是将问题分解为对象集合的一种合理方式。

将对象看作是服务提供者还有一个附加的好处：它有助于提高对象的内聚性(cohesiveness)。高内聚是软件设计的基本质量要求之一：这意味着一个软件构件（例如一个对象，尽管它也有可能被用来指代一个方法或一个对象库）的各个方面“组合(fit together)”得很好。人们在设计对象时所面临的一个问题是将过多的功能都填塞在一个对象中。例如，在你的检查打印模式模块中，你可以设计一个对象，它了解所有的格式和打印技术。你可能会发现这些功能对于一个对象来说太多了，你需要的是三个甚至更多个对象，其中，一个对象可以是所有可能的支票排版的目录，它可以被用来查询有关如何打印一张支票的信息；另一个对象或是对象集合可以是一个通用的打印接口，它知道有关所有不同类型的打印机的信息（但是不包含任何有关簿记的内容，它更应该是一个需要购买而不是自己编写的对象）；第三个对象通过调用另外两个对象的服务来完成打印任务。因此，每个对象都有一个它所能提供服务的高内聚的集合。在良好的面向对象设计中，每个对象都可以很好地完成一项任务，但是它并不试图多更多的事。就像在这里看到的，不仅允许某些对象可以通过购买获得（打印机接口对象），而且还使对象在某处重用成为可能（支票排版目录对象）。

将对象作为服务提供者看待是一件伟大的简化工具，它不仅在设计过程中非常有用，而且当其他人试图理解你的代码或重用某个对象时（如果他们看出了这个对象所能提供的服务的价值所在的话），它会使将对象调整到适应其设计的过程变得简单得多。

被隐藏的具体实现

将程序开发人员按照角色分为类创建者(class creator，那些创建新数据类型的程序员)和客户端程序员⁴(client programmer，那些在其应用中使用数据类型的类消费者)是大有裨益的。客户端程序员的目标是收集各种用来实现快速应用开发(Rapid Application Development)的类。类创建者的目标是构建类，该类只向客户端程序员暴露必需的部分，而隐藏其它所有部分。为什么要这样呢？因为如果加以隐藏，那么客户端程序员将不能够访问它，这意味着类创建者可以任意修改被隐藏的部分，而不用担心对其他任何人造成影响。被隐藏的部分通常代表对象内部脆弱的部分，它们很容易被粗心的或不知内情的客户端程序员所毁坏，因此将实现隐藏起来可以减少程序的Bug。

实现隐藏的概念再怎么强调也不会过分。在任何相互关系中，具有关系所涉及的各方都遵守的边界是十分重要的事情。当你创建一个类库(Library)时，你就建立了与客户端程序员之间的关系，他们同样也是程序员，但是他们是使用你的类库来构建应用，或者是构建更大的类库的程序员。如果所有的类成员(Member)对任何人都是可用的，那么客户端程序员就可以对类作任何事情，而不受任何约束。即使你希望客户端程序员不要直接操作你的类中的某些成员，但是如果没有任何访问控制，将无法阻止此事发生。所有东西都将赤裸裸地暴露于世前。

⁴ 关于这个术语的表述，我的感谢我的朋友Scott Meyers

因此，访问控制的第一个存在原因就是让客户端程序员无法触及他们不应该触及的部分——这些部分对数据类型的内部操作来说是必需的，但并不是用户需要的用来解决特定问题的接口的一部分。这对用户来说其实是一项服务，因为他们可以很容易地看出哪些东西对他们来说很重要，而哪些东西可以忽略。

访问控制的第二个存在原因就是允许库设计者可以改变类内部的工作方式而不用担心是否会影响到客户端程序员。例如，你可能为了减轻开发任务而以某种简单的方式实现了某个特定类，但稍后你就发现你必须改写它才能使其运行得更快。如果接口和实现可以清晰地分离并得以保护，那么你就可以轻而易举地完成这项工作。

Java 使用三个关键字来在类的内部设定边界：`public`、`private`、`protected`。它们的含义和用法非常易懂。这些“访问指定词（access specifier）”决定了紧跟其后被定义的东西可以被谁使用。`public` 表示紧随其后的元素对任何人都是可用的，另一方面，`private` 这个关键字表示除类型创建者和该类型的内部方法之外的任何人都不能访问的元素。`private` 就像你与客户端程序员之间的一堵砖墙，如果有人试图访问 `private` 成员，就会在编译时刻得到错误信息。`protected` 关键字与 `private` 作用相当，差别仅在于继承类（Inheriting class）可以访问 `protected` 成员，但是不能访问 `private` 成员。稍后将会对继承（Inheritance）进行介绍。

Java 还有一种缺省（default）的访问权限，当你没有使用前面提到的任何访问指定词时，它将发挥作用。这种权限通常被称为“包访问权限（package access）”，因为在这种权限下，类可以访问在同一个包中的其它类的成员，但是在包之外，这些成员如同 `private` 一样。

复用具体实现

一旦类被开发并被测试完成，那么它就应该（理想情况下）代表一个有用的代码单元。事实证明，这种复用性(reusability)并不容易达到我们所希望的那种程度，产生一个可复用的对象设计需要丰富的经验和敏锐的洞察力。但是一旦你拥有了这样的一个设计，它就会请求被复用。代码复用是面向对象程序设计语言所提供的最了不起的优点之一。

最简单的复用某个类的方式就是直接使用该类的一个对象，此外你也可以将该类的一个对象置于某个新的类中。我们称其为“创建一个成员对象”。新的类可以由任意数量、任意类型的其它对象以任意可以实现新的类中想要的功能的方式所组成。因为你在使用现有的类合成新的类，所以这种概念被称为组合（composition），如果组合式动态发生的，那么它通常被称为聚合(aggregation)。组合经常被视为“has-a”（拥有）关系，就像我们常说的“小汽车拥有引擎”一样。



（这张UML图用实心菱形声明有一辆小汽车，它表明了组合关系。我通常采用最简单的形式：仅仅是一条没有菱形的线来表示关联（association）。⁵）

⁵ 通常对于大多数图来说，这样表示已经足够了，你并不需要关心你所使用的是聚合还是组合。

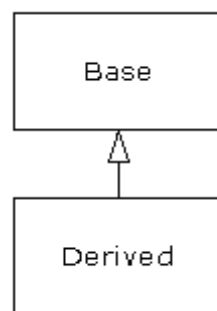
组合带来了极大的灵活性。新类的成员对象通常都被声明为 **private**，使得使用该类的客户端程序员不能访问它们。这也使得你可以在不干扰现有客户端代码的情况下，修改这些成员。你也可以在运行时刻修改这些成员对象，以实现动态修改程序的行为。下面将要讨论的继承（**inheritance**）并被具备这样的灵活性，因为编译器必须对通过集成而创建的类施加编译时刻的限制。

由于继承在面向对象程序设计中如此重要，所以它经常被高度强调，于是程序员新手就会有这样的印象：处处都应该使用继承。这会导致难以使用并过分复杂的设计。实际上，在建立新类时，你应该首先考虑组合，因为它更加简单而灵活。如果你采用这种方式，你的设计会变得更加清晰。一旦有了一些经验之后，你便能够看透必须使用继承的场合。

继承：复用接口

对象这种观念，本身就是十分方便的工具，使得你可以通过概念（**concept**）将数据和功能封装到一起，因此你可以对问题域的观念给出恰当的表示，而不用受制于必须使用底层机器语言。这些概念用关键字 **class** 来表示，形成了编程语言中的基本单位。

遗憾的是，这样做还是有很多麻烦，在创建了一个类之后，即使另一个新类与其具有相似的功能，你还是得重新创建一个新类。如果我们能够以现有的类为基础，复制它，然后通过添加和修改这个副本来创建新类那就要好得多了。通过继承便可以达到这样的效果，不过也有例外，当源类（被称为基类(**base class**)、超类(**super class**)或父类(**parent class**)）发生变动时，被修改的“副本”（被称为导出类(**derived class**)、继承类(**inherited class**)或子类(**subclass, child class**)）也会反映出这些变动。

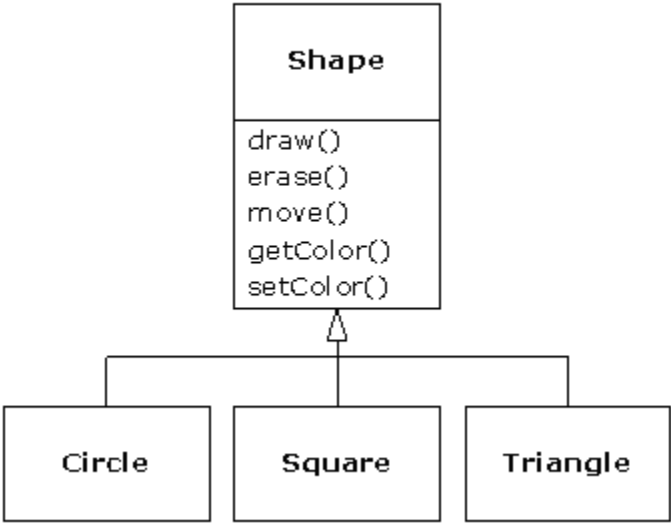


（这张 UML 图中的箭头从导出类指向基类，就像稍后你会看到的，通常会存在一个以上的导出类）

类型不仅仅只是描述了作用于一个对象集合之上的约束条件，同时还有与其它类型之间的关系。两个类型可以有相同的特性和行为，但是其中一个类型可能比另一个含有更多的特性，并且可以处理更多的消息（或以不同的方式来处理消息）。继承使用基类和导出类的概念表示了这种类型之间的相似性。一个基类包含其所有导出类共享的特性和行为。你可以创建一个基类来表示系统中某些对象的核心概念，从基类中导出其它的类，来表示此核心可以被实现的各种不同方式。

以垃圾回收机（trash-recycling machine）为例，它用来归类散落的垃圾。trash 是基类，每一件垃圾都有重量、价值等特性，可以被切碎、熔化或分解。在此基础上，可以通过添加额外的特性（例如瓶子有颜色）或行为（例如铝罐可以被压碎，铁罐可以被磁化）导出更具体的垃圾类型。此外，某些行为可能不同（例如纸的价值依赖其类型和状态）。你可以通过使用继承来构建一个类型层次结构（type hierarchy）来表示你的待解问题相关联的类型。

第二个例子是经典的在计算机辅助设计系统或游戏仿真系统中可能被泳道的几何形（shape）的例子。基类是 shape，每一个 shape 都具有尺寸、颜色、位置等，同时每一个 shape 都可以被绘制、擦除、移动和着色等。在此基础上，可以导出（继承出）具体的几何形状——圆形、正方形、三角形等——每一种都具有额外的特性和行为，例如某些形状可以被翻转。某些行为可能并不相同，例如面积计算。类型层次结构同时体现了几何形状之间的相似性和相异性。

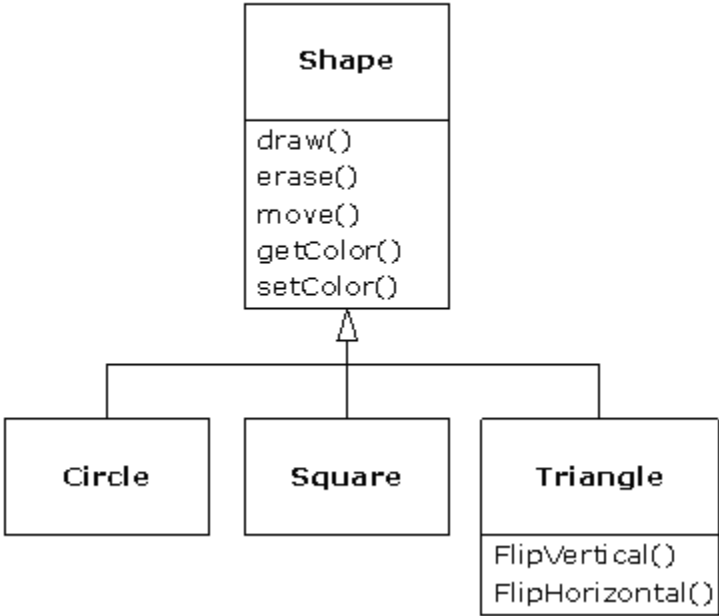


将解决方案转换成为问题术语的描述是大有裨益的，因为你不需要在问题描述和解决方案描述之间建立众多的中介模型。通过使用对象，类型层次结构成为了主要模型，因此，你可以直接从真实世界中对系统进行描述过渡到用代码对系统进行描述。事实上，对使用面向对象设计的人们来说，困难之一是从开始到结束太过于简单。对于训练有素、善于寻找复杂的解决方案的头脑来说，可能会在一开始被这种简单性给难倒。

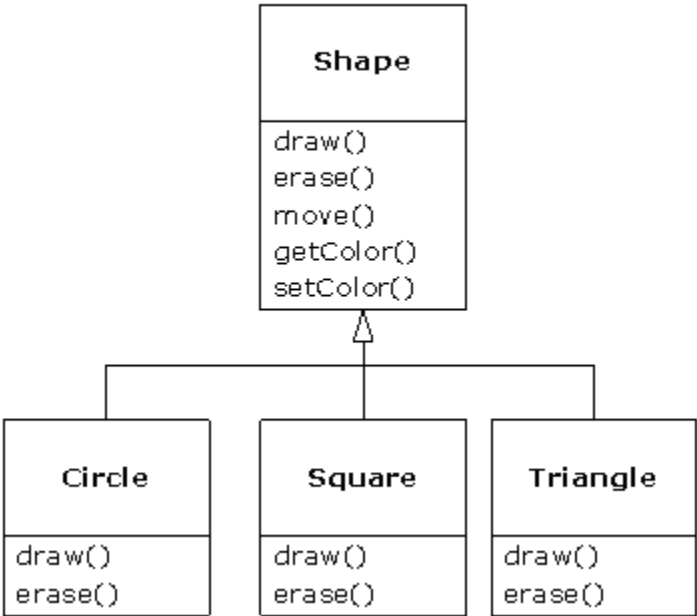
当你继承现有类型时，也就创造了新的类型。这个新的类型不仅包括现有类型的所有成员（尽管 private 成员被隐藏了起来，并且不可访问），而且更重要的是它复制了基类的接口。也就是说，所有可以发送给基类对象的消息同时也可以发送出导出类。由于我们通过可发送消息的类型可知类的类型，所以这也就意味着导出类与基类具有相同的类型。在前面的例子中，“一个圆形也就是一个几何形状”。通过继承而产生的类型等价（type equivalence）是理解面向对象程序设计方法内涵的重要门槛。

由于基类和导出类具有相同的基础接口，所以伴随此接口的必定有某些具体实现。也就是说，当对象接收到特定消息时，必须有某些代码去执行。如果你只是简单地继承一个类而并不作其他任何事，那么在基类接口中的方法将会直接继承到导出类中。这意味着导出类的对象不仅与基类拥有相同的类型，而且还拥有相同的行为，这样做并没有什么特别的意义。

有两种方法可以使基类与导出类产生差异。第一种方法非常直接：直接在导出类中添加新方法。这些新方法并不是基类接口的一部分。这意味着基类不能直接满足你的所有需求，因此你必需添加更多方法。这种对继承简单而基本的使用方式，有时对你的问题来说确实是一种完美的解决方式。但是，你应该仔细考虑是否存在你的基类也需要这些额外方法的可能性。这种设计的发现与迭代过程在面向对象程序设计中会经常发生。



虽然继承有时可能意味着在接口中添加新方法（尤其是在以 `extends` 关键字表示继承的 Java 中），但并非总需如此。第二种以及其它使导出类和基类之间产生差异的方法是改变现有基类的方法的行为。这被称之为重载（overriding）。



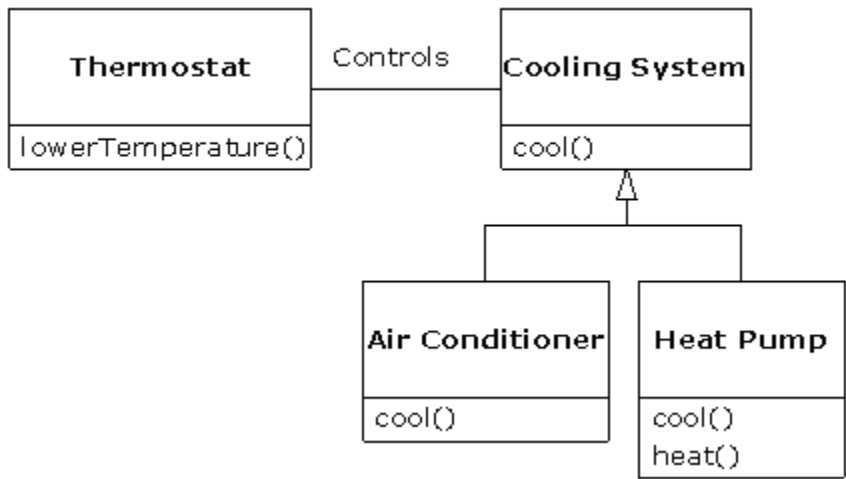
要想重载某个方法，可以直接在导出类中创建该方法的新定义即可。你可以说：“此时，我

正在使用相同的接口方法，但是我想在新类型中做些不同的事情。”

是一个（is-a）与像是一个（is-like-a）关系

对于继承可能会引发某种争论：继承应该只重载基类的方法（而并不添加在基类中没有的新方法）吗？如果这样做，就意味着导出类和基类是完全相同的类型，因为它们具有完全相同的接口。结果你可以用一个导出类对象来完全替代一个基类对象。这可以被视为“纯粹替代（pure substitution）”，通常称之为“替代法则（substitution principle）”。在某种意义上，这是一种处理继承的理想方式。我们经常将这种情况下的基类与导出类之间的关系称为“is-a”关系，因为你可以说“一个圆形就是一个几何形状”。判断是否继承，就是要确定你是否可以用 is-a 来描述类之间的关系，并使之具有实际意义。

有时你必须在导出类型中添加新的接口元素，这样也就扩展了接口并创建了新的类型。这个新的类型仍然可以替代基类，但是这种替代并不完美，因为基类无法访问你新添加的方法。这种情况我们可以描述为“is-like-a”关系。新类型具有旧类型的接口，但是它还包含其他方法，所以你不能说它们完全相同。以空调为例，假设你的房子里已经布线安装好了所有的冷气设备的控制器，也就是说，你的房子具备了让你控制冷气设备的接口。想象一下，如果空调坏了，你用一个既能制冷又能制热的热泵替换了它，那么这个热泵就“is-like-a（像是一个）”空调，但是它可以做更多的事。因为你的房子的控制系统被设计为只能控制冷气设备，所以它只能和新对象中的制冷部分进行通信。尽管新对象的接口已经被扩展了，但是现有系统除了源接口之外，对其他东西一无所知。



当然，在你看过这个设计之后，你会发现很显然，**Cooling System** 这个基类不够一般化，应该将其更名为“温度控制系统”，使其可以包括制热功能，这样我们就可以套用替代法则了。这张图说明了在真实世界中进行设计时可能会发生的事情。

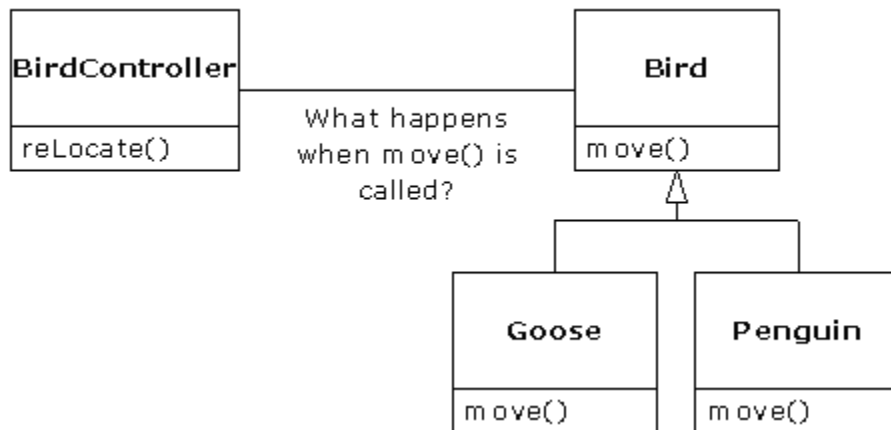
当你看到替代法则时，很容易会认为这种方式“纯粹替代”是唯一可行的方式，而且事实上言此方式，你的设计会显得很好。但是你会发现有时同样很明显你必须在导出类接口中添加新方法。只要仔细审视，两种方法的使用场合应该是相当明显的。

伴随多态的可互换对象

在处理类型的层次结构时，你经常想把一个对象不要当作它所属的特定类型来对待，而是将其当作其基类的对象来对待。这使得你可以编写出不依赖于特定类型的代码。在 `shape` 的例子中，方法都是用来操作泛化（`generic`）形状的，不管它们是圆形、正方形、三角形还是其他什么尚未定义的形状。所有的几何形状都可以被绘制、被擦除、被移动，所以这些方法都是直接对一个 `shape` 对象发送消息，并不用担心这个对象如何处理该消息。

这样的代码是不会受添加新类型的影响的，而且添加新类型是扩展一个面向对象程序已处理新情况的最常用方式。例如，你可以从 `shape` 中导出一个新的子类型 `pentagon`（无边形），而并不需要修改处理泛化几何形状的方法。通过导出新的子类型而轻松扩展设计的能力是封装改动的基本方式之一。这种能力可以极大地改善我们的设计，同时也降低了软件维护的代价。

但是，在试图将导出类型的对象当作他们的泛化基类对象来看待时（把圆形看作是几何形状，把自行车看作是交通工具，把鸬鹚看作是鸟等等），仍然存在一个问题。如果某个方法是要泛化几何形状绘制自己，泛化交通工具前进，或者是泛化的鸟类移动，那么编译器在编译时是不可能知道应该执行哪一段代码的。这就是关键所在：当发送这样的消息时，程序员并不想知道哪一段代码将被执行；绘图（`draw`）方法可以被同等地应用于圆形、正方形、三角形之上，而对象会依据自身的具体类型来执行恰当的代码。如果你不需要知道哪一段代码会被执行，那么当你添加新的子类型时，不需要更改方法调用的代码，就能够执行不同的代码。因此，编译器无法精确地了解哪一段代码将会被执行，那么它该怎么办呢？例如，在下面的图中，`BirdController` 对象仅仅处理泛化的 `Bird` 对象，而不了解它们的确切类型。从 `BirdController` 的角度看，这么做非常方便，因为不需要编写特别的代码来判定要处理的 `Bird` 对象的确切类型或是 `Bird` 对象的行为。当 `move()` 方法被调用时，即便忽略 `Bird` 的具体类型，也会产生正确的行为（鹅跑、飞或游泳，企鹅跑或游泳），那么，这又是如何发生的呢？



这个问题的答案，也是面向对象程序设计的最重要的妙诀：编译器不可能产生传统意义上的函数调用（`function call`）。一个非面向对象（`non-OOP`）编译器产生的函数调用会引起所谓的“前期绑定（`early binding`）”，这个术语你可能以前从未听说过，因为你从未想过函数调用的其他方式。这么做意味着编译器将产生对一个具体函数名字的调用，而链接器（`linker`）将这个调用解析到将要被执行代码的绝对地址（`absolute address`）。在 `OOP` 中，程序直到运行时刻才能够确定代码的地址，所以当消息发送到一个泛化对象时，必须采用其他的机制。

为了解决这个问题，面向对象程序设计语言使用了“后期绑定（late binding）”的概念。当你向对象发送消息时，被调用的代码直到运行时刻才能被确定。编译器确保被调用方法存在，并对调用参数（argument）和返回值（return value）执行类型检查（无法提供此类保证的语言被称为是弱类型的（weakly typed）），但是并不知道将会被执行的确切代码。

为了执行后期绑定，Java 使用一小段特殊的代码来替代绝对地址调用。这段代码使用在对象中存储的信息来计算方法体的地址（这个过程将在第 7 章中详述）。这样，根据这一小段代码的内容，每一个对象都可以具有不同的行为表现。当你向一个对象发送消息时，该对象就能够知道对这条消息应该做些什么。

在某些语言中，你必须明确地声明希望某个方法具备后期绑定属性所带来的灵活性（C++ 是使用 virtual 关键字来实现的）。在这些语言中，方法在缺省情况下不是动态绑定的。而在 Java 中，动态绑定是缺省行为，你不需要添加额外的关键字来实现多态（polymorphism）。

在来看看几何形状的例子。整个类族（其中所有的类都基于相同一致的接口）在本章前面已有图示。为了说明多态，我们要编写一段代码，它忽略类型的具体细节，仅仅和基类交互。这段代码和类型特定信息是分离的（decoupled），这样做使代码编写更为简单，也更易于理解。而且，如果通过继承机制添加一个新类型，例如 Hexagon，你编写的代码对 Shape 的新类型的处理与对已有类型的处理会同样出色。正因为如此，可以称这个程序是可扩展的（extensible）。

如果用 Java 来编写一个方法（后面很快你就会学到如何编写）：

```
void doStuff(Shape s) {  
    s.erase();  
    // ...  
    s.draw();  
}
```

这个方法可以与任何 Shape 交谈，因此它是独立于任何它要绘制和擦除的对象的具体类型的。如果程序中其他部分用到了 doStuff() 方法：

```
Circle c = new Circle();  
Triangle t = new Triangle();  
Line l = new Line();  
doStuff(c);  
doStuff(t);  
doStuff(l);
```

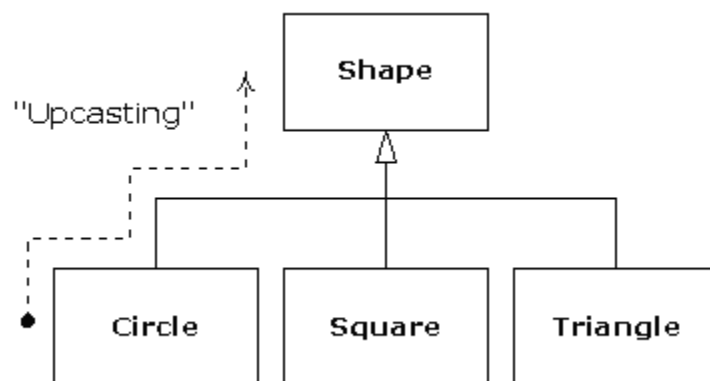
对 doStuff() 的调用会被自动地正确处理，而不管对象的确切类型。

这是一个相当令人惊奇的诀窍。看看下面这行代码：

```
doStuff(c);
```

如果被传入到预期接收 **Shape** 的方法中，究竟会发生什么呢？由于 **Circle** 可以被 **doStuff()** 看作是 **Shape**，也就是说，**doStuff()** 可以发送给 **Shape** 的任何消息，**Circle** 都可以接收，那么，这么做是完全安全且合乎逻辑的。

我们把将导出类看作是它的基类的过程称为“向上转型（upcasting）”。 “转型（cast）”这个名称的灵感来自于模型铸造的塑模动作，而“向上（up）”这个词来源于继承图的典型布局方式：通常基类在顶部，而导出类在其下部散开。因此，转型为一个基类就是在继承图中向上移动，即“向上转型（upcasting）”。



一个面向对象程序肯定会在某处包含向上转型，因为这正是你如何将自己从必须知道确切类型中解放出来的关键。让我们再看看在 **doStuff()** 中的代码：

```
s.erase();  
// ...  
s.draw();
```

注意这些代码并不是说“如果你是 **Circle**，请这样做；如果你是 **Square**，请那些做；……”。如果你编写了那种检查 **Shape** 实际上所有可能类型的代码，那么这段代码肯定是杂乱不堪的，而且你需要在每次添加了新类型的 **Shape** 之后去修改这段代码。这里你所要表达的意思仅仅是“你是一个 **Shape**，我知道你可以 **erase()** 和 **draw()** 你自己，那么去做吧，但是要注意细节的正确性。”

doStuff() 的代码给人印象深刻之处在于，不知何故，总是做了该做的。调用 **Circle** 的 **draw()** 方法所执行的代码与调用 **Square** 或 **Line** 的 **draw()** 方法所执行的代码是不同的，但是当 **draw()** 消息被发送给一个匿名的（anonymous）的 **Shape** 时，也会基于该 **Shape** 的实际类型产生正确的行为。这相当神奇，因为就象在前面提到的，当 **Java** 编译器在编译 **doStuff()** 的代码时，并不能确切知道 **doStuff()** 要处理的确切类型。所以通常会期望它的编译结果是调用基类 **Shape** 的 **erase()** 和 **draw()** 版本，而不是具体的 **Circle**、**Square** 或是 **Line** 的版本。正是因为多态才使得事情总是能够被正确处理。编译器和运行系统会处理相关的细节，你需要马上知道的只是事情会发生，更重要的是怎样通过它来设计。当你向一个对象发送消息时，即使涉及向上转型，该对象也知道要执行什么样的正确行为。

抽象基类和接口

通常在一个设计中，你会希望基类仅仅表示其导出类的接口，也就是说，你不希望任何人创建基类的实际对象，而只是希望他们将对象向上转型到基类，所以它的接口将派上用场。这是通过使用 `abstract` 关键字把类标识成为抽象类来实现的。如果有人试图创建一个抽象类的对象，编译器都会加以阻止。这是支持某种特殊设计的工具。

你也可以使用 `abstract` 关键字来描述尚未被实现的方法，就象一个存根，用来表示“这是一个从此类中继承出的所有类型都具有的接口方法，但是此刻我没有为它设计任何具体实现。”抽象方法只能在抽象类内部创建，当该类被继承时，抽象方法必须被实现，否则继承类仍然是一个抽象类。创建抽象方法使得你可以将一个方法置于接口中而不必被迫为此方法提供可能毫无意义的方法体。

`Interface`（接口）这个关键字比抽象类的概念更进了一步，它压根不允许有任何方法定义。接口是一个非常方便而通用的工具，因为它提供了接口与实现的完美分离。此外，只要你愿意，你就可以将多个接口组合到一起，与之相对照的，你要继承多个一般类或抽象类却是不可能的。

对象的创建、使用和生命周期

从技术上说，OOP 只是涉及抽象数据类型、继承和多态，但是其他议题至少也同样重要，本节将涵盖这些议题。

对象最重要的要素之一便是它们的生成和销毁。对象的数据位于何处？怎样控制对象的生命周期？关于此存在着不同的处理哲学。C++认为效率控制是最重要的议题，所以提供选择给程序员。为了追求最大的执行速度，对象的存储空间和生命周期可以在编写程序时确定，这可以通过将对象置于堆栈（它们有时被称为自动变量（`automatic variable`）或限域变量（`scoped variable`））或静态存储区域内来实现。这种方式将存储空间分配和释放置于优先考虑的位置，某些情况下这样控制非常有价值，但是，也牺牲掉了灵活性，因为你必须在编写程序时知道对象确切的数量、生命周期和类型。如果你试图解决更一般化的问题，例如计算机辅助设计、仓库管理或者是空中交通控制，这种方式都显得过于受限了。

第二种方式是在被称为堆（`heap`）的内存池中动态地创建对象。在这种方式中，你直到运行时刻才知道需要多少对象？它们的生命周期如何？以及它们的具体类型是什么？这些问题的答案只能在程序运行时相关代码被执行到的那一刻才能确定。如果你需要一个新对象，你可以在你需要的时刻直接在堆中创建。因为存储空间是在运行时刻被动态管理的，所以需要大量的时间在堆中分配存储空间，这可能要远远大于在堆栈中创建存储空间的时间。（在堆栈中创建存储空间通常只需要一条将栈顶指针向下移动的汇编指令，另一条汇编指令对应释放存储空间所需的将栈顶指针向上移动。创建堆存储空间的时间以来于存储机制的设计）。动态方式有这样一个逻辑假设：对象趋向于变得复杂，所以查找和释放存储空间的开销不会对对象的创建造成重大冲击。动态方式所带来的更大的灵活性正是解决一般化编程问题的要点所在。

Java完全采用了第二种方式⁶。每当你想要创建新对象时，就要使用new关键字来构建此对象的动态实例。

还有另一个议题，就是对象生命周期。对于允许在堆栈上创建对象的语言，编译器可以确定对象存活的时间有多久，并可以自动销毁它。然而，如果你在堆上创建对象，编译器就会对它的生命周期一无所知。在像 C++ 这类的语言中，你必须通过编程方式来确定何时销毁对象，这可能会因为你不能正确处理而导致内存泄漏（这在 C++ 程序中是常见的问题）。Java 提供了被称为“垃圾回收器（garbage collector）”的机制，它可以自动发现对象何时不再被使用，并继而销毁它。垃圾回收器非常有用，因为它减少了你必须考虑的议题和你必须编写的代码。更重要的是，垃圾回收器提供了更高层的保障，可以避免暗藏的内存泄漏问题（这个问题已经使许多 C++ 项目折戟沉沙）。

集合（collection）与迭代器（iterator）

如果你不知道在解决某个特定问题时需要多少个对象，或者它们将存活多久，那么你就不可能知道如何存储这些对象。你如何才能知道需要多少空间来创建这些对象呢？答案是你不可能知道，因为这类信息只有在运行时刻才能获得。

对于面向对象设计中的大多数问题而言，这个问题的解决方案似乎过于简单：创建另一种对象的类型。解决这个特定问题的新的对象类型持有对其它对象的引用。当然，你可以用在大多数语言中都可获得的数组类型来实现相同的功能。但是这个通常被称为容器（container，也被称为集合（collection），但是 Java 类库以不同的含义使用这个术语，所以本书将使用容器这个词）的新对象将在任何需要时可扩充自己的容量以容纳你放置于其中的所有东西。因此你不需要知道将来会把多少个对象置于容器中，只需要创建一个容器对象，然后让它处理所有细节。

幸运的是，好的 OOP 语言都具有作为开发包一部分的一组容器。在 C++ 中，容器是标准 C++ 类库的一部分，有时也称为标准模板类库（Standard Template Library, STL）。Object Pascal 在其可视化构件库（Visual Component Library）中具有容器。Smalltalk 提供了一个非常完备的容器集。Java 在其标准类库中也有容器。在某些类库中，通用容器足够满足所有的需要，但是在其它类库（例如 Java）中，具有满足不同需要的不同类型的容器，例如 List 类（列表，用于存储序列），Map 类（散列表，也被称为关联数组，用来建立对象之间的关联），Set 类（集类，用于存储一类对象）。容器类库还可能包括 Queue（队列）、Tree（树）、Stack（堆栈）等。

所有容器都有某种方式来处理元素的置入和取出。某些通用的方法用来在容器中添加元素，而另一些用来将元素取出。但是取出元素可能问题会更多一些，因为单一选取（single-selection）的方法是很受限的。如果你想操作或是比较容器中的一组元素时，用什么方式来替代单一选取呢？

⁶ 稍候您将看到，原始类型只是一种特例。

解决的方法是迭代器（iterator），它是一个用来选取容器中的元素，并把它呈现给迭代器用户的对象。作为一个类，它也提供了某种抽象层次。这种抽象可以用来把容器的细节从访问容器的代码中分离出来。容器通过迭代器被抽象为仅仅是一个序列（sequence）。迭代器允许你遍历这个序列而不用担心底层的结构，也就是说，不用关心它是一个 `Arraylist`、`LinkedList`、`Stack` 还是其他什么东西。这给你提供了极大的灵活性，使得你不用干扰你的程序代码就可以十分方便地修改底层数据结构。Java 的 1.0 和 1.1 版本有一个为所有容器类设计的被称为 `Enumeration` 的标准迭代器，Java 2 增加了一个完备得多的容器类库，其中包含一个被称为 `Iterator` 的比老式的 `Enumeration` 能做得更多的迭代器。

从设计的观点来看，你真正需要的只是一个可以被操作，从而解决问题的序列。如果单一类型的序列可以满足你的所有需要，那么就没有理由设计不同种类的序列了。有两个原因使得你还是需要对容器有所选择。第一，不同容器提供了不同类型的接口和外部行。堆栈与队列就具备不同的接口和行为，也不同于集合（set）和列表。其中某种容器提供的问题解决方案可能比其它容器要灵活的多。第二，不同的容器对于某些操作具有不同的效率。最好的例子就是两种 `List` 的比较：`ArrayList` 和 `LinkedList`。它们都是具有相同接口和外部行为的简单的序列，但是它们对某些操作所花费的代价却有天壤之别。在 `ArrayList` 中随机访问元素是一个花费固定时间的操作，但是，对 `LinkedList` 来说，随即选取元素需要在列表中移动，其代价是高昂的，访问越靠近表尾的元素，花费的时间越长。另一方面，如果你想在序列中间插入一个元素，`LinkedList` 的开销却比 `ArrayList` 要小。上述以及其它操作的效率，依序列底层结构的不同而存在很大的差异。在设计阶段，你开始可以使用 `LinkedList`，在优化系统性能时，改用 `ArrayList`。基类 `List` 和迭代器所带来的抽象把你在容器之间进行转换时对代码产生的影响降到了最低。

单根继承结构

在 OOP 中有一个议题，自 C++ 面世以来变得非常瞩目，那就是是否所有的类最终都继承自单一的基类。在 Java 中（事实上还包括除 C++ 以外的所有 OOP 语言）的答案是 yes，这个终极基类的名字为 `Object`。事实证明，单根继承结构带来了很多好处。

在单根继承结构中的所有对象都具有一个共用接口，所以它们归根到底都是相同的基本类型。另一种（C++ 所提供的）是你无法确保所有对象都属于同一个基本类型。从向后兼容的角度看，这么做能够更好地适应 C 模型，而且受限较少，但是当你要进行完全的面向对象程序设计时，你必须都要构建自己的继承体系，使得它可以提供其他 OOP 语言内置的便利。并且在你获得的任何新类库中，总会用到一些不兼容的接口，你需要花费力气（并有可能要通过多重继承）来使得新接口融入你的设计之中。这么做以换取 C++ 额外的灵活性是否值得呢？如果你需要的话——你在 C 上面投资巨大——那这么做就很有价值。如果你刚刚从头开始，那么像 Java 这样的选择通常会更高效高产。

单根继承结构（例如 Java 所提供的）保证所有对象都具备某些功能。因此你了解在你的系统中你可以在每个对象上都可以执行的某些基本操作。单根继承结构以及在堆上创建所有对象，极大地简化了参数传递（这在 C++ 中是十分复杂的话题之一）。

单根继承结构使垃圾回收器（内置于 Java 中）的实现变得容易得多。其必需的支持功能可置于基类中，这样，垃圾回收器就可以发送恰当的消息给系统中的每一个对象。如果没有单

根继承结构以及通过引用来操作对象的系统特性，要实现垃圾回收器非常困难。

由于所有对象都保证具有运行时刻类型信息（run time type information），因此你不会因无法确定对象的类型而陷入僵局。这对异常处理这样的系统级操作显得尤其重要，并且给编程带来了更大的灵活性。

向下转型（downcasting）与模板/泛型（template/generic）

为了复用上述容器，我们让它们都可以存储Java中的一个通用类型：**Object**。单根继承结构意味着所有东西都是对象，所以可以存储**Object**的容器可以存储任何东西⁷。这使得容器很容易被复用。

要使用这样的容器，你只需在其中置入对象引用（object reference），稍后还可以将它们取回。但是由于容器只存储 **Object**，所以当你将对象引用置入容器时，它必须被向上转型为 **Object**，因此它会丢失其身份。当你把它取回时，你获取了一个对 **Object** 对象的引用，而不是对你置入时的那个类型对象的引用。所以，你怎样才能将它变回先前你置入容器中的具有实用接口的对象呢？

这里再度用到了转型，但这一次不是向继承结构的上层转型为一个更泛化的类型，而是向下转型为更具体的类型。这种转型的方式称为向下转型（downcasting）。你可以知道向上转型是安全的，例如 **Circle** 是一种 **Shape** 类型，但是你无法知道某个 **Object** 是 **Circle** 还是 **Shape**，所以除非你确切知道你要处理的对象的类型，否则向下转型几乎是不安全的。

然而向下转型并非彻底是危险的，因为如果你向下转型为错误的类型，你会得到被称为异常（exception）的运行时刻错误，稍后我会介绍什么是异常。尽管如此，当你从容器中取出对象引用时，还是必须要由某种方式来记住这些它们究竟是什么类型，这样你才能执行正确的向下转型。

向下转型和运行时刻的检查需要额外的程序运行时间和程序员心血。那么创建知道自己所保存对象的类型的容器，从而消除向下转型的需求和犯错误的可能不是更有意义吗？其解决方案被称为参数化类型（parameterized type）机制。参数化类型就是编译器可以自动定制作用于特定类型之上的类。例如，通过使用参数化类型，编译器可以定制一个只接纳和取出 **Shape** 对象的容器。

参数化类型是 C++的重要组成部分，部分原因是 C++压根没有单根继承结构。在 C++中，实现参数化类型的关键字是“**template**（模板）”。Java 目前并没有参数化类型，因为通过使用单根继承结构可以达到相同的目的，尽管其机制显得比较笨拙。但是，目前已经有了一份采用与 C++ 模板极其相似的语法来实现参数化类型的提案，我们期待在下一个 Java 版本中看到参数化类型。

⁷ 很遗憾，原始类型排除在外。本书稍后会详细讨论这一点。

确保正确清除

每个对象为了生存都需要资源，尤其是内存。当我们不再一个对象时，它必须被清除掉使其占有的资源可以被释放和重用。在相对简单的编程情况下，怎样清除对象看起来似乎不是什么挑战：你创建了对象，根据需要使用它，然后它应该被销毁。然而，你很可能会遇到相对复杂的情况。

例如，假设你正在为某个机场设计空中交通管理系统（同样的模型在仓库货柜管理系统、录像带出租系统或是宠物寄宿店也适用）。一开始问题似乎很简单：创建一个容器来保存所有的飞机，然后为每一架进入控制交通控制区域的飞机创建一个新的飞机对象，并将其置于容器中。对于清除工作，只需在飞机离开此区域时删除相关的飞机对象即可。

但是，可能还另有某个系统记录着有关飞机的数据，也许这些数据不需要像主要的控制功能那样立刻受到人们的注意。例如，它可能记录着所有飞离飞机场的小型飞机的飞行计划。因此你需要有第二个容器用来存放小型飞机，无论何时，只要创建的是小型飞机对象，那么它同时也应该置入第二个容器内。然后某个后台进程在空闲时间对第二个容器内的对象执行操作。

现在问题变得更困难了：你怎样才能知道何时销毁这些对象呢？当你处理完某个对象之后，系统其他的某部分可能正在处理它。在其他许多场合中也会遇到同样的问题，在必须明确删除对象的编程系统中（例如 C++），此问题会变得十分复杂。

Java 的垃圾回收器被设计用来处理内存释放问题（尽管它不包括清除对象的其他方面）。垃圾回收器“知道”对象何时不再被使用，并自动释放该对象的内存。这与所有对象都是继承自单根基类 `Object`，以及你只能以一种方式创建对象——在堆上创建这两个特性一起，使得用 Java 编程的过程较之用 C++ 编程要简单得多，你要做的决策和要克服的障碍都要少得多。

垃圾回收与效率和灵活性

如果这么做完美无瑕，那为什么 C++ 没有采用呢？因为你必须要为编程的方便付出代价，这个代价就是运行时刻的开销。就像前面提到的，在 C++ 中，你在堆栈中创建对象，在这种情况下，它们可以自动被清除（但是无法得到在运行时刻你想要得到的灵活性）。在堆栈上创建对象是为对象分配和释放存储空间最有效的途径。在堆上创建对象可能代价就要高昂得多。总是从某个基类继承以及所有的方法调用都是多态的也需要较小的开销。但是垃圾回收器是一个特殊的问题，因为你从来都不确切了解它将于何时启动并将持续多久。这意味着一个 Java 程序的执行速度会有前后不一致的情况，因此你在某些场合不能使用它，例如强调程序的执行速度要一致的场合。（此类程序通常被称为实时程序，尽管不是所有的实时编程问题都是如此严苛。）

C++ 语言的设计者努力争取 C 程序员的支持，（他们几乎已经成功，但是）却不想添加任何影响速度的功能，也不想添加任何能够使程序员在选择使用 C 的场合转而选择 C++ 的新功能。这个目标是实现了，但是付出的代价是在用 C++ 编程时复杂性极高。

异常处理：处理错误

自从编程语言问世以来，错误处理就始终是最困难的问题之一。因为设计一个良好的错误处理机制非常困难，所以许多语言直接略去这个问题，将其交给程序库设计者处理，而这些设计者也只是提出了一些不彻底的方法，这些方法可用于许多很容易就可以绕过此问题的场合，而且其解决方式通常也只是忽略此问题。大多数错误处理机制的主要问题在于，它们都依赖于程序员自身的警惕性，这种警惕性是靠遵循人们已经达成一致的惯例而保持的，而这种惯例并不是编程语言所强制的。如果程序员不够警惕——通常是因为他们太忙——这些机制就很容易被忽视。

异常处理将错误处理直接置于编程语言中，有时甚至置于操作系统中。异常是一种对象，它从出错地点被“抛出（thrown）”，并被适当的专门被设计用来处理特定类型异常的异常处理器“捕获（caught）”。异常处理就像是与程序正常执行路径并行的，在错误发生时执行的另一条路径。因为它是另一条完全分离的执行路径，所以它不会干扰正常的执行代码。这使得代码编写变得简单了，因为你不需要被迫定期检查错误。此外，被抛出的异常不像方法返回的错误值和方法设置的用来表示错误条件的标志位那样可以被忽略。异常不能被忽略，所以它保证一定会在某处被处理。最后需要指出的是：异常提供了一种从错误状况进行可靠恢复的途径。现在不再是只能推出程序，你可以经常进行校正，并恢复程序的执行，这些都有助你编写出健壮性好得多的程序。

Java 的异常处理在众多的编程语言中格外引人注目，因为 Java 一开始就内置了异常处理，而且强制你必须使用它。如果你没有编写正确的处理异常的代码，那么你就会得到一条编译时刻的出错消息。这种得到确保的一致性有时会使得错误处理非常容易。

值得注意的是，异常处理不是面向对象的特征，尽管在面向对象语言中异常通常被表示成为一个对象。异常处理在面向对象语言出现之前就已经存在了。

并发（concurrency）

在计算机编程中有一个基本概念，就是在同一时刻处理多个任务（task）的思想。许多程序设计问题都需要程序能够停下正在做的工作，转而处理某个其它问题，然后再返回主进程（main process）。有许多方法可以实现这个目的。最初，程序员们用所掌握的有关机器底层的知识来编写中断服务程序（interrupt service routine），主进程的挂起（supension）是通过硬件终端来触发的。尽管这么做可以解决问题，但是其难度太大，而且不能够移植，所以使得将程序移植到新型号的机器上时，既费时又费力。

有时中断对于处理时间临界（time-critical）的任务是必需的，但是对于大量的其它问题，我们只是想把问题切分成多个可独立运行的部分，从而提高程序的响应能力。在程序中，这些彼此独立运行的部分称之为线程（thread），上述概念被称为“并发（concurrency）”或“多线程（multithreading）”。多线程最常见的例子就是用户界面。通过使用线程，用户可以在撤

下按钮后快速得到一个响应，而不用强制等待直到程序完成当前任务为止。

通常，线程只是一种为单一处理器分配执行时间的手段。但是如果操作系统支持多处理器，那么每个线程都可以被指派给不同的处理器，并且它们是在真正地并行执行。在语言级别上多线程所带来的便利之一便是程序员不用再操心机器上有多个处理器还是只有一个处理器。由于程序被逻辑化分为线程，所以如果机器拥有多个处理器，那么程序将在不需要特殊调整的情况下执行得更快。

所有这些都使得线程看起来相当简单，但是有一个隐患：共享资源。如果有超过一个的并行线程都要访问同一项资源，那么就会出问题。例如，两个进程不能同时向一台打印机发送信息。为了解决这个问题，可以共享的资源，例如打印机，必须在被使用期间锁定。因此，整个过程是：某个线程锁定某项资源，完成其任务，然后释放资源锁，使其它线程可以使用这项资源。

Java 的线程机制是内置于其中的，它使此复杂课题变得简单得多了。线程机制被对象层次所支持，因此线程的执行可以用对象来表示。Java 同时也提供了限制性资源锁定功能，它可以锁定任何对象所占用的内存（毕竟这也算是某种共享资源），使得同一时刻只能有一个线程在使用它。这是通过 `synchronized` 关键字来实现的。其它类型的资源必须由程序员显式地锁定，通常是通过创建一个表示锁的对象，所有线程在访问资源之前先检查这个对象。

持久性

当你创建了一个对象之后，只要你需要它，它就一直存活着，但是在程序终止后，它无论如何都不能存活了。在某些场合，如果对象在程序非执行状态下仍然能够存活，并保存其相关信息，将非常有用。当你下一次重新启动程序时，这个对象能够重生，并且拥有与上一次程序执行时相同的信息。当然，你可以通过将信息写入文件或数据库中达到相同的效果，但是在“万物皆为对象”的精神下，能够将对象声明为持久的（`persistent`），并让语言系统为你处理所有细节，不是非常方便吗？

Java 提供对“轻量级持久性（`lightweight persistent`）”的支持，这意味着你可以很容易地将对象存储在磁盘上，并在以后取回它们。称之为“轻量级”是因为你仍然得创建显式的调用来执行存储和取回操作。轻量级持久性可以通过对象序列化（`object serialization`，第 12 章介绍）或 Java 数据对象（JDO, Java Data Object，在《企业 Java 编程思想（Thinking in Enterprise Java）》一书中有介绍）来实现。

Java 与 Internet

如果 Java 仅仅只是众多的程序设计语言中的一种，你可能就会问：为什么它如此重要？为什么它促使计算机编程语言向前迈进了革命性的一步？如果从传统的程序设计观点看，问题的答案似乎不太明显。尽管 Java 对于解决传统的单机程序设计问题非常有用，但同样重要的是，它能够解决在万维网（`world wide web`）上的程序设计问题。

Web 是什么？

Web 一词乍一看有点神秘，就象“网上冲浪（surfing）”、“表现（presence）”、“主页（home page）”一样。我们回头审视它的真实面貌有助于对它的理解，但是要这么做就必须先理解客户/服务器（client/server）系统，它使计算技术中另一个充满了诸多疑惑的话题。

客户/服务器计算技术

客户/服务器系统的核心思想是：系统具有一个中央信息存储池（central repository of information），用来存储某种数据，它通常存在于数据库中，你可以根据需要将它分发给某个人员或机器集群。客户/服务器概念的关键在于信息存储池的位置集中于中央，这使得它可以被修改，并且这些修改将被传播给信息消费者。总之，信息存储池是用于分发信息的软件，信息与软件的宿主机（或机器的集群）被称为服务器（server）。宿主于远程机器上的软件与服务器进行通信，以获取信息、处理信息，然后将它们显示在被称为客户（client）的远程机器上。

客户机/服务器计算技术的基本概念并不复杂。问题在于你只有单一的服务器，却要同时为多个客户服务。通常，这都会涉及数据库管理系统，因此设计者“权衡”数据置于数据表（table）中的结构，以取得最优的使用效果。此外，系统通常允许客户在服务器中插入新的信息。这意味着你必须保证一个客户插入的新数据不会覆盖另一个客户插入的新数据，也不会在将其添加到数据库的过程中丢失（这被称为事务处理（transaction processing））。如果客户端软件发生变化，那么它必须被重新编译、调试并安装到客户端机器上，事实证明这比你想象中的要更加复杂与费力。如果想支持多种不同类型的计算机和操作系统，问题将更麻烦。最后还有一个最重要的性能问题：可能在任意时刻都有成百上千的客户向服务器发出请求，那么随便多么小的延迟都会产生重大影响。为了将延迟最小化，程序员努力地减轻处理任务的负载，通常是分散给客户端机器处理，但有时也会使用所谓“中间件（middleware）”将负载分散给在服务器端的其它机器。（中间件也被用来提高可维护性（maintainability））

分发信息这个简单思想的复杂性实际上是有很多不同层次的，这使得整个问题可能看起来高深莫测得让人绝望。但是它仍然至关重要：算起来客户/服务器计算技术大概占了所有程序设计行为的一半，从制定订单、信用卡交易到包括股票市场、科学计算、政府、个人在内的任意类型的数据分发。过去我们所作的，都是针对某个问题发明一个单独的解决方案，所以每一次都要发明一个新的方案。这些方案难以开发并难以使用，而且用户对每一个方案都要学习新的接口。因此，整个客户/服务器问题需要彻底地解决。

Web 就是一台巨型服务器

Web 实际上就是一个巨型客户/服务器系统，但是比其稍微差一点，因为所有的服务器和客

户机都同时共存于同一个网络中。你不需要了解这些，因为你所要关心的只是在某一时刻怎样连接到一台服务器上，并与之进行交互（即便你可能要满世界地查找你想要的服务器）。

最初只有一种很简单的单向过程（one-way process）：你对某个服务器产生一个请求，然后它返回给你一个文件，你的机器（也就是客户机）上的浏览器软件根据本地机器的格式来解读这个文件。但是很快人们就希望能够做得更多，而不仅仅是从服务器传递网页。他们希望实现完整的客户/服务器能力，使得客户可以将信息反馈给服务器。例如，在服务器上进行数据库查找、将新信息添加到服务器以及下订单（这需要比原始系统提供的安全性更高的安全保障）。这些变革，正是我们在 Web 发展过程中一直目睹的。

Web 浏览器是向前跨进的一大步，它包含了这样的概念：一段信息不经修改就可以在任意型号的计算机上显示。然而，浏览器仍然相当原始，很快就因为加诸于其上的种种需要而陷入困境。浏览器并不具备显著的交互性，而且它趋向于使服务器和 Internet 阻塞，因为在任何时候，只要你需要完成通过编程来实现的任务，就必须将信息发回到服务器去处理。这使得即便是发现你的请求中的拼写错误也要花去数秒甚至是数分钟的时间。因为浏览器只是一个视图工具，因此它甚至不能执行最简单的计算任务。（另一方面，它是安全的，因为它在你的本地机器上不会执行任何程序，而这些程序有可能包含 bug 和病毒。）

为了解决这个问题，人们采用了各种不同的方法。首先，图形标准得到了增强，使得在浏览器中可以播放质量更好的动画和视频。剩下的问题通过引入在客户端浏览器中运行程序的能力就可以解决。这被称为“客户端编程（client-side programming）”。

客户端编程

Web 最初的“服务器-浏览器”设计是为了能够提供交互性的内容，但是其交互性完全由服务器提供。服务器产生静态页面，提供给只能解释并显示它们的客户端浏览器。基本的 HTML（HyperText Markup Language，超文本标记语言）包含有简单的数据收集机制：文本输入框（text-entry box）、复选框（check box）、单选框（radio box）、列表（list）和下拉式列表（drop-down list）等，以及只能被编程用来实现复位（reset）表单上的数据或提交（submit）表单上的数据给服务器的按钮。这种提交动作传递给所有的 Web 服务器都提供的通用网关接口（common gateway interface, CGI）。提交内容会告诉 CGI 应该如何处理它。最常见的动作就是运行一个在服务器中通常被命名为“cgi-bin”的目录下的一个程序。（当你点击了网页上的按钮时，如果你观察你的浏览器窗口顶部的地址，有时你可以看见“cgi-bin”的字样混迹在一串冗长不知所云的字符中。）几乎所有的语言都可以用来编写这些程序，Perl 已经成为了最常见的选择，因为它被设计用来处理文本，并且是解释型语言，因此无论服务器的处理器和操作系统如何，它都可以被安装于其上。然而，Python（我的最爱，请查看 www.Python.org）以对其产生了重大的冲击，因为它更强大且更简单。

当今许多有影响力的网站都是完全构建于 CGI 之上的，实际上你几乎可以通过 CGI 做任何事。然而，构建于 CGI 程序之上的网站可能会迅速变得过于复杂而难以维护，并同时产生响应时间过长的問題。CGI 程序的响应时间依赖于必须发送的数据量的大小，以及服务器和 Internet 的负载，此外，CGI 程序的初始化也相当慢。Web 的最初设计者们并没有预见到网络带宽（bandwidth）被人们开发的各种应用迅速耗尽。例如，任何形式的动态图形处理几

乎都不可能连贯地执行，因为图形交互格式（graphic interchange format, GIF）的文件对每一种版本的图形都必须在服务器端创建，并发送给客户端。再比如，你肯定处理过像验证输入表单那样简单的事情：你按下网页上的提交（Submit）按钮；数据被封装发送回服务器；服务器启动一个 CGI 程序来检查发现错误，并将错误组装为一个用来通知你的 HTML 页面，然后将这个页面发回给你；你之后必须回退一个页面，然后重新再试。这个过程不仅很慢，而且不太优雅。

问题的解决方法就是客户端编程（client-side programming）。大多数运行 web 浏览器的机器都是能够执行大型任务的强有力的引擎。在使用原始的静态 HTML 方式的情况下，它们只是空闲地愣在那里，等着服务器送来下一个页面。客户端编程意味着 Web 浏览器被用来执行任何它可以完成的工作，使得返回给用户的结果更加迅捷，而且使得你的网站更加具有交互性。

客户端编程的问题是：它与通常意义上的编程十分不同，参数几乎相同，而平台却不同。Web 浏览器就象一个功能受限的操作系统。当然，你仍然的编写程序，而且还得处理那些令人头晕眼花的成堆的问题，并以客户端编程的方式来产生解决方案。本节剩下的部分将带你纵览有关客户端编程的话题和方式。

插件（Plug-in）

客户端编程所迈出的最重要的一步就是插件（plug-in）的开发。通过这种方式，程序员可以下载一段代码，并将其插入到浏览器中适当的位置，以此来为浏览器添加新功能。它告诉浏览器：从现在开始，你可以执行这个新行为了（你只需要下载一次插件即可）。某些更快更强大的行为都是通过插件添加到服务器中的，但是编写插件并不是件轻松的事，也不像你希望的那样成为构建某特定网站的过程中而作的事情。插件对于客户端编程的价值在于：它允许专家级的程序员不需经过浏览器生产厂商的许可，就可以开发某种新语言，并将其添加到服务器中。因此，插件提供了一个“后门（back door）”，使得可以创建新的客户端编程语言（但是并不是所有的客户端编程语言都是以插件的形式实现的）。

脚本语言（scripting language）

插件引发了脚本语言（scripting language）的大爆炸。通过使用某种脚本语言，你可以将客户端程序的源代码直接嵌入到 HTML 页面中，解释这种语言的插件在 HTML 页面被显示时自动激活。脚本语言先天就相当易于理解，因为它们只是作为 HTML 页面一部分的简单文本，当服务器收到要获取该个页面的请求时，它们可以被快速加载。此方法的缺点是你的代码会被暴露给任何人去浏览（或窃取）。但是，通常你不会使用脚本语言去做相当复杂的事情，所以这个缺点并不太严重。

这也点出了在 Web 浏览器内部使用的脚本语言实际上总是被用来解决特定类型的问题，主要是用来创建更丰富、更具有交互性的图形化用户界面（GUI, graphic user interface）。但是，脚本语言确实可以解决客户端编程中所遇到的百分之八十的问题。你的问题可能正好落在这

百分之八十的范围之内，由于脚本语言提供了更容易、更快捷的开发方式，因此你应该在考虑诸如 Java 或 ActiveX 之类的更复杂的解决方案之前，先考虑脚本语言。

最常被讨论的浏览器脚本语言包括：JavaScript（它与 Java 并没有任何关系，它之所以这样被命名只是因为想赶上 Java 的市场浪潮），VBScript（它看起来很像 Visual BASIC），和 Tcl/Tk（流行的 GUI 构建语言）。还有一些脚本语言没有列在这里，当然还有更多的脚本语言还处于开发阶段。

JavaScript 可能是最被广泛支持的一种脚本语言。网景（Netscape）的 Navigator 和微软（Microsoft）的 Internet Explorer（IE）都提供对它的内置支持。遗憾的是，JavaScript 在这两种浏览器中的风格有很大的不同（可以从 www.Mozilla.com 上自由下载的 Mozilla 浏览器支持有可能在某天被普遍支持的 ECMAScript）。此外，从市场上可以得到的有关 JavaScript 的书可能比有关其它浏览器语言的书籍要更多，而且某些工具能够用 JavaScript 自动地生成页面。但是，如果你已经熟练掌握了 Visual BASIC 或 Tcl/Tk，那么使用这些脚本语言比学习全新一种脚本语言要更有生产效率。因为你已经可以投入全部精力去解决有关 Web 的相关问题。

Java

如果脚本语言可以解决客户端编程百分之八十的问题的话，那么剩下那百分之二十（那才是真正难啃的硬骨头）又该怎么办呢？Java 是处理它们最流行的解决方案。Java 不仅是一种功能强大的、被构建为安全的、跨平台的、国际化的编程语言，而且它还在不断地被扩展，以提供更多的语言功能，以及更多的能够优雅地处理在传统编程语言中很难解决的问题的类库，例如多线程（multithreading）、数据库访问（database access）、网络编程（network programming）和分布式计算（distributed computing）。Java 是通过 applet 以及使用 Java Web Start 来进行客户端编程的。

Applet 是只在 Web 浏览器中运行的小程序。Applet 是作为网页的一部分被自动下载的（就象网页中的图片被自动下载一样）。当 applet 被激活时，它便开始执行程序。这正是它优雅之处：它提供了一种一旦用户需要客户端软件时，就可以自动地从服务器分发客户端软件给用户的方法。当用户获取了最新版本的客户端软件时，并不会产生错误，而且也不需要很麻烦的重新安装过程。因为 Java 的这种设计方式，使得程序员只需创建单一的程序，而只要一台计算机有浏览器，且浏览器具有内置的 Java 解释器（大多数的机器都如此），那么这个程序就可以在这台计算机上运行。由于 Java 是一种成熟的编程语言，所以在创建了到服务器的请求之前和之后，你可以在客户端尽可能多地做些事情。例如，你不必跨网络地发送一张请求表单去检查你是否填写了错误的日期或其它参数，你的客户端计算机就可以快速地标出错误数据，而不用等待服务器作出标记并传回一张图片给你。你不仅立即就获得了高速度和快速的响应能力，而且也降低了网络流量和服务器负载，从而不会使整个 Internet 的速度都慢了下来。

Java applet 胜过脚本语言程序的优势之一就是它是以被编译过的形式存在的，因此其源代码对客户端来说是不可见的。另一方面，虽然反编译 Java applet 并不需要花费多少力气，但是隐藏你的代码通常并不是一个重要的话题。有另外两个因素是很重要。就象你在本书稍后的

部分会看到的那样，如果编译过的 applet 很大的话，那么就需要额外的时间去下载它。脚本语言程序只是被作为 Web 页面的一部分文本而集成到了 Web 页面中（通常比较小，并减少了对服务器的访问）。这对 Web 网站的响应能力来说很重要。另一个因素是非常重要的“学习曲线（learning curve）”。如果你是一个 Visual BASIC 程序员，那么转而学习 VBScript 可能是最快的解决方案（假设你可以限制你的客户只能用 windows 平台），而且由于它或许能够解决大多数典型的客户/服务器问题，所以你可能很那对学习 Java 报以公正的态度。如果你已经对脚本语言很有经验了，那么你在考虑付诸于 Java 之前，应该先看看 JavaScript 或 VBScript 是否满足你的要求，这样做对你会很有好处的，因为它们也许能够更方便地满足你的要求，而且使你更具生产力。

.NET 和 C#

曾几何时，Java applet 的主要竞争对手是微软的 ActiveX，尽管它要求客户端必须运行 Windows 平台。从那以后，微软以 .NET 平台和 C# 编程语言的形式推出了与 Java 全面竞争的对手。.NET 平台大致相当于 Java 虚拟机（virtual machine）和 Java 类库（library），而 C# 毫无疑问具有与 Java 类似之处。这当然是微软在编程语言与编程环境这块竞技场上所做出的最出色的成果。当然，他们有相当大的有利条件可以利用：他们可以看得到 Java 在什么方面做得好，在什么方面做得还不够好，然后基于此去构建，并要具备 Java 不具备的优点。这是自从 Java 出现以来，它所碰到的真正的竞争，如果事情真如微软所想，那么其结果只能是 Sun 的 Java 设计者们认真仔细地去研究 C#，去发现为什么程序员们可能会转而使用它，然后通过对 Java 做出根本的改进而对微软做出回应。

目前，.NET 主要受攻击的地方和人们所关心的最重要的问题就是微软是否会允许将它完全地移植到其它平台上。他们宣称这么做没有问题，而且 Mono 项目（www.go-mono.com）已经有了一个在 Linux 上运行的 .NET 的部分实现，但是，只要此实现完成，并且微软不会排斥其中的任何部分之日还为来临，.NET 作为一种跨平台的解决方案都仍旧是异常高风险的赌博。

要想学习更多的有关 .NET 和 C# 的知识，请阅读 Larry O'Brien 和 Bruce Eckel 撰写的《C# 编程思想（Think in C#）》一书（Prentice Hall，2003）。

安全性（Security）

通过 Internet 自动下载并运行程序听起来就像是病毒制作者的梦想。当你在某个网站上点击之后，可能会随 HTML 页面自动下载任意数量的东西：GIF 文件、脚本代码、编译过得 Java 代码和 ActiveX 控件。这些东西有些是良性的，例如 GIF 就是无害的，脚本语言能作的事情也很有限。在 Java 的设计中，applet 也只能运行在受安全保护的“沙盒（sandbox）”中，沙盒使 applet 无法写磁盘或是访问沙盒之外的内存。

微软的 ActiveX 正好位于走了另一个极端。使用 ActiveX 编程就像对 windows 编程——你可以随心所欲做任何事。因此如果你点击某页面下载了一个 ActiveX 控件，那么此控件就可能

会破坏你的磁盘上的文件系统。当然，你在计算机上加载的程序如果不被限制运行在 Web 浏览器内部，就也有可能造成这样的破坏。下载自 BBS 的病毒长久以来一直是一个严重的问题，Internet 迅猛的发展速度更是加剧了这个问题。

“数字签名 (digital signature)” 看起来像是一种解决方案，凭此签名可以检验代码以确定代码的作者。这种做法基于这样的思想：病毒能够发作是因为它的创建者可以匿名，因此如果消除匿名行为，那么就可以强制每个人都要为自己的行为负责。这看起来是一个不错的计划，因为它使得程序更加实用，但是我怀疑它是否真的能根除恶意的祸害。而且，如果程序中存在非故意的破坏性 Bug，那么它仍然会引发问题。

Java 的方法是通过沙盒来防止这类问题的发生。存在于本地浏览器中的 Java 解释器在加载 applet 的时候检查其是否含有不恰当的指令，特别是，applet 不能往磁盘上写文件或从磁盘上删除文件（这正是病毒赖以生存与发作的基础）。Applet 通常被认为是安全的，而这正是可靠的客户/服务器系统的要点所在，所以在 Java 语言中的任何可能会滋生病毒 bug 都会被快速修复。（值得注意的是，浏览器软件事实上都会强制执行这些安全限制，某些浏览器甚至允许你选择不同的安全级别，以提供不同级别的系统访问能力。）

你可能会质疑，不能向本地磁盘写文件的限制是否过于严苛。例如，你可能想构建一个本地数据库或存储数据以备在稍后下线之后仍能使用。尽管人们最初的梦想是最终要让所有人都可以在线进行所有重要的工作，但是很快这个梦想就被证明是不切实际的（尽管有朝一日，低成本的网络设备可以满足绝大多数用户的需要）。“签名 applet” 是问题的解决方案之一，它们使用公钥加密 (public-key encryption) 来验证 applet 是否确实来自于它所宣称的来源。虽然签名 applet 仍然可能会毁掉你的磁盘，但是既然你现在已经让 applet 的创建者负责任，那么他们就不会有恶意的行为。Java 提供了一个数字签名 (digital signature) 的框架，因此，如果需要的话，你最终是可以让 applet 步出沙盒之外的。第 14 章包含了一个怎样对 applet 签名的例子。

此外，Java Web Start 也是一种相对比较新的，用来便捷地部署不需要在 Web 浏览器中就可以独立运行的程序的方法。这项技术在解决许多与“在浏览器内部运行程序”相关的客户端问题方面很有潜力。Web Start 程序可以是签过名的，或者是在每一次执行有可能危及本地系统的操作时，向客户端索要相应的权限。第 14 章有一个简单的例子以及有关 Java Web Start 的解释。

数字签名遗漏了一个重要的问题，那就是人们在 Internet 上游荡的速度。如果你下载了一个有问题的程序，并且这个程序执行了某些不恰当的动作，那么需要多久你才能发现它造成的损害呢？也许是数天，甚至是数周。到那时，你怎样才能追踪到这个造成损害的程序呢？你又怎样才能知道在那时它干了哪些好事呢？

Internet 与 Intranet

Web 是最常用的解决客户/服务器问题的方案，因此，即便是解决这个问题一个子集，特别是在一个公司内部典型的客户/服务器问题，也一样可以使用这项技术。如果采用传统

的客户/服务器方式，你可能会遇到客户端计算机有多种型号的问题，也可能会遇到安装新的客户端软件的麻烦，而它们都可以很方便地通过 **Web** 浏览器和客户端编程得以解决。当 **Web** 技术仅限用于特定公司的信息网络时，它就被称为 **Intranet**（企业内部网）。**Intranet** 比 **Internet** 提供了更高的安全性，因为你可以从物理上控制对公司内部服务器的访问。从培训的角度看，似乎一旦人们理解了浏览器的基本概念后，对他们来说，处理网页和 **applet** 的外观差异就会容易得多，因此对新型系统的学习曲线也就减缓了。

安全问题把我们带到了一个客户端编程世界自动形成的领域。如果你的程序运行在 **Internet** 之上，那么你就不可能知道它将运行在什么样的平台之上，因此，你要格外地小心，不要传播由 **Bug** 的代码。你需要跨平台的、安全的语言，就像脚本语言和 **Java**。

如果你的程序运行与 **Intranet** 上，那么你可能会受到不同的限制。企业内所有的机器都采用 **Intel/Windows** 平台并不是什么稀奇的事。在 **Intranet** 上，你可以对你自己的代码质量负责，并且在发现 **Bug** 之后可以修复它们，此外，你可能已经有了以前使用更传统的客户/服务器方式编写的遗产代码，因此，你必须在每一次作升级时都要在物理上重装客户端程序。在安装升级程序时所浪费的时间是迁移到浏览器方式上的最主要的原因，因为在浏览器方式下，升级是透明的、自动的（**Java Web Start** 也是解决此问题的方式之一）。如果你身处这样的 **Intranet** 之中，那么最有意义的方式就是选择一条能够使用现有代码库最短的捷径，而不是用一种新语言重新编写你的代码。

当你面对各种令人眼花缭乱的解决客户端编程问题的方案时，最好的方法就是进行性价比分析。认真考虑你的问题的各种限制，然后思考那种解决方案可以成为最短的捷径。既然客户端编程仍然需要编程，那么针对你的特殊应用选取最快的开发方式，总是最好的做法。为那些在程序开发中不可避免的问题提早作准备是一种积极的态度。

服务器端编程

前面的讨论没有涉及服务器端编程的话题。当你产生了到服务器的请求后，会发生什么呢？大部分时候，请求只是要求“给我发送一个文件”，之后浏览器会以某种适当的形式解释这个文件，例如将其作为 **HTML** 页面、图片、**Java applet** 或脚本程序等来解释。更复杂的到服务器的请求通常涉及数据库事务。常见的情形是复杂的数据库查询请求，然后服务器将结果组装成为一个 **HTML** 页面发回给客户端。（当然，如果客户端通过 **Java** 或脚本程序具备了更多的智能，那么服务器可以将原始的数据发回，然后在客户端组装，这样会更快，而且服务器的负载将更小。）另一种常见情形是，当你要加入一个团体或下订单时，可能想在数据库中注册你的名字，这将涉及对数据库的修改。这些数据库请求必须通过服务器端的某些代码来处理，这就是所谓的服务器端编程。过去，服务器端编程都是通过使用 **Perl**、**Python**、**C++** 或其它某种语言来编写 **CGI** 程序而实现的，这使得更加复杂的系统出现了。其中包括基于 **Java** 的 **Web** 服务器，它让你用 **Java** 编写被称为 **Servlet** 的程序来实现服务器端编程。**Servlet** 及其衍生物 **JSP**，是许多开发网站的公司迁移到 **Java** 上的两个主要的原因，尤其是因为它们消除了处理具有不同能力的浏览器时所遇到的问题（这些话题在《企业 **Java** 编程思想（Thinking in Enterprise Java）》一书中有论述）。

应用

Java 引起人们的注意很大程度上始于 applet。Java 确实是一种通用的编程语言，至少在理论上可以解决各种用其它语言能够解决的问题。正像前面指出的那样，可能存在其它更有效的方式去解决客户/服务器问题。当你离开 applet 的竞技场时（同时也从其限制中解脱了，例如不能写磁盘的限制），你便步入了通用应用系统的世界，这里的系统都是独立运行的，不需要浏览器，就像其它普通程序一样。在这里，Java 的威力不仅在于它的可移植性（portability），还包括它的可编程性（programmability）。就在你阅读此书的同时，Java 已经具备了许多功能，让你创建健壮的程序，而花费的时间比使用 Java 之前的任何编程语言都更少。

不过你要意识到这是好坏掺半的事。你为 Java 带来的改进所付出的代价是降低了执行速度（尽管在此领域人们正在努力，例如，被称为“hotspot”的性能改善技术已经添加到新版本的 Java 中了）。像任何语言一样，Java 也具有使其不适合解决某类问题的天生的限制。然而，Java 是快速进化的语言，而且每当发布一个新版本时，它都会因为能够解决更多的问题而变得越来越具有吸引力。

Java 为什么成功

Java 能够取得如此的成功是因为它在设计时的目标就定位在要解决当今程序员们所面临的众多问题。Java 的基本目标之一就是要提高生产率。生产率来源于许多方面，但是 Java 希望在语言方面相对于它前辈有明显的提高，从而为程序员提供更大的便利。

系统易于表达、易于理解

被设计用来适应问题的“类”，在表达问题的能力上显得更强。这意味着当你编写代码时，你是在用问题空间的术语（“将垫圈放进盒子中”）而不是计算机，也就是解空间的术语（“设置芯片上的一位，表示继电器将被关闭”）来描述解决方案。你可以用更高层的概念来处理问题，并且只用一行代码就可以做更多的事。

易于表达所带来的另一个好处就是易于维护，维护（如果报告可信的话）在程序生命周期中所占的成本比例极大。如果程序易于理解，那么它必定易于维护。同时，这也降低了创建和维护文档的费用。

通过类库得到最大的支持

创建程序最快捷的方式就是使用已经编写好的代码：类库。Java 的主要目标之一就是要使类库更易于使用。这是通过将类库转型为新的数据类型（类）来实现的，因此，引入类库意味

着在语言中添加了新的数据类型。因为 **Java** 编译器会留意类库的使用方式——确保正确的初始化和垃圾回收，并保证其方法被正确调用，因此，你只需专注于你想让类库做些什么，而不必关心应如何去做。

错误处理

在 **C** 语言中，错误处理一直是一个声名狼藉的问题，而且经常被忽视——经常只能靠上帝保佑了。如果你在构建一个大型的、复杂的程序，那么没有什么比在程序某处暗藏了一个错误，而你却没有任何能够暗示它在何处的线索更糟糕的事情了。**Java** 异常处理（**exception handling**）便是一种能够确保错误必须报告，而且必须有所动作作为其响应的机制。

大型程序设计

许多传统语言在程序大小和复杂度方面都有内置的限制。例如，**BASIC** 可能对某类问题地解决能力非常强，可以快速创建解决方案，但是如果程序长度超过数页，或者超出该语言正常的题域之外，它就会像“在非常粘稠的液体中游泳”。没有明确的界线来表示何时你所使用的语言会导致最终的失败，即使有这样的界线，你也会忽视它。你总不能说：“我的 **BASIC** 程序太大了，我不得不用 **C** 来重写它！”相反，你会试着将几行代码硬塞进去，以便在程序中增加新功能。因此，不知不觉中，你就付出了额外的开销。

Java 具备编写大型程序的能力——也就是说，它消除了小型程序和大型程序之间的复杂度界线。在编写“**hello, world**”风格的小应用程序时，你当然不必使用 **OOP**，但是当你需要用到时，这些功能随手可得。而且，对小型程序和大型程序，编译器都会一视同仁地、积极地找出因 **Bug** 而产生的错误。

Java 与 C++

Java 看起来很像 **C++**，因此，很自然地，看起来 **C++** 将会被 **Java** 取代。但是我开始怀疑这种逻辑了。**C++** 仍然有某些功能是 **Java** 不具备的，尽管关于 **Java** 终有一日会与 **C++** 一样快，甚至更快的承诺层出不穷，我们也看到了 **Java** 在稳步地提高，但是至今并没有什么令人瞩目的突破。而且，人们对 **C++** 仍持续地保持着兴趣，因此我不认为 **C++** 会在近期内消亡。所有的编程语言看起来都不会永远地消亡。

于是我开始考虑：**Java** 的能力适用的战场与 **C++** 的稍有不同，**C++** 并不会去尝试为某类问题量身订造。当然，它也采用了大量的方法来解决各种特定问题。某些 **C++** 工具结合了类库、构件模型和代码生成工具，以解决开发视窗型终端用户应用（微软 **Windows** 应用）过程中的问题。然而，绝大多数 **Windows** 应用的开发者使用的是什么呢？是微软的 **Visual BASIC (VB)**，尽管 **VB** 所产生的代码在程序仅仅只有几页长的情况下就已经变得难以管理了（而且其语法也肯定让人迷惑不解）。虽然 **VB** 如此成功、如此流行，它却不是一个很好的语言设计范例。

如果能够在拥有VB的简易性及其强大威力的同时，而又不会产生难以管理的代码，那该有多好啊。这正是我认为Java终会光芒四散的原因所在：它会是下一个VB⁸。你可能会，也可能不会因听到这种说法而感到害怕，但是想想看，Java作了那么多的事情，都是为了使程序员能够更容易地解决诸如网络、跨平台UI之类的应用级问题，可是它仍然具备了编程语言的设计特征，它允许创建非常大型且极具灵活性的代码。此外，Java的类型检查和错误处理相对于绝大多数其他语言来说，都有很大的提高，这使得你可以在编程生产力方面产生显著的阶越。

如果你基本上是从头开发所有的代码，那么在简单性方面要胜过C++的Java可以显著地缩短你的开发时间。有传言（我是从一些原先使用C++，后来转投Java阵营的开发团队那里听来的）称，用Java的开发速度超过C++的两倍。如果Java的性能对你来说不是问题，或者你可以以某种方式加以弥补，那么纯粹考虑“时间-市场”因素，你是很难不选择Java而选择C++的。

最大的问题还是性能。在原始的Java解释器中，解释型的Java运行速度曾经非常慢，甚至比C要慢20至50倍。随着时间的推移，这一点已经有了很大的改进（特别是最近的Java版本），但是仍然有很大的差距。论及计算机，无非就是速度。如果你在计算机上做事情的速度并没有快很多，也许你就会宁愿手工完成它。（我曾听过有人建议，如果你需要较快的执行速度，那么你可以先使用Java开发以获取较短的开发时间，然后再使用某种工具和支持类库将你编写的代码翻译成为C++代码。）

使Java适用于许多开发项目的关键，就是出现了能够提升速度的技术，例如所谓“即时（just-in-time, JIT）”编译器、Sun自己的“hotspot”技术，以及“本地代码编译器（native code compiler）”。当然，所有的本地代码编译器抹煞了编译过的代码可以跨平台执行这一非常吸引用户的特性，但是它们同时也带来了接近C和C++的执行速度。而且，跨平台编译Java程序比起C和C++来要容易得多。（理论上讲，你只须重新编译，但是其他语言也都这么承诺过。）

总结

本章试图让你体验一下面向对象程序设计和Java中各种宽泛的话题，包括为什么面向对象程序设计与众不同，以及为什么Java格外与众不同。

OOP和Java也许并不适合所有的人。重要的是正确评估你自己的需求，并决定Java是否能够最好地满足这些需求，或者你使用其它编程系统（包括你当前正在使用的）是更好的选择。如果你知道你的需求在可预见的未来会变得非常特殊化，并且Java可能不能满足你的具体限制，那么你就应该去考察其它的选择（我特别推荐你看看Python，www.Python.org）。即使最终你选择了Java作为你的编程语言，你至少要理解还有哪些选项可供选择，并且对为什么选择这个方向要有清楚的认识。

⁸ 微软总是在强调它不象C#和.NET“那么快”，许多人都产生过这样的问题：VB的程序员是否愿意转而使用其它语言，是否会使用Java, C#, 或者是VB.NET

你知道过程型语言看起来像什么样子：数据定义和函数调用。想了解此类程序的含义，你得忙上一阵，需要通读函数调用和低层概念，以在你那脑海里建立一个模型。这正是我们在设计过程式程序时，需要中介表示方式的原因。这些程序总是容易把人搞糊涂，因为它们使用的表示术语更加面向计算机而不是你要解决的问题。

因为 Java 在你能够在过程型语言中找到的概念的基础上，又添加了许多新概念，所以你可能会很自然地假设：Java 程序中的 `main()` 方法比 C 程序中等价的方法要复杂得多。但是，你会感到很惊喜：编写良好的 Java 程序通常比 C 程序要简单的多，而且也易于理解得多。你看到的只是有关下面两部分内容的定义：用来表示问题空间概念的对象（而不是有关计算机表示方式的相关内容），以及发送给这些对象的用来表示在此空间内的行为的消息。面向对象程序设计来给人们的喜悦之一就是：对于设计良好的程序，通过阅读它就可以很容易地理解其代码。通常，其代码也会少很多，因为许多问题都可以通过重用现有的类库代码而得到解决。

第二章 一切都是对象

尽管 Java 是基于 C++ 的，但是相比之下，Java 是一种更“纯粹”的面向对象程序设计语言。

C++ 和 Java 都是杂合型语言 (hybrid language)。但是，Java 的设计者认为这种杂合性并不像在 C++ 中那么重要。杂合型语言允许多种编程风格；C++ 之所以成为一种杂合型语言主要是因为它支持与 C 语言的向后兼容。因为 C++ 是 C 的一个超集，所以势必包括许多 C 语言不具备的特性，这些特性使 C++ 在某些方面显得过于复杂。

Java 语言假设我们只进行面向对象的程序设计。也就是说，在你开始用它设计之前，必须将你的思想转换到面向对象的世界中来（除非你的思维方式早已转换过来了）。这个入门基本功，可以使你具备使用编程语言编程的能力，这种能力使你在学习和使用其他 OOP 语言时，更加容易。在本章，我们将看到 Java 程序的基本组成，并体会到在 Java 程序中的一切，甚至 Java 程序本身都是对象。

用引用 (reference) 操纵对象

每种编程语言都有自己的数据操纵方式。有时候，程序员必须注意将要处理的数据是什么类型。你是直接操纵对象，还是用某种基于特殊语法的间接表示（例如 C 和 C++ 里的指针）来操纵对象？

所有这一切在 Java 里都得到了简化。一切都被视为对象，因此可采用单一固定的语法。尽管一切都“看作”对象，但操纵的标识符实际上是对象的一个“引用” (reference)¹。你可以将这一情形想象成用遥控器（引用）来操纵电视机（对象）。你只要握住这个遥控器，就能保持与电视机的连接。当有人想改变频道或者减小音量时，你实际操控的是遥控器（引用），再由遥控器来调控电视机（对象）。如果你想在房间里四处走走，同时仍能调控电视机；那么你只需携带遥控器（引用）而不是电视机（对象）。

此外，即使没有电视机，遥控器亦可独立存在。也就是说，你拥有一个引用，并不一定需要有一个对象与它关联。因此，如果你想操纵一个词或句子，你可以创建一个 String 引用：

```
String s;
```

¹ 这可能会引起争论。有人认为“很明显，它是一个指针”。但是这种说法是基于底层实现的某种假设。并且，Java 中的引用，在语法上更接近 C++ 的引用而不是指针。本书的第一版本中，我选择发明了一个用于此的新术语“句柄 (handle)”，因为，Java 的引用和 C++ 的引用毕竟存在一些重大差异。我当时正在脱离 C++ 阵营，而且也不想使那些已经习惯 C++ 语言的程序员（我想他们将来会是最大的热衷于 Java 的群体）感到迷惑。第二版本中，我决定换回这个最广泛使用的术语——“引用”。并且，那些从 C++ 阵营转换过来的人们，理应更会处理引用，而不是仅仅理解“引用”这个术语，因而，他们也会全心全意投入其中的。尽管如此，还是有人不同意用“引用”这个术语。我曾经读到的一本书这样说：“Java 所支持的‘按址传递’是完全错误的”，因为 Java 对象标识符（按那位作者所说）实际上是“对象引用。”并且他接着说任何事物都是“按值传递”的。也许有人会赞成这种精确却让人费解的解释，但我认为我的这种方法可以简化概念上的理解并且不会伤害到任何事物（啊，那些语言专家可能会说我在撒谎，但我认为我只是提供了一个合适的抽象罢了。）

但这里所创建的只是引用，并不是对象。如果此时向 `s` 发送一个消息，就会返回一个运行时错误。这是因为此时 `s` 实际上没有与任何事物相关联（即没有电视机）。因此，一种安全的做法是：创建一个引用的同时便进行初始化。

```
String s = "asdf";
```

但这里用到了 Java 语言的一个特性：字符串可以用带引号的文本初始化。通常，你必须对对象使用一种更通用的初始化方法。

必须由你创建所有对象

一旦创建了一个引用，就希望它能与一个新的对象相连接。我们通常用 `new` 关键字来实现这一目的。`New` 关键字的意思是“给我一个新对象。”所以前面的例子可以写成：

```
String s = new String("asdf");
```

它不仅表示“给我一个新的字符串”，而且通过提供一个初始字符串，给出了怎样产生这个 **String** 的信息。

当然，字符串类型并非唯一存在的类型，Java 提供了数量众多的现成类型。重要的是，你可以自行创建类型。事实上，这是 Java 程序设计中一项基本行为，你会在本书以后章节中慢慢学到。

存储到什么地方

程序运行时，对象是怎么进行放置安排的呢？特别是内存是怎样分配的呢？对这些方面的了解会对你有很大的帮助。有六个不同的地方可以存储数据：

1. 寄存器（**register**）。这是最快的存储区，因为它位于不同于其他存储区的地方——处理器内部。但是寄存器的数量极其有限，所以寄存器由编译器根据需求进行分配。你不能直接控制，也不能在程序中感觉到寄存器存在的任何迹象。
2. 堆栈（**stack**）。位于通用 RAM（**random-access memory**，随机访问存储器）中，但通过它的“堆栈指针”可以从处理器那里获得直接支持。堆栈指针若向下移动，则分配新的内存；若向上移动，则释放那些内存。这是一种快速有效的分配存储方法，仅次于寄存器。创建程序时，Java 编译器必须知道存储在堆栈内所有数据的确切大小和生命周期，因为它必须生成相应的代码，以便上下移动堆栈指针。这一约束限制了程序的灵活性，所以虽然某些 Java 数据存储于堆栈中——特别是对对象引用，但是 Java 对象并不存储于其中。
3. 堆（**heap**）。一种通用性的内存池（也存在于 RAM 区），用于存放所有的 Java 对象。堆不同于堆栈的好处是：编译器不需要知道要从堆里分配多少存储区域，也不必知道存储的数据在堆里存活多长时间。因此，在堆里分配存储有很大的灵活性。当你需要

创建一个对象时，只需用 `new` 写一行简单的代码，当执行这行代码时，会自动在堆里进行存储分配。当然，为这种灵活性必须要付出相应的代价。用堆进行存储分配比用堆栈进行存储需要更多的时间（如果确实可以在 Java 中像在 C++ 中一样用栈保存对象）。

4. 静态存储（static storage）。这里的“静态”是指“在固定的位置”（尽管也在 RAM 里）。静态存储里存放程序运行时一直存在的数据。你可用关键字 `Static` 来标识一个对象的特定元素是静态的，但 Java 对象本身从来不会存放在静态存储空间里。
5. 常量存储（constant storage）。常量值通常直接存放在程序代码内部，这样做是安全的，因为它们永远不会被改变。有时，在嵌入式系统中，常量本身会和其它部分隔离开，所以在这种情况下，可以选择将其存放在 ROM（read-only memory，只读存储器）中。
6. 非 RAM 存储（non-RAM storage）。如果数据完全存活于程序之外，那么它可以不受程序的任何控制，在程序没有运行时也可以存在。其中两个基本的例子是“流对象（streamed object）”和“持久化对象（persistent object）”。在“流对象”中，对象转化成字节流，通常被发送给另一台机器。在“持久化对象”中，对象被存放于磁盘上，因此，即使程序终止，它们仍可以保持自己的状态。这种存储方式的技巧在于：把对象转化成可以存放在其它媒介上的事物，在需要时，可恢复成常规的、基于 RAM 的对象。Java 提供对轻量级持久化（*lightweight persistence*）的支持，未来的 Java 版本可能会为持久化提供更全面的解决方案。

特例：基本类型（primitive type）

有一系列类型经常在程序设计中用到，它们需要特殊对待。你可以把它们想象成“基本（primitive）”类型。之所以特殊对待，是因为 `new` 将对象存储在“堆”里，故用 `new` 创建一个对象——特别是小的、简单的变量，往往不是很有效。因此，对于这些类型，Java 采取与 C 和 C++ 相同的方法。也就是说，不用 `new` 来创建变量，而是创建一个并非是“引用”的“自动”变量。这个变量拥有它的“值”，并置于堆栈中，因此更加高效。

Java 要确定每种基本类型所占存储空间的大小。它们的大小并不像其它大多数语言那样随机器硬件架构的变化而变化。这种所占存储空间大小的不变性是 Java 程序具有可移植性的原因之一。

基本类型	大小	最小值	最大值	包装器类型
boolean	—	—	—	Boolean
char	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15}	$+2^{15}-1$	Short
int	32-bit	-2^{31}	$+2^{31}-1$	Integer
long	64-bit	-2^{63}	$+2^{63}-1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double

void	—	—	—	Void
-------------	---	---	---	-------------

所有数值类型都有正负号，所以不要去寻找无符号的数值类型。

布尔类型所占存储空间的大小没有明确指定，仅定义为能够获取代表 **true** 或 **false** 的值。

基本类型具有的包装器类，使你可以在堆中创建一个非基本对象，用来表示对应的基本类型。例如：

```
char c = 'x';
Character C = new Character(c);
```

也可以这样用：

```
Character C = new Character('x');
```

这样做的原因，在以后的章节中会说明。

高精度数字（high-precision number）

Java 提供了两个用于高精度计算的类：**BigInteger** 和 **BigDecimal**。虽然它们大体上属于“包装器类”的范畴，但二者都没有对应的基本类型。

不过，这两个类包含的方法，与基本类型所能执行的操作相似。也就是说，能作用于 **int** 或 **float** 的操作，也同样能作用于 **BigInteger** 或 **BigDecimal**。只不过必须以方法调用方式取代运算符方式来实现。由于这么做复杂了许多，所以运算速度会比较慢。在这里，我们以速度换取了精度。

BigInteger 支持任意精度的整数（integer）。也就是说，在运算中，你可以准确表示任何大小的整数值，而不会丢失任何信息。

BigDecimal 支持任何精度的定点数（fixed-point number），例如，你可以用它进行精确的货币计算。

关于调用这两个类的构造器和方法的详细信息，请查阅 **JDK** 文档。

Java 中的数组（Array）

几乎所有的程序设计语言都支持数组。在 C 和 C++ 中使用数组是很危险的，因为 C 和 C++ 中的数组就是内存块。如果一个程序要访问其自身内存块之外的数组，或在数组初始化前使用内存（程序中常见错误），都会产生难以预料的后果。

Java 的主要目标之一是安全性，所以许多在 C 和 C++ 里困扰程序员的问题在 Java 里不会再出现。Java 确保数组会被初始化，而且不能在它的范围之外被访问。这种范围检查，是以每个数组上少量的内存开销及运行时的索引校验为代价的。但由此换来的是安全性和效率的提高，因此付出的代价是值得的。

当你创建一个数组对象时，实际上就是创建了一个引用数组，并且每个引用都会自动被初始化为一个特定值，该值拥有自己的关键字 **null**。一旦 Java 看到 **null**，就知道这个引用还没有指向某个对象。在使用任何引用前，必须为其指定一个对象；如果你试图使用一个还是 **null** 的引用，在运行时将会报错。因此，常犯的数组错误在 Java 中就可以避免。

你还可以创建用来存放基本数据类型的数组。同样地，编译器也能确保这种数组的初始化，因为它会将这种数组所占的内存全部置零。

数组将在以后的章节中详细讨论。

永远不需要销毁对象

在大多数程序设计语言中，变量生命周期的概念，占据了程序设计工作中非常重要的部分。变量需要存活多长时间？如果想要销毁对象，那什么时刻进行呢？变量生命周期的混乱往往会导致大量的程序 bug，本节将介绍 Java 是怎样通过替我们完成所有的清除工作，来大大地简化这个问题的。

作用域（scoping）

大多数过程型语言都有作用域（scope）的概念。作用域决定了在其内定义的变量名的可见性和生命周期。在 C、C++ 和 Java 中，作用域由花括号的位置决定。例如：

```
{
    int x = 12;
    // Only x available
    {
        int q = 96;
        // Both x & q available
    }
    // Only x available
    // q "out of scope"
}
```

在作用域里定义的变量只可用于作用域结束之前。

任何位于 “//” 之后到行末的文本都是注释。

缩排格式使 Java 代码更易于阅读。由于 Java 是一种自由格式（free-form）的语言，所以，空格、制表符、换行都不会影响程序的执行结果。

注意，尽管在 C 和 C++ 中是合法的，但是在 Java 中却不能像下面这样做：

```
{
    int x = 12;
    {
        int x = 96; // Illegal
    }
}
```

编译器将会通告变量 X 已经定义过。所以，在 C 和 C++ 里将一个较大作用域的变量“隐藏”起来的做法，在 Java 里是不允许的。因为 Java 设计者认为这样做会导致程序混乱。

对象作用域（scope of object）

Java 对象不具备和基本类型一样的生命周期。当你用 new 创建一个 Java 对象时，它可以存活于作用域之外。所以假如你有下面的代码：

```
{
    String s = new String("a string");
} // End of scope
```

引用 s 在作用域终点就消失了。然而，s 指向的 String 对象仍继续占据内存空间。在这一小段代码中，我们似乎无法再访问这个对象，因为它唯一的引用已超出了作用域的范围。在后继章节中，你将会看到：在程序执行过程中，怎样传递和复制对象引用。

事实证明，由 new 创建的对象，只要你需要，就会一直保留下去。这样，许多 C++ 编程问题在 Java 中就完全消失了。在 C++ 中，最难的问题似乎在于：程序员并不能从语言本身中获得任何帮助，以确保在需要调用对象时，该对象仍可用。更重要的是：在 C++ 中，一旦使用完对象后，你必须确保要销毁对象。

这样便带来一个有趣的问题。如果 Java 让对象继续存在，那么靠什么才能防止这些对象填满内存空间，进而阻塞你的程序呢？这正是 C++ 里可能会发生的问题。则正是 Java 神奇之所在。Java 有一个“垃圾回收器”，用来监视用 new 创建的所有对象，并辨别那些不会再被引用的对象。随后，释放这些对象的内存空间，以便供其它新的对象使用。也就是说，你根本不必担心内存回收的问题。你只需要创建对象，一旦不再需要，它们就会自行消失。这样做就消除了这类编程问题：即所谓的“内存溢出”，即由于程序员忘记释放内存而产生的问题。

创建新的数据类型：类

如果一切都是对象，那么是什么决定了某一类对象的外观与行为呢？换句话说，是什么确定了对象的类型？你可能期望有一个名为“**type**”的关键字，当然它必须还要有相应的含义。然而，从历史发展角度来看，大多数面向对象的程序设计语言习惯用关键字“**Class**”，来表示“我准备告诉你一种新类型的对象看起来像什么样子。”**class** 这个关键字（以后会频繁使用，本书以后就不再用粗体字表示）之后紧跟着的是新类型的名称。例如：

```
class ATypeName { /* Class body goes here */ }
```

这就引入了一种新的类型，尽管类主体仅包含一条注释语句(星号和斜杠以及其中的内容就是注释，本章后面再讨论)。因此，你还不能用它做太多的事情。然而，你已经可以用 **new** 来创建这种类型的对象：

```
ATypeName a = new ATypeName();
```

但是，直到你定义了它的所有方法之前，你都还没有办法能够让它去做更多的事情（也就是说，不能向它发送任何有意义的消息）。

域（field）和方法（method）

一旦定义了一个类（在 **Java** 你所做的全部工作就是定义类，产生那些类的对象，以及发送消息给这些对象），就可以在类中设置两种类型的元素：域（**field**，有时被称作数据成员（**data member**））和方法（有时被称作成员函数（**member function**））。域可以是任何类型的对象，可以通过其引用与其进行通讯；也可以是基本类型（不是引用）中的一种。如果域是一个对象的引用，那么你必须用一种被称为构造器（**constructor**，在第 4 章详述）的特殊方法进行初始化，以便使其与一个实际的对象（如前所示，使用 **new** 来实现）相关联。但若是一种基本类型，则可在类中域被定义处进行初始化。（以后会看到：引用也可以在域定义处进行初始化。）

每个对象都有用来存储它的域的空间；域不能在对象间共享，下面是一个具有某些域的分类：

```
class DataOnly {  
    int i;  
    float f;  
    boolean b;  
}
```

尽管这个类什么也不能做，但是你可以创建它的一个对象：

```
DataOnly d = new DataOnly();
```

你可以给域赋值，但首先必须知道如何引用一个对象的成员。具体的实现为：在对象引用之后紧接着的一个句点，然后再接着是对象内部的成员名称：

```
objectReference.member
```

例如：

```
d.i = 47;
d.f = 1.1f; // ‘f’ after number indicates float constant
d.b = false;
```

你想修改的数据也有可能位于对象所包含的其它对象中。在这种情况下，只需要在使用连接句点即可。例如：

```
myPlane.leftTank.capacity = 100;
```

DataOnly 类除了保存数据外不能再做更多事情，因为它没有任何成员方法。如果你了解成员方法的运行机制，就得先了解“参数（argument）”和“返回值（return value）”的概念，稍后将对此作简略描述。

基本成员默认值

若类的某个成员是基本数据类型，即使没有进行初始化，Java 也会确保它获得一个默认值。如下表所示：

基本类型	默认值
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

千万要小心：当变量作为一个类的成员使用时，Java 才确保给定其默认值，以确保那些是基本类型的成员变量得到初始化（C++没有此功能），防止产生程序错误。但是，这些初始值对你的程序来说，可能是不正确的，甚至是不合法的。所以最好明确地对变量进行初始化。

然而上述确保初始化的方法并不适用于“**局部**”变量（即并非某个类的属性）。因此，如果在某个方法中有这样定义：

```
int x;
```

那么变量 `x` 得到的可能是任意值（与 `C` 和 `C++` 中一样），而不会被自动初始化为零。所以在使用 `x` 前，应先对其赋一个适当的值。如果你忘记了这么做，`Java` 会在编译时返回给你一个错误，告诉你此变量没有初始化，这正是 `Java` 优于 `C++` 的地方。（许多 `C++` 编译器会对未初始化变量给予警告，而 `Java` 则视为是错误）。

方法 (Method)、参数 (argument) 和返回值 (return value)

许多程序设计语言（像 `C` 和 `C++`）用“函数 (function)”这个术语来描述命名子程序。而在 `Java` 里却常用“方法 (method)”这个术语，来表示“做某些事情的方式”。实际上，继续把它看作是函数也无妨。尽管这只是用词上的差别，但本书将沿用 `Java` 的惯用法；用“方法”，而不是“函数”来描述。

`Java` 的方法决定了一个对象能够接收什么样的消息。通过本节学习，你将会了解到定义一个方法是多么的简单。

方法的基本组成部分包括：名字、参数、返回值和方法体。下面是它最基本的形式：

```
returnType methodName( /* Argument list */ ) {  
    /* Method body */  
}
```

返回类型是指调用方法后返回的数据类型。参数表列出了要传给方法的类型和名称信息。方法名和参数表的组合在一起唯一地标识某个方法。

`Java` 中的方法只能作为类的一部分来创建。方法只有通过对象才能被调用²，且这个对象必须能执行这个方法调用。如果试图在某个对象上调用并不具备的方法，那么在编译时，就会得到一条错误消息。通过对象调用方法时，需要先列出对象名，紧接着是句点，然后是方法名和参数表。即

```
objectName.methodName(arg1, arg2, arg3);
```

例如，假设有一个方法 `f()`，不带任何参数，返回类型是 `int`。如果有个名为 `a` 的对象，可以通过它调用 `f()`，那么你就可以这样写：

```
int x = a.f();
```

返回值的类型必须要与 `x` 的类型兼容。

² 稍后您将会学到 `static` 方法，它是针对类调用的，并不依赖于对象的存在。

这种调用方法的行为通常被称为“发送消息给对象”。在上面的例子中，消息是 `f()`，对象是 `a`。面向对象的程序设计通常简单地归纳为“向对象发送消息”。

参数列表（argument list）

方法的参数列表指定了要传递给方法什么样的信息。正如你所料想的那样，这些信息像Java中的其它信息一样，采用的都是对象形式。因此，在参数列表中必须指定每个传入对象的类型及名字。像Java中任何传递对象的场合一样，这里传递的实际上也是引用³，并且引用的类型必须正确。如果参数被设为**String**类型，则必须传递一个**String**对象；否则，编译器将抛出错误。

假设某个方法接受 **String** 为其参数，下面是其具体定义，它必须置于某个类的定义内才能被正确编译。

```
int storage(String s) {  
    return s.length() * 2;  
}
```

此方法告诉你，需要多少个字节才能容纳一个特定的 **String** 对象中的信息（字符串中的每个字符都是 16 位或 2 个字节，或长整型，以此来提供对 Unicode 字符集的支持）。此方法的参数类型是 **String**，参数名是 `s`。一旦将 `s` 传递给此方法，就可以把它当作其它对象一样进行处理（可以给它传递消息）。在这里，`s` 的 `length()` 方法被调用，它是 **String** 类提供的方法之一，会返回字符串包含的字符数。

通过上面例子，还可以了解到 **return** 关键字的用法，它包括两方面：首先，它代表“已经做完，离开此方法”。其次，如果此方法产生了一个返回值，这个值要放在 **return** 语句后面。在这种情况下，返回值是通过计算 `s.length() * 2` 这个表达式得到的。

你可以定义方法返回任意想要的类型，如果不想返回任何值，可以指示此方法返回 **void**(空)。下面是一些例子：

```
boolean flag() { return true; }  
float naturalLogBase() { return 2.718f; }  
void nothing() { return; }  
void nothing2() {}
```

若返回类型是 **void**，**return** 关键字的作用只是用来退出方法。因此，没有必要到方法结束时才离开，可在任何地方返回。但如果返回类型不是 **void**，那么无论在何处返回，编译器都会强制返回一个正确类型的返回值。

³ 对于前面所提到的特殊数据类型 **boolean**, **char**, **byte**, **short**, **int**, **long**, **float**, 和 **double**来说是一个例外。通常，尽管你传递的是对象，而实际上传递的是对象的引用。

到此为止，你或许已经觉得：程序似乎只是一系列带有方法的对象组合，这些方法以其它对象为参数，并发送消息给其它对象。大体上确实是这样，但在以后章节中，你将会学到怎样在一个方法内进行判断，做一些更细致的底层工作。至于本章，你只需要理解消息发送就足够了。

构建一个 Java 程序

在构建自己的第一个 Java 程序前，你还必须了解其它一些问题。

名字可视性（Name visibility）

名字管理对任何程序设计语言来说，都是一个重要问题。如果你在程序的某个模块里使用了一个名字，而其他人在这个程序的另一个模块里也使用了相同的名字，那么怎样才能区分这两个名字并防止二者互相冲突呢？这个问题在 C 语言中尤其严重，因为程序往往包含许多难以管理的名字。C++类(Java 类基于此)将函数包于其内，从而避免了与其它类中的函数名冲突。然而，C++仍允许全局数据和全局函数的存在，所以还是有可能发生冲突。为了解决这个问题，C++通过几个关键字，引入了“名字空间”的概念。

Java 采用了一种全新的方法，能够避免上述所有问题。为了给一个类库生成不会与其它名字混淆的名字，Java 采用了与 Internet 域名相似的指定符。实际上，Java 设计者希望程序员反过来使用自己的 Internet 域名，因为这样可以保证它们肯定是独一无二的。由于我的域名是 **BruceEckel.com**，所以我的各种奇奇怪怪的应用工具类库就被命名为 **com.bruceeckel.utility.foibles**。反转域名后，句点就用来代表子目录的划分。

在 Java1.0 和 Java1.1 中，扩展名 **com, edu, org, net** 等约定为大写形式。所以上面的库名应该写成：**COM.bruceeckel.utility.foibles**。然而，在 Java2 开发到一半时，设计者们发现这样做会引起一些问题，因此，现在整个包名都是小写了。

Java2 的这种机制意味着所有的文件都能够自动存活于它们自己的名字空间内，而且同一个文件内的每个类都有唯一的标识符。所以不必学习特殊的语言知识来解决这个问题——Java 语言本身已经为你解决了这个问题。

运用其它构件

如果你想在自己的程序里使用预先定义好的类，那么编译器就必须知道怎么定位它们。当然，这个类可能就在发出调用的那个源文件中；在这种情况下，你就可以直接使用这个类——即使这个类在文件的后面才会被定义（Java 消除了“向前引用”问题，故不必考虑它）。

如果那个类位于其它文件中，又会怎样呢？你可能会认为编译器应该有足够的智慧，能够直

接找到它的位置。但事实并非如此。想象下面的情况，如果你想使用一个特定名字类，但其定义却不只一份（假设这些定义各不相同）。更糟糕的是，假设你正在写一个程序，在构建过程中，你想将某个新类添加到类库中，但却与已有的某个类名冲突。

为了解决这个问题，你必须消除所有可能的混淆情况。为实现这个目的，你可以使用关键字 **import** 来准确地告诉编译器你想要的类是什么。**Import** 指示编译器导入一个包，也就是一个类库（在其它语言中，一个库不仅包含类，还可能包括方法和数据；但是 Java 中的所有的代码都必须写在类里）。

大多时候，我们使用与编译器附在一起的 Java 标准类库里的构件。有了这些构件，你就不必写一长串的反转域名。举例来说，只须像下面这么书写就行了：

```
import java.util.ArrayList;
```

这行代码告诉编译器，你想使用 Java 的 **ArrayList** 类。但是，**util** 包含了数量众多的类，有时你想使用其中的几个，同时又不想明确地逐一声明。那么你可以使用通配符“*”来很容易地实现这个目的：

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

用这种方法一次导入一群类的方式倒是比一个一个地导入类的方式更常用。

Static 关键字

通常来说，当你创建类时，就是在描述那个类的对象的外观与行为。除非你用 **new** 创建那个类的对象，否则，你实际上并未获得任何东西。执行 **new** 来创建对象时，数据存储空间才被分配，其方法才供外界调用。

但是有两种情形，用上述方法是无法解决的。一种情形是，你只想为某特定数据分配一份存储空间，而不去考虑究竟要创建多少对象，还是甚至根本就不创建任何对象。另一种情形是，你希望某个方法不与包含它的类的任何对象关联在一起。也就是说，即使没有创建对象，也能够调用这个方法。通过 **Static** 关键字，可以满足这两方面的需要。当你声明一个事物是 **Static** 时，就意味着这个数据或方法不会与包含它的那个类的任何对象实例关联在一起。所以，即使从未创建某个类的任何对象，也可以调用其 **Static** 方法或访问其 **Static** 数据。通常，你必须创建一个对象，并用它来访问数据或方法。因为非 **Static** 数据和方法必须知道它们一起运作的特定对象。由于在用 **Static** 方法前，不需要创建任何对象；所以对于 **Static** 方法，不能只是简单地通过调用其它方法，而没有指定某个对象，来直接访问非 **Static** 成员或方法（因为非 **Static** 成员或方法必须与某一特定对象关联）。

有些面向对象语言采用“类数据（class data）”和“类方法（class method）”两个术语，代表那些数据和方法只是为了整个类，而不是类的某个特定对象而存在的。有时，一些 Java 文献里也用到这两个术语。

只须将 **Static** 关键字放在定义之前，就可以将域或方法设为 **Static**。例如，下面的代码就生成了一个 **Static** 域，并对其进行了初始化：

```
class StaticTest {  
    static int i = 47;  
}
```

现在，即使你创建了两个 **StaticTest** 对象，**StaticTest.i** 也只是一份存储空间，这两个对象共享同一个 **i**。再看看下面代码：

```
StaticTest st1 = new StaticTest();  
StaticTest st2 = new StaticTest();
```

在这里，**st1.i** 和 **st2.i** 指向同一存储空间，因此它们具有相同的值 47。

有两种方法引用一个 **static** 变量。如前例所示，你可以通过一个对象去定位它，如 **st2.i**；也可以通过其类名直接引用，而这对于非静态成员则不行。（最好用这个方法引用 **static** 变量，因为它强调了变量的静态性。）

```
StaticTest.i++;
```

其中，++运算符对变量进行加 1 操作。此时，**st1.i** 和 **st2.i** 仍具有相同的值 48。

类似逻辑也被应用于静态方法。你既可以像其它方法一样，通过一个对象来引用某个静态方法，也可以通过特殊的语法形式 **ClassName.method()** 加以引用。定义静态方法的方式也与静态变量的定义方式相似。

```
class StaticFun {  
    static void incr() { StaticTest.i++; }  
}
```

可以看到，**StaticFun** 的 **incr()** 方法通过++运算符将静态数据 **i** 加 1。你可以用典型的方式，通过对象来调用 **incr()**：

```
StaticFun sf = new StaticFun();  
sf.incr();
```

或者，因为 **incr()** 是一个静态方法，所以你也可以直接通过它的类直接调用：

```
StaticFun.incr();
```

尽管当 **static** 作用于某个域时，肯定会改变数据创建的方式（因为一个 **static** 域对每个类来说都只有一份存储空间，而非 **static** 域则是对每个对象有一个存储空间），但是如果 **static** 作用于某个方法，差别却没有那么大。**Static** 方法的一个重要用法就是在不创建任何对象的

前提下，就可以调用它。正如我们将会看到的那样，这一点对定义 **main()** 方法时很重要。这个方法是运行一个应用时的入口点。

和其它任何方法一样，**static** 方法可以创建或使用与其类型相同的被命名对象，因此，**static** 方法常常拿来做“牧羊人”的角色，负责看护与其隶属同一类型的实例群。

你的第一个 Java 程序

最后，让我们编写一个真正的程序。此程序开始是打印一个字符串，然后是打印当前日期，这里用到了 Java 标准库里的 **Date** 类。

```
// HelloDate.java
import java.util.*;

public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
}
```

在每个程序文件的开头，必须声明 **import** 语句，以便引入在文件代码中需要用到所有额外类。注意，在这里说它们“额外”，是因为有一个特定类会自动被导入到每一个 Java 文件中：**java.lang**。打开你的 web 浏览器，查找 Sun 公司提供的文档（若没有从 java.sun.com 下载 JDK 文档，现在开始下载⁴）。在包列表里，可以看到 Java 配套提供的各种类库。请点击其中的 **java.lang**，就会显示出这个类库所包含的全部类的列表。由于 **java.lang** 是默认导入到每个 Java 文件中的，所以它的所有类都可以被直接使用。**java.lang** 里没有 **Date** 类，所以必须导入另外一个类库才能使用它。若不知某个特定类在哪个类库里，可在 Java 文档中选择“Tree”。便可以看到 Java 配套提供的每一个类。接下来，用浏览器的“查找”功能查找 **Date**。这样做，你就可以发现它以 **java.util.Date** 的形式被列出。于是我们知道它位于 **Util** 类库中，并且必须书写 **import java.util.*** 才能使用 **Date** 类。

现在返回文档最开头的部分，选择 **java.lang**，接着是 **System**，可以看到 **System** 类有许多属性；若选择 **out**，就会发现它是一个静态 **PrintStream** 对象。因为是静态的，所以不需要创建任何东西，**out** 对象便已经存在了，只须直接使用即可。但我们能够用 **out** 对象做些什么事情，是由它的类型 **PrintStream** 决定的。**PrintStream** 在描述文档中是以超链接形式显示，所以很方便进行查看，只须点击它，就可以看到能够为 **PrintStream** 调用的所有方法。方法的数量不少，本书后面再详加讨论。现在我们只对 **println()** 方法感兴趣，它的实际作用是“将我给你的数据打印到控制台，完成后换行”。因此，在任何 Java 程序中，一旦需要将某些数据打印到控制台，就可以这样写 **System.out.println("things")**。

⁴ Sun 提供的 Java 编译器和文档没有包含在本书的 CD 中，因为它会定期改变。读者可自行下载最新版本。

类的名字必须和文件名相同。如果你像现在这样创建一个独立运行的程序，那么文件中必须存在某个类与该文件同名（否则，编译器会报错），且那个类必须包含一个名为 **main()** 的方法，形式如下所示：

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

其中，**public** 关键字意指这是一个可由外部调用的方法（第 5 章将详细描述）。**main()** 方法的参数是一个字符串数组。在这个程序中并未用到 **args**，但是 **Java** 编译器会要求必须这样做，因为 **args** 要用作存储命令行参数。

打印日期的这行代码很是有趣的：

```
System.out.println(new Date());
```

在这里，传递的参数是一个 **Date** 对象，一旦创建它之后，就可以直接将它的值发送给 **println()**。当这条语句执行完毕后，**Date** 对象就不再被使用，因此垃圾回收器会发现这一情况，并在任意时候将其回收。因此，我们就没必要去关心怎样清除它了。

编译运行

为了编译运行这个程序以及本书中其它所有的程序，首先，你必须要有个 **Java** 开发环境。目前，有相当多的第三方厂商提供开发环境，但是在本书中，我们假设你使用的是 **Sun** 免费提供的 **JDK**（**Java Developer's Kit**）开发环境。若使用其它的开发系统⁵，请查找该系统的相应文档，以便决定怎样编译和运行程序。

请登录到 java.sun.com 这个网站，那里会有相关的信息和链接，引导你下载和安装与你的机器平台相兼容的 **JDK**。

安装好 **JDK** 后，还需要设定好路径信息，以确保计算机能找到 **javac** 和 **java** 这两个文件。然后请下载并解压本书提供的源代码（从 www.BruceEckel.com 处可以获得），它会为书中每一章自动创建一个子目录。请转到 **c02** 子目录下，并键入：

```
javac HelloDate.java
```

正常情况下，这行命令应该不会产生任何响应。如果有任何错误消息返回给你，都说明还没能正确安装好 **JDK**，需进行检查并找出问题所在。

如果没有返回任何回应消息，在命令提示符下键入：

```
java HelloDate
```

⁵ IBM 的“jikes”编译器也是一个常用的编译器，它比 **Sun** 的 **javac** 要明显的快。

接着，便可看到程序中的消息和当天日期被输出。

整个这个过程也是本书中每一个程序的编译和运行过程。然而，你还会看到在本书所附源代码中，每一章都有一名为**build.xml**的文件，该文件提供一个“ant”命令，来自动构建该章的所有文件。Build文件和 Ant（以及在哪里下载）将在第 15 章详细讨论，一旦安装好 Ant（可从<http://jakarta.apache.org/ant>下载），便可直接在命令行提示符下键入**ant**，来编译和运行每一章的程序了。

注释和嵌入式文档

Java 里有两种注释风格。一种是传统的 C 语言风格的注释，C++也继承了这种风格。此种注释以“/*”开始，随后是注释内容，并可跨越多行，最后以“*/”结束。注意，许多程序员在连续的注释内容的每一行都以一个“*”开头，所以经常看到像下面这样的写法：

```
/* This is a comment
 * that continues
 * across lines
 */
```

但请记住，进行编译时，/*和*/之间的所有东西都会被忽略，所以上述注释与下面这段注释并没有什么两样：

```
/* This is a comment that
continues across lines */
```

第二种风格的注释也源于 C++。这种注释是“单行注释”，以一个“//”起头，直到句末。这种风格的注释因为书写容易，所以更方便、更常用。你无需在键盘上寻找“/”，再寻找“*”（而只需按两次同样的键），而且不必考虑结束注释。下面是这类注释的例子：

```
// This is a one-line comment
```

注释文档

Java 语言一项更好的设计就是：程序代码的编写并不是唯一重要的事情——文档编写的重要性并不亚于程序代码本身。代码说明文档撰写的最大问题，大概就是对文档的维护了。如果文档与代码是分离的，那么在每次修变代码时，都需要修变相应的文档，这会成为一件相当麻烦的事情。解决的方法似乎很简单：将代码同文档“链接”起来。为达到这个目的，最简单的方法是将所有东西都放在同一个文件内。然而，为实现这一目的，还必须使用一种特殊的注释语法，来标记文档；此外也还需一个工具，用于提取这些注释，并将其转换成有用的形式。这正是 Java 所做的。

Javadoc 便是用于提取注释的工具，它是 JDK 安装的一部分。它采用了 Java 编译器的某些技术，查找程序内的特殊注释标签。它不仅解析由这些标签标记的信息，也将毗邻注释的类名或方法名解析出来。如此，我们就可以用最少的工作量，生成相当好的程序文档。

javadoc 输出的是一个 HTML 文件，你可以用自己的 Web 浏览器来查看。这样，该工具就使得我们只需创建和维护单一的源文件，并能自动生成有用的文档。有了 javadoc，我们就有了创建文档的标准；我们可以期望甚至要求所有的 Java 类库都提供相关的文档。

此外，如果你想对 javadoc 处理过的信息执行特殊的操作（例如：以不同格式输出），那么你可以通过编写你自己的，被称为“doclets”的 Javadoc 处理器来实现。关于 Doclets，将在第 15 章介绍。

下面仅对基本的 javadoc 进行简单介绍和概述。全面翔实的描述可从 java.sun.com 提供的可下载的 JDK 文档中找到。（注意此文档并没有与 JDK 一块打包，需单独下载）。解压缩该文档之后，查阅“Tooldocs”子目录（或点击“Tooldocs”链接）。

语法

所有 javadoc 命令都只能始于“/**”注释，和一般注释一样，结束于“*/”。使用 javadoc 的方式主要有两种：嵌入 HTML，或使用“文档标签（doc tag）”。所谓“文档标签”是一些以“@”字符开头的命令，且该“@”字符要置于注释行的最前面（即忽略前导“/**”之后的最前面）。而“行内文档标签（inline doc tag）”则可以出现在 javadoc 注释中的任何地方，它们也是以“@”字符开头，但要括在花括号内。

共有三种类型的注释文档，分别对应于注释位置后面的三种元素：类、变量和方法。也就是说，类注释正好位于类定义之前；变量注释正好位于变量定义之前；而方法定义也正好位于方法定义的前面。如下面这个简单的例子所示：

```
/** A class comment */
public class DocTest {
    /** A variable comment */
    public int i;
    /** A method comment */
    public void f() {}
}
```

注意，javadoc 只能为“public（公共）”和“protected（受保护）”成员进行文档注释。“private（私有）”和包内可访问成员（参阅第 5 章）的注释会被忽略掉，所以输出结果中看不到它们（不过可以用-private 进行标记，以便对 private 成员一并处理）。这样做是有道理的，因为只有 public 和 protected 成员才能在文件之外被使用，这是客户端程序员所期望的。至于所有对类所作的注释，则都会包含在输出结果中。

上述代码的输出结果是一个 HTML 文件，它与其他 Java 文档具有相同的标准格式。因此，用户会非常熟悉这种格式，从而方便地导航到用户自己设计的类。输入上述代码，然后通过 javadoc 处理产生 HTML 文件，最后通过浏览器观看生成的结果，这样做是非常值得的。

嵌入式 HTML

javadoc 将 HTML 命令嵌入到它所生成的 HTML 文档中。这使你能够充分利用 HTML 的功能。当然，其主要目的还是为了对代码进行格式化。下面列出一个例子：

```
/**
 * <pre>
 * System.out.println(new Date());
 * </pre>
 */
```

你也可以像在其他 Web 文档中那样运用 HTML，对普通文本按照你自己所描述的进行格式化。

```
/**
 * You can <em>even</em> insert a list:
 * <ol>
 * <li> Item one
 * <li> Item two
 * <li> Item three
 * </ol>
 */
```

注意，在文档注释中，位于每一行最开头的星号和前导空格都会被 javadoc 丢弃。Javadoc 会对所有内容重新格式化，使其与标准的文档外观一致。不要在嵌入式 HTML 中使用标题标签，例如：<h1>或<hr>，因为 javadoc 会插入自己的标题，而你的标题可能会对它造成干扰。

所有类型的注释文档——类、变量和方法——都支持嵌入式 HTML。

一些标签实例

这里将介绍一些可用于代码文档的 javadoc 标签。在使用 javadoc 处理重要事情之前，你应该先到可下载的 JDK 文档那里查阅 javadoc 参考，以获取全面的 javadoc 的使用方法。

@see: 引用其他类

@see 标签允许你引用其他类的文档。javadoc 会在其生成的 HTML 文件中，用 **@see** 标签链接到其他文档。格式如下：

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

每种格式都会在生成的文档中加入一个具有超链接的“See Also”条目。但是 javadoc 不会检查你所提供的超链接是否有效。

{@link package.class#member label}

该标签与 **@see** 极其相似，只是它可以用于行内，并且是用“label”作为超链接文本而不用“See Also”。

{@docRoot}

该标签产生到文档根目录的相对路径，用于文档树页面的显式超链接。

{@inheritDoc}

该标签从当前这个类的最直接的基类中继承相关文档到当前的文档注释中。

@version

该标签的格式如下：

```
@version version-information
```

其中，“version-information”可以是任何你认为适合作为版本说明的重要信息。如果 javadoc 命令行使用了“-version”标记，那么就可以从生成的 HTML 文档中提取出版本信息。

@author

该标签的格式如下：

```
@author author-information
```

其中，“author-information”，望文生义你也知道，应该是你的姓名，也可以包括电子邮件地址或者其他任何适宜的信息。如果 javadoc 命令行使用了“-author”标签，那么就可以从生成的 HTML 文档中提取作者信息。

你可以使用多个标签，以便列出所有作者，但是它们必须连续放置。全部作者信息会合并到同一段落，置于生成的 HTML 中。

@since

该标签允许你指定程序代码最早使用的版本，你将会在 HTML Java 文档中看到它被用来指定所用的 JDK 版本。

@param

该标签用于方法文档中，形式如下：

```
@param parameter-name description
```

其中，“parameter-name”是方法的参数列表中的标识符，“description”的文本可延续数行，终止于新的文档标签出现之前。你可以使用任意数量的此标签，大约每个参数都有一个这样的标签。

@return

该标签用于方法文档，格式如下：

```
@return description
```

其中，“description”用来描述返回值的含义，可以延续数行。

@throws

“异常”（Exception）将在第 9 章论述。简言之，它们是由于某个方法调用失败而“抛出”的对象。尽管在调用一个方法时，只有出现一个异常对象，但是某个特殊方法可能会产生任意多、不同类型的异常，所有这些异常都需要进行说明。所以，异常标签的格式如下：

```
@throws fully-qualified-class-name description
```

其中，“fully-qualified-class-name”给出了一个异常类的完整名字，而且该异常类已经在某处定义过。而“description”（同样可以延续数行）告诉你为什么此特殊类型的异常会在方法调用中出现。

@deprecated

该标签用于指出一些旧特性已由改进的新特性所取代，建议用户不要再使用这些旧特性，因为在不久的将来，它们很可能会被移除。如果使用一个标记为@deprecated 的方法，则会引起编译器的警告。

文档示例

下面还是我们的第一个 Java 程序，但是这次加上了文档注释：

```
//: c02:HelloDate.java
import java.util.*;

/** The first Thinking in Java example program.
 * Displays a string and today's date.
 * @author Bruce Eckel
 * @author www.BruceEckel.com
 * @version 2.0
 */
public class HelloDate {
    /** Sole entry point to class & application
     * @param args array of string arguments
     * @return No return value
     * @exception exceptions No exceptions thrown
     */
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
}
```

```
}  
} ///:~
```

第一行采用了我自己独特的方法，是一行用一个“:”作为特殊记号的注释行，来说明此注释行含有源文件名。该行包含了文件的路径信息（此时，c02 代表第 2 章），随后是文件名⁶。最后一行也是一行注释，这个“///:~”标志着源代码清单的结束。这样一来，就可以将这些源代码自动从本书文本中提取出，再用编译器对其进行检查，然后执行。

代码风格

在“Java编程语言编码协定（*Code Conventions for the Java Programming Language*）”⁷中，代码风格是这样规定的：类名的首字母要大写；如果类名由几个单词构成，那么把它们并在一起（也就是说，不要用下划线来分隔名字），其中每个内部单词的首字母都采用大写形式。例如：

```
class AllTheColorsOfTheRainbow { // ...
```

这种风格有时称作“驼峰风格（camel-casing）”。几乎其它的所有内容：方法、域（成员变量）以及对象引用名称等，公认的风格与类的风格一样，只是标识符的第一个字母采用小写。例如：

```
class AllTheColorsOfTheRainbow {  
    int anIntegerRepresentingColors;  
    void changeTheHueOfTheColor(int newHue) {  
        // ...  
    }  
    // ...  
}
```

当然，用户还必须键入所有这些长名字，并且不能输错，因此，一定要格外仔细。

Sun 程序库中的 Java 代码遵循左右（open-and-close）花括号的摆放方式，本书中也是此种格式。

总结

通过本章的学习，大家已接触相当多的关于如何编写一个简单程序的 Java 编程知识。此外，

⁶ 最初，我用Python（见www.Python.org）创建了一个工具，它能够用这份信息提取出代码文件，并放在适当的子目录下，然后创建makefiles。此版本中，所有的文件存放在并发版本系统中（CVS），用Visual BASIC的应用（VBA）宏可将其自动整合到本书中。主要是因为CVS，所以此方法在代码维护方面似乎运作的更好。

⁷ 网址是：<http://java.sun.com/docs/codeconv/index.html>。为了精简本书和研讨会的陈述，这些指南的所有条款并没有全部遵循。

对 Java 语言及它的一些基本思想也有了一个总体认识。然而到目前为止，所有示例都是“这样做，再那样做，接着再做另一些事情”这种形式。如果想让程序做出选择，例如，“假如所做的结果是红色，就那样做；否则，就做另一些事情”。这将又怎么进行呢？Java 为此种基本编程行为所提供的支持，将会在下一章讲述。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 参照本章的 `HelloDate.java` 这个例子，创建一个“Hello, World”程序，该程序只要输出这句话即可。你所编写的类里只需一个方法（“main”方法会在程序启动时被执行）。记住要把它设为 `static` 形式，并指定参数列表——即使根本不会用到这个列表。用 `javac` 进行编译，再用 `java` 运行它。如果你使用的是不同于 `JDK` 的开发环境，请学习如何在你的环境中进行编译和运行。
2. 找出含有 **ATypeName** 的代码段，将其改写成完整的程序，然后编译运行。
3. 将 **DataOnly** 代码段改写成程序，然后编译运行。
4. 修改练习 3，将 **DataOnly** 中的数据在 `main()` 方法中赋值并打印出来。
5. 编写一个程序，含有本章所定义的 `storage()` 方法的代码段，并调用之。
6. 将 **StaticFun** 的代码段改写成完整的可运行程序。
7. 编写一个程序，打印出从命令行获得的三个参数。为此，需要对代表命令行参数的 `String` 数组建立索引。
8. 将 **AllTheColorsOfTheRainbow** 这个示例改写成程序，然后编译运行。
9. 找出 **HelloDate.java** 的第二版本，也就是那个简单注释文档的示例。对该文件执行 `javadoc`，然后通过 `web` 浏览器观看运行结果。
10. 将 **docTest** 改写成文件并编译之，然后用 `javadoc` 运行它。通过 `web` 浏览器查看结果文档。
11. 在练习 10 的文档中加入一个 `HTML` 列表项。
12. 使用练习 1 的程序，加入注释文档。用 `javadoc` 提取此注释文档，并产生一个 `HTML` 文件，然后通过 `web` 浏览器查看结果。
13. 找到第 4 章的 **Overloading.java** 示例，并为它加入 `javadoc` 文档。然后用 `javadoc` 提取此注释文档，并产生一个 `HTML` 文件，最后，通过 `web` 浏览器查看结果。

第三章 控制程序流

就象任何有感知的生物一样，程序必须能操纵自己的世界，在执行过程中作出判断与选择。

在 Java 中，我们利用操作符操纵对象和数据，并用执行控制语句作出选择。Java 是建立在 C++ 基础之上的，所以对 C 和 C++ 程序员来说，应该非常熟悉 Java 的大多数语句和操作符。当然，Java 也作了一些改进与简化工作。

如果你觉得很难理解这一章的内容，请先阅读随书附带的多媒体光盘中的《Foundations for java》。其中包括有声演讲、幻灯片、练习以及解答，这些能带领你快速掌握学习 java 所必需的基础知识。

使用 Java 操作符

操作符接受一个或多个参数，并生成一个新值。参数的形式与普通的调用方法不同，但效果是相同的。加号 (+)、减号和负号 (-)、乘号 (*)、除号 (/) 以及等号 (=) 的用法与其他编程语言都是类似的。

操作符作用于操作数，以生成一个新值。以外，操作符可能会改变操作数自身的值，这被称为“副作用” (Side Effect)。那些能改变操作数的运算，最普遍的用途就是用来产生副作用。但要记住，使用此类操作符生成的值，与使用没有副作用的操作符生成的值，没有什么区别。

几乎所有的操作符都只能操作“基本类型 (Primitives)”。唯一的例外是“=”、“==”和“!=”，它们能操作所有的对象（也是对象易令人迷惑的地方）。除此以外，String 类支持“+”和“+=”。

优先级

当一个表达式中存在多个操作符时，操作符的优先级就决定了各部分的计算顺序。Java 对计算顺序作出了特别的规定。其中，最简单的规则就是先乘除后加减。程序员经常会忘记其他优先级规则，所以应该用括号明确规定计算顺序。例如：

```
a = x + y - 2/2 + z;
```

为上述表达式加上括号后，就有了一个不同的含义。

```
a = x + (y - 2)/(2 + z);
```

赋值

赋值使用等号操作符“=”。它的意思是“取得右边的值（通常称为右值 *rvalue*），把它复制给左边(通常称为左值 *lvalue*)”。右值可以是任何常数、变量或者表达式（只要它能生成一个值就行）。但左值必须是一个明确的、已命名的变量。也就是说，它必须有一个物理空间以存储等号右边的值。举例来说，可将一个常数赋给一个变量：

```
a=4;
```

但是不能把任何东西赋给一个常数——常数不能作为左值（比如不能说 `4=a`）。

对基本数据类型的赋值是很简单的。基本类型存储了实际的数值，而并非指向一个对象的引用，所以在为其赋值的时候，是直接将一个地方的内容复制到了另一个地方。例如，对基本数据类型使用“`a=b`”，那么 `b` 的内容就复制给 `a`。若接着又修改了 `a`，那么 `b` 根本不会受这种修改的影响。作为程序员，这正是大多数情况下我们所期望的。

但是在为对象“赋值”的时候，情况却发生了变化。对一个对象进行操作时，我们真正操作的是对对象的引用。所以倘若“将一个对象赋值给另一个对象”，实际是将“引用”从一个地方复制到另一个地方。这意味着假若对对象使用“`c=d`”，那么 `c` 和 `d` 都指向原本只有 `d` 指向的那个对象。下面这个例子将向大家阐释这一点。

```
//: c03:Assignment.java
// Assignment with objects is a bit tricky.
import com.bruceeckel.simpletest.*;

class Number {
    int i;
}

public class Assignment {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Number n1 = new Number();
        Number n2 = new Number();
        n1.i = 9;
        n2.i = 47;
        System.out.println("1: n1.i: " + n1.i +
            ", n2.i: " + n2.i);
        n1 = n2;
        System.out.println("2: n1.i: " + n1.i +
            ", n2.i: " + n2.i);
        n1.i = 27;
        System.out.println("3: n1.i: " + n1.i +
            ", n2.i: " + n2.i);
    }
}
```

```

        monitor.expect(new String[] {
            "1: n1.i: 9, n2.i: 47",
            "2: n1.i: 47, n2.i: 47",
            "3: n1.i: 27, n2.i: 27"
        });
    }
} ///:~

```

首先，请注意一些新加入的东西。请看下面这行代码：

```
import com.bruceeckel.simpletest.*;
```

它是用来导入“simpletest”库，这是一个专门用来测试本书中的代码的库文件。我们会在第十五章详细解释它。在 Assignment 类的开头，你会看到下面的代码：

```
static Test monitor = new Test();
```

这行代码生成了 simpletest 库中 Test 类的一个实例，命名为 monitor。最后，在 main() 函数的结尾处，有下面的语句：

```

monitor.expect(new String[] {
    "1: n1.i: 9, n2.i: 47",
    "2: n1.i: 47, n2.i: 47",
    "3: n1.i: 27, n2.i: 27"
});

```

这是我们期望程序产生的输出，用字符串对象数组来表示。在程序运行时，不仅打印输出，而且还会与这个数组进行比较，检查输出与这个数组是否一致。所以在本书中，如果你看到某个程序中使用了 simpletest，那么这个程序一定调用了 expect() 函数来向你展示程序的输出应该是什么样子。这样你就能看到经过校验之后的程序输出。

Number 类非常简单，它的两个实例（n1 和 n2）是在 main() 里创建的。对每个 Number 类对象的属性 i 都赋予了一个不同的值，然后，将 n2 赋给 n1，接着又修改了 n1。在许多编程语言中，我们可能会期望 n1 和 n2 总是相互独立的。但由于赋值行为操作的是一个对象的引用，所以你可以从 expect() 的输出中看出，修改 n1 的同时也改变了 n2！这是由于 n1 和 n2 包含的是相同的引用，它们指向相同的对象。（原本 n1 包含的对对象的引用，是指向一个值为 9 的对象。在对 n1 赋值的时候，这个引用被覆盖，也就是丢失了；而那个没有再被引用了的对象会由“垃圾回收器”自动清除）。

这种特殊的现象通常称作“别名现象（aliasing）”，是 Java 操作对象的一种基本方式。在这个例子中，如果你想避免别名应该怎么办呢？你可以这样写：

```
n1.i = n2.i;
```

这样便可以保持两个对象彼此独立，而不是将 `n1` 和 `n2` 绑定到相同的对象。但您很快就会意识到，直接操作对象内部的属性容易导致混乱，并且，违背了良好的面向对象程序设计的原则。这可不是一个小问题，我留待附录 A 中再详细论述，我们会在那里专门讨论别名问题。而从现在开始大家就应该留意，为对象赋值可能会产生意想不到的结果。

方法调用中的别名问题

将一个对象传递给方法时，也会产生别名问题。

```
//: c03:PassObject.java
// Passing objects to methods may not be what
// you're used to.
import com.bruceeckel.simpletest.*;

class Letter {
    char c;
}

public class PassObject {
    static Test monitor = new Test();
    static void f(Letter y) {
        y.c = 'z';
    }
    public static void main(String[] args) {
        Letter x = new Letter();
        x.c = 'a';
        System.out.println("1: x.c: " + x.c);
        f(x);
        System.out.println("2: x.c: " + x.c);
        monitor.expect(new String[] {
            "1: x.c: a",
            "2: x.c: z"
        });
    }
} ///:~
```

在许多编程语言中，方法 `f()` 似乎要在它的作用域内复制参数 `Letter` 类的对象 `y` 的一个副本。但实际上，只是传递了一个引用。所以下面这行代码：

```
y.c = 'z';
```

实际改变的是 `f()` 之外的对象。`expect()` 展示了输出结果。

别名引起的问题，以及解决它的办法是很复杂的话题。尽管要到阅读了附录 A 之后，你才

能得到所有的答案，但是你现在就应该知道它的存在，并在使用中注意这个陷阱。

算术操作符 (Mathematical operator)

Java 的基本算术操作符与其他大多数程序设计语言是相同的。其中包括加号(+)、减号(-)、除号(/)、乘号(*)以及模数(%)，从整数除法中获得余数)。整数除法会直接去掉结果的小数位，而不是四舍五入的进位。

Java 也使用一种简化符号同时进行运算与赋值操作。它用操作符后紧跟一个等号来表示，这对于 Java 中的所有操作符都适用，只要其有实际意义就行。例如，要将 x 加 4，并将结果赋回给 x，可以这么写：x+=4。

下面这个例子展示了各种算术操作符的用法：

```
//: c03:MathOps.java
// Demonstrates the mathematical operators.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class MathOps {
    static Test monitor = new Test();
    // Shorthand to print a string and an int:
    static void printInt(String s, int i) {
        System.out.println(s + " = " + i);
    }
    // Shorthand to print a string and a float:
    static void printFloat(String s, float f) {
        System.out.println(s + " = " + f);
    }
    public static void main(String[] args) {
        // Create a random number generator,
        // seeds with current time by default:
        Random rand = new Random();
        int i, j, k;
        // Choose value from 1 to 100:
        j = rand.nextInt(100) + 1;
        k = rand.nextInt(100) + 1;
        printInt("j", j); printInt("k", k);
        i = j + k; printInt("j + k", i);
        i = j - k; printInt("j - k", i);
        i = k / j; printInt("k / j", i);
        i = k * j; printInt("k * j", i);
        i = k % j; printInt("k % j", i);
    }
}
```



```

j %= k; printInt("j %= k", j);
// Floating-point number tests:
float u,v,w; // applies to doubles, too
v = rand.nextFloat();
w = rand.nextFloat();
printFloat("v", v); printFloat("w", w);
u = v + w; printFloat("v + w", u);
u = v - w; printFloat("v - w", u);
u = v * w; printFloat("v * w", u);
u = v / w; printFloat("v / w", u);
// the following also works for
// char, byte, short, int, long,
// and double:
u += v; printFloat("u += v", u);
u -= v; printFloat("u -= v", u);
u *= v; printFloat("u *= v", u);
u /= v; printFloat("u /= v", u);
monitor.expect(new String[] {
    "% j = -?\\d+",
    "% k = -?\\d+",
    "% j \\+ k = -?\\d+",
    "% j - k = -?\\d+",
    "% k / j = -?\\d+",
    "% k \\* j = -?\\d+",
    "% k % j = -?\\d+",
    "% j %= k = -?\\d+",
    "% v = -?\\d+\\.\\d+(E-?\\d)?",
    "% w = -?\\d+\\.\\d+(E-?\\d)?",
    "% v \\+ w = -?\\d+\\.\\d+(E-?\\d)??",
    "% v - w = -?\\d+\\.\\d+(E-?\\d)??",
    "% v \\* w = -?\\d+\\.\\d+(E-?\\d)??",
    "% v / w = -?\\d+\\.\\d+(E-?\\d)??",
    "% u \\+= v = -?\\d+\\.\\d+(E-?\\d)??",
    "% u -= v = -?\\d+\\.\\d+(E-?\\d)??",
    "% u \\*= v = -?\\d+\\.\\d+(E-?\\d)??",
    "% u /= v = -?\\d+\\.\\d+(E-?\\d)?"
});
}
} ///:~

```

首先注意用于打印的一些便捷方法: `printInt()` 方法先打印一个 `String`, 再打印一个 `int`; 而 `printFloat()` 先打印一个 `String`, 再打印一个 `float`。

为了生成数字, 程序首先会创建一个 `Random` 类的对象。由于在创建过程中没有传递任何参

数，所以 Java 将当前时间作为随机数生成器的种子。通过 `Random` 类的对象，程序可生成许多不同类型的随机数字。做法很简单，只需调用不同的方法即可：`nextInt()`，`nextFloat()`，（你也可以调用 `nextLong()` 或者 `nextDouble()`）。

当使用随机数生成器产生的结果时，取模运算（%）可将结果限制在上限为运算对象最大值减 1 的范围内（本例是 99）。

正则表达式（regular expression）

由于上面的例子程序使用随机数生成输出，且每一次的输出都与下一次输出不同，所以 `expect()` 函数无法知道确切的输出。要解决这个问题，可以在 `expect()` 中使用正则表达式。这是 Java JDK 1.4 中引入的新特性（但在 Perl 和 Python 中早就有了）。虽然要到第十二章我们才用到了这个非常强大的工具，但是现在我们还是需要学习一些正则表达式的知识，以便理解这些语句。这里我们只需要读懂 `expect()` 语句，如果你想全面地了解正则表达式，可以参考 JDK 文档中的 `java.util.regex.Pattern`。

正则表达式是使用通用术语（**general terms**）来描述字符串的一种方法。你可以说：“如果一个字符串包含这些东西，那么它与我要找的东西相匹配。”例如，要表达一个数可能有，也可能没有负号，你可以在负号后面跟一个问号，就像下面这样：

```
-?
```

要表示一个整数，你可以描述其具有一位或多位数字。在正则表达式中，一个数位用 `'\d'` 表示，但在 Java 的字符串类型中，我们必须添加一个反斜线才能“转义”表达出反斜线：`'\\d'`。在正则表达式中要说明有“一个或多个前述的表达式”，就要使用 `'+'`。所以要表达“可能有负号，后面有一个或多个数位”，我们要这样写：

```
-?\\d+
```

这就是前面代码 `expect()` 语句中的第一行。

在 `expect()` 方法中的各行的开头部分，`'%%'` 不是正则表达式的语法（注意，包含空格是为了可读性），而是一个标记，`simpletest` 使用它来表示这一行余下的部分是一个正则表达式。所以在标准的正则表达式中是不会看到它的，它只在 `simpletest` 的 `expect()` 语句中出现。

对于其他的字符，只要不是正则表达式专有的特殊字符，都要求精确匹配。所以在第一行代码中：

```
%% j = -?\\d+
```

`'j='` 需要准确匹配。但是，在第三行中，`'j+k'` 中的 `'+'` 需要转义，因为它是正则表达式的特殊字符，同样的还有 `'*'`。通过前面的介绍，这一行余下的代码你应该能理解了。在后

面，`expect()` 还会使用到正则表达式其他的特性，到时候我们再解释。

一元加、减操作符

一元减号 (-) 和一元加号 (+) 与二元加号和减号都是相同的符号。根据表达式的书写形式，编译器会自动判断出使用的是哪一种。例如下述语句：

```
x = -a;
```

它的含义是显然的。编译器能正确识别下述语句：

```
x = a * -b;
```

但读者会被搞糊涂，所以最好更明确地写成：

```
x = a * (-b);
```

一元减号用于转变数据的符号，而一元加号只是为了与一元减号相对应，它实际并不做任何事情。

自动递增 (increment) 和递减 (decrement)

和 C 类似，Java 提供了丰富的快捷运算。这些快捷运算使编码更方便，同时也使得代码更容易阅读，但是有时可能使代码阅读起来更困难。

递增和递减运算是两种相当不错的快捷运算（常称作“自动递增”和“自动递减”运算）。其中，递减操作符是“--”，意为“减少一个单位”；递增操作符是“++”，意为“增加一个单位”。举个例子来说，假设 `a` 是一个 `int`（整数）值，则表达式 `++a` 就等价于 `(a = a + 1)`。递增和递减操作符不仅改变了变量，并且以变量的值作为生成的结果。

这两个操作符各有两种使用方式，通常称为“前缀式”和“后缀式”。“前缀递增”表示 `++` 操作符位于变量或表达式的前面；而“后缀递增”表示 `++` 操作符位于变量或表达式的后面。类似地，“前缀递减”意味着 `--` 操作符位于变量或表达式的前面；而“后缀递减”意味着 `--` 操作符位于变量或表达式的后面。对于前缀递增和前缀递减（如 `++a` 或 `--a`），会先执行运算，再生成值。而对于后缀递增和后缀递减（如 `a++` 或 `a--`），会先生成值，再执行运算。下面是一个例子：

```
//: c03:AutoInc.java
// Demonstrates the ++ and -- operators.
import com.bruceeckel.simpletest.*;

public class AutoInc {
```

```

static Test monitor = new Test();
public static void main(String[] args) {
    int i = 1;
    System.out.println("i : " + i);
    System.out.println("++i : " + ++i); // Pre-increment
    System.out.println("i++ : " + i++); // Post-increment
    System.out.println("i : " + i);
    System.out.println("--i : " + --i); // Pre-decrement
    System.out.println("i-- : " + i--); // Post-decrement
    System.out.println("i : " + i);
    monitor.expect(new String[] {
        "i : 1",
        "++i : 2",
        "i++ : 2",
        "i : 3",
        "--i : 2",
        "i-- : 2",
        "i : 1"
    });
}
} ///:~

```

从中可以看到，对于前缀形式，我们在执行完运算后才得到值。但对于后缀形式，则是在运算执行之前就得到值。它们是（除那些涉及赋值的操作符以外）唯一具有“副作用”的操作符。也就是说，它们会改变运算对象，而不仅仅是使用自己的值。

递增操作符正是对“C++”这个名字的一种解释，暗示着“超越 C 一步”。在早期的一次有关 Java 的演讲中，**Bill Joy**（Java 始创人之一）声称“Java=C++--”（C 加加减减），意味着 Java 已去除了 C++ 中一些很困难而又没必要的东西，成为了一种更精简的语言。正如大家会在这本书中学到的那样，Java 的许多地方更精简了，然而并不是说 Java 就比 C++ 容易很多。

关系操作符（relational operator）

关系操作符生成的是一个“布尔”（Boolean）结果，它们计算的是操作对象之间的关系。如果关系是真实的，关系表达式会生成 **true**（真）；如果关系不真实，则生成 **false**（假）。关系操作符包括小于（<）、大于（>）、小于或等于（<=）、大于或等于（>=）、等于（==）以及不等于（!=）。等于和不等于是适用于所有内置的数据类型，而其他比较符不适用于 **boolean** 类型。

测试对象的等价性

关系操作符==和!=都适用于所有对象，但它们的含义通常会使第一次接触 Java 的程序员感到迷惑。下面是一个例子：

```
//: c03:Equivalence.java
import com.bruceeckel.simpletest.*;

public class Equivalence {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);
        monitor.expect(new String[] {
            "false",
            "true"
        });
    }
} ///:~
```

表达式 `System.out.println(n1 == n2)` 将打印出括号内的比较式的布尔值结果。你可能认为输出结果肯定先是 `true`，再是 `false`，因为两个 `Integer` 对象都是相同的。但是尽管对象的内容相同，对象的引用却是不同的，而 `==` 和 `!=` 比较的就是对象的引用。所以输出结果实际上先是 `false`，再是 `true`。这自然会使第一次接触关系操作符的人感到惊奇。

如果想比较两个对象的实际内容是否相同，又该如何操作呢？此时，必须使用所有对象都适用的特殊方法 `equals()`。但这个方法不适用于“基本类型”，基本类型直接使用 `==` 和 `!=` 即可。下面举例说明如何使用：

```
//: c03:EqualsMethod.java
import com.bruceeckel.simpletest.*;

public class EqualsMethod {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1.equals(n2));
        monitor.expect(new String[] {
            "true"
        });
    }
} ///:~
```

正如我们预计的那样，此时得到的结果是 `true`。但事情并不总是这么简单！假设您创建了自己的类，就象下面这样：

```
//: c03:EqualsMethod2.java
import com.bruceeckel.simpletest.*;

class Value {
    int i;
}

public class EqualsMethod2 {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
        monitor.expect(new String[] {
            "false"
        });
    }
} ///:~
```

此时的结果又变回了 `false`！这是由于 `equals()` 的默认行为是比较引用。所以除非在自己的新类中重载 `equals()` 方法，否则不可能表现出我们希望的行为。不幸的是，我们要到第 7 章才学习重载，到第 11 章才学习如何恰当地定义 `equals()`。但在这之前，请留意 `equals()` 的这种行为方式，或许能够避免一些“灾难”。

大多数 Java 类库都实现了用来比较对象的内容，而非比较对象引用的 `equals()` 方法。

逻辑运算符（logical operator）

逻辑运算“与”（`&&`）、“或”（`||`）、“非”（`!`）能根据参数的逻辑关系，生成一个布尔值（`true` 或 `false`）。下面这个例子就使用了关系和逻辑操作符。

```
//: c03:Bool.java
// Relational and logical operators.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class Bool {
    static Test monitor = new Test();
    public static void main(String[] args) {
```

```

Random rand = new Random();
int i = rand.nextInt(100);
int j = rand.nextInt(100);
System.out.println("i = " + i);
System.out.println("j = " + j);
System.out.println("i > j is " + (i > j));
System.out.println("i < j is " + (i < j));
System.out.println("i >= j is " + (i >= j));
System.out.println("i <= j is " + (i <= j));
System.out.println("i == j is " + (i == j));
System.out.println("i != j is " + (i != j));
// Treating an int as a boolean is not legal Java:
//! System.out.println("i && j is " + (i && j));
//! System.out.println("i || j is " + (i || j));
//! System.out.println("!i is " + !i);
System.out.println("(i < 10) && (j < 10) is "
    + ((i < 10) && (j < 10)) );
System.out.println("(i < 10) || (j < 10) is "
    + ((i < 10) || (j < 10)) );
monitor.expect(new String[] {
    "% i = -?\\d+",
    "% j = -?\\d+",
    "% i > j is (true|false)",
    "% i < j is (true|false)",
    "% i >= j is (true|false)",
    "% i <= j is (true|false)",
    "% i == j is (true|false)",
    "% i != j is (true|false)",
    "% \\(i < 10\\) && \\(j < 10\\) is (true|false)",
    "% \\(i < 10\\) \\||\\ \\(j < 10\\) is (true|false)"
});
}
} ///:~

```

在 `expect()` 语句包含的正则表达式中, 使用小括号能把表达式分组, 而 `'|'` 表示逻辑“与”。所以:

```
(true|false)
```

意指: 字符串的这部分不是 `'true'` 就是 `'false'`。因为这些字符在正则表达式中有特殊的含义, 所以如果你希望它们像普通字符一样出现在表达式中, 就必须用一个 `'\\'` 来转义。

逻辑与、逻辑或、逻辑非操作只可应用于布尔值。与在 C 及 C++ 中不同的是: 不可将一个非布尔值当作布尔值在逻辑表达式中使用。在前面的代码中用 `“//!”` 注释掉的语句, 就是错误

的用法。后面的两个表达式先使用关系比较运算，生成布尔值，然后再对产生的布尔值进行逻辑运算。

注意，如果在应该使用 `String` 值的地方使用了布尔值，它会自动转换成适当的文本。

在上述程序中，可将整数类型替换成除布尔型以外的其他任何基本数据类型。但要注意，对浮点数的比较是非常严格的。即使一个数仅在小数部分与另一个数存在极微小的差异，仍然认为它们是“不相等”的。即使一个数只比零大一点点，它仍然是“非零”值。

短路 (Short-circuiting)

当进行逻辑运算时，我们会遇到一种“短路”现象。即一旦能够明确无误的确定整个表达式的值，我们就不再计算表达式余下的部分了。因此，一个完整的逻辑表达式靠后的部分有可能不会被运算。下面是演示短路现象的一个例子：

```
//: c03:ShortCircuit.java
// Demonstrates short-circuiting behavior.
// with logical operators.
import com.bruceeckel.simpletest.*;

public class ShortCircuit {
    static Test monitor = new Test();
    static boolean test1(int val) {
        System.out.println("test1(" + val + ")");
        System.out.println("result: " + (val < 1));
        return val < 1;
    }
    static boolean test2(int val) {
        System.out.println("test2(" + val + ")");
        System.out.println("result: " + (val < 2));
        return val < 2;
    }
    static boolean test3(int val) {
        System.out.println("test3(" + val + ")");
        System.out.println("result: " + (val < 3));
        return val < 3;
    }
    public static void main(String[] args) {
        if(test1(0) && test2(2) && test3(2))
            System.out.println("expression is true");
        else
            System.out.println("expression is false");
        monitor.expect(new String[] {
            "test1(0)",
```



```

        "result: true",
        "test2(2)",
        "result: false",
        "expression is false"
    });
}
} ///:~

```

每个测试都会比较参数，并返回真或假。它也会打印信息告诉你测试正在被调用。这些测试都作用于下面这个表达式：

```
if(test1(0)) && test2(2) && test3(2))
```

你会很自然地认为所有这三个测试都会得以执行。但输出结果却并不是这样。第一个测试生成结果 **true**，所以表达式计算会继续下去。然而，第二个测试产生了一个 **false** 结果。由于这意味着整个表达式肯定为 **false**，所以为什么还要继续计算剩余的表达式呢？那只是浪费。“短路”一词的由来正源于此。事实上，如果所有的逻辑表达式都有一部分不必计算，我们将获得潜在的性能提升。

位操作符（bitwise operator）

位操作符允许我们操作一个基本数据类型中的整数型值的单个“比特（bit）”，即二进制位。位操作符会对两个参数对应的位执行布尔代数运算，并最终生成一个结果。

位操作符来源于 C 语言面向底层的操作，那时我们经常需要直接操纵硬件，设置硬件寄存器内的二进制位。Java 的设计初衷是嵌入电视机顶盒内，所以这种低级操作仍被保留了下来。但是，我们可能不会过多地使用到位运算符。

如果两个输入位都是 1，则按位“与”操作符（&）生成一个输出位 1；否则生成一个输出位 0。如果两个输入位里只要有一个是 1，则按位“或”操作符（|）生成一个输出位 1；只有在两个输入位都是 0 的情况下，它才会生成一个输出位 0。如果两个输入位的某一个为 1，但不全都是 1，那么“异或”操作（^）生成一个输出位 1。按位“非”（~，也称为取补运算，ones complement operator）属于一元操作符；它只对一个操作数进行操作（其他位操作是二元运算）。按位“非”生成与输入位相反的值——若输入 0，则输出 1；输入 1，则输出 0。

位操作符和逻辑操作符都使用了同样的符号。因此，我们能方便地记住它们的含义：由于“位”是非常“小”的，所以位操作符仅使用了一位符号。

位操作符可与等号（=）联合使用，以便合并运算操作和赋值操作：&=，|=和^=都是合法的（由于~是一元操作符，所以不可与=联合使用）。

我们将布尔类型（boolean）作为一种“单比特”值对待，所以它多少有些独特的地方。我们

可对它执行按位“与”、“或”和“异或”运算，但不能执行按位“非”（大概是为了避免与逻辑 NOT 混淆）。对于布尔值，位操作符具有与逻辑操作符相同的效果，只是它们不会中途“短路”。此外，针对布尔值进行的按位运算为我们新增了一个“异或”逻辑操作符，它并未包括在“逻辑”操作符的列表中。在移位表达式中，我们被禁止使用布尔运算，原因将在下面解释。

移位操作符（shift operator）

移位操作符操作的运算对象也是二进制的“位”，但是它们只可以被用来处理整数类型（基本类型的一种）。左移位操作符（<<）能将操作符左边的运算对象向左移动操作符右侧指定的位数（在低位补 0）。“有符号”右移位操作符（>>）则将操作符左边的运算对象向右移动操作符右侧指定的位数。“有符号”右移位操作符使用了“符号扩展”：若符号为正，则在高位插入 0；若符号为负，则在高位插入 1。Java 中增加了一种“无符号”右移位操作符（>>>），它使用了“零扩展”：无论正负，都在高位插入 0。这一操作符是 C 或 C++ 没有的。

如果对 char、byte 或者 short 类型的数值进行移位处理，那么在移位进行之前，它们会自动转换为 int，并且得到的结果也是一个 int 类型的值。而右侧操作数，作为真正移位的位数，只有其二进制表示中的低 5 位才有用。这样可防止我们移位超过 int 型值所具有的位数。（译注：因为 2 的 5 次方为 32，而 int 型值只有 32 位）。若对一个 long 类型的数值进行处理，最后得到的结果也是 long。此时只会用到右侧操作数的低 6 位，以防止移位超过 long 型数值具有的位数。

移位可与等号（<<=或>>=或>>>=）组合使用。此时，操作符左边的值会移动由右边的值指定的位数，再将得到的结果赋回左边的变量。但在进行“无符号”右移结合赋值操作时，可能会遇到一个问题：如果对 byte 或 short 值进行这样的移位运算，得到的可能不是正确的结果。它们会先被转换成 int 类型，再进行右移操作。然后被截断，赋值给原来的类型，在这种情况下可能得到-1 的结果。下面这个例子演示了这种情况：

```
//: c03:URShift.java
// Test of unsigned right shift.
import com.bruceeckel.simpletest.*;

public class URShift {
    static Test monitor = new Test();
    public static void main(String[] args) {
        int i = -1;
        System.out.println(i >>>= 10);
        long l = -1;
        System.out.println(l >>>= 10);
        short s = -1;
        System.out.println(s >>>= 10);
        byte b = -1;
        System.out.println(b >>>= 10);
    }
}
```

```

        b = -1;
        System.out.println(b>>>10);
        monitor.expect(new String[] {
            "4194303",
            "18014398509481983",
            "-1",
            "-1",
            "4194303"
        });
    }
} ///:~

```

在最后一个移位运算中，结果没有赋回给 `b`，而是直接打印出来，所以其结果是正确的。

下面这个例子向大家阐释了如何应用涉及“按位”操作的所有操作符：

```

//: c03:BitManipulation.java
// Using the bitwise operators.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class BitManipulation {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Random rand = new Random();
        int i = rand.nextInt();
        int j = rand.nextInt();
        printBinaryInt("-1", -1);
        printBinaryInt("+1", +1);
        int maxpos = 2147483647;
        printBinaryInt("maxpos", maxpos);
        int maxneg = -2147483648;
        printBinaryInt("maxneg", maxneg);
        printBinaryInt("i", i);
        printBinaryInt("~i", ~i);
        printBinaryInt("-i", -i);
        printBinaryInt("j", j);
        printBinaryInt("i & j", i & j);
        printBinaryInt("i | j", i | j);
        printBinaryInt("i ^ j", i ^ j);
        printBinaryInt("i << 5", i << 5);
        printBinaryInt("i >> 5", i >> 5);
        printBinaryInt("(~i) >> 5", (~i) >> 5);
        printBinaryInt("i >>> 5", i >>> 5);
    }
}

```

```

printBinaryInt("(~i) >>> 5", (~i) >>> 5);

long l = rand.nextLong();
long m = rand.nextLong();
printBinaryLong("-1L", -1L);
printBinaryLong("+1L", +1L);
long ll = 9223372036854775807L;
printBinaryLong("maxpos", ll);
long llN = -9223372036854775808L;
printBinaryLong("maxneg", llN);
printBinaryLong("1", 1);
printBinaryLong("~1", ~1);
printBinaryLong("-1", -1);
printBinaryLong("m", m);
printBinaryLong("1 & m", 1 & m);
printBinaryLong("1 | m", 1 | m);
printBinaryLong("1 ^ m", 1 ^ m);
printBinaryLong("1 << 5", 1 << 5);
printBinaryLong("1 >> 5", 1 >> 5);
printBinaryLong("(~1) >> 5", (~1) >> 5);
printBinaryLong("1 >>> 5", 1 >>> 5);
printBinaryLong("(~1) >>> 5", (~1) >>> 5);
monitor.expect("BitManipulation.out");
}

static void printBinaryInt(String s, int i) {
    System.out.println(
        s + ", int: " + i + ", binary: ");
    System.out.print("  ");
    for(int j = 31; j >= 0; j--)
        if(((1 << j) & i) != 0)
            System.out.print("1");
        else
            System.out.print("0");
    System.out.println();
}

static void printBinaryLong(String s, long l) {
    System.out.println(
        s + ", long: " + l + ", binary: ");
    System.out.print("  ");
    for(int i = 63; i >= 0; i--)
        if(((1L << i) & l) != 0)
            System.out.print("1");
        else
            System.out.print("0");
}

```

```

        System.out.println();
    }
} ///:~

```

程序末尾调用了两个方法：`printBinaryInt()`和`printBinaryLong()`。它们分别接受一个 `int` 或 `long` 值的参数，并用二进制格式输出，同时附有简要的说明文字。你可以暂时忽略它们具体是如何实现的。

请注意这里是用 `System.out.print()`，而不是 `System.out.println()`。`print()`方法不自动换行，所以我们能在同一行里输出多个信息。

上面的例子中，`expect()` 以一个文件名作参数，它会从这个文件中读取预期的行（其中可以有，也可以没有正则表达式）。对于那些太长，不适宜列在书里的输出，这种做法很有用。这个文件的扩展名是“.out”，是所发布的代码的一部分，可以从www.BruceEckel.com下载。如果有兴趣的话，可以打开这个文件，看看正确的输出应该是什么（或者你自己直接运行一下前面这个程序）。

上面的例子展示了对 `int` 和 `long` 的所有按位运算的效果，还展示了 `int` 和 `long` 的最小值、最大值、+1 和-1 值，以及它们的二进制形式，以使大家了解它们在机器中的具体形式。注意，最高位表示符号：0 为正，1 为负。下面列出例子中关于 `int` 部分的输出：

```

-1, int: -1, binary:
11111111111111111111111111111111
+1, int: 1, binary:
00000000000000000000000000000001
maxpos, int: 2147483647, binary:
01111111111111111111111111111111
maxneg, int: -2147483648, binary:
10000000000000000000000000000000
i, int: 59081716, binary:
00000011100001011000001111110100
~i, int: -59081717, binary:
11111100011110100111110000001011
-i, int: -59081716, binary:
11111100011110100111110000001100
j, int: 198850956, binary:
00001011110110100011100110001100
i & j, int: 58720644, binary:
000000111000000000000000110000100
i | j, int: 199212028, binary:
0000101111011111011101111111100
i ^ j, int: 140491384, binary:
00001000010111111011101001111000
i << 5, int: 1890614912, binary:

```

```

01110000101100000111111010000000
i >> 5, int: 1846303, binary:
00000000000111000010110000011111
(~i) >> 5, int: -1846304, binary:
11111111111000111101001111100000
i >>> 5, int: 1846303, binary:
00000000000111000010110000011111
(~i) >>> 5, int: 132371424, binary:
00000111111000111101001111100000

```

数字的二进制表示形式被称为“有符号的 2 的补码”。

三元操作符 if-else

这种操作符比较特别，因为它有三个操作对象。但它确实属于操作符的一种，因为它最终也会生成一个值。这与本章下一节中介绍的普通 if-else 语句是不同的。其表达式采取下述形式：

```
boolean-exp ? value0 : value1
```

如果“布尔表达式”的结果为 true，就计算 “value0”，而且这个计算结果也就是操作符最终产生的值。如果“布尔表达式”的结果为 false，就计算“value1”，同样，它的结果也就成为了操作符最终产生的值。

当然，也可以换用普通的 if-else 语句（在后面介绍），但三元操作符更加简洁。尽管 C 引以为傲的就是它是一种简练的语言，而且三元操作符的引入多半就是为了体现这种高效率的编程，但假如你打算频繁使用它，还是要先多作一些思量——因为它很容易就会产生可读性极差的代码。

这种条件操作符的使用目的，有时是为了它的“副作用”，有时是为了它运算生成的值。一般而言，我们需要的是其运算的结果值，这正是这个三元操作符与 if-else 不同之处。下面便是一个例子：

```

static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
}

```

可以看出，上面的代码比起不用三元操作符来说显得非常紧凑。

```

static int alternative(int i) {
    if (i < 10)
        return i * 100;
}

```

```
else
    return i * 10;
}
```

第二种形式更易理解，而且不需要太多的录入。所以在选择使用三元操作符时，请务必仔细考量。

逗号操作符（comma operator）

在 C 和 C++ 中，逗号不仅作为函数自变量列表的分隔符使用，也作为进行后续计算的一个操作符使用。在 Java 中，需要用到逗号的唯一场合就是 for 循环，本章稍后会对此详加解释。

字符串操作符 +

这个操作符在 Java 中有一项特殊用途：连接不同的字符串。这一点已经在前面的例子中展示过了。尽管与 + 的传统使用方式不太一样，但我们还是很自然地使用 + 来做这件事情。在 C++ 中，这个主意非常不错，所以引入了“操作符重载（operator overriding）”机制，以便 C++ 程序员可以为几乎所有操作符增加功能。但非常遗憾，与 C++ 的另外一些限制结合在一起，使得操作符重载成为了一种非常复杂的特性，程序员在设计自己的类时必须对此有非常周全的考虑。与 C++ 相比，尽管操作符重载在 Java 中更易实现，但仍然过于复杂。所以 Java 程序员不能象 C++ 程序员那样设计自己的重载操作符。

我们注意到字符串相加时有一些有趣的现象。如果表达式以一个字符串起头，那么后续所有操作对象都必须是字符串型(请记住，编译器会把双引号内的字符序列转成字符串)：

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

在这里，Java 编译器会将 x, y 和 z 转换成它们的字符串形式在作处理，而不是先把它们加到一起。如果使用下述语句：

```
System.out.println(x + sString);
```

Java 会把 x 转换成字符串。

使用操作符时常犯的错误

使用操作符时一个常犯的错误就是，虽然你对一个表达式如何计算有点不确定，却不愿意使

用括号。这个问题在 Java 中仍然存在。

在 C 和 C++ 中，一个特别常见的错误如下：

```
while(x = y) {  
    //...  
}
```

程序员很明显是想测试是否“相等”（==），而不是进行赋值操作。在 C 和 C++ 中，如果 y 是一个非零值，那么这种赋值的结果肯定是 true，而这样便会得到一个无穷循环。在 Java 中，这个表达式的结果并不是布尔值，而编译器期望的是一个布尔值。由于 Java 不会自动地将 int 数值转换成布尔值，所以在编译时会抛出一个编译期错误，从而有效地阻止了我们更进一步地去运行程序。所以这个陷阱在 Java 中永远不会出现。（唯一不会得到编译错误的情况是 x 和 y 都为布尔值。在这种情况下，x = y 属于合法表达式。而在前面的例子中，则可能是一个错误。）

Java 中有一个与 C 和 C++ 中类似的问题，即使用按位与和按位或代替逻辑与和逻辑或。按位与和按位或使用单字符（&或|），而逻辑与和逻辑或使用双字符（&&或||）。就象“=”和“==”一样，键入一个字符当然要比键入两个简单。Java 编译器可防止这个错误发生，因为它不允许我们随便的把一种类型当作另一种类型来用。

类型转换操作符（casting operator）

“类型转换”（Cast）的原意是“与一个模型匹配”。在适当的时候，Java 会将一种数据类型自动转换成另一种。例如，假设我们为某浮点变量赋以一个整数值，编译器会将 int 自动转换成 float。类型转换运算允许我们显式地进行这种类型的转换，或者在不能自动进行转换的时候强制进行类型转换。

要想执行类型转换，需要将希望得到的数据类型（包括所有修饰符）置于小括号内，放在要进行类型转换的值的左边。下面是一个例子：

```
void casts() {  
    int i = 200;  
    long l = (long)i;  
    long l2 = (long)200;  
}
```

正如你所看到的，既可对一个数值进行类型转换，亦可对一个变量进行类型转换。但在这儿展示的两种情况，类型转换均是多余的，因为编译器在必要的时候会自动进行 int 值到 long 值的提升。但是你仍然可以做这样“多余的”事，以提醒自己需要留意，也可以使代码更清楚。在其他情况下，可能只有先进行类型转换，代码编译才能通过。

在 C 和 C++ 中，类型转换有时会让人头痛。但是在 Java 中，类型转换则是一种比较安全的

操作。然而，如果要执行一种名为“窄化转换”（Narrowing Conversion）的操作（也就是说，当你将能容纳更多信息的数据类型转换成无法容纳那么多信息的类型时），就有可能面临信息丢失的危险。此时，编译器会强制我们进行类型转换，这实际上相当于说：“这可能是一件危险的事情，如果您无论如何一定要让我这么做，您必须显式地进行类型转换。”而对于“扩展转换”（Widening conversion），则不必进行显式地类型转换，因为新类型肯定能容纳原来类型的信息，不会造成任何信息的丢失。

Java 允许我们把任何基本数据类型转换成别的基本数据类型，但布尔型值除外，后者根本不允许进行任何类型转换处理。“类”不允许进行类型转换。为了将一种类转换成另一种，必须采用特殊的方法（字符串是一种特殊的情况。本书后面会讲到，一个对象可以在其所属类的类族之间可以进行类型转换；例如，“橡树”可转型为“树”；反之亦然。但不能把它转换成类族以外的类型，如“岩石”）。

直接常量（Literal）

一般说来，如果在一个程序里使用了“直接常量”（Literal），编译器可以准确地知道要生成什么样的类型，但有时候却是模棱两可的。如果发生这种情况，必须对编译器加以适当的“指导”，用与直接量搭配某些字符来增加一些信息。下面这段代码向大家展示了这些字符。

```
//: c03:Literals.java

public class Literals {
    char c = 0xffff; // max char hex value
    byte b = 0x7f; // max byte hex value
    short s = 0x7fff; // max short hex value
    int i1 = 0x2f; // Hexadecimal (lowercase)
    int i2 = 0X2F; // Hexadecimal (uppercase)
    int i3 = 0177; // Octal (leading zero)
    // Hex and Oct also work with long.
    long n1 = 200L; // long suffix
    long n2 = 200l; // long suffix (but can be confusing)
    long n3 = 200;
    //! long 16(200); // not allowed
    float f1 = 1;
    float f2 = 1F; // float suffix
    float f3 = 1f; // float suffix
    float f4 = 1e-45f; // 10 to the power
    float f5 = 1e+9f; // float suffix
    double d1 = 1d; // double suffix
    double d2 = 1D; // double suffix
    double d3 = 47e47d; // 10 to the power
} ///:~
```

十六进制数（Base 16），适用于所有整数数据类型，以一个前缀 0x 或 0X 来表示，后面跟随

0-9 或大小写的 a-f。如果试图将一个变量初始化成超出自身表示范围的值（无论这个值的数值形式如何），编译器都会向我们报告一条错误信息。注意在前面的代码中，已经给出了 char、byte 以及 short 能表示的最大值。如果超出范围，编译器会将其值自动转换成一个 int，并告诉我们需要对这一次赋值进行“窄化转型”。这样我们就可清楚地知道自己的操作是否越界了。

八进制数（Base 8）是由前缀 0 以及后续的 0-7 的数字组成的。在 C、C++ 或者 Java 中，没有二进制数直接常量的表示方法。

直接常量后面的后缀字符标志了它的类型。若为 大写或小写的 L，代表 long；大写或小写的 F，代表 float；大写或小写的 D，则代表 double。

指数总是采用一种很不直观的记号方法：1.39e-47f。在科学与工程领域，“e”代表自然对数的基数，约等于 2.718（Java 中的 Math.E 给出了更精确的 double 型的值）。它在象 $1.39 \times e^{-47}$ 这样的指数表达式中使用，意味着 1.39×2.718^{-47} 。然而，设计 FORTRAN 语言的时候，设计师们很自然地决定 e 代表“10 的幂次”。这种做法很奇怪，因为 FORTRAN 最初是面向科学与工程领域的设计的，它的设计者们对引入这样容易混淆的概念应该很敏感才对¹。但不管怎样，这种惯例在 C、C++ 以及 Java 中被保留下来了。所以倘若您习惯将 e 作为自然对数的基数使用，那么在 Java 中看到象 “1.39e-47f” 这样的表达式时，请转换您的思维，它真正的含义是 1.39×10^{-47} 。

注意如果编译器能够正确地识别类型，就不必在数值后附加字符。例如下述语句：

```
long n3 = 200;
```

这里不存在含混不清的地方，所以 200 后面的一个 L 是多余的。然而，对于下述语句：

```
float f4 = 1e-47f; // 10 to the power
```

编译器通常会将指数作为双精度数（double）处理，所以假如没有这个尾随的 f，就会收到一条出错提示，告诉我们必须使用类型转换将 double 转换成 float。

¹ John Kirkham 写道：“我开始用计算机是在 1962 年，使用的是 IBM 1620 机器上的 FORTRAN II。那时候，从 60 年代到 70 年代，FORTRAN 一直都是使用大写字母。之所以会出现这一情况，可能是由于早期的输入设备大多是老式电传打字机，使用 5 位 Baudot 码，那是不包括小写字母的。乘幂表达式中的 ‘E’ 也肯定是大写的，所以不会与自然对数的基数 ‘e’ 发生冲突，后者必然是小写的。‘E’ 这个字母的含义其实很简单，就是 ‘Exponential’ 的意思，即 ‘指数’ 或 ‘幂数’，代表数字系统的基数——一般都是 10。当时，八进制也在程序员中广泛使用。尽管我自己未看到它的使用，但假若我在乘幂表达式中看到一个八进制数字，我就会认为基数是 8。我记得第一次看到用小写 ‘e’ 表示指数是在 70 年代末期。我当时也觉得它极易产生混淆。产生这个问题是因为 FORTRAN 已经渐渐引入了小写字母，但并非一开始就有。如果你真的想使用自然对数的基数，有现成的函数可供利用，但它们都是大写的。”

提升 (Promotion)

如果对基本数据类型执行算术运算或按位运算，大家会发现，只要它们“比 `int` 小”（即 `char`、`byte` 或者 `short`），那么在运算之前，这些值会自动转换成 `int`。这样一来，最终生成的结果就是 `int` 类型。如果你想要把结果赋值给较小的类型，就必须使用类型转换（如果要把结果赋回给较小的类型，就可能出现信息丢失）。通常，表达式中出现的最大的数据类型决定了表达式最终结果的数据类型。如果将一个 `float` 值与一个 `double` 值相乘，结果就是 `double`；如果将一个 `int` 和一个 `long` 值相加，则结果为 `long`。

Java 没有 “sizeof”

在 C 和 C++ 中，`sizeof()` 操作符能满足我们的一项特殊需要：获取为数据存储分配的字节数。在 C 和 C++ 中，最需要使用 `sizeof()` 的原因是为了“移植”。不同的数据类型在不同的机器上可能有不同的大小，所以在进行一些与存储空间有关的运算时，程序员必须清楚那些类型具体有多大。例如，一台计算机可用 32 位来保存整数，而另一台只用 16 位保存。显然，在第一台机器中，程序可保存更大的值。您可以想象，移植是令 C 和 C++ 程序员颇为头痛的一个问题。

Java 不需要 `sizeof()` 操作符来满足这方面的需要，因为所有数据类型在所有机器中的大小都是相同的。我们不必考虑移植问题——它已经被设计在语言中了。

再论优先级 (precedence)

在我举办的一次培训班中，有人抱怨操作符的优先级太难记了。一名学生推荐用一句话来帮助记忆：“Ulcer Addicts Really Like C A lot”（肠疡患者是 C 程序员的写照）。

助记词	操作符类型	操作符
Ulcer	Unary	+ - ++ - [[rest...]]
Addicts	Arithmetic (and shift)	* / % + - << >>
Really	Relational	> < >= <= == !=
Like	Logical (and bitwise)	&& & ^
C	Conditional (ternary)	A > B ? X : Y
A Lot	Assignment	= (and compound assignment like *=)

当然，对于移位和位操作符，上表并不是完美的助记方法；但对于其他运算来说，它确实很管用。

操作符总结

下面这个例子向大家展示了哪些基本数据类型能进行哪些特定的运算。基本上这是一个不断重复的程序，只是每次使用了不同的基本数据类型。文件编译时不会报错，因为那些会导致错误的行已用`//!`注释掉了。

```
//: c03:AllOps.java
// Tests all the operators on all the primitive data types
// to show which ones are accepted by the Java compiler.

public class AllOps {
    // To accept the results of a boolean test:
    void f(boolean b) {}
    void boolTest(boolean x, boolean y) {
        // Arithmetic operators:
        //! x = x * y;
        //! x = x / y;
        //! x = x % y;
        //! x = x + y;
        //! x = x - y;
        //! x++;
        //! x--;
        //! x = +y;
        //! x = -y;
        // Relational and logical:
        //! f(x > y);
        //! f(x >= y);
        //! f(x < y);
        //! f(x <= y);
        f(x == y);
        f(x != y);
        f(!y);
        x = x && y;
        x = x || y;
        // Bitwise operators:
        //! x = ~y;
        x = x & y;
        x = x | y;
        x = x ^ y;
        //! x = x << 1;
        //! x = x >> 1;
        //! x = x >>> 1;
        // Compound assignment:
        //! x += y;
```

```

    //! x -= y;
    //! x *= y;
    //! x /= y;
    //! x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! char c = (char)x;
    //! byte B = (byte)x;
    //! short s = (short)x;
    //! int i = (int)x;
    //! long l = (long)x;
    //! float f = (float)x;
    //! double d = (double)x;
}
void charTest(char x, char y) {
    // Arithmetic operators:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char)+y;
    x = (char)-y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = (char)~y;
    x = (char)(x & y);
    x = (char)(x | y);

```

```

x = (char)(x ^ y);
x = (char)(x << 1);
x = (char)(x >> 1);
x = (char)(x >>> 1);
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean b = (boolean)x;
byte B = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void byteTest(byte x, byte y) {
    // Arithmetic operators:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte)+ y;
    x = (byte)- y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);

```

```

    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
    x = (byte)(x >> 1);
    x = (byte)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
void shortTest(short x, short y) {
    // Arithmetic operators:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Relational and logical:
    f(x > y);

```

```

f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Bitwise operators:
x = (short)~y;
x = (short)(x & y);
x = (short)(x | y);
x = (short)(x ^ y);
x = (short)(x << 1);
x = (short)(x >> 1);
x = (short)(x >>> 1);
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void intTest(int x, int y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;

```



```

x++;
x--;
x = +y;
x = -y;
// Relational and logical:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Bitwise operators:
x = ~y;
x = x & y;
x = x | y;
x = x ^ y;
x = x << 1;
x = x >> 1;
x = x >>> 1;
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
short s = (short)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void longTest(long x, long y) {

```

```

// Arithmetic operators:
x = x * y;
x = x / y;
x = x % y;
x = x + y;
x = x - y;
x++;
x--;
x = +y;
x = -y;
// Relational and logical:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Bitwise operators:
x = ~y;
x = x & y;
x = x | y;
x = x ^ y;
x = x << 1;
x = x >> 1;
x = x >>> 1;
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;

```

```

    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
}
void floatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;

```

```

    //! x ^= y;
    //! x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
}
void doubleTest(double x, double y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;

```

```

x /= y;
x %= y;
//! x <= 1;
//! x >= 1;
//! x >>= 1;
//! x &= y;
//! x ^= y;
//! x |= y;
// Casting:
//! boolean b = (boolean)x;
char c = (char)x;
byte B = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
}
} ///:~

```

注意，能够对作用于布尔型值的运算非常有限。我们只能赋予它 `true` 和 `false` 值，并测试它为真还是为假，而不能将布尔值相加，或对布尔值进行其他任何运算。

在 `char`、`byte` 和 `short` 中，我们可看到算术运算中数据类型提升的效果。对这些类型的任何一个进行算术运算，都会获得一个 `int` 结果。必须将其显式地类型转换回原来的类型（窄化转换可能会造成信息的丢失），以将值赋给原本的类型。但对于 `int` 值，却不必进行类型转化，因为所有数据都已经属于 `int` 类型。但不要放松警惕，认为一切事情都是安全的。如果对两个足够大的 `int` 值执行乘法运算，结果就会溢出。下面这个例子向大家展示了这一点：

```

//: c03:Overflow.java
// Surprise! Java lets you overflow.
import com.bruceeckel.simpletest.*;

public class Overflow {
    static Test monitor = new Test();
    public static void main(String[] args) {
        int big = 0x7fffffff; // max int value
        System.out.println("big = " + big);
        int bigger = big * 4;
        System.out.println("bigger = " + bigger);
        monitor.expect(new String[] {
            "big = 2147483647",
            "bigger = -4"
        });
    }
}

```

```
} ///:~
```

你不会从编译器那里收到出错或警告信息，运行时也不会出现异常。这说明 Java 确实是好东西，却也没有“那么”好！

对于 `char`、`byte` 或者 `short`，混合赋值并不需要类型转换。即使它们执行类型提升，也会获得与直接算术运算相同的结果。而在另一方面，省略类型转换可使代码更加简练。

可以看到，除 `boolean` 以外，任何一种基本类型都可通过类型转换变为其他基本类型。再一次提醒您，当类型转换成一种较小的类型时，必须留意“窄化转换”的结果。否则会在类型转化过程中不知不觉地丢失了信息。

流程控制

Java 使用了 C 的所有流程控制语句，所以如果你以前用过 C 或 C++ 编程，那么你应该非常熟悉了。大多数过程型编程语言都具有某些形式的控制语句，它们通常在各种语言间是共通的。在 Java 中，涉及的关键字包括 `if-else`、`while`、`do-while`、`for` 以及选择语句 `switch`。然而，Java 并不支持被过度诽谤的 `goto` 语句（它仍是解决某些特殊问题的最便利的方法）。在 Java 中，你仍然可以进行类似 `goto` 那样的跳转，但比起典型的 `goto`，有很多的限制。

true 和 false

所有条件语句都利用条件表达式的真或假来决定执行流程。这里有一个条件表达式的例子：`A==B`。它用条件操作符“`==`”来判断 A 值是否等于 B 值。该表达式返回 `true` 或 `false`。在本章前面接触到的所有关系操作符都可拿来构造一个条件语句。注意 Java 不允许我们将一个数字作为布尔值使用，虽然这在 C 和 C++ 里是允许的（真是非零，而假是零）。如果想在布尔测试中使用一个非布尔值，比如在 `if(a)` 中，那么首先必须用一个条件表达式将其转换成布尔值，例如 `if(a!=0)`。

if-else

`if-else` 语句或许是控制程序流程最基本的形式。其中的 `else` 是可选的，所以可按下述两种形式来使用 `if`：

```
if (Boolean-expression)
    statement
```

或

```

if(Boolean-expression)
    statement
else
    statement

```

条件表达式必须产生一个布尔结果，`statement` 指用分号结尾的一个简单语句，或一个复合语句——封闭在花括号内的一组简单语句。在本书任何地方，只要提及“语句”这个词，就可能是简单语句或复合语句。

作为 if-else 的一个例子，下面这个 `test()` 方法可以告诉你，你猜的数是大于，小于还是等于目标数：

```

//: c03:IfElse.java
import com.bruceeckel.simpletest.*;

public class IfElse {
    static Test monitor = new Test();
    static int test(int testval, int target) {
        int result = 0;
        if(testval > target)
            result = +1;
        else if(testval < target)
            result = -1;
        else
            result = 0; // Match
        return result;
    }
    public static void main(String[] args) {
        System.out.println(test(10, 5));
        System.out.println(test(5, 10));
        System.out.println(test(5, 5));
        monitor.expect(new String[] {
            "1",
            "-1",
            "0"
        });
    }
} ///:~

```

最好将流程控制语句缩进排列，使读者能方便地确定起始与终止。

return

`return` 关键字有两方面的用途：指定一个方法返回什么值（假设它没有 `void` 返回值），并立即返回那个值。可据此改写上面的 `test()` 方法，使其利用这些特点：

```
//: c03:IfElse2.java
import com.bruceeckel.simpletest.*;

public class IfElse2 {
    static Test monitor = new Test();
    static int test(int testval, int target) {
        if(testval > target)
            return +1;
        else if(testval < target)
            return -1;
        else
            return 0; // Match
    }
    public static void main(String[] args) {
        System.out.println(test(10, 5));
        System.out.println(test(5, 10));
        System.out.println(test(5, 5));
        monitor.expect(new String[] {
            "1",
            "-1",
            "0"
        });
    }
} ///:~
```

不必加上 `else`，因为方法在执行了 `return` 后不再继续执行。

迭代（Iteration）

`while`，`do-while` 和 `for` 用来控制循环，有时将它们划分为“迭代语句”。语句会重复执行，除非用于控制的布尔表达式得到“假”的结果。`while` 循环的格式如下：

```
while(Boolean-expression)
    statement
```

在循环刚开始时，会计算一次“布尔表达式”的值。而在语句的下一次迭代开始前也要重新计算一次。

下面这个简单的例子可产生随机数，直到符合特定的条件为止：


```

//: c03:WhileTest.java
// Demonstrates the while loop.
import com.bruceeckel.simpletest.*;

public class WhileTest {
    static Test monitor = new Test();
    public static void main(String[] args) {
        double r = 0;
        while(r < 0.99d) {
            r = Math.random();
            System.out.println(r);
            monitor.expect(new String[] {
                "%d\\d+\\.\\d+E?-?\\d*"
            }, Test.AT_LEAST);
        }
    }
} ///:~

```

上面的例子用到了 `Math` 库里的 `static`（静态）方法 `random()`。该方法的作用是产生 0 和 1 之间（包括 0，但不包括 1）的一个 `double` 值。`while` 的条件表达式意思是说：“一直循环下去，直到数字等于或大于 0.99”。由于它的随机性，使得每运行一次这个程序，都会获得大小不同的一系列数字。

在 `expect()` 语句中，你可以看到 `Test.AT_LEAST` 标志跟随在所期望的字符串后面。`expect()` 语句可以包含多个不同的标志来修改它的行为；这里出现的这个标志表明：`expect()` 方法至少应该能够看到后面所展示的行，但是也可以出现其它行（它将忽略这些行）。这里，它表明“你至少应该看到一个双精度的数值。”

do-while

`do-while` 的格式如下：

```

do
    statement
while(Boolean-expression);

```

`while` 和 `do-while` 唯一的区别就是 `do-while` 中的语句至少会执行一次，即便表达式第一次就被计算为 `false`。而在 `while` 循环结构中，如果条件第一次就为 `false`，那么其中的语句根本不会执行。在实际应用中，`while` 比 `do-while` 更常用一些。

for

`for` 循环在第一次迭代之前要进行初始化。随后，它会进行条件测试，而且在每一次迭代结

束时，进行某种形式的“步进（Stepping）”。for 循环的形式如下：

```
for(initialization; Boolean-expression; step)
    statement
```

初始表达式，布尔表达式，或者步进运算，都可以为空。每次迭代前会测试布尔表达式。若获得的结果是 `false`，就会执行在 `for` 语句后面的代码。每次迭代的末尾，会做一次步进运算。

`for` 循环通常用于执行“计数”任务：

```
//: c03:ListCharacters.java
// Demonstrates "for" loop by listing
// all the lowercase ASCII letters.
import com.bruceeckel.simpletest.*;

public class ListCharacters {
    static Test monitor = new Test();
    public static void main(String[] args) {
        for(int i = 0; i < 128; i++)
            if(Character.isLowerCase((char)i))
                System.out.println("value: " + i +
                    " character: " + (char)i);
        monitor.expect(new String[] {
            "value: 97 character: a",
            "value: 98 character: b",
            "value: 99 character: c",
            "value: 100 character: d",
            "value: 101 character: e",
            "value: 102 character: f",
            "value: 103 character: g",
            "value: 104 character: h",
            "value: 105 character: i",
            "value: 106 character: j",
            "value: 107 character: k",
            "value: 108 character: l",
            "value: 109 character: m",
            "value: 110 character: n",
            "value: 111 character: o",
            "value: 112 character: p",
            "value: 113 character: q",
            "value: 114 character: r",
            "value: 115 character: s",
            "value: 116 character: t",
            "value: 117 character: u",
```

```

        "value: 118 character: v",
        "value: 119 character: w",
        "value: 120 character: x",
        "value: 121 character: y",
        "value: 122 character: z"
    });
}
} ///:~

```

注意，变量 `i` 是在程序用到它的地方被定义的，也就是在 `for` 循环的控制表达式里面。而不是在由花括号划分的块开始的地方定义的。`i` 的作用域就是 `for` 控制的表达式的范围内。

这个程序也使用了 `java.lang.Character` 包装器 (wrapper) 类，这个类不但能把 `char` 型的值包装进对象，还提供了一些别的有用的方法。这里用到了 `static isLowerCase()` 方法来检查是否小写字母。

对于象 C 那样的传统的过程型语言，要求所有变量都在一个块的开头定义。以便编译器在创建这个块的时候，可以为那些变量分配空间。而在 Java 和 C++ 中，则可在整个块的范围内分散变量声明，在真正需要的地方才加以定义。这样便可形成更自然的编码风格，也更易理解。

可以在一个 `for` 语句中定义多个变量，但它们必须具有同样的类型：

```

for(int i = 0, j = 1; i < 10 && j != 11; i++, j++)
    // body of for loop

```

其中，`for` 语句内的 `int` 定义同时覆盖了 `i` 和 `j`。只有 `for` 循环才具备在控制表达式里定义变量的能力。对于其他任何条件或循环语句，都不可采用这种方法。

逗号操作符

本章前面已经提到了逗号操作符 (注意不是逗号分隔符；后者用于分隔函数的不同自变量) Java 里唯一用到逗号操作符的地方就是 `for` 循环的控制表达式。在控制表达式的初始化和步进控制部分，我们可使用一系列由逗号分隔的语句。而且那些语句均会独立执行。前面的例子已运用了这种能力，下面则是另一个例子：

```

//: c03:CommaOperator.java
import com.bruceeckel.simpletest.*;

public class CommaOperator {
    static Test monitor = new Test();
    public static void main(String[] args) {
        for(int i = 1, j = i + 10; i < 5;
            i++, j = i * 2) {

```

```

        System.out.println("i= " + i + " j= " + j);
    }
    monitor.expect(new String[] {
        "i= 1 j= 11",
        "i= 2 j= 4",
        "i= 3 j= 6",
        "i= 4 j= 8"
    });
}
} ///:~

```

你可以看到，无论在初始化还是在步进部分，语句都是顺序执行的。此外，尽管初始化部分可设置任意数量的定义，但都属于同一类型。

break 和 continue

在任何循环语句的主体部分，都可用 `break` 和 `continue` 控制循环的流程。其中，`break` 用于强行退出循环，不执行循环中剩余的语句。而 `continue` 则停止执行当前的迭代，然后退回循环起始处，开始下一次迭代。

下面这个程序向大家展示了 `break` 和 `continue` 在 `for` 和 `while` 循环中的例子：

```

//: c03:BreakAndContinue.java
// Demonstrates break and continue keywords.
import com.bruceeckel.simpletest.*;

public class BreakAndContinue {
    static Test monitor = new Test();
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            if(i == 74) break; // Out of for loop
            if(i % 9 != 0) continue; // Next iteration
            System.out.println(i);
        }
        int i = 0;
        // An "infinite loop":
        while(true) {
            i++;
            int j = i * 27;
            if(j == 1269) break; // Out of loop
            if(i % 10 != 0) continue; // Top of loop
            System.out.println(i);
        }
    }
}

```

```

        monitor.expect(new String[] {
            "0",
            "9",
            "18",
            "27",
            "36",
            "45",
            "54",
            "63",
            "72",
            "10",
            "20",
            "30",
            "40"
        });
    }
} ///:~

```

在这个 `for` 循环中，`i` 的值永远不会到达 100。因为一旦 `i` 到达 74，`break` 语句就会中断循环。通常，只有在不知道中断条件何时满足时，才需要这样使用 `break`。只要 `i` 不能被 9 整除，`continue` 语句就会使程序流程返回循环的最开头（这使 `i` 值递增）。如果能够整除，则将值显示出来。

第二部分向大家展示了一个“无穷循环”的情况。然而，循环内部有一个 `break` 语句，可中止循环。除此以外，大家还会看到 `continue` 移回循环顶部，而不完成剩余的内容（所以只有在 `i` 值能被 10 整除时才打印出值）。输出结果之所以显示 0，是由于 `0%9` 等于 0。

无穷循环的第二种形式是 `for(;;)`。编译器将 `while(true)` 与 `for(;;)` 看作同一回事。所以具体选用哪个取决于自己的编程习惯。

臭名昭著的“goto”

在编程语言中很早就有 `goto` 关键字了。事实上，`goto` 起源于汇编语言的程序控制：“若条件 A 成立，则跳到这里；否则跳到那里”。如果阅读由编译器生成的最终的汇编代码，就会发现程序控制里包含了许多跳转。（Java 编译器生成它自己的“汇编代码”，但是这个代码是运行在 Java 虚拟机上的，而不是直接运行在 CPU 硬件上。）

`goto` 语句是在源码级上的跳转，这使其招致了不好的声誉。若一个程序总是从一个地方跳到另一个地方，还有什么办法能识别程序的控制流程呢？随着 Edsger Dijkstra 著名的《Goto considered harmful》论文的出版，众人开始痛斥 `goto` 的不是，甚至建议从关键字集合中扫地出门。

这种情况下，通常中庸之道是最好的。真正的问题并不在于使用 `goto`，而在于 `goto` 的滥用。而且在一些少见的情况下，`goto` 是组织控制流程的最佳手段。

尽管 `goto` 仍是 Java 的一个保留字，但并未在语言中得到正式的使用；Java 没有 `goto`。然而，在 `break` 和 `continue` 这两个关键字的身上，我们仍然能看出一些 `goto` 的影子。它并不属于一次跳转，而是中断循环语句的一种方法。之所以把它们纳入 `goto` 问题中一起讨论，是由于它们使用了相同的机制：标签（`label`）。

“标签”是后面跟一个冒号的标识符，就象下面这样：

`label1:`

在 Java 中，标签起作用的唯一的地方是在迭代语句之前。在标签和迭代之间置入任何语句都是不明智的。而在迭代之前设置标签的唯一理由是：我们希望在其中嵌套另一个循环或者一个开关。这是由于 `break` 和 `continue` 关键字通常只中断当前循环，但若随同标签使用，它们就会中断所有进行中的循环，转到标签所在的地方：

```
label1:
outer-iteration {
    inner-iteration {
        //...
        break; // 1
        //...
        continue; // 2
        //...
        continue label1; // 3
        //...
        break label1; // 4
    }
}
```

在状况 1 处，`break` 中断内部循环，回到外部循环。在状况 2 处，`continue` 移回内部循环的起始处。状况 3 处，`continue label1` 同时中断内部循环以及外部循环，直接转到 `label1` 处。随后，它实际是继续循环，但却从外部循环重新开始。在状况 4 处，`break label1` 也会中断所有循环，并回到 `label1` 处，但并不重新进入循环。也就是说，它实际是完全中止了两个循环。

下面是 Label 用于 `for` 循环的一个例子：

```
//: c03:LabeledFor.java
// Java's "labeled for" loop.
import com.bruceeckel.simpletest.*;

public class LabeledFor {
    static Test monitor = new Test();
    public static void main(String[] args) {
```

```

int i = 0;
outer: // Can't have statements here
for(;; true ;) { // infinite loop
    inner: // Can't have statements here
    for(; i < 10; i++) {
        System.out.println("i = " + i);
        if(i == 2) {
            System.out.println("continue");
            continue;
        }
        if(i == 3) {
            System.out.println("break");
            i++; // Otherwise i never
                // gets incremented.
            break;
        }
        if(i == 7) {
            System.out.println("continue outer");
            i++; // Otherwise i never
                // gets incremented.
            continue outer;
        }
        if(i == 8) {
            System.out.println("break outer");
            break outer;
        }
        for(int k = 0; k < 5; k++) {
            if(k == 3) {
                System.out.println("continue inner");
                continue inner;
            }
        }
    }
}

// Can't break or continue to labels here
monitor.expect(new String[] {
    "i = 0",
    "continue inner",
    "i = 1",
    "continue inner",
    "i = 2",
    "continue",
    "i = 3",
    "break",

```

```

        "i = 4",
        "continue inner",
        "i = 5",
        "continue inner",
        "i = 6",
        "continue inner",
        "i = 7",
        "continue outer",
        "i = 8",
        "break outer"
    });
}
} ///:~

```

注意，`break` 会中断 `for` 循环，而且在抵达 `for` 循环的末尾之前，递增表达式不会执行。由于 `break` 跳过了递增表达式，作为弥补，在 `i==3` 的情况下，我们直接对 `i` 执行递增运算加 1。在 `i==7` 的情况下，`continue outer` 语句会跳到循环顶部，而且也会跳过递增，所以这里也对 `i` 直接加 1 递增。

如果没有 `break outer` 语句，就没有办法从一个内部循环里找到跳出外部循环的路径。这是由于 `break` 本身只能中断最内层的循环（对于 `continue` 同样如此）。

当然，如果想在中断循环的同时退出方法，简单地用一个 `return` 即可。

下面这个例子向大家展示了带标签的 `break` 以及 `continue` 语句在 `while` 循环中的用法：

```

//: c03:LabeledWhile.java
// Java's "labeled while" loop.
import com.bruceeckel.simpletest.*;

public class LabeledWhile {
    static Test monitor = new Test();
    public static void main(String[] args) {
        int i = 0;
    outer:
        while(true) {
            System.out.println("Outer while loop");
            while(true) {
                i++;
                System.out.println("i = " + i);
                if(i == 1) {
                    System.out.println("continue");
                    continue;
                }
            }
        }
    }
}

```



```

        if(i == 3) {
            System.out.println("continue outer");
            continue outer;
        }
        if(i == 5) {
            System.out.println("break");
            break;
        }
        if(i == 7) {
            System.out.println("break outer");
            break outer;
        }
    }
}

monitor.expect(new String[] {
    "Outer while loop",
    "i = 1",
    "continue",
    "i = 2",
    "i = 3",
    "continue outer",
    "Outer while loop",
    "i = 4",
    "i = 5",
    "break",
    "Outer while loop",
    "i = 6",
    "i = 7",
    "break outer"
});
}
} ///:~

```

同样的规则亦适用于 `while`:

- (1) 一般的 `continue` 会退回最内层循环的开头（顶部），并继续执行。
- (2) 带标签的 `continue` 会到达标签的位置，并重新进入紧接在那个标签后面的循环。
- (3) 一般的 `break` 会中断并跳出当前循环。
- (4) 带标签的 `break` 会中断并跳出标签所指的循环。

大家要记住的重点是：在 Java 里需要使用标签的唯一理由就是有循环嵌套，而且你想从不止一个嵌套中 `break` 或 `continue`。

在 Dijkstra 的“Goto 有害”的论文中，他最反对的就是标签，而非 `goto`。他发现在一个程序里

随着标签的增多，产生的错误也越来越多。由于标签和 `goto` 在程序执行流程图中引入了循环 (cycles)，使我们难以对程序作静态分析。但是，Java 的标签不会造成这种问题，因为它们的应用场合已经受到限制了，不能用什么特别的方式改变程序的控制流程。由此也引出了一个有趣的问题：通过限制语句的能力，反而能使一项语言特性更加有用。

开关 (switch)

`switch` 有时也被划归为一种“选择语句”。根据一个整数表达式的值，`switch` 语句可以从一系列代码中选出一段去执行。它的格式如下：

```
switch(integral-selector) {
    case integral-value1 : statement; break;
    case integral-value2 : statement; break;
    case integral-value3 : statement; break;
    case integral-value4 : statement; break;
    case integral-value5 : statement; break;
    // ...
    default: statement;
}
```

其中，“整数选择因子 (Integral-selector)”是一个能够产生整数值的表达式。`switch` 能将整数选择因子的结果与每个整数值 (integral-value) 相比较。若发现相符的，就执行对应的语句 (简单或复合语句)。若没有发现相符的，就执行 `default` 语句。

在上面的定义中，大家会注意到每个 `case` 均以 `break` 结尾。这样可使执行流程跳转至 `switch` 主体的末尾。这是构建 `switch` 语句的一种传统方式，但 `break` 是可选的。若省略 `break`，会继续执行后面的 `case` 语句，直到遇到一个 `break` 为止。尽管通常不想出现这种情况，但对有经验的程序员来说，也许能够善加利用。注意最后的 `default` 语句没有 `break`，因为执行流程已到了 `break` 的跳转目的地。当然，如果考虑到编程风格方面的原因，完全可以在 `default` 语句的末尾放置一个 `break`，尽管它并没有任何实际的用处。

`switch` 语句是实现多路选择 (也就是说从一系列执行路径中挑选一个) 的一种干净利落的方法。但它要求使用一个选择因子，并且必须是 `int` 或 `char` 那样的整数值。例如，假若将一个字符串或者浮点数作为选择因子使用，那么它们在 `switch` 语句里是不会工作的。对于非整数类型，则必须使用一系列 `if` 语句。

下面这个例子可随机生成字母，并判断它们是元音还是辅音字母：

```
//: c03:VowelsAndConsonants.java
// Demonstrates the switch statement.
import com.bruceeckel.simpletest.*;

public class VowelsAndConsonants {
```

```

static Test monitor = new Test();
public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
        char c = (char) (Math.random() * 26 + 'a');
        System.out.print(c + ": ");
        switch(c) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u': System.out.println("vowel");
                       break;
            case 'y':
            case 'w': System.out.println("Sometimes a vowel");
                       break;
            default: System.out.println("consonant");
        }
        monitor.expect(new String[] {
            "% [aeiou]: vowel| [yw]: Sometimes a vowel| " +
            "[^aeiouyw]: consonant"
        }, Test.AT_LEAST);
    }
}
} ///:~

```

由于 `Math.random()` 会产生 0 到 1 之间的一个值，所以只需将其乘以想获得的数字范围的上界（对于英语字母，这个数字是 26），再加上作为偏移量的数字范围下界，就可以得到合适的随机数。

尽管我们在这儿表面上要处理的是字符，但 `switch` 语句实际使用的是字符的整数值。在 `case` 语句中，使用单引号引用的字符也会产生整数值，用来进行比较。

请注意 `case` 语句能够堆叠在一起，为一段代码形成多重匹配，即只要符合多种条件中的一种，就执行那段特别的代码。这时也应注意将 `break` 语句置于特定 `case` 的末尾，否则控制流程会简单地移下，处理后面的 `case`。

在 `expect()` 里的正则表达式用 ‘|’ 划分成了三种可能。正则表达式中的 ‘[]’，表示的是一个字符‘集’，所以第一种可能是“a, e, i, o, u 中的一个，后跟一个冒号和单词 ‘vowel’”；第二种可能是 y 或者 w，然后是 “Sometimes a vowel.”；第三种可能是以一个 ‘^’ 开始，意指“不是字符集中的字符”，所以它说明的是所有非元音的匹配字符。

计算细节

仔细观察下面这个语句：

```
char c = (char)(Math.random() * 26 + 'a');
```

`Math.random()` 会产生一个 `double` 值，所以 26 会转换成 `double` 类型，以便执行乘法运算。这个运算也会产生一个 `double` 值。这意味着为了执行加法，必须将 `'a'` 转换成一个 `double`。最后使用显式的类型转换将 `double` 结果转换回 `char`。

我们的第一个问题是，类型转换会对 `char` 作什么样的处理呢？换言之，假设一个值是 29.7，我们把它造型成一个 `char`，那么结果值到底是 30 还是 29 呢？答案可从下面这个例子中得到：

```
//: c03:CastingNumbers.java
// What happens when you cast a float
// or double to an integral value?
import com.bruceeckel.simpletest.*;

public class CastingNumbers {
    static Test monitor = new Test();
    public static void main(String[] args) {
        double
            above = 0.7,
            below = 0.4;
        System.out.println("above: " + above);
        System.out.println("below: " + below);
        System.out.println("(int)above: " + (int)above);
        System.out.println("(int)below: " + (int)below);
        System.out.println("(char)('a' + above): " +
            (char)('a' + above));
        System.out.println("(char)('a' + below): " +
            (char)('a' + below));
        monitor.expect(new String[] {
            "above: 0.7",
            "below: 0.4",
            "(int)above: 0",
            "(int)below: 0",
            "(char)('a' + above): a",
            "(char)('a' + below): a"
        });
    }
} ///:~
```

所以答案就是：将一个 `float` 或 `double` 值转型成整数值后，总是将小数部分“砍掉”（而不是四舍五入）。

第二个问题与 `Math.random()` 有关。它会产生 0 和 1 之间的值，但是否包括值 1 呢？用正

规的数学语言表达，它到底是(0,1)、[0,1]、(0,1)，还是[0,1)呢（方括号表示“包括”，圆括号表示“不包括”）？同样地，一个示范程序向我们揭示了答案：

```
//: c03:RandomBounds.java
// Does Math.random() produce 0.0 and 1.0?
// {RunByHand}

public class RandomBounds {
    static void usage() {
        System.out.println("Usage: \n\t" +
            "RandomBounds lower\n\tRandomBounds upper");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length != 1) usage();
        if(args[0].equals("lower")) {
            while(Math.random() != 0.0)
                ; // Keep trying
            System.out.println("Produced 0.0!");
        }
        else if(args[0].equals("upper")) {
            while(Math.random() != 1.0)
                ; // Keep trying
            System.out.println("Produced 1.0!");
        }
        else
            usage();
    }
} ///:~
```

为运行这个程序，只需在命令行键入下述命令即可：

```
java RandomBounds lower
或
java RandomBounds upper
```

在这两种情况下，我们都必须人工中断程序，所以会发现`Math.random()`“似乎”永远都不会产生 0.0 或 1.0。但这正是实验可能欺骗我们的地方。要知道 0 和 1 之间有 2^{62} 个不同的双精度小数²，如果产生全部的数字，花费的时间会超过一台电脑，甚至做此试验的人的

² Chuck Allison 提到：在一个浮点数系统中，可表达的数的总数量是：

$$2(M-m+1)b^{(p-1)} + 1$$

其中 b 是基数（通常是 2）， p 是精度（尾数中的位数）， M 是指数的最大值， m 是指数的最小值。IEEE754 的规范是：

$$M = 1023, m = -1022, p = 53, b = 2$$

所以能够表示的数的总个数有：

$$\begin{aligned} & 2(1023+1022+1)2^{52} \\ & = 2((2^{10}-1) + (2^{10}-1))2^{52} \end{aligned}$$

寿命。事实是，在`Math.random()`的输出中包括了 0.0，用数学语言，输出值范围是 $[0,1)$ 。

总结

本章总结了大多数编程语言都具有的基本特性：运算、操作符优先级、类型转换以及选择和循环等等。现在，我们已经作好了准备，以使自己更靠近面向对象的程序设计的世界。在下一章里，我们将讨论对象的初始化与清除，再后面则讲述隐藏实现细节（implementation hiding）这一核心概念。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 在本章的“优先级”一节中有两个表达式。把这两个表达式放到一个程序中，验证它们会产生不同的结果。
2. 把 `ternary()` 和 `alternative()` 方法放进一个能运行的程序中。
3. 修改“if-else”和“return”两节中的 `test()` 方法，让它判断 `testval` 是否在参数 `begin` 和 `end` 之间（闭集）。
4. 写一个打印从 1 到 100 的值的程序。
5. 用 `break` 语句修改练习 4 的程序，让它在 47 的时候退出。再用 `return` 来代替 `break`。
6. 写一个接收两个字符串参数的方法，用各种布尔值的比较关系来比较这两个字符串，然后把结果打印出来。做 `==` 和 `!=` 比较的同时，用 `equals()` 作测试。在 `main()` 里面用几个不同的字符串对象调用这个方法。
7. 写一个会随机生成 25 个整型值的程序。对每一个值，用 `if-else` 语句判断其是大于、小于，还是等于下一个随机生成的值。
8. 修改练习 7 的程序，用“无穷循环”的 `while` 语句来包裹这段代码。这样它就能不停的执行，直到你敲键盘来中断程序了（通常是 `Ctrl+C`）。
9. 写一个嵌套了两层 `for` 循环的程序。用取模操作符 `(%)` 来检测质数，并把它打印出来。（所谓质数是指只能被 1 和它自己整除的自然数）。
10. 写一个 `switch` 开关语句，为每个 `case` 打印一个消息。然后把这个 `switch` 放进 `for` 循环来测试每个 `case`。先测试每个 `case` 后面都有 `break` 的程序，然后把 `break` 删了，看看会怎样。

$$\begin{aligned} &= (2^{10}-1)2^{54} \\ &= 2^{64} - 2^{54} \end{aligned}$$

其中一半的数（指数在 $[-1022, 0]$ 范围内）小于 1（包括正数和负数）。所以上面表达式的 $1/4$ ，也就是有 $2^{62} - 2^{52} + 1$ 个数（接近 2^{62} ）属于 $[0,1)$ 。可以参考下面的网址 <http://www.freshsources.com/1995006a.htm>（此文档的最后部分）。

第四章 初始化与清除

随着计算机革命的发展，“不安全”的编程方式已逐渐成为编程代价高昂的主因之一。

“初始化 (initialization)”和“清除 (cleanup)”正是涉及安全的两个问题。许多 C 程序的错误都源于程序员忘记初始化变量。特别是在使用程序库时，如果用户不知道如何初始化库的构件，或者是用户必须初始化的其它东西，更是如此。清理也是个特殊的问题，当你使用完一个元素时，它对你也就不会有什么影响了，所以你很容易把它忘记。这样一来，这个元素占用的资源就会一直得不到释放，等待你的将是资源（尤其是内存）用尽的后果。

C++引入了“构造器 (constructor)”的概念。这是一个在创建对象时被自动调用的特殊方法。Java 中也采用了构造器，并额外提供了“垃圾回收器”。对于不再使用的内存资源，垃圾回收器能自动将其释放。本章将讨论初始化和清理的相关问题，以及 Java 对它们提供的支持。

以构造器确保初始化

可以假想为编写的每个类都定义一个 `initialize()` 方法。此名称提醒你在使用其对象之前，应首先调用 `initialize()`。然而，这同时意味着用户必须记得自己去调用此方法。在 Java 中，通过提供“构造器”这种特殊方法，类的设计者可确保每个对象都会得到初始化。当对象被创建时，如果其类具有构造器，Java 就会在用户有能力操作对象之前自动调用相应的构造器，所以初始化动作得以确保。

接下来的问题就是如何命名这个方法。有两个问题：第一，你取的任何名字都可能与类的某个成员名称相冲突；第二，调用构造器是编译器的责任，所以必须让编译器知道应该调用哪个方法。C++语言中采用的方案看来最简单且更符合逻辑，所以在 Java 中也得到了应用：即构造器采用与类相同的名称。考虑到在初始化期间要自动调用构造器，这种作法就顺理成章了。

以下就是一个带有构造器的简单类：

```
//: c04:SimpleConstructor.java
// Demonstration of a simple constructor.
import com.bruceeckel.simpletest.*;

class Rock {
    Rock() { // This is the constructor
        System.out.println("Creating Rock");
    }
}

public class SimpleConstructor {
    static Test monitor = new Test();
```



```

public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
        new Rock();
    monitor.expect(new String[] {
        "Creating Rock",
        "Creating Rock",
        "Creating Rock",
        "Creating Rock",
        "Creating Rock",
        "Creating Rock",
        "Creating Rock",
        "Creating Rock",
        "Creating Rock",
        "Creating Rock"
    });
}
} ///:~

```

现在，在创建对象时：

```
new Rock();
```

将会为对象分配存储空间，并调用相应的构造器。这就确保了在你能操作对象之前，它已经被恰当地初始化了。

请注意，由于构造器的名称必须与类名完全相同，所以“每个方法首字母小写”的编码风格并不适用于构造器。

和其他方法一样，构造器也能带有形式参数，以便指定对象的具体创建方式。对上述例子稍加修改，即可使构造器接受一个参数：

```

///: c04:SimpleConstructor2.java
/// Constructors can have arguments.
import com.bruceeckel.simpletest.*;

class Rock2 {
    Rock2(int i) {
        System.out.println("Creating Rock number " + i);
    }
}

public class SimpleConstructor2 {
    static Test monitor = new Test();
    public static void main(String[] args) {

```

```

for(int i = 0; i < 10; i++)
    new Rock2(i);
monitor.expect(new String[] {
    "Creating Rock number 0",
    "Creating Rock number 1",
    "Creating Rock number 2",
    "Creating Rock number 3",
    "Creating Rock number 4",
    "Creating Rock number 5",
    "Creating Rock number 6",
    "Creating Rock number 7",
    "Creating Rock number 8",
    "Creating Rock number 9"
});
}
} ///:~

```

有了构造器形式参数，你就可以在初始化对象时提供实际参数。假设类 `Tree` 有一个构造器，它接受一个整型变量来表示树的高度，你就可以这样创建一个 `Tree` 对象：

```
Tree t = new Tree(12); // 12-foot tree
```

如果 `Tree(int)` 是 `Tree` 类中唯一的构造器，那么编译器将不会允许你以其他任何方式创建 `Tree` 对象。

构造器有助于减少错误，并使代码更易于阅读。从概念上讲，“初始化”与“创建”是彼此独立的，然而在上面的代码中，你却找不到对类似 `initialize()` 方法的直接调用。在 `Java` 中，“初始化”和“创建”被捆绑在一起，两者不能分离。

构造器比较特殊，因为它没有返回值。这与返回值为空（`void`）明显不同。对于空返回值，尽管方法本身不会自动返回什么，但你仍可选择让它返回别的东西。构造器则不会返回任何东西，你别无选择（`new` 表达式确实返回了对新建对象的引用，但构造器本身并没有任何返回值）。假如构造器具有返回值，并且允许你自行选择返回类型，那么势必得让编译器知道该如何处理此返回值。

方法重载（method overloading）

任何程序设计语言都具备的一项重要特性就是对名字的运用。当你创建一个对象时，也就给此对象分配到的存储空间取了一个名字。所谓方法则是给某个动作取的名字。使用名字来描述系统，可使程序更易于理解和修改。就好比书写散文——目的是让读者易于理解。

通过使用名字，你可以引用所有的对象和方法。起得好的名字可以使你的代码更易于理解。

将人类语言中存在细微差别的概念“映射”到程序设计语言中时，问题随之而生。在日常生活中，相同的词可以表达多种不同的含义——它们被“重载”了。特别是含义之间的差别很小时，这种方式十分有用。你可以说“清洗衬衫”、“清洗车”以及“清洗狗”。但如果硬要这样说就显得很愚蠢：“以洗衬衫的方式洗衬衫”、“以洗车的方式洗车”以及“以洗狗的方式洗狗”。这是因为听众根本不需要对执行的行动作加以任何明确的区分。大多数人类语言具有很强的“冗余”性，所以你即使漏掉了几个词，仍然可以推断出含义。你不需要为每个概念都使用不同的词汇——从具体的语境中就可以推断出含义。

大多数程序设计语言（尤其是C）要求你为每个函数都提供一个独一无二的标识符。所以绝不能用了一个名为 `print()` 的函数来显示整数之后，又用另一个 `print()` 显示浮点数——每个函数都要有唯一的名称。

在Java（和C++）里，构造器是另一个强制重载方法名的原因。既然构造器的名字已经由类名所决定，就只能有一个构造器名。那么要想用多种方式创建一个对象该怎么办呢？假设你要创建一个类，既可以用标准方式进行初始化，也可以从文件里读取信息来初始化。这就需要两个构造器：一个不带形式参数（即“缺省（default）”构造器¹，也称“无参（no-arg）”构造器），另一个取字符串作为形式参数——该字符串表示初始化对象所需的文件名称。由于都是构造器，所以它们必须有相同的名字，即类名。为了让方法名相同而形式参数不同的构造器同时存在，必须用到“方法重载”。同时，尽管方法重载是构造器所必需的，但它亦可应用于其他方法，且用法同样方便。

下面这个例子同时示范了构造器和普通方法的重载：

```
//: c04:Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Tree {
    int height;
    Tree() {
        System.out.println("Planting a seedling");
        height = 0;
    }
    Tree(int i) {
        System.out.println("Creating new Tree that is "
            + i + " feet tall");
        height = i;
    }
    void info() {
        System.out.println("Tree is " + height + " feet tall");
    }
}
```

¹在一些Sun公司的Java文档中，他们却使用了一个笨拙但很直观的名字“无参数的构造器”，不过“缺省构造器”已使用了多年，所以我还是用这个。

```

    }
    void info(String s) {
        System.out.println(s + ": Tree is "
            + height + " feet tall");
    }
}

public class Overloading {
    static Test monitor = new Test();
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        // Overloaded constructor:
        new Tree();
        monitor.expect(new String[] {
            "Creating new Tree that is 0 feet tall",
            "Tree is 0 feet tall",
            "overloaded method: Tree is 0 feet tall",
            "Creating new Tree that is 1 feet tall",
            "Tree is 1 feet tall",
            "overloaded method: Tree is 1 feet tall",
            "Creating new Tree that is 2 feet tall",
            "Tree is 2 feet tall",
            "overloaded method: Tree is 2 feet tall",
            "Creating new Tree that is 3 feet tall",
            "Tree is 3 feet tall",
            "overloaded method: Tree is 3 feet tall",
            "Creating new Tree that is 4 feet tall",
            "Tree is 4 feet tall",
            "overloaded method: Tree is 4 feet tall",
            "Planting a seedling"
        });
    }
} ///:~

```

创建 `Tree` 对象的时候，既可以不含参数，也可以用树的高度当参数。前者表示一棵树苗，后者表示已有一定高度的树木。要支持这种创建方式，得有一个缺省构造器，和一个采用现有高度作为参数的构造器。

或许你还想通过多种方式调用 `info()` 方法。例如，你想显示额外信息，可以用 `info(String)` 方法；没有的话就用 `info()`。要是明显相同的概念却使用了不同的名字，那一定会让人很

纳闷。好在有了方法重载，你得以为两者使用相同的名字。

区分重载方法

要是有几个方法名字相同，Java 要怎样才能知道你指的哪一个呢？其实规则很简单：每个重载的方法都必须有一个独一无二的参数类型列表。

稍加思考，你就会觉得这是合理的。毕竟，对于名字相同的方法，除了参数类型带来的差异以外，还有什么办法能把它们区别开呢？

甚至形式参数顺序的不同也足以区分两个方法（不过，一般情况下别这么做，因为这会使代码难以维护）：

```
//: c04:Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Tree {
    int height;
    Tree() {
        System.out.println("Planting a seedling");
        height = 0;
    }
    Tree(int i) {
        System.out.println("Creating new Tree that is "
            + i + " feet tall");
        height = i;
    }
    void info() {
        System.out.println("Tree is " + height + " feet tall");
    }
    void info(String s) {
        System.out.println(s + ": Tree is "
            + height + " feet tall");
    }
}

public class Overloading {
    static Test monitor = new Test();
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
```

```

        Tree t = new Tree(i);
        t.info();
        t.info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
    monitor.expect(new String[] {
        "Creating new Tree that is 0 feet tall",
        "Tree is 0 feet tall",
        "overloaded method: Tree is 0 feet tall",
        "Creating new Tree that is 1 feet tall",
        "Tree is 1 feet tall",
        "overloaded method: Tree is 1 feet tall",
        "Creating new Tree that is 2 feet tall",
        "Tree is 2 feet tall",
        "overloaded method: Tree is 2 feet tall",
        "Creating new Tree that is 3 feet tall",
        "Tree is 3 feet tall",
        "overloaded method: Tree is 3 feet tall",
        "Creating new Tree that is 4 feet tall",
        "Tree is 4 feet tall",
        "overloaded method: Tree is 4 feet tall",
        "Planting a seedling"
    });
}
} ///:~

```

上例两个 `print()` 方法虽然声明了相同的参数，但顺序不同，因此得以区分。

涉及基本类型的重载

基本类型能从一个“较小”的类型自动提升至一个“较大”的类型，此过程一旦牵涉到重载，可能会造成一些混淆。以下例子说明了将基本类型传递给重载方法时发生的情况：

```

///: c04:PrimitiveOverloading.java
// Promotion of primitives and overloading.
import com.bruceeckel.simpletest.*;

public class PrimitiveOverloading {
    static Test monitor = new Test();
    void f1(char x) { System.out.println("f1(char)"); }
    void f1(byte x) { System.out.println("f1(byte)"); }
    void f1(short x) { System.out.println("f1(short)"); }
}

```

```

void f1(int x) { System.out.println("f1(int)"); }
void f1(long x) { System.out.println("f1(long)"); }
void f1(float x) { System.out.println("f1(float)"); }
void f1(double x) { System.out.println("f1(double)"); }

void f2(byte x) { System.out.println("f2(byte)"); }
void f2(short x) { System.out.println("f2(short)"); }
void f2(int x) { System.out.println("f2(int)"); }
void f2(long x) { System.out.println("f2(long)"); }
void f2(float x) { System.out.println("f2(float)"); }
void f2(double x) { System.out.println("f2(double)"); }

void f3(short x) { System.out.println("f3(short)"); }
void f3(int x) { System.out.println("f3(int)"); }
void f3(long x) { System.out.println("f3(long)"); }
void f3(float x) { System.out.println("f3(float)"); }
void f3(double x) { System.out.println("f3(double)"); }

void f4(int x) { System.out.println("f4(int)"); }
void f4(long x) { System.out.println("f4(long)"); }
void f4(float x) { System.out.println("f4(float)"); }
void f4(double x) { System.out.println("f4(double)"); }

void f5(long x) { System.out.println("f5(long)"); }
void f5(float x) { System.out.println("f5(float)"); }
void f5(double x) { System.out.println("f5(double)"); }

void f6(float x) { System.out.println("f6(float)"); }
void f6(double x) { System.out.println("f6(double)"); }

void f7(double x) { System.out.println("f7(double)"); }

void testConstVal() {
    System.out.println("Testing with 5");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
}

void testChar() {
    char x = 'x';
    System.out.println("char argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testByte() {
    byte x = 0;
    System.out.println("byte argument:");

```

```

        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testShort() {
        short x = 0;
        System.out.println("short argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testInt() {
        int x = 0;
        System.out.println("int argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testLong() {
        long x = 0;
        System.out.println("long argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testFloat() {
        float x = 0;
        System.out.println("float argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testDouble() {
        double x = 0;
        System.out.println("double argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    public static void main(String[] args) {
        PrimitiveOverloading p =
            new PrimitiveOverloading();
        p.testConstVal();
        p.testChar();
        p.testByte();
        p.testShort();
        p.testInt();
        p.testLong();
        p.testFloat();
        p.testDouble();
        monitor.expect(new String[] {
            "Testing with 5",
            "f1(int)",
            "f2(int)",
            "f3(int)",
            "f4(int)",

```



```
"f5(long)",
"f6(float)",
"f7(double)",
"char argument:",
"f1(char)",
"f2(int)",
"f3(int)",
"f4(int)",
"f5(long)",
"f6(float)",
"f7(double)",
"byte argument:",
"f1(byte)",
"f2(byte)",
"f3(short)",
"f4(int)",
"f5(long)",
"f6(float)",
"f7(double)",
"short argument:",
"f1(short)",
"f2(short)",
"f3(short)",
"f4(int)",
"f5(long)",
"f6(float)",
"f7(double)",
"int argument:",
"f1(int)",
"f2(int)",
"f3(int)",
"f4(int)",
"f5(long)",
"f6(float)",
"f7(double)",
"long argument:",
"f1(long)",
"f2(long)",
"f3(long)",
"f4(long)",
"f5(long)",
"f6(float)",
"f7(double)",
"float argument:",
```

```

        "f1(float)",
        "f2(float)",
        "f3(float)",
        "f4(float)",
        "f5(float)",
        "f6(float)",
        "f7(double)",
        "double argument:",
        "f1(double)",
        "f2(double)",
        "f3(double)",
        "f4(double)",
        "f5(double)",
        "f6(double)",
        "f7(double)"
    });
}
} ///:~

```

通过程序输出，你会发现常数值 5 被当作 `int` 值处理。所以如果有某个重载方法接受 `int` 型参数，它就会被调用。至于其他情况，如果传入的实际参数类型“小于”方法中声明的形式参数类型，实际参数的类型就会被“提升”。`char` 型略有不同，如果无法找到恰好接受 `char` 参数的方法，就会把 `char` 直接提升至 `int` 型。

如果传入的实际参数“大于”重载方法声明的形式参数，会出现什么情况呢？修改上述程序，就能得到答案：

```

///: c04:Demotion.java
// Demotion of primitives and overloading.
import com.bruceeckel.simpletest.*;

public class Demotion {
    static Test monitor = new Test();
    void f1(char x) { System.out.println("f1(char)"); }
    void f1(byte x) { System.out.println("f1(byte)"); }
    void f1(short x) { System.out.println("f1(short)"); }
    void f1(int x) { System.out.println("f1(int)"); }
    void f1(long x) { System.out.println("f1(long)"); }
    void f1(float x) { System.out.println("f1(float)"); }
    void f1(double x) { System.out.println("f1(double)"); }

    void f2(char x) { System.out.println("f2(char)"); }
    void f2(byte x) { System.out.println("f2(byte)"); }
    void f2(short x) { System.out.println("f2(short)"); }
}

```

```

void f2(int x) { System.out.println("f2(int)"); }
void f2(long x) { System.out.println("f2(long)"); }
void f2(float x) { System.out.println("f2(float)"); }

void f3(char x) { System.out.println("f3(char)"); }
void f3(byte x) { System.out.println("f3(byte)"); }
void f3(short x) { System.out.println("f3(short)"); }
void f3(int x) { System.out.println("f3(int)"); }
void f3(long x) { System.out.println("f3(long)"); }

void f4(char x) { System.out.println("f4(char)"); }
void f4(byte x) { System.out.println("f4(byte)"); }
void f4(short x) { System.out.println("f4(short)"); }
void f4(int x) { System.out.println("f4(int)"); }

void f5(char x) { System.out.println("f5(char)"); }
void f5(byte x) { System.out.println("f5(byte)"); }
void f5(short x) { System.out.println("f5(short)"); }

void f6(char x) { System.out.println("f6(char)"); }
void f6(byte x) { System.out.println("f6(byte)"); }

void f7(char x) { System.out.println("f7(char)"); }

void testDouble() {
    double x = 0;
    System.out.println("double argument:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
}

public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
    monitor.expect(new String[] {
        "double argument:",
        "f1(double)",
        "f2(float)",
        "f3(long)",
        "f4(int)",
        "f5(short)",
        "f6(byte)",
        "f7(char)"
    });
}

```

```
} ///:~
```

在这里，方法接受“较小”的基本类型作为参数。如果传入的实际参数“较大”，你就得在圆括号里写上类型名称，做必要的类型转换。如果不这样做，编译器就会报错。

你应该注意到这是一种“窄化转换”，这意味着在类型转换过程中可能会丢失信息。这也正是编译器强制你必须明确进行类型转换的原因。

以返回值区分重载方法

你可能会这么想：“在区分重载方法的时候，为什么只能以类名和方法的形参列表作为标准呢？能否考虑用方法的返回值来区分呢？”比如下面两个方法，虽然它们有同样的名字和形式参数，但你却可以很容易地区分它们：

```
void f() {}  
int f() {}
```

在类似 `int x=f()` 的语境中，如果编译器可根据上下文明确判断出语义，那么的确可以据此区分重载方法。不过，有时你并不关心方法的返回值，你想要的是方法调用的其他效果（常被称为“为了副作用而调用”）。你可能会调用方法而忽略其返回值：

```
f();
```

此时 Java 如何才能判断该调用哪一个 `f()` 呢？别人该如何理解这种代码呢？因此，根据方法的返回值来区分重载方法是行不通的。

缺省构造器

如前所述，缺省构造器（又名“无参”构造器）是没有形式参数的。它的作用是创建一个“基本对象”。如果你写的类中没有构造器，则编译器会自动帮你创建一个缺省构造器。例如：

```
///: c04:DefaultConstructor.java  
  
class Bird {  
    int i;  
}  
  
public class DefaultConstructor {  
    public static void main(String[] args) {  
        Bird nc = new Bird(); // Default!  
    }  
}
```

```
} ///:~
```

其中下面这一行：

```
new Bird();
```

创建了一个新对象，并调用其缺省构造器——即使你没有明确定义它。没有它的话，就无法创建对象。但是，如果你已经定义了一个构造器（无论是否有参数），编译器就不会帮你自动创建缺省构造器：

```
class Hat {  
    Hat(int i) {}  
    Hat(double d) {}  
}
```

现在，要是你这么写：

```
new Hat();
```

编译器就会报错：没有找到匹配的构造器。这就好比，要是你没有提供任何构造器，编译器会认为：“你需要一个构造器，让我给你制造一个吧”，但假如你已写了一个构造器，编译器则会认为：“啊，你已写了一个构造器，所以你知道你在做什么；你是刻意省略了缺省构造器。”

this 关键字

如果有同一类型的两个对象，分别是 **a** 和 **b**。你可能想知道，如何才能分别为这两个对象调用 **f()** 呢：

```
class Banana { void f(int i) { /* ... */ } }  
Banana a = new Banana(), b = new Banana();  
a.f(1);  
b.f(2);
```

如果只有一个 **f()** 方法，它是如何知道被 **a** 还是被 **b** 所调用的呢？

为了能用简便、面向对象的语法来编写代码——即“发送消息给对象”，编译器做了一些幕后工作。它暗自把“所操作对象的引用”作为第一个参数传递给 **f()**。所以上述两个方法的调用就变成了这样：

```
Banana.f(a, 1);  
Banana.f(b, 2);
```

这是内部的表示形式。你并不能这样书写代码，并试图通过编译。但这种写法的确能帮你了解实际发生的事情有所了解。

假设你希望在方法的内部获得对当前对象的引用。由于这个引用是由编译器“偷偷”传入的，所以没有标识符可用。因此，有个可供你使用的专门的关键字：**this**。**this** 关键字只能在方法内部使用，表示对“调用方法的那个对象”的引用。**this** 的用法和其它对象引用并无不同。但要注意，如果在方法内部调用同一个类的方法，就不必使用 **this**，直接调用即可。当前方法中的 **this** 引用会自动应用于同一类中的其他方法。所以你能写这样的代码：

```
class Apricot {
    void pick() { /* ... */ }
    void pit() { pick(); /* ... */ }
}
```

在 `pit()` 内部，你可以写 `this.pick()`，但无此必要²。编译器能帮你自动添加。只有当你需要明确指出当前对象的引用时，才需要使用 **this** 关键字。例如，当需要返回对当前对象的引用时，就常常在 `return` 语句里这么写：

```
//: c04:Leaf.java
// Simple use of the "this" keyword.
import com.bruceeckel.simpletest.*;

public class Leaf {
    static Test monitor = new Test();
    int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() {
        System.out.println("i = " + i);
    }
    public static void main(String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().increment().print();
        monitor.expect(new String[] {
            "i = 3"
        });
    }
} ///:~
```

²有些人执意将 **this** 放在每一个方法调用和字段引用前，认为这样“更清楚更明确”。但是，千万别这么做。我们使用高级语言的原因之一就是它们能帮我们做一些事情。要是你把 **this** 放在一些没必要的地方，就会使读你程序的人不知所措，因为别人写的代码不会到处使用 **this**。遵循一种一致而直观的编程风格能节省时间和金钱。

由于 `increment()` 通过 `this` 关键字返回了对当前对象的引用，所以很容易在一条语句里对同一个对象执行多次操作。

在构造器中调用构造器

如果你为一个类写了多个构造器，有时可能想在一个构造器中调用另一个构造器，以避免重复代码。你可用 `this` 关键字做到这一点。

通常你写 `this` 的时候，都是指“这个对象”或者“当前对象”，而且它本身表示对当前对象的引用。在构造器中，如果为 `this` 添加了参数列表，那么就有了不同的含义：这将产生对符合此参数列表的某个构造器的明确调用。这样，就有了一个直接的途径来调用其它构造器：

```
//: c04:Flower.java
// Calling constructors with "this."
import com.bruceeckel.simpletest.*;

public class Flower {
    static Test monitor = new Test();
    int petalCount = 0;
    String s = new String("null");
    Flower(int petals) {
        petalCount = petals;
        System.out.println(
            "Constructor w/ int arg only, petalCount= "
            + petalCount);
    }
    Flower(String ss) {
        System.out.println(
            "Constructor w/ String arg only, s=" + ss);
        s = ss;
    }
    Flower(String s, int petals) {
        this(petals);
        //!   this(s); // Can't call two!
        this.s = s; // Another use of "this"
        System.out.println("String & int args");
    }
    Flower() {
        this("hi", 47);
        System.out.println("default constructor (no args)");
    }
    void print() {
```

```

    //! this(11); // Not inside non-constructor!
    System.out.println(
        "petalCount = " + petalCount + " s = " + s);
}
public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
    monitor.expect(new String[] {
        "Constructor w/ int arg only, petalCount= 47",
        "String & int args",
        "default constructor (no args)",
        "petalCount = 47 s = hi"
    });
}
} ///~

```

构造器 `Flower(String s,int petals)`表明：尽管你可以用 `this` 调用一个构造器，但你却不能用相同的方法调用两个构造器。此外，你必须将构造器调用置于最起始处，否则编译器会报错。

这个例子也展示了 `this` 的另一种用法。由于参数 `s` 的名称和数据成员 `s` 的名字相同，所以会产生歧义。使用 `this.s` 来引用数据成员就能解决这个问题。在 `Java` 程序代码中经常出现这种写法，本书中也常这么写。

`print()`方法表明，除构造器之外，编译器禁止你在其他任何方法中调用构造器。

static 的含义

了解 `this` 之后，你就能更全面地理解“静态（`static`）方法”的含义。静态方法就是没有 `this` 的方法。在“静态方法”的内部不能调用“非静态方法”³，反过来倒是可以的。而且你可以在没有创建任何对象的前提下，仅仅通过类本身来调用静态方法。这实际上正是静态方法存在的主要原因。它很象是 `C` 语言中的全局函数。`Java` 中禁止使用全局函数，但你在类中置入静态方法就可以访问其它静态方法和静态字段。

有些人认为静态方法不是“面向对象”的，因为它们的确具有全局函数的语义；使用静态方法时，由于不存在 `this`，所以不是通过“向对象发送消息”的方式来完成的。的确，要是你在代码中出现了大量的静态方法，就该重新考虑自己的设计了。然而，`static` 的概念有其实用之处，许多时候都要用到它。至于它是否真的“面向对象”，就留给理论家去讨论吧。事实上，`Smalltalk` 语言里的“类方法”就是与静态方法相对应的。

³这不是完全不可能。如果你传递一个对象的引用到静态方法里，然后通过这个引用（和 `this` 效果相同），你就可以调用非静态方法和访问非静态数据成员了。但通常要达到这样的效果，你只需写一个非静态方法即可。

清除 (cleanup): 终结 (finalization) 和垃圾回收 (garbage collection)

程序员都了解初始化的重要性，但常常会忘记同样重要的清除工作。毕竟，谁需要清除一个 `int` 呢？但在使用程序库时，把一个对象用完后就“弃之不顾”的做法并非总是安全的。当然，Java 有垃圾回收器来回收无用对象占据的内存资源。但也有特殊情况：假定你的对象（并非使用 `new`）获得了一块“特殊”的内存区域，由于垃圾回收器只知道释放那些经由 `new` 分配的内存，所以它不知道该如何释放该对象的这块“特殊”内存。为了应对这种情况，Java 允许你在类中定义一个名为 `finalize()` 的方法。它的工作原理“应该”是这样的：一旦垃圾回收器准备好释放对象占用的存储空间，将首先调用其 `finalize()` 方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。所以要是你打算用 `finalize()`，就能在“垃圾回收时刻”做一些重要的清除工作。

这里有一个潜在的编程陷阱，因为有些程序员（特别是 C++ 程序员）刚开始可能会误把 `finalize()` 当作 C++ 中的“析构函数”（C++ 中销毁对象必须用到这个函数）。所以有必要明确区分一下：在 C++ 中，对象一定会被“销毁”（如果程序中没有错误的话）；而 Java 里的对象却并非总是被“垃圾回收”的。或者换句话说：

1. 对象可能不被回收。
2. 垃圾回收并不等于“析构”。

牢记这些，你就能远离困扰。这意味着在你不再需要某个对象之前，如果必须执行某些动作，那么你得自己去做。Java 并未提供“析构函数”或相似的概念，要做类似的清除工作，你必须自己动手创建一个执行清除工作的普通方法。例如，假设某个对象在创建过程中，会将自己绘制到屏幕上。要是你不明确地从屏幕上将其擦除，它可能永远得不到清除。如果在 `finalize()` 里加入某种擦除功能，当“垃圾回收”发生时（不能保证一定会发生），`finalize()` 得到了调用，图像就会被擦除。要是“垃圾回收”没有发生，图像就会一直保留下来。

也许你会发现，只要程序没有濒临存储空间用完的那一刻，对象占用的空间就总也得不到释放。如果程序执行结束，并且垃圾回收器一直都没有释放你创建的任何对象的存储空间，则随着程序的退出，那些资源会全部交还给操作系统。这个策略是恰当的，因为垃圾回收本身也有开销，要是不使用它，那就不用支付这部分开销了。

`finalize()` 用途何在？

此时，你已经明白了不该将 `finalize()` 作为通用的清除方法。那么，`finalize()` 的真正用途是什么呢？

这引出了要记住的第三点：

3. 垃圾回收只与内存有关。

也就是说，垃圾回收器存在的唯一原因是为了回收程序不再使用的内存。所以对于与垃圾回收有关的任何行为来说（尤其是 `finalize()` 方法），它们也必须同内存及其回收有关。

但这是否意味着要是对象中含有其他对象，`finalize()` 就应该明确释放那些对象呢？不——无论对象是如何创建的，垃圾回收器都会负责释放对象占据的所有内存。这就将对 `finalize()` 的需求限制到特殊情况之下：你通过某种非“创建对象”的方式为对象分配了存储空间。不过，你也看到了，Java 中一切皆为对象，那这种特殊情况是怎么回事呢？

看来之所以要有 `finalize()`，是由于你可能在分配内存时，采用了类似 C 语言中的做法而非 Java 中的通常做法。这种情况主要发生在使用“本地方法”的情况下，它是在 Java 中调用非 Java 代码的一种方式（关于本地方法的讨论见本书电子版第二版，本书所附光盘和 www.BruceEckel.com 网站上均有收录）。本地方法目前只支持 C 和 C++。但它们可以调用其它语言写的代码，所以你实际上可以调用任何代码。在非 Java 代码中，也许会调用类似 C 的 `malloc()` 函数，用它分配存储空间，而且除非调用了 `free()` 函数，否则存储空间将不会得到释放，从而造成内存泄露。当然，`free()` 是 C 和 C++ 中的函数，所以你需要在 `finalize()` 中用本地方法调用它。

至此，你或许已经明白了不要过多地使用 `finalize()`⁴。对，它确实不是进行普通的清除工作的合适场所。那么，普通的清除工作应该在哪执行呢？

你必须执行清除

为清除一个对象，用户必须在进行清除的时刻调用执行清除动作的方法。听起来似乎很简单，但却与 C++ 中的“析构函数”的概念稍有抵触。在 C++ 中，所有对象都会被销毁，或者说，“应该”被销毁。如果在 C++ 中创建了一个局部对象（就是在堆栈上创建，Java 中可不行），此时的销毁动作发生在以“右花括号”为边界的、此对象作用域的末尾处进行。如果对象是用 `new` 创建的（类似于 Java），那么当程序员调用 C++ 的 `delete()` 时（Java 没有这个命令），就会调用相应的析构函数。如果程序员忘了，那么永远不会调用析构函数，就会出现内存泄露，对象的其他部分也不会得到清除。这种错误很难跟踪，这也是让 C++ 程序员转向 Java 的一个主要因素。

相反，Java 不允许创建局部对象，你必须使用 `new`。在 Java 中，也没有“`delete`”来释放对象，因为垃圾回收器会帮助你释放存储空间。甚至可以肤浅地认为，正是由于垃圾收集机制的存在，使得 Java 没有析构函数。然而，随着学习的深入，你就会明白垃圾回收器的存在并不能完全代替析构函数。（而且你绝对不能直接调用 `finalize()`，所以这也不是一个恰当的途径。）如果你希望进行除释放存储空间之外的清除工作，你还是得明确调用某个恰当的 Java 方法。这就等同于使用析构函数了，而且没有它方便。

记住，无论是“垃圾回收”还是“终结”，都不保证一定会发生。如果 Java 虚拟机（JVM）

⁴ Joshua Bloch 在题为“避免使用终结函数”一节中走的更远，他提到：“终结函数无法预料，常常是危险的，总之是多余的。”《Effective Java》，第 20 页，(Addison-Wesley 2001)。

并未面临内存耗尽的情形，它是不会浪费时间在回收垃圾以恢复内存上的。

终结条件

通常，你不能指望`finalize()`，你必须创建其它的“清除”方法，并且明确地调用它们。看来，`finalize()`只能存在于程序员很难用到的一些晦涩用法里了。不过，`finalize()`还有一个有趣的用法，它并不依赖于每次都要对`finalize()`进行调用，这就是对象“终结条件”⁵的验证。

当你对某个对象不再感兴趣，也就是它可以被清除时，这个对象应该处于某种状态，使它占用的内存可以被安全地释放。例如，要是对象代表了一个打开的文件，在对象被回收前程序员应该关闭这个文件。只要对象中存在没有被适当清除的部分，你的程序就存在很隐晦的错误。`finalize()`的价值在于可以用来最终发现这种情况，尽管它并不总是会被调用。如果某次`finalize()`的动作使得 `bug` 被发现，那你就可据此找出问题所在——这才是你真正关心的。

以下是个简单的例子，示范了可能的使用方式：

```
//: c04:TerminationCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.
import com.bruceeckel.simpletest.*;

class Book {
    boolean checkedOut = false;
    Book(boolean checkOut) {
        checkedOut = checkOut;
    }
    void checkIn() {
        checkedOut = false;
    }
    public void finalize() {
        if(checkedOut)
            System.out.println("Error: checked out");
    }
}

public class TerminationCondition {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Book novel = new Book(true);
        // Proper cleanup:
        novel.checkIn();
    }
}
```

⁵这个术语是在由Bill Venners (www.artima.com)和我一同开的培训班上发明的。

```
// Drop the reference, forget to clean up:
new Book(true);
// Force garbage collection & finalization:
System.gc();
monitor.expect(new String[] {
    "Error: checked out"}, Test.WAIT);
}
} ///:~
```

本例的终结条件是：所有的 **Book** 对象在被当作垃圾回收前都应该被签入（check in）。但在 `main()` 方法中，由于程序员的错误，有一本书未被签入。要是没有 `finalize()` 来验证终结条件，将很难发现这种错误。

注意，`System.gc()` 用于强制终结动作的进行（在写程序的时候这么做可以加速调试过程）。即使不这么做的话，通过重复的执行程序（假设程序将分配大量的存储空间而导致垃圾回收动作的执行），最终还是很有可能找出错误的 **Book** 对象的。

垃圾回收器如何工作

在你以前所用过的程序语言中，如果在堆上分配对象的代价十分高昂，你自然会觉得 **Java** 中所有对象（基本类型除外）都在堆上分配的方式也非常高昂。然而，垃圾回收器对于对象的创建，却具有明显的效果。听起来很奇怪——存储空间的释放竟然会影响存储空间的分配——但这确实是某些 **Java** 虚拟机的工作方式。这也意味着，**Java** 从堆分配空间的速度，可以和其它语言从堆栈上分配空间的速度相媲美。

打个比方，你可以把 **C++** 里的堆想象成一个院子，里面每个对象都负责管理自己的地盘。一段时间以后，对象可能被销毁，但地盘必须被重用。在某些 **Java** 虚拟机中，堆的实现截然不同：它更象一个传送带，你每分配一个新对象，它就往前移动一格。这意味着对象存储空间的分配速度非常快。**Java** 的“堆指针”只是简单地移动到尚未分配的区域，其效率比得上 **C++** 在堆栈上分配空间的效率。当然，实际过程中还存在诸如簿记工作的少量额外开销，但不会有象查找可用空间这样的大动作。

也许你已经意识到了，**Java** 中的堆未必完全象传送带那样工作。要真是那样的话，势必会导致频繁的内存页面调度（这将极大影响性能），并最终耗尽资源。其中的秘密在于垃圾回收器的介入。当它工作时，将一面回收空间，一面使堆中的对象紧凑排列，这样“堆指针”就可以很容易移动到更靠近传送带的开始处，也就尽量避免了页面错误。通过垃圾回收器对对象重新排列，实现了一种高速的、有无限空间可供分配的堆模型。

你得更好地理解不同垃圾回收器模式的工作机制，才能明白上述方式如何工作。“引用记数（reference counting）”是一种简单但速度很慢的垃圾回收技术。每个对象都含有一个引用计数器，当有引用连接至对象时，引用计数加 1。当引用离开作用域或被置为 `null` 时，引用计数减 1。虽然管理引用记数的开销不大，但需要在整个程序生命周期中持续地开销。垃圾回收器会在含有全部对象的列表上遍历，当发现某个对象的引用计数为 0 时，就释放其占用的

空间。这种方法有个缺陷，如果对象之间存在循环引用，可能会出现“对象应该被回收，但引用计数却不为零”的情况。对垃圾回收器而言，定位这样存在交互引用的对象组所需的工作量极大。引用记数常用来说明垃圾收集的工作方式，似乎从未被应用于任何一种 Java 虚拟机实现中。

在一些更快的模式中，垃圾回收器并非基于引用记数技术。它们依据的思想是：对任何“活”的对象，一定能最终追溯到其存活在堆栈或静态存储区之中的引用。这个引用链条可能会穿过数个对象层次。由此，如果你从堆栈和静态存储区开始，遍历所有的引用，就能找到所有“活”的对象。对于发现的每个引用，你必须追踪它所引用的对象，然后是此对象包含的所有引用，如此反复进行，直到“根源于堆栈和静态存储区的引用”所形成的网络全部被访问为止。你所访问过的对象必须都是“活”的。注意，这就解决了“存在交互引用的整体对象”的问题，这些对象根本不会被发现，因此也就被自动回收了。

在这种方式下，Java 虚拟机将采用一种“自适应”的垃圾回收技术。至于如何处理找到的存活对象，取决于不同的 Java 虚拟机实现。有一种作法名为“停止——复制”(stop-and-copy)。这意味着，先暂停程序的运行，(所以它不属于后台回收模式)，然后将所有存活的对象从当前堆复制到另一个堆，没有被复制的全部都是垃圾。当对象被复制到新堆时，它们是一个挨着一个的，所以新堆保持紧凑排列，然后就可以按前述方法简单、直接地分配新空间了。

当把对象从一处搬到另一处时，所有指向它的那些引用都必须修正。位于堆或静态存储区的引用可以直接被修正，但可能还有其它指向这些对象的引用，它们在遍历的过程中才能被找到。你可以想象有个表格，将旧地址映射至新地址，这样就可以在遍历的同时进行修改了。

对于这种所谓的“复制式回收器”而言，有两个原因会降低效率。首先，你得有两个堆，然后你得在这两个分离的堆之间来回捣腾，从而得维护比实际需要多一倍的空间。某些 Java 虚拟机对此问题的处理方式是，按需从堆中分配几块较大的内存，复制动作发生在这些大块内存之间。

第二个问题在于复制。你的程序进入稳定状态之后，可能只会产生少量垃圾，甚至没有垃圾。尽管如此，复制式回收器仍然会将所有内存自一处复制到另一处，这很浪费。为了避免这种情形，一些 Java 虚拟机会进行检查：要是没有新垃圾产生，就会转换到另一种工作模式（此即“自适应”）。这种模式称为“标记——清扫 (mark-and-sweep)”，Sun 公司早期版本的 Java 虚拟机使用了这种技术。对一般用途而言，“标记——清扫”方式速度相当慢，但是当你知道你只会产生少量垃圾甚至不会产生垃圾时，它的速度就很快了。

“标记——清扫”所依据的思路同样是从堆栈和静态存储区出发，遍历所有的引用，进而找出所有存活的对象。每当它找到一个存活对象，就会给对象设一个标记，这个过程中不会回收任何对象。只有全部标记工作完成的时候，清除动作才会开始。在清除过程中，没有标记的对象将被释放，不会发生任何复制动作。所以剩下的堆空间是不连续的，垃圾回收器要是希望得到连续空间的话，就得重新整理剩下的对象。

“停止——复制”的意思是这种垃圾回收方式不是在后台进行的；相反，垃圾回收动作发生的同时，程序将会被暂停。在 Sun 公司的文档中你会发现，许多参考文献将垃圾回收视为低优先级的后台进程，但事实上垃圾回收器并非以这种方式实现——至少 Sun 公司早期版

本的 Java 虚拟机中并非如此。当可用内存数量较低时，Sun 版中的垃圾回收器才会被激活，同样，“标记——清扫”工作也必须在程序暂停的情况下才能进行。

如前文所述，这里讨论的 Java 虚拟机，内存分配单位是较大的“块”。如果对象较大，它会占用单独的块。严格来说，“停止——复制”要求你在释放旧有对象之前，必须先把所有存活对象从旧堆复制到新堆，这将导致大量内存复制行为。有了块之后，垃圾回收器在回收的时候就可以往废弃的块里拷贝对象了。每个块都用相应的“代数（generation count）”记录它是否还存活。通常，如果块在某处被引用，其代数会增加；垃圾回收器将对上次回收动作之后新分配的块进行整理。这对处理大量短命的临时对象很有帮助。垃圾回收器会定期进行完整的清除动作——大型对象仍然不会被复制（只是其代数会增加），内含小型对象的那些块则被复制并整理。Java 虚拟机会进行监视，如果所有对象都很稳定，垃圾回收器的效率降低的话，就切换到“标记——清扫”方式；同样，Java 虚拟机会注意“标记——清扫”的效果，要是堆空间出现很多碎片，就会切换回“停止——复制”方式。这就是“自适应”技术。你可以给它个罗嗦的称呼：“自适应的、分代的、停止——复制、标记——清扫”式垃圾回收器。

Java 虚拟机中有许多附加技术用以提升速度。尤其是与加载器操作有关的，被称为“即时”（Just-In-Time, JIT）编译的技术。这种技术可以把程序全部或部分翻成本地机器码（这本来是 Java 虚拟机的工作），程序运行速度因此得以提升。当需要装载某个类（通常是在你为该创建第一个对象）时，编译器会先找到其 .class 文件，然后将该类的字节码装入内存。此时，有两种方案可供选择。一种是就让即时编译器编译所有代码。但这种做法有两个缺陷：这种加载动作散落在整个程序生命周期内，累加起来要花更多时间；并且会增加可执行代码的长度（字节码要比即时编译器展开后的本地机器码小很多），这将导致页面调度，从而降低程序速度。另一种做法称为“惰性编译（lazy evaluation）”，意思是即时编译器只在必要的时候才编译代码。这样，从不会被执行的代码也许就压根不会被 JIT 所编译。新版 JDK 中的 Java HotSpot 技术就采用了类似方法，代码每次被执行的时候都会做一些优化，所以执行的次数越多，它的速度就越快。

成员初始化

Java 尽力保证：所有变量在使用前都能得到恰当的初始化。对于定义于方法内部的局部变量，Java 以编译时刻错误的形式来贯彻这种保证。所以如果你这么写：

```
void f() {
    int i;
    i++; // Error -- i not initialized
}
```

就会得到一条出错消息，告诉你 i 可能尚未初始化。当然，编译器也可以为 i 赋予一个缺省值，但是未初始化这种情况看起来更像是程序员的疏忽，所以采用缺省值反而会掩盖这种失误。所以强制程序员提供一个初始值，往往能够帮助找出程序里的 bug。

要是类的数据成员是基本类型，情况就会变得有些不同。因为类中的任何方法都可以初始化

或用到这个数据，所以强制用户一定得在使用数据前将其初始化成一个适当的值并不现实。然而，任其含有无意义的值同样也是不安全的。因此，一个类的所有基本类型数据成员都会保证有一个初始值。可用下面这段小程序看到这些值：

```
//: c04:InitialValues.java
// Shows default initial values.
import com.bruceeckel.simpletest.*;

public class InitialValues {
    static Test monitor = new Test();
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    void print(String s) { System.out.println(s); }
    void printInitialValues() {
        print("Data type      Initial value");
        print("boolean        " + t);
        print("char            [" + c + "]");
        print("byte            " + b);
        print("short           " + s);
        print("int             " + i);
        print("long            " + l);
        print("float           " + f);
        print("double          " + d);
    }
    public static void main(String[] args) {
        InitialValues iv = new InitialValues();
        iv.printInitialValues();
        /* You could also say:
        new InitialValues().printInitialValues();
        */
        monitor.expect(new String[] {
            "Data type      Initial value",
            "boolean        false",
            "char            [" + (char)0 + "]",
            "byte            0",
            "short           0",
            "int             0",
            "long            0",
        })
    }
}
```



```

        "float"        0.0",
        "double"       0.0"
    });
}
} ///:~

```

可见尽管数据成员的初值没有给出，但它们确实有初值（char 值为 0，所以显示为空白）。所以你至少不会冒“未初始化变量”的风险了。

稍后你会看到，在类里定义一个对象引用时，如果不将其初始化，此引用就会获得一个特殊值 null（这是 Java 关键字）。

指定初始化

如果想为某个变量赋初值，该怎么做呢？有一种很直接的办法，就是在定义类成员变量的地方为其赋值（注意在 C++ 里不能这样做，尽管 C++ 的新手们总想这样做）。以下代码片段修改了 InitialValues 类成员变量的定义，直接提供了初值。

```

class InitialValues {
    boolean b = true;
    char c = 'x';
    byte B = 47;
    short s = 0xff;
    int i = 999;
    long l = 1;
    float f = 3.14f;
    double d = 3.14159;
    // . . .
}

```

你也可以用同样的方法初始化非基本类型的对象。如果 Depth 是一个类，你可以象下面这样创建一个对象并初始化它：

```

class Measurement {
    Depth d = new Depth();
    // . . .
}

```

如果你没有为 d 指定一个初始值就尝试使用它，就会出现运行期错误，告诉你产生了一个“异常（Exception）”（在第 9 章中详述）。

你甚至可以通过调用某个方法来提供初值：

```

class CInit {
    int i = f();
}

```



```
//...  
}
```

这个方法也可以带有参数，但这些参数必须是已经被初始化的。因此，你可以这样写：

```
class CInit {  
    int i = f();  
    int j = g(i);  
    //...  
}
```

但像下面这样写就不对了：

```
class CInit {  
    int j = g(i);  
    int i = f();  
    //...  
}
```

显然，上述程序的正确性取决于初始化的顺序，而与其编译方式无关。所以，编译器恰当地对“向前引用”发出了警告。

这种初始化方法既简单又直观。但有个限制：类 **InitialValues** 的每个对象都会具有相同的初值。有时，这正是你所希望的，但有时你却需要更大的灵活性。

构造器初始化

可以用构造器来进行初始化。在运行时刻，你可以调用方法或执行某些动作来确定初值，这为你在编程时带来了更大的灵活性。但要牢记：你无法屏蔽自动初始化的进行，它将在构造器被调用之前发生。因此，假如使用下述代码：

```
class Counter {  
    int i;  
    Counter() { i = 7; }  
    // . . .
```

那么*i*首先会被置 0，然后变成 7。对于基本类型和对象引用，包括在定义时已经指定初值的变量，这种情况都是成立的。因此，编译器不会强制你一定要在构造器的某个地方或在使用它们之前对元素进行初始化——因为初始化早已得到了保证⁶。

⁶相反，C++有构造器初始化列表来确保初始化动作发生于进入构造器体之前，并强制施行于所有对象上。见《Thinking in C++》，第二版（本书所附光盘里就有，也可以到www.BruceEckel.com上下载）。

初始化顺序

在类的内部,变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于方法定义之间,它们仍旧会在任何方法(包括构造器)被调用之前得到初始化。例如:

```
//: c04:OrderOfInitialization.java
// Demonstrates initialization order.
import com.bruceekel.simpletest.*;

// When the constructor is called to create a
// Tag object, you'll see a message:
class Tag {
    Tag(int marker) {
        System.out.println("Tag(" + marker + ")");
    }
}

class Card {
    Tag t1 = new Tag(1); // Before constructor
    Card() {
        // Indicate we're in the constructor:
        System.out.println("Card()");
        t3 = new Tag(33); // Reinitialize t3
    }
    Tag t2 = new Tag(2); // After constructor
    void f() {
        System.out.println("f()");
    }
    Tag t3 = new Tag(3); // At end
}

public class OrderOfInitialization {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Card t = new Card();
        t.f(); // Shows that construction is done
        monitor.expect(new String[] {
            "Tag(1)",
            "Tag(2)",
            "Tag(3)",
            "Card()",
            "Tag(33)",
            "f()"
        })
    }
}
```

```

    });
}
} ///:~

```

在 `Card` 类中，故意把几个 `Tag` 对象的定义散布到各处，以证明它们全都会在调用构造器或其它方法之前得到初始化。此外，`t3` 在构造器内再次被初始化。

由输出可见，`t3` 这个引用会被初始化两次：一次在调用构造器前，一次在调用期间（第一次引用的对象将被丢弃，并作为垃圾回收）。试想，如果定义了一个重载的构造器，它没有初始化 `t3`；同时在 `t3` 的定义里也没有指定缺省值，那会产生什么后果呢？所以尽管这种方法似乎效率不高，但它的确能使初始化得到保证。

静态数据的初始化

如果数据是静态的（`static`），情况并无不同：如果它属于某个基本类型，而且你也没有对它进行初始化，那么它就会获得基本类型的标准初值；如果它是一个对象引用，那么除非你新建一个对象，并指派给该引用，否则它就是空值（`null`）。

如果想在定义处进行初始化，采取的方法与非静态数据没什么不同。无论创建多少个对象，静态数据都只占用一份存储区域。但是当你要对这个静态存储区域进行初始化时，问题就来了。看看下面这个例子会更清楚一些：

```

///: c04:StaticInitialization.java
/// Specifying initial values in a class definition.
import com.bruceeckel.simpletest.*;

class Bowl {
    Bowl(int marker) {
        System.out.println("Bowl(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Table {
    static Bowl b1 = new Bowl(1);
    Table() {
        System.out.println("Table()");
        b2.f(1);
    }
    void f2(int marker) {
        System.out.println("f2(" + marker + ")");
    }
}

```

```

    }
    static Bowl b2 = new Bowl(2);
}

class Cupboard {
    Bowl b3 = new Bowl(3);
    static Bowl b4 = new Bowl(4);
    Cupboard() {
        System.out.println("Cupboard()");
        b4.f(2);
    }
    void f3(int marker) {
        System.out.println("f3(" + marker + ")");
    }
    static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
    static Test monitor = new Test();
    public static void main(String[] args) {
        System.out.println("Creating new Cupboard() in main");
        new Cupboard();
        System.out.println("Creating new Cupboard() in main");
        new Cupboard();
        t2.f2(1);
        t3.f3(1);
        monitor.expect(new String[] {
            "Bowl(1)",
            "Bowl(2)",
            "Table()",
            "f(1)",
            "Bowl(4)",
            "Bowl(5)",
            "Bowl(3)",
            "Cupboard()",
            "f(2)",
            "Creating new Cupboard() in main",
            "Bowl(3)",
            "Cupboard()",
            "f(2)",
            "Creating new Cupboard() in main",
            "Bowl(3)",
            "Cupboard()",
            "f(2)",
        });
    }
}

```

```

        "f2(1)",
        "f3(1)"
    });
}
static Table t2 = new Table();
static Cupboard t3 = new Cupboard();
} ///:~

```

Bowl 类使你得以看到类的创建，而 Table 类和 Cupboard 类在它们的类定义中加入了 Bowl 类型的静态成员。注意，在静态数据成员定义之前，Cupboard 类先定义了一个 Bowl 类型的非静态成员 b3。

由输出可见，静态初始化只有在必要时刻才会进行。如果不创建 Table 对象，也不引用 Table.b1 或 Table.b2，那么静态的 Bowl b1 和 b2 永远都不会被创建。只有在第一个 Table 对象被创建（或者第一次访问静态数据）的时候，它们才会被初始化。此后，静态对象不会再次被初始化。

初始化的顺序是先“静态”，（如果它们尚未因前面的对象创建过程而被初始化），后“非静态”。从输出结果中可以观察到这一点。

总结一下对象的创建过程会很有帮助。假设有个名为 Dog 的类：

1. 当首次创建类型为 Dog 的对象时（构造器可以看成静态方法），或者 Dog 类的静态方法 / 静态域首次被访问时，Java 解释器必须查找类路径，以定位 Dog.class 文件。
2. 然后载入 Dog.class（后面会学到，这将创建一个 Class 对象），有关静态初始化的动作都会执行。因此，静态初始化只在 Class 对象首次加载的时候进行一次。
3. 当你用 new Dog() 创建对象的时候，首先将在堆上为 Dog 对象分配足够的存储空间。
4. 这块存储空间会被清零，这就自动地将 Dog 中的所有基本类型数据设置成了默认值（对数字来说就是 0，对布尔型和字符型也相同），而引用则被设置成了 null。
5. 执行所有出现于域定义处的初始化动作。
6. 执行构造器。正如你将在第 6 章中看到的，这可能会牵涉到很多动作，尤其是涉及继承的时候。

明确进行的静态初始化

Java 允许你将多个静态初始化动作组织成一个特殊的“静态子句”（有时也叫作“静态块”）。就象下面这样：

```

class Spoon {
    static int i;
    static {
        i = 47;
    }
}

```

```
// . . .
```

尽管上面的代码看起来象个方法，但它实际只是一段跟在 `static` 关键字后面的代码。与其他静态初始化动作一样，这段代码仅执行一次：当你首次生成这个类的一个对象时，或者首次访问属于那个类的一个静态成员时（即便从未生成过那个类的对象）。例如：

```
//: c04:ExplicitStatic.java
// Explicit static initialization with the "static" clause.
import com.bruceeckel.simpletest.*;

class Cup {
    Cup(int marker) {
        System.out.println("Cup(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Cups {
    static Cup c1;
    static Cup c2;
    static {
        c1 = new Cup(1);
        c2 = new Cup(2);
    }
    Cups() {
        System.out.println("Cups()");
    }
}

public class ExplicitStatic {
    static Test monitor = new Test();
    public static void main(String[] args) {
        System.out.println("Inside main()");
        Cups.c1.f(99); // (1)
        monitor.expect(new String[] {
            "Inside main()",
            "Cup(1)",
            "Cup(2)",
            "f(99)"
        });
    }
    // static Cups x = new Cups(); // (2)
}
```

```

    // static Cups y = new Cups(); // (2)
} ///:~

```

无论是通过标为(1)的那行程序访问静态的 `c1` 对象，还是把(1)注释掉，让它去运行标为(2)的那行，`Cups` 的静态初始化动作都会得到执行。如果把(1)和(2)同时注释掉，`Cups` 的静态初始化动作就不会进行。此外，激活一行还是两行(2)代码都无关紧要，静态初始化动作只进行一次。

非静态实例初始化

Java 中也有类似的语法用来初始化每一个对象的非静态变量。例如：

```

///: c04:Mugs.java
// Java "Instance Initialization."
import com.bruceeckel.simpletest.*;

class Mug {
    Mug(int marker) {
        System.out.println("Mug(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

public class Mugs {
    static Test monitor = new Test();
    Mug c1;
    Mug c2;
    {
        c1 = new Mug(1);
        c2 = new Mug(2);
        System.out.println("c1 & c2 initialized");
    }
    Mugs() {
        System.out.println("Mugs()");
    }
    public static void main(String[] args) {
        System.out.println("Inside main()");
        Mugs x = new Mugs();
        monitor.expect(new String[] {
            "Inside main()",
            "Mug(1)",

```

```

        "Mug(2)",
        "c1 & c2 initialized",
        "Mugs()"
    });
}
} ///:~

```

你可以看到实例初始化子句：

```

{
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
}

```

看起来它与静态初始化子句一模一样，只不过少了 `static` 关键字。这种语法对于支持“匿名内部类（anonymous inner class）”（参见第 8 章）的初始化是必须的。

数组初始化

在C语言中初始化数组既容易出错，又相当麻烦。C++则通过“聚合初始化（aggregate initialization）”使其更安全⁷。在Java中，一切都是对象，也没有类似C++里“聚合”那样的概念。但它确实提供了数组，也支持数组初始化。

数组只是相同类型的、用一个标识符名称封装到一起的一个对象序列或基本类型数据序列。数组是通过方括号索引操作符`[]`来定义和使用的。要定义一个数组，只需在类型名后加上一对空方括号即可：

```
int[] a1;
```

方括号也可以置于标识符后面，其含义相同：

```
int a1[];
```

这种格式符合 C 和 C++程序员的习惯。不过，前一种格式或许更合理，毕竟它表明类型是“一个 `int` 型数组”。本书将采用这种格式。

编译器不允许你指定数组的大小。这就又把我们带回到有关“引用”的问题上。现在你拥有的只是对数组的一个引用，而且也没给数组分配任何空间。为了给数组创建相应的存储空间，你必须写初始化表达式。对于数组，初始化动作可以出现在代码的任何地方，但也可以使用一种特殊的初始化表达式，它必须在创建数组的地方出现。这种特殊的初始化是由一对花括

⁷要完整了解C++聚集初始化，参见《Thinking in C++》，第二版。

号括起来的值组成的。在这种情况下，存储空间的分配（等价于使用 `new`）将由编译器负责。例如：

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

那么，为什么还要在没有数组的时候定义一个数组引用呢？

```
int[] a2;
```

在 Java 中你可以将一个数组赋值给另一个数组，所以你可以这样：

```
a2 = a1;
```

其实你真正做的只是复制了一个引用，就象下面演示的那样：

```
//: c04:Arrays.java
// Arrays of primitives.
import com.bruceeckel.simpletest.*;

public class Arrays {
    static Test monitor = new Test();
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
            a2[i]++;
        for(int i = 0; i < a1.length; i++)
            System.out.println(
                "a1[" + i + "] = " + a1[i]);
        monitor.expect(new String[] {
            "a1[0] = 2",
            "a1[1] = 3",
            "a1[2] = 4",
            "a1[3] = 5",
            "a1[4] = 6"
        });
    }
} ///:~
```

你可以看到代码中给出了 `a1` 的初始值，但 `a2` 却没有；在本例中，`a2` 是在后面被赋值为另一个数组的。

这里有点新东西：所有数组（无论它们的元素是对象还是基本类型）都有一个固有成员，你

可以通过它获知数组内包含了多少个元素，但不能对其修改。这个成员就是 `length`。与 C 和 C++ 类似，Java 数组计数也是从第 0 个元素开始，所以能使用的最大索引数是 “`length - 1`”。要是超出这个边界，C 和 C++ 会 “默默” 地接受，并允许你访问所有内存，许多声名狼藉的程序错误由此而生。Java 则能保护你免受这一问题的困扰，一旦访问下标过界，就会出现运行期错误（即 “异常”，将在第 9 章中讨论）。当然，每次访问数组的时候都要检查边界的做法是需要在时间和代码上有所开销的，但是你无法禁用这个功能。这意味着如果数组访问发生在一些关键节点上，它们有可能会成为导致程序效率低下的原因之一。但是基于 “因特网的安全以及提高程序员生产力” 的理由，Java 的设计者认为这是值得的取舍。

如果在编写程序时，你并不能确定在数组里需要多少个元素，那么你该怎么办呢？你可以直接用 `new` 在数组里创建元素。尽管创建的是基本类型数组，`new` 仍然可以工作（不能用 `new` 创建单个的基本类型数据）。

```
//: c04:ArrayNew.java
// Creating arrays with new.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class ArrayNew {
    static Test monitor = new Test();
    static Random rand = new Random();
    public static void main(String[] args) {
        int[] a;
        a = new int[rand.nextInt(20)];
        System.out.println("length of a = " + a.length);
        for(int i = 0; i < a.length; i++)
            System.out.println("a[" + i + "] = " + a[i]);
        monitor.expect(new Object[] {
            "%% length of a = \\d+",
            new TestExpression("%% a\\[\\d+\\] = 0", a.length)
        });
    }
} ///:~
```

本例中的 `expect()` 语句里有些新内容：`TestExpression` 类。要创建 `TestExpression` 对象需要传入一个表达式，此表达式既可以是普通的字符串，也可以是如例子中的正则表达式；还要传入一个整形参数用以说明表达式重复的次数。如本例所示，`TestExpression` 类不仅可以防止无意义的代码重复，还能在运行时刻决定重复的次数。

数组的大小是通过 `Random.nextInt()` 方法随机决定的，这个方法会返回 0 到输入参数之间的一个值。这表明数组的创建确实是在运行时刻进行的。此外，程序输出表明：数组元素中的基本数据类型值会自动初始化成 “空” 值。（对于数字和字符，就是 0；对于布尔型，是 `false`）。

当然，数组也可以在定义的同时进行初始化：

```
int[] a = new int[rand.nextInt(20)];
```

如果可能的话，你应该尽量这么做。

如果数组里的元素不是基本数据类型，那么你必须使用 **new**。在这里，你会再次遇到引用问题，因为你创建的数组里每个元素都是一个引用。以整型的包装类 **Integer**（它可不是基本类型）为例：

```
//: c04:ArrayClassObj.java
// Creating an array of nonprimitive objects.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class ArrayClassObj {
    static Test monitor = new Test();
    static Random rand = new Random();
    public static void main(String[] args) {
        Integer[] a = new Integer[rand.nextInt(20)];
        System.out.println("length of a = " + a.length);
        for(int i = 0; i < a.length; i++) {
            a[i] = new Integer(rand.nextInt(500));
            System.out.println("a[" + i + "] = " + a[i]);
        }
        monitor.expect(new Object[] {
            "%% length of a = \\d+",
            new TestExpression("%% a\\[\\d+\\] = \\d+", a.length)
        });
    }
} ///:~
```

这里，即便使用 **new** 创建数组之后：

```
Integer[] a = new Integer[rand.nextInt(20)];
```

它还只是一个引用数组，并且直到通过创建新的 **Integer** 对象，并且把对象赋值给引用，初始化进程才算结束：

```
a[i] = new Integer(rand.nextInt(500));
```

如果你忘记了创建对象，并且试图使用数组中的空引用，就会在运行时刻产生“异常”。

注意一下打印语句中 **String** 对象的形成，**Integer** 对象的引用会自动转型，从而产生一个代表对象内部值的字符串。

也可以用花括号括起来的列表来初始化对象数组。有两种形式：

```
//: c04:ArrayInit.java
// Array initialization.

public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
            new Integer(1),
            new Integer(2),
            new Integer(3),
        };
        Integer[] b = new Integer[] {
            new Integer(1),
            new Integer(2),
            new Integer(3),
        };
    }
} ///:~
```

第一种写法有时很有用，但由于数组的大小在编译期间就决定了，所以很受限制。初始化列表的最后一个逗号是可选的（这一特性使维护长列表变得更容易）。

第二种形式提供了一种方便的语法来创建对象并调用方法，以获得与 C 的“可变参数列表”（C 通常把它简称为“varargs”）一致的效果。这可以应用于参数个数或类型未知的场合。由于所有的类都直接或间接继承于 `Object` 类（随着本书的进展，你会对它有更深入的认识），所以你可以创建以 `Object` 数组为参数的方法，并这样调用：

```
//: c04:VarArgs.java
// Using array syntax to create variable argument lists.
import com.bruceeckel.simpletest.*;

class A { int i; }

public class VarArgs {
    static Test monitor = new Test();
    static void print(Object[] x) {
        for(int i = 0; i < x.length; i++)
            System.out.println(x[i]);
    }
    public static void main(String[] args) {
        print(new Object[] {
            new Integer(47), new VarArgs(),
        });
    }
}
```

```

        new Float(3.14), new Double(11.11)
    });
    print(new Object[] { "one", "two", "three" });
    print(new Object[] { new A(), new A(), new A() });
    monitor.expect(new Object[] {
        "47",
        "% VarArgs@\p{XDigit}+",
        "3.14",
        "11.11",
        "one",
        "two",
        "three",
        new TestExpression("% A@\p{XDigit}+", 3)
    });
}
} ///:~

```

你可以看到 `print()` 方法使用 `Object` 数组作为参数，然后遍历数组，打印每个对象。标准 Java 库中的类能输出有意义的内容，但这里建立的类（`A` 和 `VarArgs`）的对象，打印出的内容只是类的名称以及后面紧跟着的一个 '@' 符号。代码里出现的正则表达式 `\p{XDigit}`，表示一个十六进制数字。后面的 '+' 表示会有一个或多个十六进制数字。于是，缺省行为（如果你没有定义 `toString()` 方法的话，后面会讲这个方法的）就是打印类的名字和对象的地址。

多维数组

在 Java 中创建多维数组也很方便：

```

///: c04:MultiDimArray.java
// Creating multidimensional arrays.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class MultiDimArray {
    static Test monitor = new Test();
    static Random rand = new Random();
    public static void main(String[] args) {
        int[][] a1 = {
            { 1, 2, 3, },
            { 4, 5, 6, },
        };
        for(int i = 0; i < a1.length; i++)
            for(int j = 0; j < a1[i].length; j++)
                System.out.println(

```

```

        "a1[" + i + "][" + j + "] = " + a1[i][j]);
// 3-D array with fixed length:
int[][][] a2 = new int[2][2][4];
for(int i = 0; i < a2.length; i++)
    for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length; k++)
            System.out.println("a2[" + i + "][" + j + "][" + k + "] = " + a2[i][j][k]);
// 3-D array with varied-length vectors:
int[][][] a3 = new int[rand.nextInt(7)][][];
for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[rand.nextInt(5)][];
    for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[rand.nextInt(5)];
}
for(int i = 0; i < a3.length; i++)
    for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length; k++)
            System.out.println("a3[" + i + "][" + j + "][" + k + "] = " + a3[i][j][k]);
// Array of nonprimitive objects:
Integer[][] a4 = {
    { new Integer(1), new Integer(2)},
    { new Integer(3), new Integer(4)},
    { new Integer(5), new Integer(6)},
};
for(int i = 0; i < a4.length; i++)
    for(int j = 0; j < a4[i].length; j++)
        System.out.println("a4[" + i + "][" + j + "] = " + a4[i][j]);
Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];
    for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i * j);
}
for(int i = 0; i < a5.length; i++)
    for(int j = 0; j < a5[i].length; j++)
        System.out.println("a5[" + i + "][" + j + "] = " + a5[i][j]);
// Output test
int ln = 0;
for(int i = 0; i < a3.length; i++)

```

```

        for(int j = 0; j < a3[i].length; j++)
            for(int k = 0; k < a3[i][j].length; k++)
                ln++;
monitor.expect(new Object[] {
    "a1[0][0] = 1",
    "a1[0][1] = 2",
    "a1[0][2] = 3",
    "a1[1][0] = 4",
    "a1[1][1] = 5",
    "a1[1][2] = 6",
    new TestExpression(
        "% a2\\[\\d\\]\\[\\d\\]\\[\\d\\] = 0", 16),
    new TestExpression(
        "% a3\\[\\d\\]\\[\\d\\]\\[\\d\\] = 0", ln),
    "a4[0][0] = 1",
    "a4[0][1] = 2",
    "a4[1][0] = 3",
    "a4[1][1] = 4",
    "a4[2][0] = 5",
    "a4[2][1] = 6",
    "a5[0][0] = 0",
    "a5[0][1] = 0",
    "a5[0][2] = 0",
    "a5[1][0] = 0",
    "a5[1][1] = 1",
    "a5[1][2] = 2",
    "a5[2][0] = 0",
    "a5[2][1] = 2",
    "a5[2][2] = 4"
});
}
} ///:~

```

注意打印部分的代码中使用了 **length**，所以它不会依赖于固定数组的大小。

第一个例子演示了基本数据类型的多维数组。你可用花括号标出数组中的每个向量：

```

int[][] a1 = {
    { 1, 2, 3, },
    { 4, 5, 6, },
};

```

每对方括号都会帮你访问下一级数组。

第二个例子用 `new` 分配了一个三维数组。这里整个数组分配是一次完成的：

```
int[][][] a2 = new int[2][2][4];
```

第三个例子表明，数组中用以构成矩阵的那些向量可以有任意的长度：

```
int[][][] a3 = new int[rand.nextInt(7)][][];
for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[rand.nextInt(5)][];
    for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[rand.nextInt(5)];
}
```

第一个 `new` 创建了数组，其第一维的长度是由随机数确定的，其他维的长度则没有定义。位于 `for` 循环内的第二个 `new` 则会决定第二维的长度，直到碰到第三个 `new`，第三维的长度才得以确定。

通过输出结果可见：如果没有明确指定初始化值，数组值就会自动置为零。

你可以用类似的方式处理非基本类型的对象数组。第四个例子演示了用花括号把多个 `new` 表达式组织到一起的能力：

```
Integer[][] a4 = {
    { new Integer(1), new Integer(2)},
    { new Integer(3), new Integer(4)},
    { new Integer(5), new Integer(6)},
};
```

第五个例子展示了如何逐步创建非基本类型的对象数组：

```
Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];
    for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
}
```

`i*j` 只是为了在 `Integer` 里放入一个有趣的值。

总结

构造器，这种精巧的初始化机制，应该给了你很强的暗示：初始化在 Java 中占有至关重要的地位。C++的发明人 Bjarne Stroustrup 在设计 C++期间，在针对 C 语言的生产率所作的最初调查中发现，大量编程错误都源于不正确的初始化。这种错误很难发现，并且不恰当的清除也会导致类似问题。构造器使你能保证正确的初始化和清除（没有正确的构造器调用，编译器就不会允许你创建对象），所以你得到了完全的控制，也很安全。

在 C++中，“析构”相当重要，因为用 `new` 创建的对象必须明确被销毁。在 Java 中，垃圾回收器会自动为对象释放内存，所以在很多场合下 Java 中类似的清除方法就不太需要了（不过正如本章指出的，当要用到的时候，你就只能自己动手了）。在你不需要类似析构函数行为的时候，Java 的垃圾回收器可以极大简化编程工作，而且在处理内存的时候也更安全。有些垃圾回收器甚至能清除其他资源，比如图形和文件句柄。然而，垃圾回收器确实也增加了运行期的开销。而且 Java 解释器从来就很慢，所以这种开销到底造成了多大的影响却很难看出。随着时间的推移，Java 在性能方面已经取得了长足的进步，但速度问题仍然是妨碍它涉足某些特定编程领域的障碍。

由于要保证所有对象都被创建，构造器实际上要比这里讨论的更复杂。特别当你通过“组合”或“继承”生成新类的时候，这种保证仍然成立，并且需要一些附加的语法来提供支持。在后面的章节中，你将学习到有关组合、继承以及它们对构造器造成的影响等方面的知识。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 创建一个带缺省构造器（即无参构造器）的类，在构造器中打印一条消息。为这个类创建一个对象。
2. 为练习 1 中的类添加一个重载的构造器，令其接受一个字符串参数，并在构造器中把你自己的消息和接收的参数一起打印出来。
3. 创建一个数组，其元素引用上题中的类的对象，但不要实际创建对象并赋值给数组。运行程序时注意它是否会打印调用构造器时所产生的初始化消息。
4. 创建对象，并赋值给数组，最终完成练习 3。
5. 创建一个字符串对象数组，并为其每个元素赋予一个字符串对象。使用 `for` 循环将所有内容打印出来。
6. 创建一个名为 `Dog` 的类，具有重载的 `bark()` 方法。此方法应根据不同的基本数据类型进行重载，并根据被调用的版本，打印出不同类型的狗吠（`barking`）、咆哮（`howling`）等信息。编写 `main()` 来调用所有不同版本的方法。
7. 修改练习 6 的程序，让两个重载方法各自接受两个类型不同的参数，但二者顺序相反。验证其是否工作。
8. 创建一个没有构造器的类，并在 `main()` 中创建其对象，用以验证编译器是否真的自动加入了缺省构造器。

9. 编写具有两个方法的类，在第一个方法内调用第二个方法两次：第一次调用时不使用 `this`，第二次调用时使用 `this`。
10. 编写具有两个重载构造器的类，并在第一个构造器中通过 `this` 调用第二个构造器。
11. 编写具有 `finalize()` 方法的类，并在其中打印消息。在 `main()` 中为该类创建一个对象。试解释这个程序的行为。
12. 修改练习 11 的程序，让你的 `finalize()` 总会被调用。
13. 编写名为 `Tank` 的类，此类的状态可以是满的或空的。其“终结条件”是：对象被清除时必须处于空状态。请编写 `finalize()` 以检验终结条件是否成立。在 `main()` 中测试 `Tank` 可能发生的几种使用方式。
14. 编写一个类，内含未初始化的整型和字符型成员，打印其值以检验 Java 的缺省初始化动作。
15. 编写一个类，内含未初始化的字符串引用。证明这个引用会被 Java 初始化为 `null`。
16. 编写一个类，内含一个字符串字段，在定义处将其初始化，另一个字符串字段由构造器初始化。这两种方法有什么分别？
17. 编写一个类，拥有两个静态字符串字段，其中一个在定义处初始化，另一个在静态块中初始化。现在，加入一个静态方法用以打印出两个字段值。请证明它们都会在被使用之前完成初始化动作。
18. 编写一个类，内含字符串字段，并采用“实例初始化”方式初始化。试描述此种方式的用途（请提出一个和本书所言不同的用途）。
19. 编写一个方法，能够产生二维双精度型数组并加以初始化。数组的容量由方法的形式参数决定，其初值必须落在另外两个形式参数所指定的区间之内。编写第二个方法，打印出第一个方法所产生的数组。在 `main()` 中通过产生不同容量的数组并打印其内容来测试这两个方法。
20. 重复练习 19，但改为三维数组。
21. 将本章中的 `ExplicitStatic.java` 中标记为 (1) 的程序行注释掉，并验证其静态初始化子句并未被调用。然后再将标记为 (2) 的其中一行程序代码解除注释，并验证其静态初始化子句的确被调用了。最后再将标记为 (2) 的另一行程序代码解除注释，并验证其静态初始化子句只会执行一次。

第五章 隐藏具体实现

在面向对象设计中，要考虑的一个基本问题是“如何将变动的事物与保持不变的事物相互隔离”。

这对程序库（library）而言尤为重要。该程序库的使用者（客户端程序员，client programmer）必须能够信赖他所使用的那部分程序库，并且能够知道如果程序库出现了新版本，他们并不需要改写代码。从另一个方面来说，程序库的开发者必须有权限进行修改和改进，并确保客户代码不会因为这些改动而受到影响。

这一目标可以通过达成协议来加以实现。例如，程序库开发者必须同意在改动程序库中的 class 时不得删除任何现有方法，因为那样会破坏客户端程序员的代码。但是，与之相反的情况会更加棘手。在有域存在的情况下，程序库开发者要怎样才能知道究竟都有哪些域已经被客户端程序员所调用了呢？这对于方法仅为类的实现的一部分，因此并不想让客户端程序员直接使用的情况来说同样如此。但如果程序开发者想要移除旧的实现而要添加新的实现时，结果将会怎样呢？改动任何一个成员都有可能破坏客户端程序员的代码。于是程序库开发者会手脚被缚，无法对任何事物进行改动。

为了解决这一问题，Java 提供了访问权限修饰词(access specifier)供程序库开发人员来向客户端程序员指明哪些是可用的，哪些是不可用的。访问权限控制的等级，从最大权限到最小权限依次为：**public**, **protected**，包访问权限（没有关键词），和 **private**。根据前述内容，你可能会认为，作为一名程序库设计员，你会尽可能将一切方法都定为 **private**，而仅向客户端程序员公开你愿意让他们使用的方法。这样做是完全正确的，但是对于那些经常使用别的语言（特别是 c 语言）编写程序并在访问事物时不受任何限制的人而言，却是与他们的直觉相违背的。到了本章末，你将会信服 Java 的访问权限控制的价值。

不过，构件程序库（component library）的概念以及对于谁有权取用该程序库构件的控制都还是不完善的。其中仍旧存在着如何将构件捆绑到一个内聚的程序库单元中的问题。对于这一点，Java 用关键字 **package** 加以控制，而访问权限会因类是存在于一个相同的包，还是存在于一个单独的包而受到影响。为此，要开始学习本章，你首先要学习如何将程序库构件置于包中，然后你就会理解访问权限修饰词的全部含义。

包（package）：程序库单元

当你使用关键字 **import** 来导入整个程序库时，如：

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

这个包就变为可用的了。这将把作为标准的 Java 发布的一部分的整个 utility 程序库都引入到程序中来。例如，**java.util** 中有一个叫作 **ArrayList** 的类，你现在既可以用全称 **java.util.ArrayList** 来指定（这样你就不必使用 **import** 语句了），也可以仅指定为 **ArrayList**（缘于 **import**）。

如果你想引入一个单一的类，你可以在 **import** 语句中命名该类

```
import java.util.ArrayList;
```

现在，你就可以不受任何限定而直接使用 **ArrayList** 了。但是，这样做 **java.util** 中的其他类就都变为是不可用的了。

我们之所以要导入，就是要提供一个管理名字空间（name spaces）的机制。所有类成员的名称都是彼此隔离的。**A** 类中的方法 **f()** 与 **B** 类中具有相同参数列表(argument list)的方法 **f()** 不会彼此冲突。但是如果类名称相互冲突又该怎么办呢？假设你编写了一个 **Stack** 类并安装到了一台机器上，而该机器上已经有了一个别人编写的 **Stack** 类，我们该如何解决？名字之间的潜在冲突使得在 **java** 中对名称空间进行完全控制，并能够不受 Internet 的限制创建唯一的名称就成为了非常重要的事情。

到目前为止，书中大多数示例都存于单一文件之中，并专为本机使用（local use）而设计，因而尚未受到包名（package name）的干扰。（此时类名称被放于“缺省包”中）这当然也是一种选择，而且出于简易性的考虑，在本书其它任何部分都尽可能地使用了此方法。不过如果你正在准备编写能够与其他 **java** 程序在同一台机器上共存的 **java** 程序的话，你就需要考虑如何防止类名称之间的冲突。

如果你编写一个 **java** 源代码文件，此文件通常被称为编译单元（compilation unit）（有时也被称为转译单元（translation unit））。每个编译单元都必须有一个后缀名 **.java**，而在编译单元之中则可以有一个 **public** 类，该类的名称必须于文件的名称相同（包括大小写，但不包括文件的后缀名 **.java**）。每个编译单元只能有一个 **public** 类，否则编译器就不会接受。如果在该编译单元之中还有额外的类的话，那么在包之外的世界是无法看见这些类的，这是因为它们不是 **public** 类，而且它们主要是被用于为主 **public** 类提供支持。

当你编译一个 **.java** 文件时，在 **.java** 文件中每个类都会有一个输出文件，而该输出文件的名称与 **.java** 文件中每个类的名称又恰好相同，只是多了一个后缀名 **.class**。因此，你在编译少量 **.java** 文件之后，会得到大量的 **.class** 文件。如果你已经用编译型语言编写过程序，那么对于编译器产生一个中间文件（通常是一个“obj”文件），然后再与通过链接器（linker，用以创建一个可执行文件）或程序库产生器（librarian，用以创建一个程序库）产生的其它同类文件捆绑在一起的情况，你可能早已习惯。但这并不是 **java** 的工作方式。一个 **java** 可运行程序是一组可以打包并压缩为一个 **java** 文档文件（JAR，用 **Java** 的 **jar** 文档生成器）的 **.class** 文件。**Java** 解释器（interpreter）负责对这些文件的查找、装载和解释。¹

程序库实际上是一组类文件。其中每个文件都有一个 **public** 类（并非强制的，但这很典型），因此每个文件都是一个构件（component）。如果你希望这些构件（每一个都有它们自己分离开的 **.java** 和 **.class** 文件）从属于同一个群组，就可以使用关键字 **package**。

当你在文件起始处写道：

```
package mypackage;
```

就表示你在声明该编译单元是名为 **mypackage** 的程序库的一部分（如果使用了一个 **package** 语句，就它必须是文件中除注释以外的第一句程序代码）。或者，换种说法，你正在声明该编译单元中的 **public** 类名称是位于 **mypackage** 名称的遮蔽之下。任何想要使用该名称的人都必须指定全名或是在与 **mypackage** 的结合中使用关键字 **import**（使

¹ **java** 中并不强求必须要使用解释器。因为存在用来生成一个单一的可执行文件的本地代码 **java** 编译器。

用前面给出的选择)。请注意，Java 包的命名规则全部使用小写字母，包括中间的字也是如此。

例如，假设文件的名称是 **MyClass.java**，这就意味着在该文件中有且只有一个 **public** 类，该类的名称必须是 **MyClass**（注意大小写）：

```
package mypackage;
public class MyClass {
    // ...
}
```

现在，如果有人想用 **MyClass** 或者是 **mypackage** 中的任何其他 **public** 类，就必须使用关键字 **import** 来使得 **mypackage** 中的名称可以被使用。另一个选择是给出完整的名称：

```
mypackage.MyClass m = new mypackage.MyClass();
```

关键字 **import** 可使之更加简洁：

```
import mypackage.*;
// ...
MyClass m = new MyClass();
```

身为一名程序库设计员，很有必要牢记 **package** 和 **import** 关键字允许你做的，是将单一的全局名字空间分割开，使得无论多少人使用 Internet 并用 java 编写类，都不会出现名称冲突问题。

创建独一无二的包名

你也许会发现，既然一个包从未真正将被打包的东西包装成一个单一的文件，并且一个包可以由许多文件构成，那么情况就有点复杂了。为了避免这种情况的发生，一种合乎逻辑的作法就是将特定包的所有 **.class** 文件都置于一个单一目录之下。也就是说，利用操作系统的层次化的文件结构来解决这一问题。这是 java 解决混乱问题的一种方式，你还会在我们介绍 **jar** 工具的时候看到另一种方式。

将所有文件收入一个子目录还可以解决了另外两个问题：怎样创建独一无二的名称以及怎样查找有可能隐藏于目录结构中某处的类。正如我们在第二章中所介绍的那样，这些任务是通过将 **.class** 文件所在的路径位置编码成 **package** 名称来实现的。按照惯例，**package** 名称的第一部分是反顺序的类的创建者的 Internet 域名。如果你遵照惯例，Internet 域名应是独一无二的，因此你的 **package** 名称也将是独一无二的，也就不会出现名称冲突的问题了。（也就是说，只有在你将自己的域名给了别人，而他又以你曾经使用过的路径名称来编写 java 程序代码时，才会出现冲突。）当然，如果你没有自己的域名，你就得构造一组不大可能与他人重复的组合，例如你的姓名，来创立一些独一无二的 **package** 名称。如果你打算发布你的 java 程序代码，稍微花点力气去取得一个域名，还是很有必要的。

此技巧的第二部分是吧 **package** 名称分解为你机器上的一个目录。所以当 java 程序运行

并且需要加载.class 文件的时候(当需要为特定的类生成对象或首次访问类的 **static** 成员时, 程序会自动进行加载), 它就可以确定.class 文件在目录上的位置。

Java解释器(interpreter)的运行过程如下: 首先, 找出环境变量CLASSPATH²(可以通过操作系统, 有时也可通过在你的机器上安装用Java 或 Java-based 工具的安装程序来设置。) CLASSPATH包含一个或多个目录, 用来作为查找.class文件的根目录。从根目录开始, 解释器获取包的名称并将每个句点替换成反斜杠以从CLASSPATH 根中产生一个路径名称(于是, **package foo.bar.baz** 就变成 **foo\bar\baz** 或 **foo/bar/baz** 或其他什么可能的东西, 这一切取决于你的操作系统)。得到的路径会与CLASSPATH中的各个不同的项相连接, 解释器就在这些目录中查找与你所要创建的类相关的名称的.class文件。(解释器还会去查找某些相对于它所在位置的标准目录)。

为了理解这一点, 以我的域名 **bruceeckel.com** 为例。把它的顺序倒过来, **bruceeckel.com** 就成了我所创建的类的一个别具一格的名称。(com, edu, org 等扩展名先前在 java 包中都是大写的, 但在 java2 中一切都已改观, 包的整个名称全都变成了小写。)若我决定再创建一个名为 **simple** 的程序库, 我可以将该名称进一步细分, 于是我可以得到一个包的名称:

```
package com.bruceeckel.simple;
```

现在, 这个包名称就可以用作下面两个文件的名称空间保护伞了:

```
//: com:bruceeckel:simple:Vector.java
// Creating a package.
package com.bruceeckel.simple;

public class Vector {
    public Vector() {
        System.out.println("com.bruceeckel.simple.Vector");
    }
} ///: ~
```

你在创建自己的包时, 将会发现 **package** 语句必须是文件中的第一行非注释程序代码。第二个文件看起来也极其相似:

```
//: com:bruceeckel:simple:List.java
// Creating a package.
package com.bruceeckel.simple;

public class List {
    public List() {
        System.out.println("com.bruceeckel.simple.List");
    }
} ///: ~
```

² 当提及环境变量时, 将用到大写字母(CLASSPATH)。

这两个文件均被置于我的系统的子目录下：

```
C:\DOC\JavaT\com\bruceeckel\simple
```

如果你沿此路径向回看，你可以看到包的名称 **com.bruceeckel.simple**，但此路径的第一部分怎么办呢？它将由环境变量 CLASSPATH 关照，在我的机器上是：

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

你可以看到，CLASSPATH 可以包含多个可选择的查询路径。

但在使用 JAR 文件时会有一点变化。你得在 classpath 中将 JAR 文件的名称写清楚，而不仅是指明它所在位置的目录。因此，对于一个名为 **grape.jar** 的 JAR 文件，你的 classpath 应包括：

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

一旦 classpath 得以正确建立，下面的文件就可以放于任何目录之下：

```
//: c05:LibTest.java
// Uses the library.
import com.bruceeckel.simpletest.*;
import com.bruceeckel.simple.*;

public class LibTest {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Vector v = new Vector();
        List l = new List();
        monitor.expect(new String[] {
            "com.bruceeckel.simple.Vector",
            "com.bruceeckel.simple.List"
        });
    }
} ///: ~
```

当编译器碰到要导入 **simple** 类库的 **import** 语句时，就开始在 CLASSPATH 所指定的目录中查找，以查找子目录 **com\bruceeckel\simple**，然后从已编辑的文件中找出名称相符者（对 **Vector** 而言是 **Vector.class**，对 **List** 而言是 **List.class**）。请注意，**Vector** 和 **List** 中的类以及要使用的方法都必须是 **public** 的。

对于使用java的新手而言，设立CLASSPATH是如此麻烦的一件事（在我最初使用时是这样的）。为此Sun将java2 中的JDK被改造得更聪明了一些。在安装后，你会发现，即使你未设立CLASSPATH，你也可以编辑并运行基本的java程序。然而，要编辑和运行本书的

源码包（从www.BruceEckel.com网站可以取得），你就得向你的CLASSPATH中添加本书程序代码树中的基目录了。

冲突

如果将两个含有相同名称的程序库以‘*’形式同时导入，将会出现什么情况呢？例如，假设某程序这样写到：

```
import com.bruceeckel.simple.*;
import java.util.*;
```

由于 **java.util.*** 也含有一个 **Vector** 类，这就存在潜在的冲突。但是只要你不写那些导致冲突的程序代码，就不会有什么问题——这样很好，否则你就得做很多的类型检查工作来防止那些根本不会出现的冲突。

如果你现在要创建一个 **Vector** 类的话，就会产生冲突：

```
Vector v = new Vector();
```

这行到底取用的是哪个 **Vector** 类？编译器不知道，读者同样也不知道。于是编译器提出错误信息，强制你必须指明。举例说明，如果我想要一个标准的 Java **Vector** 类，我就得这样写：

```
java.util.Vector v = new java.util.Vector();
```

由于这样可以完全指明该 **Vector** 类的位置（配合 CLASSPATH），所以除非我还要使用 **java.util** 中的其他东西，否则我就没有必要写 **import java.util.*** 语句了。

定制工具库

具备了这些知识以后，你现在就可以创建自己的工具库来减少或消除重复的程序代码了。例如，要给 **System.out.println()** 创建一个别名以减少输入负担。下面这个类可以作为名为 **tools** 的包的一部分：

```
//: com:bruceeckel:tools:P.java
// The P.rint & P.rintln shorthand.
package com.bruceeckel.tools;

public class P {
    public static void rint(String s) {
        System.out.print(s);
    }
    public static void rintln(String s) {
        System.out.println(s);
    }
}
```



```
    }
} ///:~
```

你可以使用这个便捷工具来打印 **String**，无论它是否需要换行(**P.println()**)。

你可以猜到，这个文件的位置一定是在某个以一个 **CLASSPATH** 位置开始，然后接着是 **com/bruceeckel/tools** 的目录下。编译完之后，你就可以用 **import** 语句在你系统上的任何地方使用 **P.class** 文件了。

```
//: c05:ToolTest.java
// Uses the tools library.
import com.bruceeckel.tools.*;
import com.bruceeckel.simpletest.*;

public class ToolTest {
    static Test monitor = new Test();
    public static void main(String[] args) {
        P.println("Available from now on!");
        P.println("" + 100); // Force it to be a String
        P.println("" + 100L);
        P.println("" + 3.14159);
        monitor.expect(new String[] {
            "Available from now on!",
            "100",
            "100",
            "3.14159"
        });
    }
} ///:~
```

请注意，只要将所有对象放于 **String** 表达式之中，就可以轻易地将它们强行转换为 **String** 的表达形式。在前例中，以一个空 **String** 开始的表达式就是这种方法。但是这样做引出了一个有趣的现象。如果你调用 **System.out.println(100)**，它无需将 100 转型成 **String** 就可以工作。通过某些额外的重载，你也可以用 **P** 类达到同样的目的。（这在本章末将作为一个练习提出。）

因此从现在开始，只要你实现了某个新的有用的工具，你就可以把它添加到自己的 **tools** 目录或是 **util** 目录之下。

用 **imports** 改变行为

Java 没有 c 的“条件编译 (conditional compilation)”功能，该功能可以使你不必更改任何程序代码，就能够切换开关并产生不同的行为。**java** 去掉此功能的原因可能是因为 c 在绝大多数情况下是用此功能来解决跨平台问题的，即程序代码的不同部分是根据不同的平台

来编译的。由于 **java** 自身可以自动跨越不同的平台，因此这个功能对 **java** 而言是没有必要的。

然而，条件编译还有其他一些有价值的用途。调试就是一个很常见的用途。调试功能在开发过程中是使能的，而在打包的产品中是被禁止的。你可以通过修改被引入的 **package** 的方法来实现这一目的，修改的方法是将在你程序中用到的代码从调试版改为发布版。这一技术可以适用于任何种类的条件代码。

对使用包（**Package**）的忠告

务必记住，无论何时你创建包，你都已经在给定包的名称的时候隐含地指定了目录结构。这个包必须位于其名称所指定的目录之中，而该目录必须是在以 **CLASSPATH** 开始的目录中可以查询到的。最初用关键字 **package** 做实验，可能会有一点不顺，因为除非你遵守“包的名称对应目录路径”的规则，否则你将会受到许多出乎意料的运行时信息，告知无法找到特定的类，哪怕是这个类就位于同一个目录之中。如果你收到类似信息，就用注释掉 **package** 语句的方法试一下，如果这样可行的话，你就可以知道问题出在哪里了。

Java 访问权限修饰词（**access specifier**）

public, **protected** 和 **private** 这几个 **java** 访问权限修饰词在使用时，是置于你的类中每个成员的定义之前的，无论它是一个域或是一个方法。每个访问权限修饰词仅控制它所修饰的特定定义的访问权。这一点与 **c++** 截然不同。在 **c++** 中，访问权限修饰词可以控制其后的所有定义，除非另有访问权限修饰词出现。

无论如何，每个定义都需要某种为它指定的访问权限。在以下几节中，你将学习各类访问权限，首先是默认的访问权限。

包访问权限

如果你根本没有给定任何访问权限，例如像本章前面的所有示例，将会出现什么情况呢？默认访问权限没有任何关键字，但通常是指包访问权限（*package access*，有时也表示成为“friendly”）。这就意味着当前的包中的其他类对那个成员有访问权限，但对于在这个包之外的所有类，这个成员却是 **private**。由于一个编译单元，即一个文件，只能隶属于一个单一的包，所以经由包访问权限，处于某个单一编译单元之中的所有类彼此之间都是自动可访问的。

包访问权限允许你将包内所有相关的类组合起来，以使它们彼此之间可以轻松地相互作用。等你把类组织起来放进一个包内之后，也就给它们的包访问权限的成员赋予了相互访问的权限，你拥有了该包内的程序代码。“只有你拥有的程序代码才可以访问你所拥有的其他程序代码”这种想法是合理的。你应该说，包访问权限为把类群聚在一个包中的做法提供了意义和理由。在许多语言中，你在文件内组织定义的方式是任意的，但在 **java** 中，则要强制你

以一种合理的方式对它们加以组织。另外，你可能还想要排除这样的类：它们不应该访问在当前包中所定义的类。

类掌握着控制哪些代码对自己的成员享有访问权的权力。不存在任何获得访问权的其它捷径。其他包内的类不能一出现就说：“嗨，我是 **Bob** 的朋友。”，并且还想看到 **Bob** 的 **protected**，包访问权限和 **private** 成员。取得对某成员的访问权的唯一途径是：

1. 使该成员成为 **public**。于是，无论是谁，无论在哪里，都可以访问该成员。
2. 通过不加访问权限修饰词并将其他类放置于同一个包内的方式给成员赋予包访问权。于是包内的其他类也就可以访问该成员了。
3. 第 6 章将会介绍到继承技术，届时你将会看到继承而来的类可以与访问 **public** 成员一样地访问 **protected** 成员（但访问 **private** 成员却不行）。只有在两个类都处于同一个包内时，它才可以访问包访问权限的成员。但现在不必担心了。
4. 提供访问器（accessor）和变异器（mutator）方法（也称作“get/set”方法），以读取和改变数值。如你在第 4 章中看到的一样，对 OOP 而言，这是最优雅的方式，而对 JavaBeans 来说，也是它的基本原理。

public:接口访问权限

当你使用关键字 **public**，就意味着 **public** 之后紧跟着的成员声明对每个人都是可用的，尤其是使用程序库的客户端程序员更是如此。假设你定义了一个包含下面编译单元的 **dessert** 包：

```
//: c05:dessert:Cookie.java
// Creates a library.
package c05.dessert;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    void bite() { System.out.println("bite"); }
} ///:~
```

记住，**Cookie.java** 产生的类文件必须位于名为 **dessert** 的子目录之中，该子目录在 **c05**（参阅本书第 5 章）之下，而 **c05** 则必须位于 CLASSPATH 指定的众多路径的其中之一底下。不要错误地认为 java 总是将当前目录视作是查找行为的起点之一。如果你的 CLASSPATH 之中缺少一个 '.' 作为路径之一的話，java 就不会查找那里。

现在如果你创建了一个使用 **Cookie** 的程序：

```
//: c05:Dinner.java
// Uses the library.
import com.bruceeckel.simpletest.*;
```

```
import c05.dessert.*;

public class Dinner {
    static Test monitor = new Test();
    public Dinner() {
        System.out.println("Dinner constructor");
    }
    public static void main(String[] args) {
        Cookie x = new Cookie();
        //! x.bite(); // Can't access
        monitor.expect(new String[] {
            "Cookie constructor"
        });
    }
} /// ~
```

你就可以创建一个 **Cookie** 对象, 因为 **Cookie** 的构造器是 **public** 而其自身也是 **public** 的。(此后我们将会对 **public** 类的概念了解更多。)但是, 由于 **bite()** 只向在 **dessert** 包中的类提供访问权, 所以 **bite()** 成员在 **Dinner.java** 之中是无法访问的, 为此编译器也禁止你使用它。

缺省包 (default package)

你会惊异地发现下面的程序代码虽然看起来是破坏了上述规则, 但它仍可以进行编译。

```
/// c05:Cake.java
// Accesses a class in a separate compilation unit.
import com.bruceeckel.simpletest.*;

class Cake {
    static Test monitor = new Test();
    public static void main(String[] args) {
        Pie x = new Pie();
        x.f();
        monitor.expect(new String[] {
            "Pie.f()"
        });
    }
} /// ~
```

在第二个处于相同目录的文件中:

```
/// c05:Pie.java
// The other class.
```

```
class Pie {
    void f() { System.out.println("Pie.f()"); }
} ///:~
```

最初你或许会认为这两个文件毫不相关，但 **Cake** 却可以创建一个 **Pie** 对象并调用它的 **f()** 方法！（记住，为了使文件可以被编译，在你的 CLASSPATH 之中一定要有‘.’。）你通常会认为 **Pie** 和 **f()** 享有包访问权限，因而是不可以为 **Cake** 所用的。它们的确享有包访问权限，但这只是部分正确的。**Cake.java** 可以访问它们的原因是因为它们同处于一个相同的目录并且没有给自己设定任何包名称。**Java** 将这样的文件自动看作是隶属于该目录的缺省包之中，于是它们为该目录中所有其他的文件都提供了包访问权限。

private: 你不可以去碰！

关键字 **private** 的意思是，除了包含该成员的类之外，其他任何类都是无法访问这个成员的。由于处于同一个包内的其他类是不可以访问 **private** 成员的，因此这等于说是自己隔离了自己。从另一方面说，让许多人共同合作来创建一个包也是不大可能的，为此 **private** 就允许你随意改变该成员，而不必考虑这样做是否会影响到包内其他的类。

缺省的包访问权限通常已经提供了充足的隐藏措施。请记住，使用类的客户端程序员是无法访问包访问权限成员的。这样做很好，因为缺省访问权限是一种我们经常使用的权限，同时也是一种在你忘记添加任何访问权限控制时能够自动得到的权限。因此，你通常会考虑哪些成员是想要明确公开给客户端程序员使用的，从而将它们声明为 **public**，而在最初，你可能不会认为你会需要使用关键字 **private**，因为没有它，照样可以工作。（这在 C++ 之中是截然相反的。）然而，事实很快就会证明，对 **private** 的使用是多么的重要，在多线程环境（multithreading）下更是如此。（正如你在第 13 章中将会看到的一样。）

此处有一个使用 **private** 的示例。

```
///: c05:IceCream.java
// Demonstrates "private" keyword.

class Sundae {
    private Sundae() {}
    static Sundae makeASundae() {
        return new Sundae();
    }
}

public class IceCream {
    public static void main(String[] args) {
        //! Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
}
```

```
} ///: ~
```

这是一个说明**private**终有其用的示例：你可能想要控制如何去创建对象，并阻止别人直接访问某个特定的构造器（或是全部构造器）。在上面的例子中，你不能通过构造器来创建**Sundae**对象，而必须调用**makeASundae()**方法来达到此目的。³

任何你确认为只是该类的一个“助手”方法的方法，你都可以把它指定为 **private**，以确保你不会在包内的其他地方误用到它，于是也就防止了你会去改变或是删除这个方法。将方法指定为 **private** 确保了你拥有这种选择权。

这对于类中的 **private** 域同样适用。除非你必须公开底层实现细目（此种境况很少见），否则你就应该将所有的域指定为 **private**。然而，不能因为在类中某个对象的引用是 **private**，就认为其他的对象无法拥有该对象的 **public** 引用。（参见附录 A 中关于别名机制的讨论）。

Protected: 继承访问权

要理解 **protected** 的访问权限，我们在内容上需要作一点跳跃。首先，你要知道在本书介绍继承（第 6 章）之前，你并不用真正理解本节的内容。但为了内容的完整性，这里还是提供了一个使用 **protected** 的简要介绍和示例。

关键字 **protected** 处理的是一个被称为继承（inheritance）的概念，借助此项技术，我们可以获取一个现有类——我们将其作为基类引用，然后将新成员添加该现有类中，而不必触及这个现有类。你还可以改变该类的现有成员的行为。为了从现有类中继承，你要声明你的新类 **extends**（扩展）了一个现有类，就象这样：

```
class Foo extends Bar {
```

至于类定义中的其他部分看起来都是一样的。

如果你创建了一个新包，并继承自另一个包中的某个类，那么你唯一可以访问的成员就是源包的 **public** 成员。（当然，如果你在同一个包内执行继承工作，你就可以操纵所有的拥有包访问权限的成员）。有时，基类的创建者会希望获得一个特定的成员，并赋予派生类，而不是所有类以访问权。这就需要 **protected** 来完成这一工作。**protected** 也提供包访问权限，也就是说，相同包内的其他类可以访问 **protected** 元素。

如果你回顾一下先前的例子 **Cookie.java**，就可以得知下面的类是不可以调用拥有包访问权限的成员 **bite()** 的：

```
///: c05:ChocolateChip.java
/// Can't use package-access member from another package.
import com.bruceeckel.simpletest.*;
```

³ 此例还有另一个效果：既然缺省构造器是唯一定义的构造器，并且它是**private**的，那么它将阻碍对此类的继承。（在第 6 章我们将介绍这个问题。）

```
import c05.dessert.*;

public class ChocolateChip extends Cookie {
    private static Test monitor = new Test();
    public ChocolateChip() {
        System.out.println("ChocolateChip constructor");
    }
    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        //! x.bite(); // Can't access bite
        monitor.expect(new String[] {
            "Cookie constructor",
            "ChocolateChip constructor"
        });
    }
} ///:~
```

有关于继承技术的一个很有趣的事情是如果类 **Cookie** 之中存在一个方法 **bite()** 的话，那么该方法同时也存在于任何一个从 **Cookie** 继承而来的类之中。但是由于 **bite()** 有包访问权限而且它位于另一个包内，所以我们在这个包内是无法使用它的。当然，你也可以把它指定为 **public**，但是这样做所有的人就都有了访问权限，而且很可能这并不是你所希望的。如果我们将类 **Cookie** 象这样加以更改：

```
public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    protected void bite() {
        System.out.println("bite");
    }
}
```

那么，**bite()** 在包 **dessert** 之中仍旧拥有包访问权限，但是它对于所有继承自 **Cookie** 的类而言，也是可以使用的。然而，它不是 **public** 的。

接口（Interface）和实现（implementation）

访问权限的控制常被视作是“具体实现的隐藏（implementation hiding）”。把数据和方法包装进类中，与具体实现的隐藏结合到一起，常被称作是“封装（encapsulation）”⁴。其结果是一个同时带有特征和行为的数据类型。

⁴ 然而，人们经常只单独将具体实现的隐藏称作封装。

出于两个很重要的原因，访问权限控制将权限的边界划在了数据类型的内部。第一个原因是设定客户端程序员可以使用和不可以使用的界限。你可以在结构中建立自己的内部机制，而不必担心客户端程序员会偶然地将内部机制当作是他们可以使用的接口的一部分。

这个原因直接引出了第二个原因，即将接口和具体实现进行分离。如果结构是用于一组程序之中，而客户端程序员除了可以向接口发送信息之外什么也不可以做的，那么你就可以随意更改所有不是 **public** 的东西（例如包访问权限，**protected** 和 **private**），而不会破坏客户端代码。

我们现在正处于一个面向对象的编程世界中。在这里 **class** 事实上被描绘成是“对象的集群”，就象你描述一群鱼或是一群鸟一样。隶属于这个类的所有对象将都具有这些特征和行为。类是对所有同类型对象的外观和行为进行描述的方式。

在最初的 OOP 语言 Simula-67 中，关键字 **class** 用于描述一种新的数据类型。这一关键字也被绝大多数面向对象语言所应用。这正是整个语言的关键点：创建新数据类型，而不仅仅只是创建含有数据和方法的盒子。

在 **java** 中，类是一个基本的 OOP 概念。类是本书中不用黑体书写的关键字之一，因为用黑体书写象类这样一个重复率如此之高的词会很繁琐。

为了清楚起见，你可能会采用一种将 **public** 成员置于开头，后面跟着 **protected**，包访问权限和 **private** 成员的创建类的形式。这样做的好处是类的使用者可以从头读起，首先阅读对他们而言最为重要的部分（即 **public** 成员，因为它们可以从文件外部被调用），等到遇见作为内部实现细节的非 **public** 成员时停止阅读：

```
public class X {
    public void pub1() { /* ... */ }
    public void pub2() { /* ... */ }
    public void pub3() { /* ... */ }
    private void priv1() { /* ... */ }
    private void priv2() { /* ... */ }
    private void priv3() { /* ... */ }
    private int i;
    // ...
}
```

这样作仅能使程序略为便于阅读，因为接口和具体实现仍旧混在一起。也就是说，你仍能看到源代码——实现部分，因为它就在类中。另外，**javadoc** 所提供的“寓文档于注解之中”的功能（第 2 章中介绍过）降低了程序代码的可读性对于客户程序员的重要性。将接口展现给某个类的使用者实际上是类浏览器（**class browser**）的任务。类浏览器是一种以非常有用的方式来查阅所有可用的类，并告诉你用它们可以做些什么（也就是显示出可用成员）的工具，它已成为任何优良的 **java** 开发工具所不可或缺的部分。

类的访问权限

在 java 中，访问权限修饰词也可以用于确定在某个程序库中的类哪些对于该库的使用者是可用的。如果你希望某个类可以为某个客户端程序员所用，你就可以通过把关键字 **public** 作用于整个类的定义来达到目的。这样做甚至可以控制客户端程序员是否能创建一个该类的对象。

为了控制某个类的访问权限，修饰词必须出现于关键字 **class** 之前。因此你可以声明：

```
public class Widget {
```

现在如果你的类库的名字是 **mylib**，那么任何客户端程序员都可以通过下面的声明访问 **Widget**

```
import mylib.Widget;
```

或

```
import mylib.*;
```

然而，这里还有一些额外的限制：

1. 每个编译单元（文件）都只能有一个 **public** 类。这表示，每个编译单元都有一个单一的公共接口，用 **public** 类来表现。该接口可以按照你的希望去包含众多的支持包访问权限的类。如果在某个编译单元内有一个以上的 **public** 类，编译器就会向你给出一个出错的信息。
2. **public** 类的名称必须完全与含有该编译单元的文件名相匹配，包括大小写。所以对于 **Widget** 而言，文件的名称必须是 **Widget.java**，而不是 **widget.java** 或 **WIDGET.java**。如果不匹配，你将会再次得到编译时间出错的提示。
3. 虽然不是很常用，但编译单元内完全不带 **public** 类也是可能的。在这种情况下，你可以随意对文件命名。

如果你获取了一个在 **mylib** 内部的类，用来完成 **Widget** 或是其他在 **mylib** 中的 **public** 类所要执行的任务，将会出现什么样的情况呢？你不想自找麻烦去为客户端程序员创建说明文档，而且你认为不久你可能会想要完全改变原有方案并将旧版本一起删除，代之以一种不同的版本。为了保留此灵活性，你需要确保客户端程序员不会依赖于你隐藏在 **mylib** 之中的任何特定实现细节。为了达到这一点，你只需将关键字 **public** 从类中拿掉，它就拥有了包访问权限。（这个类只可以用于该包之中。）

在你创建一个包访问权限的类时，它仍旧在将该类的域声明为 **private** 的时才具有意义——你应尽可能地总是将域指定为是私有的——但是通常来说，将与类（包访问权）相同的访问权限赋予方法也是很合理的。既然一个包访问权限的类通常只能被用于包内，那么如果

对你有强制要求，在此种情况下，编译器会告诉你，你只需要将这样的类的方法设定为 **public** 就可以了。

请注意，类既不可以是**private**的（这样会使得除该类之外再无其他类可以访问它）也不可以是**protected**的⁵。所以对于类的访问权限，你仅有两个选择：包访问权或是**public**。如果你不希望其他任何人对该类拥有访问权限，你可以把所有的构造器都指定为**private**，从而阻止任何人创建该类的对象，但是有一个例外，就是在该类的**static**成员内部可以创建。下面是一示例：

```
//: c05:Lunch.java
// Demonstrates class access specifiers. Make a class
// effectively private with private constructors:

class Soup {
    private Soup() {}
    // (1) Allow creation via static method:
    public static Soup makeSoup() {
        return new Soup();
    }
    // (2) Create a static object and return a reference
    // upon request.(The "Singleton" pattern):
    private static Soup ps1 = new Soup();
    public static Soup access() {
        return ps1;
    }
    public void f() {}
}

class Sandwich { // Uses Lunch
    void f() { new Lunch(); }
}

// Only one public class allowed per file:
public class Lunch {
    void test() {
        // Can't do this! Private constructor:
        //! Soup priv1 = new Soup();
        Soup priv2 = Soup.makeSoup();
        Sandwich f1 = new Sandwich();
        Soup.access().f();
    }
} ///:~
```

⁵ 事实上，一个内部类可以是**private**或是**protected**的，但那是一个特例。这将在第 7 章中介绍到。

到目前为止，绝大多数方法均返回 **void** 或是原始类型，所以此定义：

```
public static Soup access() {  
    return ps1;  
}
```

初看起来可能有点令人迷惑不解。方法名称(**access**)前面的词告知了该方法返回的东西。到目前为止，这个方法经常被用作 **void**，所以它不返回任何东西。但是你也可以返回一个对象引用，示例中就是这种情况。这个方法返回了一个对 **Soup** 类的对象的引用。

Soup 类展示了如何通过将所有构造器指定为 **private** 的方法来阻止直接创建某个类的实例。请一定要牢记如果你没有显示地至少创建一个构造器的话，就会帮你创建一个缺省构造器（不带有参数的构造器）。如果我们自己编写了缺省的构造器，那么就不会自动创建它了。如果把该构造器指定为 **private**，那么就谁也无法创建该类的对象了。但是现在别人该怎样使用这个类呢？上面的例子就给出了两个选择：第一，创建一个 **static** 方法，来生成一个新的 **Soup** 并返回一个对它的引用。如果你想要在返回引用之前在 **Soup** 上作一些额外的工作，或是如果你想要记录到底创建了多少个 **Soup** 的对象（可能要限制其数量），这种做法将会是大有裨益的。

第二个选择用到了一个设计模式（design pattern），该模式在 www.BruceEckel.com 网站 *Thinking in Patterns (with Java)* 一书中有所介绍。这种特定的模式被称为“singleton”，这是因为你始终只能创建它的单一的一个对象。**Soup** 类的对象是作为 **Soup** 的一个 **static private** 成员而生成的，所以有且仅有一个，而且你除非是通过 **public** 方法 **access()**，否则你是无法访问到它的。

正如前面所提到的，如果你没能为类访问权限指定一个访问修饰符，它就会缺省得到包访问权限。这就意味着该类的对象可以由包内任何其他类来创建，但在包外则是不行的。（一定要记住，相同目录下的所有不具有明确 **package** 声明的文件，都被视作是该目录下缺省包的一部分。）然而，如果该类的某个 **static** 成员是 **public** 的话，则客户端程序员仍旧可以调用该 **static** 成员，哪怕是他们并不能生成该类的对象。

总结

无论是在什么样的关系之中，设立一些为各成员所遵守的界限始终是很重要的。当你创建了一个程序库，你也就与该程序库的用户建立了某种关系，这些用户就是客户端程序员，他们是另外一些程序员，他们将你的程序库聚合成为一个应用程序，或是运用你的程序库来创建一个更大的程序库。

如果不制定规则，客户端程序员就可以使用某类的所有成员来随心所欲，而你可能并不希望他们径直复制其中的一些成员。在这种情况下，所有事物都要公开的。

本章讨论了类是如何被构建成程序库的：首先，介绍了一组类是如何被打包到一个程序库中的；第二，类是如何对它的成员进行访问权限控制的。

据估计，用 c 语言开发项目，在 50k 到 100k 的代码行之间的某个地方就会出现这个问题。这是因为 c 语言仅有一个单一的“名字空间”：名称开始发生冲突，引发额外的管理开销。而对于 java，关键字 **package**、包的命名架构和关键字 **import** 可以使你对名称进行完全的控制，因此诸如名称冲突的问题是很容易避免的。

进行对成员的访问权限控制有两个原因。第一个原因是为了使用户不要碰触那些他们不该碰触的工具：工具对于数据类型内部的操作是很有必要的，但是它并不属于用户在解决他们的特定的问题时所需接口的一部分。因此，将方法和域指定成 **private**，对用户而言是一种服务。因为这样他们可以很清楚地看到什么对他们重要，什么是他们可以忽略的。这样简化了他们对类的理解。

对访问权限进行控制的第二个原因，也是最重要的原因，是为了让程序库设计者可以更改类的内部工作方式，而不必担心这样会对客户端程序员产生多大的影响。最初你可能会以某种方式创建一个类，然后发现如果更改程序结构，可以大大提高运行速度。如果接口和实现可以被明确地加以隔离和保护，你就可以实现这一目的，而不必强制用户重新编写代码。

Java 的访问权限修饰符对类的创建者给予极具价值的控制力量。类的用户可以清晰地看到到底什么是他们可以用的，什么又是他们可以会忽略的。但是，确保所有用户都具有能够独立于某个类的底层实现细节的能力才是更为重要的。作为该类的创建者，如果你了解了这一点，你就可以按照自己的意愿改变底层实施细节，因为你知道客户端程序员不会因为这些改变而受到影响，他们也不可能访问到该类的那一部分。

当你具备了改变底层实施细节的能力时，你就可以随意地将你的设计升级，同时你也就具有了犯错的可能性。无论你是如何细心地计划并设计，你都有可能犯错。当你了解到你所犯错误是相对安全的时候，你就可以更加放心地进行实验，也就可以更快地学会，更快地完成你的项目。

类的 **public** 接口是用户真正能够看到的，所以这一部分是在分析和设计的过程中决定该类是否正确的重要组成部分。尽管如此，你仍然有进行改变的空间。如果你在最初无法创建出正确的接口，那么只要你不删除任何客户端程序员在他们的程序中已经用到的东西，你就可以添加更多的方法。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 编写一个用来创建一个 **ArrayList** 对象的程序，但是不明确导入 **java.util.***。
2. 在标题为“包:程序库单元”的一节中，将关于 **mypackage** 的程序代码片段，改写为一组可编译、可运行的 java 文件。
3. 在标题为“冲突”的一节中，将其代码片段改写为完整的程序，并校验实际所发生的冲突。
4. 添加 **rint()** 和 **rintln()** 的所有重载版本，以处理所有不同的 java 基本类型，从而使本章中所定义的类型 **P** 通用化。

5. 创建一个带有 **public**, **private**, **protected**, 和包访问权限域以及方法成员的类。创建该类的一个对象，看看在你试图调用所有类成员时，会得到什么类型的编译信息。请注意，处于同一个目录中的所有类都是缺省包的一部分。
6. 创建一个带有 **protected** 数据的类。运用在第一个类中处理 **protected** 数据的方法在相同的文件中创建第二个类。
7. 按照标题为“protected: 继承访问权限”一节中的指示去修改 **Cookie** 类。验证 **bite()** 不是 **public** 的。
8. 在标题为“类的访问权限”的一节中，你会发现描绘 **mylib** 和 **Widget** 的代码片段。请完成这个程序库，然后在位于 **mylib** 包之外的类中创建一个 **Widget**。
9. 创建一个目录，然后对 **CLASSPATH** 进行编辑，使之包括该新目录。把文件 **P.class**（由编辑 **com.bruceeckel.tools.P.java** 而得）复制到新目录之中并更改该文件的名称、**P** 类的内部以及方法的名称。（你也可能想要添加其他的输出信息，以观察其工作方式。）在另一个目录下再编写一个使用这个新类的程序。
10. 效仿示例 **Lunch.java** 的形式，创建一个名为 **ConnectionManager** 的类。该类管理一个 **Connection** 对象的固定数组。客户端程序员不能够直接创建 **Connection** 对象，而只能通过 **ConnectionManager** 中的某个 **static** 方法来获取它们。当 **ConnectionManager** 之中不再有对象时，它会返回 **null** 引用。在 **main()** 之中检测这些类。
11. 在 **c05/local** 目录下编写以下文件（假定 **c05/local** 目录在你的 **CLASSPATH** 中）：

```
// c05:local:PackagedClass.java
package c05.local;
class PackagedClass {
    public PackagedClass() {
        System.out.println("Creating a packaged class");
    }
}
```

然后在 **c05** 之外的另一个目录中创建下列文件：

```
// c05:foreign:Foreign.java
package c05.foreign;
import c05.local.*;
public class Foreign {
    public static void main (String[] args) {
        PackagedClass pc = new PackagedClass();
    }
}
```

解释一下为什么编译器会产生错误。如果将 **Foreign** 类置于 **c05.local** 包之中的话，会有所改变吗？

第六章 复用类

复用程序代码是 Java 众多引人注目的功能之一。但要想成为极具革命性的语言，仅是能够复制程序代码并对之加以改变还是不够的，它还必须能够做更多的事情。

上述方法常为 C 这类过程型（procedural）语言所使用，但收效并不是很好。正如 Java 中所有事物一样，问题解决都是围绕着类（class）展开的。你可以通过创建新类来复用程序代码，而不必再重头开始编写。你可以使用别人业已开发并调试好的类。

此方法中的窍门在于使用类而不破坏现有程序代码。你将会在本章中看到两种达到这一目的的方法。第一种方法非常直观：你只需在新的类中产生现有类的对象。由于新的类是由现有类的对象所组成，所以这种方法被称为组合（composition）。该方法只是复用了现有程序代码的功能，而非它的形式。

第二种方法则更细致一些，它按照现有类的类型来创建新类。你无需改变旧有类的形式，仅仅只是采用它的形式并在其中添加新代码。这种神奇的方式被称为继承（inheritance），而且编译器可以完成其中大部分工作。继承是面向对象程序的基石之一，我们将在第七章探究其含义与功能。

就组合（composition）和继承（inheritance）而言，其语法和行为大多是相似的。由于它们是利用现有类型生成新类型，所以这样做极富意义。在本章中，你将会了解到这两种程序代码重用机制。

组合（composition）语法

到目前为止，本书已多次使用组合技术。你仅需将对象引用置于新类之中即可。例如，假设你需要某个对象，它需要具有多个 string 对象、两三个基本类型数据、以及另一个类的对象。对于非基本类型的对象，你必须将其引用置于新的类中，而现在，你只需直接定义基本类型数据：

```
//: c06:SprinklerSystem.java
// Composition for code reuse.
import com.bruceeckel.simpletest.*;

class WaterSource {
    private String s;
    WaterSource() {
        System.out.println("WaterSource()");
        s = new String("Constructed");
    }
    public String toString() { return s; }
}
```

```

public class SprinklerSystem {
    private static Test monitor = new Test();
    private String valve1, valve2, valve3, valve4;
    private WaterSource source;
    private int i;
    private float f;
    public String toString() {
        return
            "valve1 = " + valve1 + "\n" +
            "valve2 = " + valve2 + "\n" +
            "valve3 = " + valve3 + "\n" +
            "valve4 = " + valve4 + "\n" +
            "i = " + i + "\n" +
            "f = " + f + "\n" +
            "source = " + source;
    }
    public static void main(String[] args) {
        SprinklerSystem sprinklers = new SprinklerSystem();
        System.out.println(sprinklers);
        monitor.expect(new String[] {
            "valve1 = null",
            "valve2 = null",
            "valve3 = null",
            "valve4 = null",
            "i = 0",
            "f = 0.0",
            "source = null"
        });
    }
} //:~

```

在上面两个类所定义的方法中，有一个很特殊：**toString()**。不久你将会了解到每一个非基本类型的对象都有一个 **toString()** 方法，而且当编译器需要一个 **string** 而你却只有一个对象时，该方法便会被调用。所以在 **sprinklerSystem.toString()** 的表达式中：

```
"source = " + source;
```

编译器将会得知你想要将一个 **string** 对象同 **watersource** 相加。由于你只能将一个 **string** 和另一个 **string** 相加，因此编译器会告诉你：“我将调用 **toString()**，把 **source** 转换成为一个 **string**！”这样做之后，它就能够将两个 **string** 连接到一起并将结果传递给 **System.out.println()**。每当你想要使你所创建的类具备这样的行为时，你仅需要编写一个 **toString()** 方法即可。

正如我们在第 2 章中所提到的，类中的基本类型数据能够自动被初始化为零。但是对象引用会被初始化为 `null`，而且如果你试图为它们调用任何方法，都会得到一个异常 (`exception`)。如果我们可以在不出现异常的前提下将其内容打印出来，将会是件有益并且有用的事情。

编译器并不是简单地为一个引用都创建缺省对象，这一点是很有意义的，因为真要是那样做的话，就会在许多情况下增加不必要的负担。如果你想初始化这些引用，可以在代码中的下列位置进行：

1. 在定义对象的地方。这意味着它们总是能够在构造器被调用之前被初始化。
2. 在类的构造器中。
3. 就在你确实需要使用这些对象之前。这种方式被称为“惰性初始化 (`lazy initialization`)”。在不必每次都生成对象的情况下，这种方式可以减少额外的负担。

以下是三种示例：

```
//: c06:Bath.java
// Constructor initialization with composition.
import com.bruceeckel.simpletest.*;

class Soap {
    private String s;
    Soap() {
        System.out.println("Soap()");
        s = new String("Constructed");
    }
    public String toString() { return s; }
}

public class Bath {
    private static Test monitor = new Test();
    private String // Initializing at point of definition:
        s1 = new String("Happy"),
        s2 = "Happy",
        s3, s4;
    private Soap castille;
    private int i;
    private float toy;
    public Bath() {
        System.out.println("Inside Bath()");
        s3 = new String("Joy");
        i = 47;
        toy = 3.14f;
        castille = new Soap();
    }
}
```



```

    }
    public String toString() {
        if(s4 == null) // Delayed initialization:
            s4 = new String("Joy");
        return
            "s1 = " + s1 + "\n" +
            "s2 = " + s2 + "\n" +
            "s3 = " + s3 + "\n" +
            "s4 = " + s4 + "\n" +
            "i = " + i + "\n" +
            "toy = " + toy + "\n" +
            "castille = " + castille;
    }
    public static void main(String[] args) {
        Bath b = new Bath();
        System.out.println(b);
        monitor.expect(new String[] {
            "Inside Bath()",
            "Soap()",
            "s1 = Happy",
            "s2 = Happy",
            "s3 = Joy",
            "s4 = Joy",
            "i = 47",
            "toy = 3.14",
            "castille = Constructed"
        });
    }
} ///:~

```

请注意，在 **Bath** 的构造器中，有一行语句在所有初始化产生之前就已经执行了。如果你没有在定义处初始化，那么除非发生了不可避免的运行期异常，否则将不能保证信息在发送给对象引用之前已经被初始化。

当 **toString()** 被调用时，它将填充 **s4** 的值，以确保所有的数据成员（**fields**）在被使用之时已被妥善初始化了。

继承（inheritance）语法

继承是所有 **OOP** 语言和 **Java** 语言的组成部分。当你在创建一个类时，你总是在继承，因此，除非你已明确指出要从其他类中继承，否则你就是在隐式地从 **Java** 的标准根源类 **object** 进行继承。

组合的语法比较平实，但要执行继承，其语法却要采用截然不同的形式。在继承过程中，你需要先声明：“新类与旧类相似。”通常，你首先给类确定一个名称，但在书写类主体的左边花括号之前，应先写下关键字 **extends**，并在其后写下基类的名称。当你这么做时，会自动得到基类中所有的数据成员和成员方法。例如：

```
//: c06:Detergent.java
// Inheritance syntax & properties.
import com.bruceeckel.simpletest.*;

class Cleanser {
    protected static Test monitor = new Test();
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public String toString() { return s; }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        System.out.println(x);
        monitor.expect(new String[] {
            "Cleanser dilute() apply() scrub()"
        });
    }
}

public class Detergent extends Cleanser {
    // Change a method:
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // Call base-class version
    }
    // Add methods to the interface:
    public void foam() { append(" foam()"); }
    // Test the new class:
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        System.out.println(x);
        System.out.println("Testing base class:");
    }
}
```

```

monitor.expect(new String[] {
    "Cleanser dilute() apply() " +
    "Detergent.scrub() scrub() foam()",
    "Testing base class:",
});
Cleanser.main(args);
}
} ///:~

```

这个程序示范了 Java 的许多特性。首先，在 **Cleanser** 的 `append()` 方法中，我们用 “+=” 操作符将几个 **String** 对象连接成 `s`，此操作符是 Java 设计者重载的用来处理 **String** 对象的两个操作符之一（另一个是 “+”）。

其次，**Cleanser** 和 **Detergent** 均含有 `main()` 方法。你可以为你的每个类都创建一个 `main()` 方法。一般我们都建议以这种方式来编写程序代码，以使测试代码被包装在类中。即使是一个程序中含有多个类，也只有命令行所调用的那个类的 `main()` 方法会被调用（只要这个 `main()` 是 `public`，其所属的类是否为 `public` 则不用考虑）。因此，在此例中，如果你的命令行是 `java Detergent`，那么 `Detergent.main()` 将会被调用。但即使 **Cleanser** 不是一个 `public` 类，如果你的命令行是 `java Cleanser`，那么 `Cleanser.main()` 仍然会被调用。这种在每个类中都设置一个 `main()` 方法的技术可使每个类的单元测试都变得简便易行。而且你在完成单元测试之后，也无需删除 `main()`，你可以将其留待下次测试。

在此例中，你可以看到 `Detergent.main()` 明确调用了 `Cleanser.main()`，并将其从命令行获取的参数传递给了它。当然，你也可以向其传递任意的 **String** 数组。**Cleanser** 中所有的方法都必须是 `public` 的，这一点非常重要。请记住，如果你没有加任何访问权限修饰符，那么成员缺省的访问权限是 `package`，它仅允许包内的成员访问。因此，在此包中，如果没有访问权限修饰符，任何人都可以使用这些方法。例如，**Detergent** 就不成问题。但是，其他包中的某个类若要从 **Cleanser** 中继承，则只能访问 `public` 成员。所以，为了继承，一般的规则是将所有的数据成员都指定为 `private`，将所有的方法指定为 `public`（稍后将会学到，`protected` 成员也可以借助导出类来访问）。当然，在特殊情况下，你必须做出调整，但上述方法的确是一个很有用的规则。

请注意，在 **Cleanser** 接口中有一组方法：`append()`、`dilute()`、`apply()`、`scrub()` 和 `toString()`。由于 **Detergent** 是由关键字 `extends` 从 **Cleanser** 导出的，所以它可以在其接口中自动获得这些方法，尽管你并不能地看到这些方法在 **Detergent** 中的显式定义。因此，你可以将继承视为是对类的复用。

正如我们在 `scrub()` 中所见，对在基类中定义的方法进行修改是可行的。在此例中，你可能想要在新版本中调用从基类继承而来的方法。但是在 `scrub()` 中，你并不能直接调用 `scrub()`，因为这样做将会产生递归，而这并不是你所期望的。为解决此问题，Java 提供了关键字 `super`。关键字 `super` 指代的是当前类所继承的基类。为此，表达式 `super.scrub()` 将调用基类版本的 `scrub()` 方法。

在继承的过程中，你并不一定非得使用基类的方法。你也可以在导出类中添加新方法，其添

加方式与在类中添加任意方法一样，即对其加以定义即可。foam()方法即为一例。

在 Detergent.main() 中，你会发现，对于一个 Detergent 对象而言，你除了可以调用 Detergent 的方法（如 foam()）之外，还可以调用 Cleanser 中所有可用的方法。

初始化基类

由于现在涉及基类和导出类这两个类，而不是只有一个类，所以要想象导出类所产生的结果对象，会有点令人迷惑。从外部来看，它就像是一个与基类具有相同接口的新类，或许还会有一些额外的方法和数据成员。但继承并不只是复制基类的接口。当你创建了一个导出类的对象时，该对象包含了一个基类的子对象（subobject）。这个子对象与你用基类直接创建的对象是一样的。二者区别在于，后者来自于外部，而基类的子对象被包装在导出类对象内部。

当然，对基类子对象的正确初始化也是至关重要的，而且也仅有一种方法来保证这一点：在构造器中调用具有执行基类初始化所需要的所有知识和能力的基类构造器来执行初始化。Java 会自动在导出类的构造器中插入对基类构造器的调用。下例展示了上述机制在三层继承关系上是如何工作的：

```
//: c06:Cartoon.java
// Constructor calls during inheritance.
import com.bruceeckel.simpletest.*;

class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

public class Cartoon extends Drawing {
    private static Test monitor = new Test();
    public Cartoon() {
        System.out.println("Cartoon constructor");
    }
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
        monitor.expect(new String[] {
            "Art constructor",
```

```

        "Drawing constructor",
        "Cartoon constructor"
    });
}
} ///:~

```

你会发现，构建过程是从基类“向外”扩散的，所以基类在导出类构造器可以访问它之前，就已经完成了初始化。即使你不为 `Cartoon()` 创建构造器，编译器也会为你合成一个缺省的构造器，该构造器将调用基类的构造器。

带参数的构造器

上例中各个类均含有缺省的构造器，即这些构造器都不带参数（`argument`）。编译器可以轻松地调用它们是因为不存在要传递什么样的参数的问题。但是，如果你的类没有缺省的参数，或者是你想调用一个带参数的基类构造器，你就必须用关键字 `super` 显式地编写调用基类构造器的语句，并且配以适当的参数列表：

```

///: c06:Chess.java
// Inheritance, constructors and arguments.
import com.bruceeckel.simpletest.*;

class Game {
    Game(int i) {
        System.out.println("Game constructor");
    }
}

class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        System.out.println("BoardGame constructor");
    }
}

public class Chess extends BoardGame {
    private static Test monitor = new Test();
    Chess() {
        super(11);
        System.out.println("Chess constructor");
    }
    public static void main(String[] args) {
        Chess x = new Chess();
        monitor.expect(new String[] {

```

```

        "Game constructor",
        "BoardGame constructor",
        "Chess constructor"
    });
}
} ///:~

```

如果你不在 `BoardGame()` 中调用基类构造器，编译器将“抱怨”无法找到符合 `Game()` 形式的构造器。而且，调用基类构造器必须是在导出类构造器中要做的第一件事（如果你做错了，编译器会提醒你）。

捕捉基类构造器异常

正如刚才所提到的，编译器会强制你首先将对基类构造器的调用置于导出类构造器之中。这意味着在此之前不会发生任何其他动作。在第九章你将看到，这种做法同样还可以防止导出类构造器捕捉任何来自基类的异常。当然，有时这样也会带来不便。

结合使用组合（composition）和继承（inheritance）

同时使用组合和继承是很常见的事。下例就展示了同时使用这两种技术，并配以必要的构造器初始化，来创建更加复杂的类：

```

///: c06:PlaceSetting.java
// Combining composition & inheritance.
import com.bruceeckel.simpletest.*;

class Plate {
    Plate(int i) {
        System.out.println("Plate constructor");
    }
}

class DinnerPlate extends Plate {
    DinnerPlate(int i) {
        super(i);
        System.out.println("DinnerPlate constructor");
    }
}

class Utensil {
    Utensil(int i) {

```

```

        System.out.println("Utensil constructor");
    }
}

class Spoon extends Utensil {
    Spoon(int i) {
        super(i);
        System.out.println("Spoon constructor");
    }
}

class Fork extends Utensil {
    Fork(int i) {
        super(i);
        System.out.println("Fork constructor");
    }
}

class Knife extends Utensil {
    Knife(int i) {
        super(i);
        System.out.println("Knife constructor");
    }
}

// A cultural way of doing something:
class Custom {
    Custom(int i) {
        System.out.println("Custom constructor");
    }
}

public class PlaceSetting extends Custom {
    private static Test monitor = new Test();
    private Spoon sp;
    private Fork frk;
    private Knife kn;
    private DinnerPlate pl;
    public PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
        frk = new Fork(i + 3);
        kn = new Knife(i + 4);
        pl = new DinnerPlate(i + 5);
    }
}

```

```

        System.out.println("PlaceSetting constructor");
    }
    public static void main(String[] args) {
        PlaceSetting x = new PlaceSetting(9);
        monitor.expect(new String[] {
            "Custom constructor",
            "Utensil constructor",
            "Spoon constructor",
            "Utensil constructor",
            "Fork constructor",
            "Utensil constructor",
            "Knife constructor",
            "Plate constructor",
            "DinnerPlate constructor",
            "PlaceSetting constructor"
        });
    }
} ///:~

```

虽然编译器强制你去初始化基类，并且要求你要在构造器起始处就要这么做，但是它并不监督你必须将成员对象也初始化，因此在这一点上你自己必须时刻注意。

确保正确清除

Java 中没有 C++ 的析构函数（**destructor**）的概念。析构函数是一种在对象被销毁时可以被自动调用的函数。其原因可能是因为在 Java 中，我们的习惯只是忘掉而不是销毁对象，并且让垃圾回收器在必要时释放其内存。

通常这样做是好事，但有时你的类可能要在其生命周期内执行一些必需的清除活动。正如我们在第四章中所提到的那样，你并不知道垃圾回收器何时将会被调用，或者它是否将被调用。因此，如果你想要某个类清除一些东西，就必须显式地编写一个特殊方法来做这件事，并确保客户端程序员知晓他们必须要调用这一方法。就像第九章（异常处理）所描述的那样，其首要任务就是，你必须将这一清除动作置于 **finally** 字句之中，以预防异常的出现。

请思考一下下面这个能在屏幕上绘制图案的计算机辅助设计系统示例：

```

///: c06:CADSystem.java
// Ensuring proper cleanup.
package c06;
import com.bruceeckel.simpletest.*;
import java.util.*;

class Shape {

```



```

Shape(int i) {
    System.out.println("Shape constructor");
}
void dispose() {
    System.out.println("Shape dispose");
}
}

class Circle extends Shape {
    Circle(int i) {
        super(i);
        System.out.println("Drawing Circle");
    }
    void dispose() {
        System.out.println("Erasing Circle");
        super.dispose();
    }
}

class Triangle extends Shape {
    Triangle(int i) {
        super(i);
        System.out.println("Drawing Triangle");
    }
    void dispose() {
        System.out.println("Erasing Triangle");
        super.dispose();
    }
}

class Line extends Shape {
    private int start, end;
    Line(int start, int end) {
        super(start);
        this.start = start;
        this.end = end;
        System.out.println("Drawing Line: " + start + ", " + end);
    }
    void dispose() {
        System.out.println("Erasing Line: " + start + ", " + end);
        super.dispose();
    }
}

```

```

public class CADSystem extends Shape {
    private static Test monitor = new Test();
    private Circle c;
    private Triangle t;
    private Line[] lines = new Line[5];
    public CADSystem(int i) {
        super(i + 1);
        for(int j = 0; j < lines.length; j++)
            lines[j] = new Line(j, j*j);
        c = new Circle(1);
        t = new Triangle(1);
        System.out.println("Combined constructor");
    }
    public void dispose() {
        System.out.println("CADSystem.dispose()");
        // The order of cleanup is the reverse
        // of the order of initialization
        t.dispose();
        c.dispose();
        for(int i = lines.length - 1; i >= 0; i--)
            lines[i].dispose();
        super.dispose();
    }
    public static void main(String[] args) {
        CADSystem x = new CADSystem(47);
        try {
            // Code and exception handling...
        } finally {
            x.dispose();
        }
        monitor.expect(new String[] {
            "Shape constructor",
            "Shape constructor",
            "Drawing Line: 0, 0",
            "Shape constructor",
            "Drawing Line: 1, 1",
            "Shape constructor",
            "Drawing Line: 2, 4",
            "Shape constructor",
            "Drawing Line: 3, 9",
            "Shape constructor",
            "Drawing Line: 4, 16",
            "Shape constructor",
            "Drawing Circle",
        });
    }
}

```

```

        "Shape constructor",
        "Drawing Triangle",
        "Combined constructor",
        "CADSystem.dispose()",
        "Erasing Triangle",
        "Shape dispose",
        "Erasing Circle",
        "Shape dispose",
        "Erasing Line: 4, 16",
        "Shape dispose",
        "Erasing Line: 3, 9",
        "Shape dispose",
        "Erasing Line: 2, 4",
        "Shape dispose",
        "Erasing Line: 1, 1",
        "Shape dispose",
        "Erasing Line: 0, 0",
        "Shape dispose",
        "Shape dispose"
    });
}
} ///:~

```

此系统中的每件事物都是某种 **Shape** (**Shape** 自身就是一种 **Object**, 因为 **Shape** 继承自根类 **Object**)。每个类都重载 **Shape** 的 **dispose()** 方法, 并运用 **super** 来调用该方法的基类版本。尽管对象生命期中任何被调用的方法都可以做执行一些必需的清除工作, 但是 **Circle**、**Triangle** 和 **Line** 这些特定的 **Shape** 类仍然都带有可以进行“绘制”的构造器。每个类都有自己的 **dispose()** 方法来将未存于内存之中的东西恢复到对象存在之前的状态。

在 **main()** 中, 你将看到 **try** 和 **finally** 这两个新的关键字, 我们将在第九章对它们进行正式介绍。关键字 **try** 表示, 下面的块(用一组大括号括起来的范围)是所谓的保护区(**guarded region**), 这意味着它需要被特殊处理。其中一项特殊处理就是无论 **try** 块是怎样退出的, 保护区后的 **finally** 子句中的代码总是要被执行的。这里 **finally** 子句表示的是“无论发生什么事, 一定要为 **x** 调用 **dispose()**”。这在第九章中, 我们将对这些关键字作全面的解释。

请注意, 在你的清除方法中, 你还必须注意对基类和成员对象的清除方法的调用顺序, 以防某个子对象(**subobject**)依赖于另一个子对象情形的发生。一般而言, 你采用的形式应该与 **C++** 编译器在其析构函数上所施加的形式相同: 首先, 执行类的所有特定的清除动作, 其顺序同生成顺序相反(通常这就要求“基类”元素仍旧存活), 然后, 就如我们所示范的那样, 调用基类的清除方法。

许多情况下, “清除”并不是问题, 你仅需要让垃圾回收器完成该动作就行。但当你必须亲自处理此事时, 你就得多做努力并多加小心。因为, 一旦涉及垃圾回收, 你能够信赖的事就不会很多了。垃圾回收器可能永远也无法被调用, 即使被调用, 它也可能以任何它想要的顺

序来回收对象。最好的办法是除了内存以外，不要依赖垃圾回收器去做任何事。如果你需要进行清除，最好是编写你自己的清除方法，但不要依赖 `finalize()`。

名称屏蔽（Name hiding）

如果 Java 的基类拥有某个已被多次重载的方法名称，那么在导出类中重新定义该方法名称，并不会屏蔽其在基类中的任何版本（这一点与 C++ 不同）。因此，无论是该层或者它的基类中对方法进行定义，重载机制都可以正常工作：

```
//: c06:Hide.java
// Overloading a base-class method name in a derived class
// does not hide the base-class versions.
import com.bruceeckel.simpletest.*;

class Homer {
    char doh(char c) {
        System.out.println("doh(char)");
        return 'd';
    }
    float doh(float f) {
        System.out.println("doh(float)");
        return 1.0f;
    }
}

class Milhouse {}

class Bart extends Homer {
    void doh(Milhouse m) {
        System.out.println("doh(Milhouse)");
    }
}

public class Hide {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Bart b = new Bart();
        b.doh(1);
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
        monitor.expect(new String[] {
            "doh(float)",
        })
    }
}
```

```

        "doh(char)",
        "doh(float)",
        "doh(Milhouse)"
    });
}
} ///:~

```

你可以看到，虽然 **Bart** 引入了一个新的重载方法（在 C++ 中若要完成这项工作是需要屏蔽基类方法的），但是在 **Bart** 中 **Homer** 的所有被重载的方法都是可用的。正如你将在下一章所看到的，使用与基类完全相同的签名（signature）及返回类型来重载具有相同名称的方法，是一件极其平常的事。但它也令人迷惑不解（这也就是为什么 C++ 不允许这样做的原因所在——防止你犯可能会犯的错误）。

组合与继承之间选择

组合和继承都允许你在新的类中设置子对象（subobject），组合是显式地这样做的，而继承则是隐式的。你或许想知道二者间的区别何在，以及怎样在二者之间做出选择。

组合技术通常用于你想要在新类中使用现有类的功能而非它的接口的情形。即，你在新类中嵌入某个对象，借其实现你所需要的功能，但新类的用户看到的只是你为新类所定义的接口，而非嵌入对象的接口。为取得此效果，你需要在新类中嵌入一个 **private** 的现有类的对象。

有时，允许类的用户直接访问新类中的组合成份是极具意义的；也就是说，将成员对象声明为 **public**。如果成员对象自身都实现了具体实现的隐藏，那么这种做法就是安全的。当用户能够了解到你在组装一组部件时，会使得端口更加易于理解。**Car** 对象即为一个好例子：

```

///: c06:Car.java
// Composition with public objects.

class Engine {
    public void start() {}
    public void rev() {}
    public void stop() {}
}

class Wheel {
    public void inflate(int psi) {}
}

class Window {
    public void rollup() {}
    public void rolldown() {}
}

```

```

class Door {
    public Window window = new Window();
    public void open() {}
    public void close() {}
}

public class Car {
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door
        left = new Door(),
        right = new Door(); // 2-door
    public Car() {
        for(int i = 0; i < 4; i++)
            wheel[i] = new Wheel();
    }
    public static void main(String[] args) {
        Car car = new Car();
        car.left.window.rollup();
        car.wheel[0].inflate(72);
    }
} ///:~

```

由于在这个例子中 `Car` 的组合也是问题分析的一部分（而不仅仅是底层设计的一部分），所以使成员成为 `public` 将有助于客户端程序员了解怎样去使用类，而且也降低了类开发者所面临的代码复杂度。但务必要记得这仅仅是一个特例，一般情况下，你应使域（`field`）成为 `private`。

在继承的时候，你会使用某个现有类，并开发一个它的特殊版本。通常，这意味着你在使用一个通用性（`general-purpose`）的类，并为了某种特殊需要而将其特殊化。略微思考一下，你就会发现，用一个“交通工具”对象来构成一部“车子”是毫无意义的，因为“车子”并不包含“交通工具”，它仅是一种（`is-a`）交通工具。其中“`is-a`（是一个）”的关系是用继承来表达的，而“`has-a`（有一个）”的关系则是用组合来表达的。

受保护的（`protected`）

现在，我们已向你介绍完了继承，关键字 `protected` 最终具有了意义。在理想世界中，仅靠关键字 `private` 就已经足够了。但在实际项目中，你经常会想要将某些事物尽可能对这个世界隐藏起来，但仍然允许让导出类的成员访问它们。关键字 `protected` 就是对这一实用主义的首肯。它指明“就类用户而言，这是 `private`，但对于任何继承于此类的导出类或其他任何位于同一个包内的类来说，却是可以进行访问的。”（在 `Java` 中，`protected` 也提供了包内访问权限。）

最好的方式是将数据成员保持为 **private**；你应当一直保留你“更改底层实现”的权利。然后你可以通过 **protected** 方法来控制你的类的继承者的访问权限。

```
//: c06:Orc.java
// The protected keyword.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Villain {
    private String name;
    protected void set(String nm) { name = nm; }
    public Villain(String name) { this.name = name; }
    public String toString() {
        return "I'm a Villain and my name is " + name;
    }
}

public class Orc extends Villain {
    private static Test monitor = new Test();
    private int orcNumber;
    public Orc(String name, int orcNumber) {
        super(name);
        this.orcNumber = orcNumber;
    }
    public void change(String name, int orcNumber) {
        set(name); // Available because it's protected
        this.orcNumber = orcNumber;
    }
    public String toString() {
        return "Orc " + orcNumber + ": " + super.toString();
    }
    public static void main(String[] args) {
        Orc orc = new Orc("Limburger", 12);
        System.out.println(orc);
        orc.change("Bob", 19);
        System.out.println(orc);
        monitor.expect(new String[] {
            "Orc 12: I'm a Villain and my name is Limburger",
            "Orc 19: I'm a Villain and my name is Bob"
        });
    }
} ///:~
```

你会发现 `change()` 可以访问 `set()`，这是因为它是 `protected` 的。你还应注意 `orc` 的 `toString()` 方法依据 `toString()` 的基类版本而被定义方式。

增量开发（incremental development）

继承技术的优点之一，就是它支持增量开发模式。你可以引入新代码而不会在现有代码中引发 **Bug**。事实上，这种模式可以将新的 **Bug** 隔离在新的代码之中。通过从现有的、功能性的类继承并添加数据成员和成员方法（并重新定义现有方法），就可以使别人可能仍在使用的现有代码既不会被改变也不会新增 **Bug**。

类被隔离得如此之干净，实在令人惊奇。你甚至不需要为了复用程序代码而调用方法的源代码，充其量只需要导入（`import`）一个包即可（这对继承和组合都适用）。

我们要认识到程序开发是一种增量过程，犹如人类的学习一样，这一点很重要。你可以尽你所能去分析，但当你开始执行一个项目时，你仍然无法知道所有的答案。如果你将项目视作是一种有机的、进化着的生命体而去培养，而不是打算像盖摩天大楼一样快速见效，你就会获得更多的成功和更迅速的回馈。

虽然就经验而言，继承是一种有用的技术，但在事情已稳定的情况下，你就得采取一个新视角来审视你的类层级结构，力求将其缩减为一种更实用的结构。记住，继承代表着对一种关系的展示，即“此新类是彼旧类的一种类型。”你的程序不应该只关心对 `bit` 的处理，而应该通过生成和操作各种类型的对象，用来自于问题空间（`problem space`）中的术语来表现一个模型（`model`）。

向上转型（upcasting）

“为新的类提供方法”并不是继承技术中最重要的一个方面。其最重要的方面是它被用来表现新类和基类之间的关系。这种关系可以用“新类是现有类的一种类型”这句话来加以概括。

这个描述并非只是一种华丽的用以解释继承的方式，而是直接由语言所支撑的特性。例如，假设有一个被称为 **Instrument** 用来代表乐器的基类和一个称为 **Wind** 的导出类。由于继承意味着基类中所有的方法在导出类中也同样有效，所以你能够向基类发送的所有信息同样也可以向导出类发送。如果 **Instrument** 类具有一个 `play()` 方法，那么 **Wind** 乐器也将同样具备。这意味着我们可以准确地说 **Wind** 对象也是一种类型的 **Instrument**。下面这个例子说明了编译器是怎样支持这一概念的：

```
//: c06:Wind.java
// Inheritance & upcasting.
import java.util.*;

class Instrument {
```



```

public void play() {}
static void tune(Instrument i) {
    // ...
    i.play();
}
}

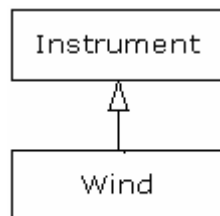
// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Upcasting
    }
} ///:~

```

在此例中，**tune()**方法可以接受 **Instrument** 引用，这实在是太有趣了。但在 **Wind.main()** 中，**tune()**方法是通过给它一个 **Wind** 引用而被调用的。鉴于 Java 对类型的检查十分严格，接受一种类型的方法同样可以接受另外一种类型就会显得很奇怪，除非你能认识到 **Wind** 对象同样也是一种 **Instrument** 对象，而且也不存在任何方法是 **tune()**可以通过 **Instrument** 来调用，同时又不存在于 **Wind** 之中的。在 **tune()**中，程序代码可以对 **Instrument** 和它所有的导出类起作用，这种将 **Wind** 引用转换为 **Instrument** 引用的动作，我们称之为“向上转型 (upcasting)”。

为什么要使用“向上转型”？

该术语的使用有其历史渊源的，并且是以传统的类继承图的绘制方法为基础的：将根置于页面的顶端，然后逐渐向下。（当然你也可以以任何你认为有效的方法进行绘制。）于是，**Wind.java** 的继承图就是：



由导出类转型成基类，在继承图上是向上移动的，因此一般称为“向上转型”。由于向上转型是从一个较专用类型向较通用类型转，所以总是很安全的。也就是说，导出类是基类的一个超集 (superset)。它可能比基类含有更多的方法，但它必须至少具备基类中所含有的方法。在向上转型的过程中，对于类接口唯一有可能发生的事情是丢失方法，而不是获取它们。这就是为什么编译器在“未曾明确表示转型”或“未曾指定特殊标记”的情况下，仍然允许向上转型的原因。

你也可以执行与向上转型相反的“向下转型（downcasting）”，但其中含有一个难题，我们将在第 10 章作为主题进行讨论。

再次探究组合与继承

在面向对象编程中，生成和使用程序代码最有可能采用的方法就是直接将数据和方法包装进一个类中，并使用该类的对象。你也可以运用组合技术使用现有类来开发新的类。而继承技术其实是不太常用的。因此，尽管在学习 **OOP** 的过程中，我们多次强调继承，但这并不意味着你一有可能就要使用它。相反地，你应当慎用这一技术，其使用场合仅限于你确信使用该技术确实有效的情况。一个用以判断你到底是该用组合还是继承的最清晰的办法，就是问一问你自己是否需要从新类向基类向上转型。如果你必须向上转型，则继承是必要的，但如果你不需要，则你应当好好考虑自己是否需要继承。下一章（多态 **polymorphism**）提出了一个使用向上转型的最具说服力的理由，但只要你记得问一下“我真的需要向上转型吗？”，你就拥有了一个很好的工具来在这两种技术中做出决定。

关键字 **final**

根据上下文环境，Java 的关键字 **final** 的含义存在着细微的区别，但通常它指的是“这是无法改变的。”你可能出于两种理由而需要阻止改变：设计或效率。由于这两个原因相差很远，所以关键字 **final** 有可能被误用。

以下几节谈论了三个可能使用到 **final** 的情况：数据、方法和类。

Final 数据

许多编程语言都有某种方法，来向编译器告知一块数据是恒定不变的。由于以下两种原因，数据的恒定不变是很有用的：

1. 它可以是一个永不改变的“编译期常量（compile-time constant）”。
2. 它可以是一个在运行期被初始化的值，而你不希望它被改变。

在编译期常量的情况下，编译器可以将该常量值带入任何可能用到它的计算式中。就是说，可以在编译期执行计算式，减轻了一些运行期的负担。在 Java 中，这类常量必须是原始的并且以关键字 **final** 表示。在对这个常量进行定义的时候，必须对其进行赋值。

一个既是 **static** 又是 **final** 的域只占有一份不能改变的存储空间。

当对对象引用而不是原始类型运用 **final** 时，其含义会有一点令人迷惑。对于原始类型，**final**

使数值恒定不变，而用于对象引用，**final** 使引用恒定不变。一旦引用被初始化指向一个对象，就无法对它进行改变以指向另一个对象。然而，对象其自身却是可以被修改的，Java 并未提供使任何对象恒定不变的途径。（但你可以自己编写类以取得使对象恒定不变的效果。）这一限制同样适用数组，它也是对象。

下面是一个用以说明 **final** 数据成员的示例：

```
//: c06:FinalData.java
// The effect of final on fields.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Value {
    int i; // Package access
    public Value(int i) { this.i = i; }
}

public class FinalData {
    private static Test monitor = new Test();
    private static Random rand = new Random();
    private String id;
    public FinalData(String id) { this.id = id; }
    // Can be compile-time constants:
    private final int VAL_ONE = 9;
    private static final int VAL_TWO = 99;
    // Typical public constant:
    public static final int VAL_THREE = 39;
    // Cannot be compile-time constants:
    private final int i4 = rand.nextInt(20);
    static final int i5 = rand.nextInt(20);
    private Value v1 = new Value(11);
    private final Value v2 = new Value(22);
    private static final Value v3 = new Value(33);
    // Arrays:
    private final int[] a = { 1, 2, 3, 4, 5, 6 };
    public String toString() {
        return id + ": " + "i4 = " + i4 + ", i5 = " + i5;
    }
    public static void main(String[] args) {
        FinalData fd1 = new FinalData("fd1");
        //! fd1.VAL_ONE++; // Error: can't change value
        fd1.v2.i++; // Object isn't constant!
        fd1.v1 = new Value(9); // OK -- not final
        for(int i = 0; i < fd1.a.length; i++)
```

```

        fd1.a[i]++; // Object isn't constant!
        //! fd1.v2 = new Value(0); // Error: Can't
        //! fd1.v3 = new Value(1); // change reference
        //! fd1.a = new int[3];
        System.out.println(fd1);
        System.out.println("Creating new FinalData");
        FinalData fd2 = new FinalData("fd2");
        System.out.println(fd1);
        System.out.println(fd2);
        monitor.expect(new String[] {
            "% fd1: i4 = \\d+, i5 = \\d+",
            "Creating new FinalData",
            "% fd1: i4 = \\d+, i5 = \\d+",
            "% fd2: i4 = \\d+, i5 = \\d+"
        });
    }
} ///:~

```

由于 **VAL_ONE** 和 **VAL_TOW** 是带有编译期数值的 **final** 原始类型，所以它们二者均可以用作编译期常量，并且没有重大区别。**VAL_THREE** 是一种更加典型的对常量进行定义的方式：定义为 **public**，则可以被用于包之外；定义为 **static** 来强调只有一份；定义为 **final** 来说明它是一个常量。请注意，带有恒定初始值（即，编译期常量）的 **final static** 原始类型全用大写字母命名，并且字与字之间用下划线来隔开。（这就像 C 常量一样，C 常量是这一命名传统的起源。）也请注意，在编译期 **i5** 的值是无法得知的，因此未用大写字母来表示。

我们不能因为某数据是 **final** 的就认为在编译期可以知道它的值。在运行期使用随机生成的数值来初始化 **i4** 和 **i5**，就说明了这一点。示例部分也展示了将 **final** 数值定义为 **static** 和非 **static** 的区别。此区别只有在数值在运行期内被初始化时才会显现，这是因为编译器对编译期数值一视同仁。（并且它们可能因优化而消失。）当你运行程序时，就会看到这个区别。请注意，在 **fd1** 和 **fd2** 中，**i4** 的值是唯一的，但 **i5** 的值是不可以通过创建第二个 **FinalData** 对象而加以改变的。这是因为它是 **static**，在装载时已被初始化，而不是每次创建新对象时都初始化。

从 **v1** 到 **v3** 的变量说明了 **final** 引用的意义。正如你在 **main()** 中所看到的，不能因为 **v2** 是 **final** 的，就认为你无法改变它的值。由于它是一个引用，**final** 意味着你无法将 **v2** 再次指向另一个新的对象。你会看到，这对数组具有同样的意义，数组则是另一种引用。（我还不知道有什么办法来使数组引用本身成为 **final**。）看起来，使引用成为 **final** 没有使原始类型成为 **final** 的用处大。

空白 final

Java 允许生成空白 final (Blank final)，所谓空白 final 是指被声明为 **final** 但又未给定初值的数据成员。无论什么情况，编译器都确保空白 final 在使用前必须被初始化。但是，空白 final

在关键字 **final** 的使用上提供了更大的灵活性，为此，一个类中的 **final** 数据成员就可以实现依对象而有所不同，却又保持其恒定不变的特性。下面即为一例：

```
//: c06:BlankFinal.java
// "Blank" final fields.

class Poppet {
    private int i;
    Poppet(int ii) { i = ii; }
}

public class BlankFinal {
    private final int i = 0; // Initialized final
    private final int j; // Blank final
    private final Poppet p; // Blank final reference
    // Blank finals MUST be initialized in the constructor:
    public BlankFinal() {
        j = 1; // Initialize blank final
        p = new Poppet(1); // Initialize blank final reference
    }
    public BlankFinal(int x) {
        j = x; // Initialize blank final
        p = new Poppet(x); // Initialize blank final reference
    }
    public static void main(String[] args) {
        new BlankFinal();
        new BlankFinal(47);
    }
} ///:~
```

你被强制在数据成员的定义处或者是每个构造器中用表达式对 **final** 进行赋值。这正是 **final** 数据成员在使用前总是被初始化的原因所在。

final 参数

Java 允许你以在参数列表中以声明的方式将参数指明为 **final**。这意味着你无法在方法中更改参数引用所指向的对象：

```
//: c06:FinalArguments.java
// Using "final" with method arguments.

class Gizmo {
    public void spin() {}
}
```

```

    }

    public class FinalArguments {
        void with(final Gizmo g) {
            //! g = new Gizmo(); // Illegal -- g is final
        }
        void without(Gizmo g) {
            g = new Gizmo(); // OK -- g not final
            g.spin();
        }
        // void f(final int i) { i++; } // Can't change
        // You can only read from a final primitive:
        int g(final int i) { return i + 1; }
        public static void main(String[] args) {
            FinalArguments bf = new FinalArguments();
            bf.without(null);
            bf.with(null);
        }
    } ///:~

```

方法 `f()` 和 `g()` 展示了当原始类型的参数被指明为 **final** 时所出现的结果：你可以读参数，但却无法修改参数。这一特性看起来仅有微不足道的用处，而且你可能还用不到。

final 方法

使用 **final** 方法的原因有两个。第一个原因是把方法锁定，以预防任何继承类修改它的意义。这是出于设计的考虑：你想要确保在继承中方法行为保持不变，并且不会被重载。

使用 **final** 方法的第二个原因是效率。如果你将一个方法指明为 **final**，就是同意编译器将针对该方法的所有调用都转为内嵌（**inline**）调用。当编译器发现一个 **final** 方法调用命令时，它会根据自己的谨慎判断，跳过插入程序代码的正常方式而执行方法调用机制（将参数压入栈，跳至方法代码处并执行，然后跳回并清除栈中的参数，处理返回值。），并且以方法体中的实际代码的复本来替代方法调用。这将消除方法调用的开销。当然，如果一个方法很大，你的程序代码就会膨胀，你可能看不到内嵌带来的任何性能提高，因此，你所取得的性能提高会因为花费于方法内的时间总量而被缩减。这意味着Java编译器能够观察到这些情况并明智地抉择是否对 **final** 方法执行内嵌动作。然而，最好是让编译器和JVM仅在你明确表示要阻止重载时，再处理效率事项，并将方法指明为 **final**¹。

¹ 不要陷入强调过早优化的陷阱。如果你已经让你的系统运行起来了，并且其速度过慢，你应该质疑你真的能够通过使用 **final** 关键字来解决此问题吗。第 15 章由有关“整形（**profiling**）”的知识，它对提高你的程序执行速度是很有帮助的。

final 和 private

类中所有的 **private** 方法都被隐含是 **final** 的。由于你无法取用 **private** 方法，所以你也无法重载之。你可以对 **private** 方法增加 **final** 修饰符，但这并不能给该方法增加任何额外的意义。

这一问题会造成混淆。因为如果你试图重载一个 **private** 方法（隐含是 **final** 的），看起来是奏效的，而且编译器也不会给出错误信息：

```
//: c06:FinalOverridingIllusion.java
// It only looks like you can override
// a private or private final method.
import com.bruceeckel.simpletest.*;

class WithFinals {
    // Identical to "private" alone:
    private final void f() {
        System.out.println("WithFinals.f()");
    }
    // Also automatically "final":
    private void g() {
        System.out.println("WithFinals.g()");
    }
}

class OverridingPrivate extends WithFinals {
    private final void f() {
        System.out.println("OverridingPrivate.f()");
    }
    private void g() {
        System.out.println("OverridingPrivate.g()");
    }
}

class OverridingPrivate2 extends OverridingPrivate {
    public final void f() {
        System.out.println("OverridingPrivate2.f()");
    }
    public void g() {
        System.out.println("OverridingPrivate2.g()");
    }
}
```

```

public class FinalOverridingIllusion {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        OverridingPrivate2 op2 = new OverridingPrivate2();
        op2.f();
        op2.g();
        // You can upcast:
        OverridingPrivate op = op2;
        // But you can't call the methods:
        //! op.f();
        //! op.g();
        // Same here:
        WithFinals wf = op2;
        //! wf.f();
        //! wf.g();
        monitor.expect(new String[] {
            "OverridingPrivate2.f()",
            "OverridingPrivate2.g()"
        });
    }
} ///:~

```

“重载”只有在某方法是基类的接口的一部分时才会出现。即，你必须能将一个对象向上转型为它的基本类型并调用相同的方法（这一点在下一章将得以阐明）。如果某方法为 **private**，它就不是基类的接口的一部分。它仅是一些隐藏于类中的程序代码，不过具有相同的名称而已。但你如果在导出类中以相同的名称生成一个 **public**、**protected** 或包访问权限（**package access**）方法的话，该方法就不会产生在基类中出现的“仅具有相同名称”的情况。此时你并没有重载方法，你仅是生成了一个新的方法。由于 **private** 方法无法触及又不能有效隐藏，所以仅仅是因为考虑到定义它的类的程序代码组织，它们才有存在的意义，除此之外，其他任何事物都不需要把它纳入考虑的范畴。

final 类

当你将某个类的整体定义为 **final** 时（通过将关键字 **final** 置于它的定义之前），你就声明了你打算继承该类，而且也不允许别人这样做。换句话说，出于某种考虑，你对该类的设计永不需要做任何变动，或者出于安全的考虑，你不希望它有子类。

```

//: c06:Jurassic.java
// Making an entire class final.

class SmallBrain {}

final class Dinosaur {

```



```

    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() {}
}

//! class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'

public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
} ///:~

```

请注意，根据你的选择，**final** 类的数据成员可以是 **final**，也可以不是。不论类是否被定义为 **final**，相同的规则都适用于 **final** 的数据成员。然而，由于 **final** 类禁止继承，所以 **final** 类中所有的方法都被隐含是 **final** 的，因为它们是不会被重载的。在 **final** 类中你可以给方法添加 **final** 修饰符，但这不会添加任何意义。

有关 **final** 的忠告

当你在设计类时，将方法指明是 **final**，应该说是明智的。你可能会觉得没人会想要重载你的方法。有时这是对的。

但请留意你所作的假设。要预见到类是如何被复用的，一般来说是很困难的，特别是对于一个通用的类而言，尤其如此。如果你将一个方法指定为 **final**，你可能失去了在其他的程序员的项目中通过继承来复用类的机会，而这仅仅是因为你无法想到对它以该种方式加以运用。

Java 标准程序库就是一个很好的这样的例子。特别是 Java 1.0/1.1 中 **Vector** 类被广泛地运用，出于效率的考虑（这近乎是一个幻想），如果所有的方法均未被指定为 **final** 的话，它还有可能会更加实用。很容易想象到，你会想要继承并重载如此基础而有用的类，但是设计者却认为这样做不太合适。这里有两个令人意外的原因。第一，**stack** 继承自 **vector**，就是说 **stack** 是个 **vector**，这从逻辑的观点看是不正确的。第二，**vector** 的许多最重要的方法，如 **addElement()** 和 **elementAt()**，是 **synchronized**。正如你在第 11 章中将要看到的那样，这将导致很大的执行开销，可能会抹煞 **final** 所带来的好处。这种情况增强了人们关于程序员无法正确猜测优化应当发生于何处的观点。如此蹩脚的设计，却要置于我们每个人都得使用的标准程序库中，也是很糟糕的。（幸运的是，Java 2 的容器相关程序库用 **ArrayList** 替代了 **Vector**。**ArrayList** 的行为要合理得多。遗憾的是仍然存在用旧容器库编写新程序代码的情

况)。

留心一下 **Hashtable**，这也是同样有趣的例子，它是又一个重要的 Java1.0/1.1 标准库类，而且不含任何 **final** 方法。如本书其他地方所提到的，某些类明显是由一些互不相关的人设计的。（你会发现名为 **Hashtable** 的方法相对于 **vector** 中的方法要简洁得多，这是又一个证据。）对于类库的使用者来说，这又是一个本不该如此轻率的事物。这种不规则的情况只能使用户付出更多的努力。这是对粗糙的设计和代码作的又一种讽刺。（请注意，Java 2 容器库用 **HashMap** 替代了 **Hashtable**。）

初始化及类的加载

在许多传统语言中，其程序是作为启动（startup）过程的一部分立刻被加载的。然后是初始化，紧接着程序开始运行。这些语言的初始化过程必须小心控制，以确保 **static** 的初始化顺序不会造成麻烦。例如，如果某个 **static** 在另一个 **static** 被初始化之前就可以被有效地使用，那么 C++ 就会出现这个问题。

Java 就不会出现这个问题，因为它采用了一种不同的加载方式。由于 Java 中的所有事物都是对象，所以许多动作就变得更加容易，加载动作仅仅是其中之一。在下一章中你将会更全面地学到，每个类的编译代码都存在于它自己的独立的文件中。该文件只在需要使用程序代码时才会被加载。一般来说，你可以说：“类的代码在初次使用时才加载。”这通常是指知道类的第一个对象被构建时才发生加载，但是当访问 **static** 数据成员或是 **static** 方法时，也会发生加载。

初次使用之处也是静态初始化（**static** 初始化）发生之处。所有的 **static** 对象和 **static** 代码段都会在加载时依程序中的顺序（即，你定义类时的书写顺序）依次初始化。当然，**static** 只会被初始化一次。

继承与初始化

检查包括继承在内的初始化全过程，以对所发生的一切有个全局性的把握，是很有益的。请看下例：

```
//: c06:Beetle.java
// The full process of initialization.
import com.bruceeckel.simpletest.*;

class Insect {
    protected static Test monitor = new Test();
    private int i = 9;
    protected int j;
    Insect() {
```

```

        System.out.println("i = " + i + ", j = " + j);
        j = 39;
    }
    private static int x1 =
        print("static Insect.x1 initialized");
    static int print(String s) {
        System.out.println(s);
        return 47;
    }
}

public class Beetle extends Insect {
    private int k = print("Beetle.k initialized");
    public Beetle() {
        System.out.println("k = " + k);
        System.out.println("j = " + j);
    }
    private static int x2 =
        print("static Beetle.x2 initialized");
    public static void main(String[] args) {
        System.out.println("Beetle constructor");
        Beetle b = new Beetle();
        monitor.expect(new String[] {
            "static Insect.x1 initialized",
            "static Beetle.x2 initialized",
            "Beetle constructor",
            "i = 9, j = 0",
            "Beetle.k initialized",
            "k = 47",
            "j = 39"
        });
    }
} ///:~

```

你在 **Beetle** 上运行 Java 时，所发生的第一件事情就是你试图访问 **Beetle.main()**（一个 **static** 方法），于是加载器开始启动并找出 **Beetle** 类被编译的程序代码（它被编译到了一个名为 **Beetle.class** 的文件之中）。在对它进行加载的过程中，编译器注意到它有一个基类（这是由关键字 **extends** 告知的），于是它继续进行加载。不管你是否打算产生一个该基类的对象，这都要发生。（请尝试将对象创建注释掉，以证明这一点。）

如果该基类还有其自身的基类，那么第二个基类就会被加载，如此类推。接下来，根基类中的静态初始化（在此例中为 **Insect**）即会被执行，然后是下一个导出类，以此类推。这种方式很重要，因为导出类的静态初始化可能会依赖于基类成员能否被正确初始化的。

至此为止，必要的类都已加载完毕，对象就可以被创建了。首先，对象中所有的原始类型都会被设为缺省值，对象引用被设为零——这是通过将对象内存设为二进制零值而一举生成的。然后，基类的构造器会被调用。在本例中，它被自动调用的。但你也可以用 **super** 来指定对基类构造器的调用（正如在 **Beetle**()构造器中的第一步操作。）基类构造器和导出类的构造器一样，以相同的顺序来经历相同的过程。在基类构造器完成之后，实例变量(instance variables)按其次序被初始化。最后，构造器的其余部分被执行。

总结

继承和组合都能从现有类型生成新类型。然而，典型的是组合将现有类型作为新类型底层实现的一部分来加以复用，而继承复用的是接口。由于导出类具有基类接口，所以它可以向上转型至基类，这对多态（polymorphism）来讲至关重要。你将在下一章看到。

尽管面向对象编程对继承极力强调，但在你开始设计时，一般你应优先选择使用组合，只在确实必要时才使用继承。因为组合更具灵活性。此外，通过对成员类型使用继承技术，你可以在运行期就改变那些成员对象的类型和行为。因此，你可以在运行期改变组合而成的对象的行为。

在设计一个系统时，你的目标应该是找到或创建某些类，其中每个类都有具体的用途，而且既不是太大（包含太多的功能而难以复用），也不是太小（不添加其他功能就无法使用）。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 创建两个带有缺省构造器（空参数列表）的类，**A** 和 **B**。从 **A** 中继承一个名为 **C** 的新类，并在 **C** 内创建一个 **B** 类的成员。不要给 **C** 编写构造器。创建一个 **C** 类的对象并观察其结果。
2. 修改练习 1 以使 **A** 和 **B** 含有带参数的构造器，以取代缺省的构造器。为 **C** 写一个构造器，并在其中执行所有的初始化。
3. 创建一个单一的类。在第二个类中，将一个引用定义为第一个类的对象。运用惰性初始化（lazy initialization）来实例化这个对象。
4. 从 **Detergent** 中继承一个新的类。重载 **scrub()**并添加一个名为 **sterilize()**的新方法。
5. 使用 **Cartoon.java** 并注解掉 **Cartoon** 类的构造器。对所发生的现象进行解释。
6. 使用 **Chess.java** 并注解掉 **Chess** 类的构造器。对所发生的现象进行解释。
7. 请证明缺省构造器是编译器为你而创建的。
8. 证明基类构造器（a）总是会被调用（b）在导出类构造器之前被调用。
9. 创建一个仅有一个非缺省构造器的基类，并创建一个带有缺省(no-arg)构造器和非缺省构造器的导出类。在导出类的构造器中调用基类的构造器。

10. 创建一个 **Root** 类，令其含有名为 **Component 1**、**Component 2**、**Component 3** 的类各一个实例（这些也由你写）。从 **Root** 中派生一个类 **stem**，也含有上述各“组成部分”。所有的类都应带有可打印出类的相关信息或缺省构造器。
11. 修改第 10 题，使每个类都仅具有非缺省的构造器。
12. 将一个适当的 **dispose()** 方法的层次结构添加到练习 11 的所有类中。
13. 创建一个类，它应带有一个被重载了三次的方法。继承一个新类，并添加一个该方法的新的重载定义，展示这四个方法在导出类中都是可以使用的。
14. 在 **Car.java** 中给 **Engine** 添加一个 **service()** 方法，并在 **main()** 中调用该方法。
15. 在包中编写一个类。你的类应具备一个 **protected** 方法。在包外部，试着调用该 **protected** 方法并解释此结果。然后，从你的类中继承一个导出类，并从该导出类的一个方法内部调用该 **protected** 方法。
16. 创建一个名为 **Amphibian** 的类。由此派生一个被称为 **Frog** 的类。在基类中设置适当的方法。在 **main()** 中，编写一个 **Frog** 并向上转型至 **Amphibian**，然后说明所有方法都可工作。
17. 修改练习 16，使 **Frog** 重载基类中方法的定义（令新定义使用相同的方法标记）。请留心 **main()** 中都发生了什么。
18. 创建一个含有 **static final** 数据成员和 **final** 数据成员的类，说明二者间的区别。
19. 创建一个含有指向某对象的空白 **final** 引用的类。在所有构造器内部都执行空白 **final** 的初始化动作。展示 Java 确保了 **final** 在使用前必须被初始化，且一旦被初始化即无法改变。
20. 创建一个带 **final** 数据成员的类。由此派生出一个类并重载此数据成员。
21. 创建一个 **final** 类并试着继承它。
22. 请证明加载类的动作仅发生一次。证明该类的第一个实体的创建或是对 **static** 成员的访问都有可能引起加载。
23. 在 **Beetle.Java** 中，从 **Beetle** 类继承一个特殊类型的甲壳虫。其形式与现有类相同，跟踪并解释其输出结果。

第七章 多态

在面向对象的程序设计语言中，多态（polymorphic）是继数据抽象和继承之后的第三种基本特性。

多态通过分离“做什么”和“怎么做”，从另一角度将接口和实现分离开来。多态不但能够改善代码的组织结构和可读性，还能够创建“可扩展的”程序，即无论在项目最初创建时，还是在需要添加新功能时，都可以进行扩充。

“封装”通过合并特征和行为来创建新的数据类型。“实现隐藏”则通过细节“私有化（private）”将接口和实现分离开来。这种类型的组织机制对那些有过程化程序设计背景的人来说，更容易理解。而多态的作用则是消除类型之间的耦合关系。在前一章中，我们已经知道继承允许将对象视为自己本身的类型或它的基类型进行处理。这种能力极为重要，因为它可以使多种类型（从同一基类导出而来的）被视为同一类型进行处理，而同一份代码也就可以毫无差别地运行在这些不同类型之上了。多态方法调用允许一种类型表现出与其他相似类型之间的区别，只要它们都是从同一基类导出而来的。这种区别是根据方法行为的不同来而表示出来的，虽然这些方法都可以通过同一个基类来调用。

在本章中，通过一些基本简单的例子（这些例子中所有与多态无关的代码都被删掉，只剩下与多态有关的部分）来深入浅出地学习多态（也称作动态绑定 dynamic binding、后期绑定 late binding 或运行时绑定 run-time binding）。

向上转型

在第 6 章中，我们已经知道对象既可以作为它自己本身的类型使用，也可以作为它的基类型使用。而这种将对某个对象的引用视为对其基类型的引用的做法被称作“向上转型（upcasting）”——因为在继承树的画法中，基类是放置在上方的。

但是，这样做也会引起一个的问题，具体看下面这个有关乐器的例子。既然几个例子都要演奏乐符（**Note**），我们就应该在包中单独创建一个 **Note** 类。

```
//: c07:music>Note.java
// Notes to play on musical instruments.
package c07.music;
import com.bruceeckel.simpletest.*;

public class Note {
    private String noteName;
    private Note(String noteName) {
        this.noteName = noteName;
    }
    public String toString() { return noteName; }
```

```

public static final Note
    MIDDLE_C = new Note("Middle C"),
    C_SHARP  = new Note("C Sharp"),
    B_FLAT   = new Note("B Flat");
    // Etc.
} ///:~

```

这是一个枚举（enumeration）类，包含固定数目的可供选择的不变对象。不能再产生另外的对象，因为其构造器是私有的。

在下面的例子中，**Wind** 是一种 **Instrument**，因此可以继承 **Instrument** 类。

```

//: c07:music:Wind.java
package c07.music;

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
    // Redefine interface method:
    public void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
} ///:~

```

```

//: c07:music:Music.java
// Inheritance & upcasting.
package c07.music;
import com.bruceeckel.simpletest.*;

public class Music {
    private static Test monitor = new Test();
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        tune(flute); // Upcasting
        monitor.expect(new String[] {
            "Wind.play() Middle C"
        });
    }
} ///:~

```

`Music.tune()` 方法接受一个 `Instrument` 引用参数，同时也接受任何导出自 `Instrument` 的类。在 `Main()` 方法中，当一个 `Wind` 引用传递到 `tune()` 方法时，就会出现这种情况，而不需要任何类型转换。这样做是允许的——因为 `Wind` 从 `Instrument` 继承而来，所以 `Instrument` 的接口必定存在于 `Wind` 中。从 `Wind` 向上转型到 `Instrument` 可能会“缩小”接口，但无论如何也不会比 `Instrument` 的全部接口更窄。

忘记对象类型

Music.java 这个程序看起来似乎有些奇怪。为什么所有人都应该故意忘记一个对象的类型呢？在进行向上转型时，就会产生这种情况；并且如果让 `tune()` 方法直接接受一个 `Wind` 引用作为自己的参数，似乎会更为直观。但这样会引发的一个重要问题是：如果你那样做，就需要为系统内 `Instrument` 的每种类型都编写一个新的 `tune()` 方法。假设按照这种推理，现在再加入 `Stringed`（弦乐）和 `Brass`（管乐）这两种 `Instrument`（乐器）：

```
//: c07:music:Music2.java
// Overloading instead of upcasting.
package c07.music;
import com.bruceeckel.simpletest.*;

class Stringed extends Instrument {
    public void play(Note n) {
        System.out.println("Stringed.play() " + n);
    }
}

class Brass extends Instrument {
    public void play(Note n) {
        System.out.println("Brass.play() " + n);
    }
}

public class Music2 {
    private static Test monitor = new Test();
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Stringed i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Brass i) {
        i.play(Note.MIDDLE_C);
    }
}
```



```

public static void main(String[] args) {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    tune(flute); // No upcasting
    tune(violin);
    tune(frenchHorn);
    monitor.expect(new String[] {
        "Wind.play() Middle C",
        "Stringed.play() Middle C",
        "Brass.play() Middle C"
    });
}
} ///:~

```

这样做行得通，但有一个主要缺点：必须为添加的每一个新 **Instrument** 类编写特定类型的方法。这意味着在开始时就需要更多的编程，这也意味着如果以后想添加类似 **Tune()** 的新方法，或者添加自 **Instrument** 导出的新类，仍需要做大量的工作。此外，如果我们忘记重载某个方法，编译器不会返回任何错误信息，这样关于类型的整个处理过程就变得难以操纵。

如果我们只写这样一个简单方法，它仅接收基类作为参数，而不是那些特殊的导出类。这样做情况会变得更好吗？也就是说，如果我们不管导出类的存在，编写的代码只是与基类打交道，会不会好呢？

这正是多态所允许的。然而，大多数程序员具有面向过程程序设计的背景，对多态的运作方式可能会感到有一点迷惑。

曲解

运行 **Music.java** 这个程序后，我们便会发现难点所在。**Wind.play()** 方法将产生输出结果。这无疑是我们所期望的输出结果，但它看起来似乎又没有什么意义。请观察一下 **tune()** 方法：

```

public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
}

```

它接受一个 **Instrument** 引用。那么在这种情况下，编译器怎样才可能知道这个 **Instrument** 引用指向的是 **Wind** 对象，而不是 **Brass** 对象或 **Stringed** 对象呢？实际上，编译器无法得知。为了深入理解这个问题，有必要研究一下“绑定（binding）”这个话题。

方法调用绑定

将一个方法调用同一个方法主体关联起来被称作“绑定（binding）”。若在程序执行前进行绑定（如果有的话，由编译器和链接程序实现），叫做“前期绑定(early binding)”。可能你以前从来没有听说过这个术语，因为它是面向过程的语言中不需要选择就默认的绑定方式。C 编译器只有一种方法调用，那就是前期绑定。

上述程序之所以令人迷惑，主要是因为提前绑定。因为，当编译器只有一个 Instrument 引用时，它无法知道究竟调用哪个方法才对。

解决的办法叫做“后期绑定（late binding）”，它的含义就是在运行时，根据对象的类型进行绑定。后期绑定也叫做“动态绑定(dynamic binding)”或“运行时绑定(run-time binding)”。如果一种语言想实现后期绑定，就必须具有某些机制，以便在运行时能判断对象的类型，以调用恰当的方法。也就是说，编译器仍不知道对象的类型，但是方法调用机制能找到正确的方法体，并加以调用。后期绑定机制随编程语言的不同而有所不同，但是我们只要想象一下就会得知，不管怎样都必须在对象中安置某种“类型信息”。

Java 中除了 `static` 和 `final` 方法（`private` 方法属于 `final`）之外，其他所有的方法都是后期绑定。这意味着通常情况下，我们不必判定是否应该进行后期绑定---它会自动发生。

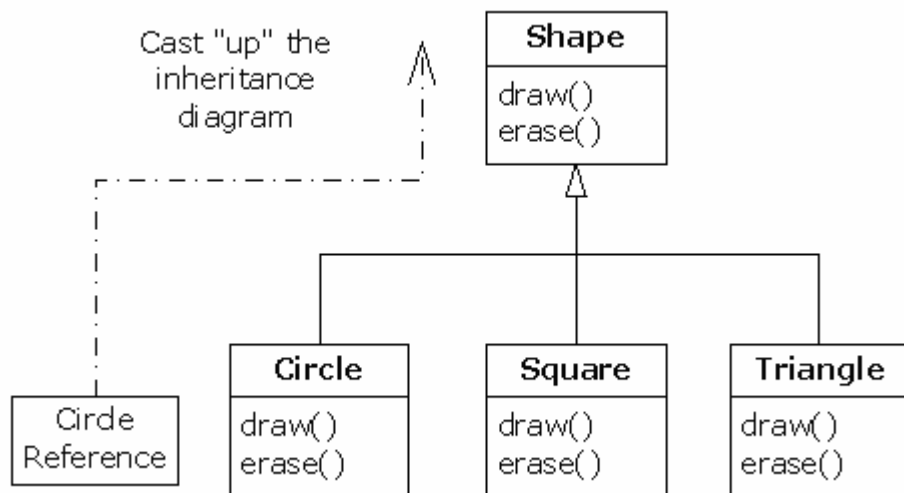
为什么要将某个方法声明为 `final` 呢？正如前一章提到的那样，它可以防止其他人重载该方法。但更重要的一点或许是：这样做可以有效地“关闭”动态绑定，或者是想告诉编译器不需要对其进行动态绑定。这样，编译器就可以为 `final` 方法调用生成更有效的代码。然而，大多数情况下，这样做对我们程序的整体性能不会产生什么改观。所以，最好根据设计来决定是否使用 `final`，而不是出于试图提高性能。

产生正确的行为

一旦知道 Java 中所有方法都是通过动态绑定实现多态这个事实之后，我们就可以编写只与基类打交道的程序代码了，并且这些代码对所有的导出类都可以正确运行。或者换种说法，发送消息给某个对象，让该对象去断定应该做什么事。

面向对象程序设计中，有一个最经典的“几何形状（shape）”例子。因为它很容易被可视化，所以经常用到；但不幸的是，它可能使初学者认为面向对象程序设计仅适用于图形化程序设计，实际当然不是这种情形了。

在“几何形状”这个例子中，包含一个 `Shape` 基类和多个导出类，如：`Circle`, `Square`, `Triangle` 等。这个例子之所以好用，是因为我们可以说“圆是一种形状”，这种说法也很容易被理解。下面的继承图展示了它们之间的关系：



向上转型可以像下面这条语句这么简单：

```
Shape s = new Circle();
```

这里，创建了一个 **Circle** 对象，并把得到的引用立即赋值给 **Shape**，这样做看似错误（将一种类型赋值给另一类型）；但实际上是没问题的，因为通过继承，**Circle** 就是一种 **Shape**。因此，编译器认可这条语句，也就不会产生错误信息。

假设我们调用某个基类方法（已被导出类所重载）：

```
s.draw();
```

同样地，我们可能会认为调用的是 **shape** 的 **draw()**，因为这毕竟是一个 **shape** 引用，那么编译器是怎样知道去做其他的事情呢？由于后期绑定（多态），程序还是正确调用了 **Circle.draw()** 方法。

下面的例子稍微有所不同：

```
//: c07:Shapes.java
// Polymorphism in Java.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Shape {
    void draw() {}
    void erase() {}
}

class Circle extends Shape {
    void draw() {
```

```

        System.out.println("Circle.draw()");
    }
    void erase() {
        System.out.println("Circle.erase()");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Square.draw()");
    }
    void erase() {
        System.out.println("Square.erase()");
    }
}

class Triangle extends Shape {
    void draw() {
        System.out.println("Triangle.draw()");
    }
    void erase() {
        System.out.println("Triangle.erase()");
    }
}

// A "factory" that randomly creates shapes:
class RandomShapeGenerator {
    private Random rand = new Random();
    public Shape next() {
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
        }
    }
}

public class Shapes {
    private static Test monitor = new Test();
    private static RandomShapeGenerator gen =
        new RandomShapeGenerator();
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
    }
}

```

```

// Fill up the array with shapes:
for(int i = 0; i < s.length; i++)
    s[i] = gen.next();
// Make polymorphic method calls:
for(int i = 0; i < s.length; i++)
    s[i].draw();
monitor.expect(new Object[] {
    new TestExpression("% (Circle|Square|Triangle)"
        + "\\draw\\(\\)", s.length)
});
}
} ///:~

```

Shape 基类为自它那里继承而来的所有导出类，建立了一个通用接口——也就是说，所有形状都可以描绘和擦除。导出类重载了这些定义，以便为每种特殊类型的几何形状提供独特的行为。

RandomShapeGenerator 是一种“工厂（factory）”，在我们每次调用 **next()** 方法时，它可以为随机选择的 **shape** 对象产生一个引用。请注意向上转型是在 **return** 语句里发生的。每个 **return** 语句取得一个指向某个 **Circle**、**Square** 或者 **Triangle** 的句柄，并将其以 **Shape** 类型从 **next()** 方法中发送出去。所以无论我们在什么时候调用 **next()** 方法时，是绝对没有可能知道它所获的具体类型到底是什么，因为我们总是只能获得一个通用的 **Shape** 引用。

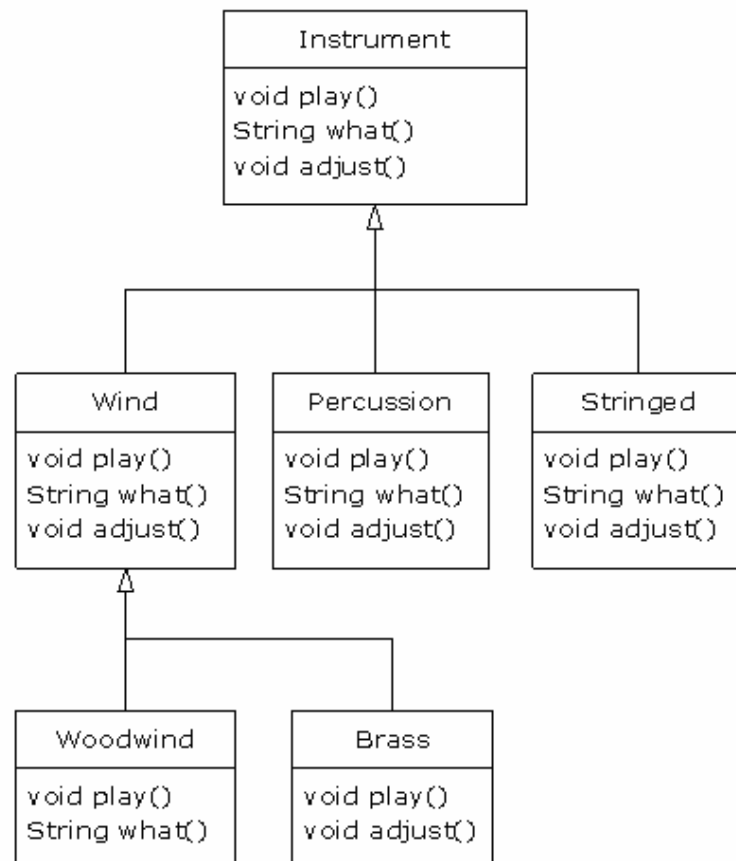
main() 包含了 **Shape** 句柄的一个数组，通过调用 **RandomShapeGenerator.next()** 来填入数据。此时，我们只知道自己拥有一些 **Shape**，不会知道除此之外的更具体情况（编译器一样不知）。然而，当我们遍历这个数组，并为每个数组元素调用 **draw()** 方法时，与各类型有关的专属行为竟会神奇般地正确发生，我们可以从运行该程序时，产生的输出结果中发现这一点。

随机选择几何形状是为了让大家理解：在编译期间，编译器不需要获得任何特殊的信息，就能进行正确的调用。对 **draw()** 方法的所有调用都是通过动态绑定进行的。

扩展性

现在，让我们返回到乐器（**Instrument**）示例。由于有多态机制，我们可根据自己的需求向系统中添加任意多的新类型，而不需更修改 **tune()** 方法。在一个设计良好的 OOP 程序中，我们的大多数或者所有方法都会遵循 **tune()** 的模型，而且只与基类接口通信。我们说这样的程序是“可扩展的”，因为我们可以从通用的基类继承出新的数据类型，从而新添一些功能。那些操纵基类接口的如方法不需要任何改动就可以应用于新类。

考虑一下：对于乐器例子，如果我们向基类中添加更多的方法，并加入一些新类，将会出现什么情况呢？如下图所示：



事实上，不需要改动 `tune()` 方法，所有的新类都能与原有类一起正确运行。即使 `tune()` 方法是存放在某个单独文件中，并且在 `Instrument` 接口中还添加了其他的新方法，`tune()` 也不需再编译就仍能正确运行。下面是上述示意图的具体实现：

```
//: c07:music3:Music3.java
// An extensible program.
package c07.music3;
import com.bruceeckel.simpletest.*;
import c07.music.Note;

class Instrument {
    void play(Note n) {
        System.out.println("Instrument.play() " + n);
    }
    String what() { return "Instrument"; }
    void adjust() {}
}

class Wind extends Instrument {
```

```

    void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
    String what() { return "Wind"; }
    void adjust() {}
}

class Percussion extends Instrument {
    void play(Note n) {
        System.out.println("Percussion.play() " + n);
    }
    String what() { return "Percussion"; }
    void adjust() {}
}

class Stringed extends Instrument {
    void play(Note n) {
        System.out.println("Stringed.play() " + n);
    }
    String what() { return "Stringed"; }
    void adjust() {}
}

class Brass extends Wind {
    void play(Note n) {
        System.out.println("Brass.play() " + n);
    }
    void adjust() {
        System.out.println("Brass.adjust()");
    }
}

class Woodwind extends Wind {
    void play(Note n) {
        System.out.println("Woodwind.play() " + n);
    }
    String what() { return "Woodwind"; }
}

public class Music3 {
    private static Test monitor = new Test();
    // Doesn't care about type, so new types
    // added to the system still work right:
    public static void tune(Instrument i) {

```

```

        // ...
        i.play(Note.MIDDLE_C);
    }

    public static void tuneAll(Instrument[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }

    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
        monitor.expect(new String[] {
            "Wind.play() Middle C",
            "Percussion.play() Middle C",
            "Stringed.play() Middle C",
            "Brass.play() Middle C",
            "Woodwind.play() Middle C"
        });
    }
} ///:~

```

新添加的方法 `what()` 返回一个 `String` 引用及类的描述说明；另一个新添加的方法 `adjust()` 则提供每种乐器的调音方法。

在 `main()` 中，当我们将某种引用置入 `orchestra` 数组中，就会自动向上转型到 `Instrument`。

可以看到，`tune()` 方法完全可以忽略它周围代码所发生的全部变化，依旧正常运行。这正是期望多态所具有的特性。我们所作的代码修改，不会对程序中其他不应受到影响的部分产生破坏。从另一方面说就是，多态是一项让程序员“将改变的事物与未变的事物分离开来”的重要技术。

缺陷：“重载”私有方法

我们试图这样做也是无可厚非的：


```

//: c07:PrivateOverride.java
// Abstract classes and methods.
import com.bruceeckel.simpletest.*;

public class PrivateOverride {
    private static Test monitor = new Test();
    private void f() {
        System.out.println("private f()");
    }
    public static void main(String[] args) {
        PrivateOverride po = new Derived();
        po.f();
        monitor.expect(new String[] {
            "private f()"
        });
    }
}

class Derived extends PrivateOverride {
    public void f() {
        System.out.println("public f()");
    }
} //::~

```

我们所期望的输出是“**public f()**”，但是由于 **private** 方法被自动认为就是 **final** 方法，而且对导出类是屏蔽的。因此，在这种情况下，**Derived** 类中的 **f()** 方法就是一个全新的方法；既然基类中 **f()** 方法在子类 **Derived** 中不可见，因此也就没有被重载。

结论就是：只有非 **private** 方法才可以被重载；但是我们还需要密切注意重载 **private** 方法的现象，虽然编译不会出错，但是不会按照我们所期望的来执行。明白地说，在导出类中，对于基类中的 **private** 方法，我们最好用一个不同的名字。

抽象类和抽象方法

在所有乐器的例子中，基类 **Instrument** 中的方法往往是“哑（dummy）”方法。若要调用这些方法，就会出现一些错误。这是因为 **Instrument** 类的目的是为它的所有导出类创建一个通用接口。

建立这个通用接口的唯一原因是，不同的子类可以用不同的方式表示此接口。它建立起一个基本形式，用来表示所有导出类的共同部分。另一种说法是将 **Instrument** 类称作“抽象基类”（或简称抽象类）。当我们想通过这个通用接口操纵一系列类时，就需创建一个抽象类。与任何基类所声明的签名相符的导出类方法，都会通过动态绑定机制来调用。（然而，

正如前一节所讲，如果方法名与基类中的相同，但是参数不同，就会出现重载，这或许不是我们想要的）

如果我们只有一个像 `Instrument` 这样的抽象类，那么该类的对象几乎没有任何意义。也就是说，`Instrument` 只是表示了一个接口，没有具体的实现内容；因此，创建一个 `Instrument` 对象没有什么意义，并且我们可能还想阻止使用者这样做。通过在 `Instrument` 的所有方法中打印出错误信息，就可以实现这个目的。但是这样做会将错误信息延迟到运行期才可获得，并且需要在客户端进行可靠、详尽的测试。所以最好是在编译期间捕获这些问题。

为此，Java提供一个叫做“抽象方法（abstract method）¹”的机制。这种方法是不完整的；仅有声明而没有方法体。下面是抽象方法声明所采用的语法：

```
abstract void f();
```

包含抽象方法的类叫做“抽象类(abstract class)”。如果一个类包含一个或多个抽象方法，该类必须被限制为是抽象的。（否则，编译器就会报错）

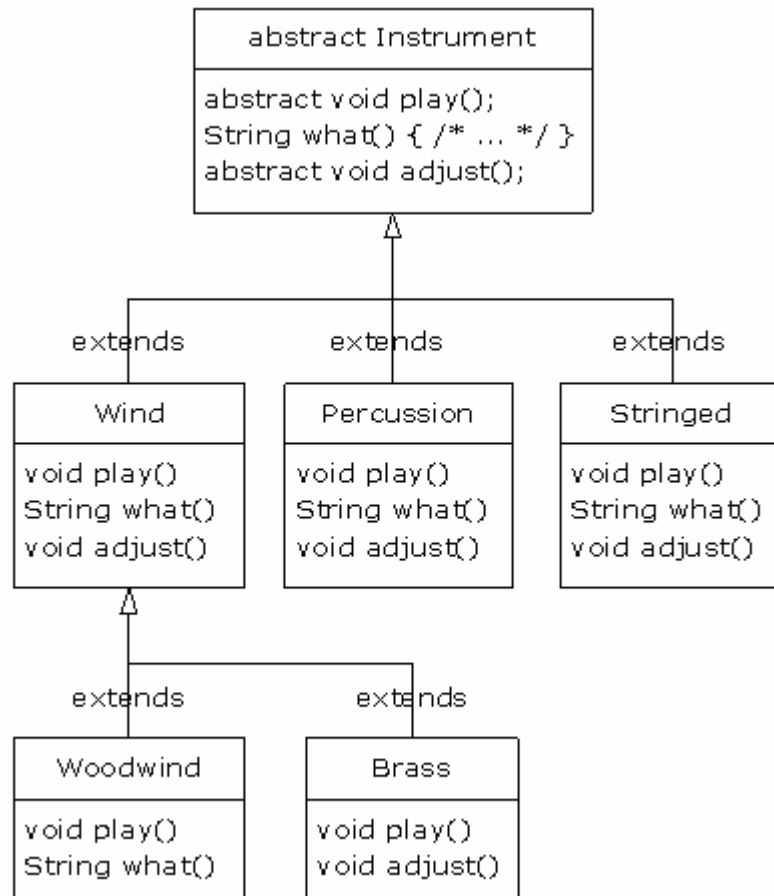
如果一个抽象类不完整，那么当我们试图产生该类的对象时，编译器会怎样处理呢？由于为一个抽象类创建对象是不安全的，所以我们会从编译器那里得到一条出错信息。这里，编译器会确保抽象类的纯粹性，我们不必担心会误用它。

如果从一个抽象类继承，并想创建该新类的对象，那么我们就必须为基类中的所有抽象方法提供方法定义。如果不这样做（可以选择不做），那么导出类便也是抽象类，且编译器将会强制我们用 **abstract** 关键字来限制修饰这个类。

我们也可能会创建一个没有任何抽象方法的抽象类。考虑这种情况：如果我们有一个类，让其包含任何 **abstract** 方法都显得没有实际意义，但是我们却想要阻止产生这个类的任何对象，那么这时这样做就很有用了。

Instrument 类可以很容易地转化成抽象类。既然使某个类成为抽象类并不需要所有的方法都是抽象的，所以仅需将某些方法声明为抽象的即可。下面所示是它的模样：

¹对于C++程序设计员来说，这相当于C++语言中的纯虚函数。



下面是修改过的“管弦乐队”的例子，其中采用了抽象类和抽象方法：

```

//: c07:music4:Music4.java
// Abstract classes and methods.
package c07.music4;
import com.bruceeckel.simpletest.*;
import java.util.*;
import c07.music.Note;

abstract class Instrument {
    private int i; // Storage allocated for each
    public abstract void play(Note n);
    public String what() {
        return "Instrument";
    }
    public abstract void adjust();
}

class Wind extends Instrument {
    public void play(Note n) {

```

```

        System.out.println("Wind.play() " + n);
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}

class Percussion extends Instrument {
    public void play(Note n) {
        System.out.println("Percussion.play() " + n);
    }
    public String what() { return "Percussion"; }
    public void adjust() {}
}

class Stringed extends Instrument {
    public void play(Note n) {
        System.out.println("Stringed.play() " + n);
    }
    public String what() { return "Stringed"; }
    public void adjust() {}
}

class Brass extends Wind {
    public void play(Note n) {
        System.out.println("Brass.play() " + n);
    }
    public void adjust() {
        System.out.println("Brass.adjust()");
    }
}

class Woodwind extends Wind {
    public void play(Note n) {
        System.out.println("Woodwind.play() " + n);
    }
    public String what() { return "Woodwind"; }
}

public class Music4 {
    private static Test monitor = new Test();
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument i) {
        // ...
    }
}

```

```

        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
        monitor.expect(new String[] {
            "Wind.play() Middle C",
            "Percussion.play() Middle C",
            "Stringed.play() Middle C",
            "Brass.play() Middle C",
            "Woodwind.play() Middle C"
        });
    }
} ///:~

```

我们可以看出，除了基类，实际上并没有什么改变。

创建抽象类和抽象方法非常有用，因为它们可以显化一个类的抽象性，并告诉用户和编译器怎样按照它所预期的方式来使用。

构造器和多态

通常，构造器异与其他种类的方法。即使涉及到多态，也仍是如此。尽管构造器并不具有多态性（它们实际上是 **Static** 方法，只不过该 **Static** 声明是隐式的），但还是非常有必要理解构造器怎样通过多态在复杂的层次结构中运作。这一理解将有助于大家避免一些令人不快的困扰。

构造器的调用顺序

构造器的调用顺序已在第 4 章进行了简要说明，并在第 6 章再次提到，但那些都是在多态引入之前介绍的。

基类的构造器总是在导出类的构造过程中被调用，而且按照继承层次逐渐向上链接，以使每个基类的构造器都能得到调用。这样做是有意义的，因为构造器具有一项特殊任务：检查对象是否被正确地构造。导出类只能访问它自己的成员，不能访问基类中的成员（基类成员通常是 `private` 类型）。只有基类的构造器才具有恰当的知识和权限对自己的元素进行初始化。因此，必须令所有构造器都得到调用，否则所有对象就不可能被正确构造。这正是编译器为什么要强制每个导出类部分都必须调用构造器的原因。在导出类的构造器主体中，如果没有明确指定调用某个基类构造器，它就会“默默”地调用缺省构造器。如果不存在缺省构造器，编译器就会报错（若某个类没有构造器，编译器会自动合成出一个缺省构造器）。

让我们来看下面这个例子，它展示了组合、继承以及多态的在构建顺序上的效果：

```
//: c07:Sandwich.java
// Order of constructor calls.
package c07;
import com.bruceeckel.simpletest.*;

class Meal {
    Meal() { System.out.println("Meal()"); }
}

class Bread {
    Bread() { System.out.println("Bread()"); }
}

class Cheese {
    Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
    Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
    Lunch() { System.out.println("Lunch()"); }
}

class PortableLunch extends Lunch {
    PortableLunch() { System.out.println("PortableLunch()"); }
}
```

```

public class Sandwich extends PortableLunch {
    private static Test monitor = new Test();
    private Bread b = new Bread();
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();
    public Sandwich() {
        System.out.println("Sandwich()");
    }
    public static void main(String[] args) {
        new Sandwich();
        monitor.expect(new String[] {
            "Meal()",
            "Lunch()",
            "PortableLunch()",
            "Bread()",
            "Cheese()",
            "Lettuce()",
            "Sandwich()"
        });
    }
} ///:~

```

在这个例子中，用其他类创建了一个复杂的类，而且每个类都有一个它声明自己的构造器。其中最重要的类是 **Sandwich**，它反映出了三层级别的继承（若将从 **Object** 的隐含继承也算在内，就是四级）以及三个成员对象。当在 **main()** 里创建一个 **Sandwich** 对象后，我们就可以看到输出结果。这也表明了这一复杂对象调用构造器要遵照下面的顺序：

1. 调用基类构造器。这个步骤会不断地反复递归下去，首先是构造这种层次结构的根，然后是下一层导出类，等等。直到最低层的导出类。
2. 按声明顺序调用成员的初始状态设置模块。
3. 调用导出类构造器的主体。

构造器的调用顺序是很重要的。当进行继承时，我们已经知道基类的一切，并且可以访问基类中任何声明为 **public** 和 **protected** 的成员。这意味着在导出类中，必须假定基类的所有成员都是有效的。一种标准方法是，构造动作一经发生，那么对象所有部分的全体成员都会得到构建。然而，在构造器内部，我们必须确保所要使用的成员都已经构建完毕。为确保这一目的，唯一的办法就是首先调用基类构造器。那么在进入导出类构造器时，在基类中可供我们访问的成员都已得到初始化。此外，在构造器中的所有成员必须有效也是因为当成员对象在类内进行定义的时候（比如上例中的 **b**、**c** 和 **l**），我们应尽可能地对它们进行初始化（也就是，通过组合方法将对象置于类内）。若遵循这一规则，那么我们就能确定所有基类成员以及当前对象的成员对象都已初始化。但遗憾的是，这种做法并不适用于所有情况，我们会在下一节看到。

继承与清除

通过组合和继承方法来创建新类时，我们永远不必担心对象的清除问题，子对象

（subobject）通常都会留给垃圾回收器进行处理。如果要是遇到清除的问题，那么我们必须不断地为我们的新类创建 **dispose()** 方法（在这里我选用此名称；你可以提出更好的）。并且由于继承的缘故，如果我们有其他作为垃圾回收一部分的特殊清除动作，就必须重载导出类中的 **dispose()** 方法。重载继承类的 **dispose()** 方法时，务必记住调用基类版本 **dispose()** 方法。否则，基类的清除动作就不会发生。下例将予以证明：

```
//: c07:Frog.java
// Cleanup and inheritance.
import com.bruceeckel.simpletest.*;

class Characteristic {
    private String s;
    Characteristic(String s) {
        this.s = s;
        System.out.println("Creating Characteristic " + s);
    }
    protected void dispose() {
        System.out.println("finalizing Characteristic " + s);
    }
}

class Description {
    private String s;
    Description(String s) {
        this.s = s;
        System.out.println("Creating Description " + s);
    }
    protected void dispose() {
        System.out.println("finalizing Description " + s);
    }
}

class LivingCreature {
    private Characteristic p = new Characteristic("is alive");
    private Description t =
        new Description("Basic Living Creature");
    LivingCreature() {
        System.out.println("LivingCreature()");
    }
}
```



```

        protected void dispose() {
            System.out.println("LivingCreature dispose");
            t.dispose();
            p.dispose();
        }
    }

    class Animal extends LivingCreature {
        private Characteristic p= new Characteristic("has heart");
        private Description t =
            new Description("Animal not Vegetable");
        Animal() {
            System.out.println("Animal()");
        }
        protected void dispose() {
            System.out.println("Animal dispose");
            t.dispose();
            p.dispose();
            super.dispose();
        }
    }

    class Amphibian extends Animal {
        private Characteristic p =
            new Characteristic("can live in water");
        private Description t =
            new Description("Both water and land");
        Amphibian() {
            System.out.println("Amphibian()");
        }
        protected void dispose() {
            System.out.println("Amphibian dispose");
            t.dispose();
            p.dispose();
            super.dispose();
        }
    }

    public class Frog extends Amphibian {
        private static Test monitor = new Test();
        private Characteristic p = new Characteristic("Croaks");
        private Description t = new Description("Eats Bugs");
        public Frog() {
            System.out.println("Frog()");
        }
    }

```

```

    }
    protected void dispose() {
        System.out.println("Frog dispose");
        t.dispose();
        p.dispose();
        super.dispose();
    }
    public static void main(String[] args) {
        Frog frog = new Frog();
        System.out.println("Bye!");
        frog.dispose();
        monitor.expect(new String[] {
            "Creating Characteristic is alive",
            "Creating Description Basic Living Creature",
            "LivingCreature()",
            "Creating Characteristic has heart",
            "Creating Description Animal not Vegetable",
            "Animal()",
            "Creating Characteristic can live in water",
            "Creating Description Both water and land",
            "Amphibian()",
            "Creating Characteristic Croaks",
            "Creating Description Eats Bugs",
            "Frog()",
            "Bye!",
            "Frog dispose",
            "finalizing Description Eats Bugs",
            "finalizing Characteristic Croaks",
            "Amphibian dispose",
            "finalizing Description Both water and land",
            "finalizing Characteristic can live in water",
            "Animal dispose",
            "finalizing Description Animal not Vegetable",
            "finalizing Characteristic has heart",
            "LivingCreature dispose",
            "finalizing Description Basic Living Creature",
            "finalizing Characteristic is alive"
        });
    }
} ///:~

```

层次结构中的每个类都包含 **Characteristic** 和 **Description** 这两种类型的成员对象，并且也必须对它们进行处理。所以万一某个子对象要依赖于其他对象，处理的顺序应该和初始化顺序相反。对于属性，则意味着与声明的顺序相反（因为属性的初始化是按照声明的顺

序进行)。对于基类(遵循 C++ 中析构函数的形式), 我们应该首先对其导出类进行清除, 然后才是基类。这是因为导出类的清除可能会调用基类中的某些方法, 所以需要使基类中的构件仍起作用而不应过早地销毁她。从输出结果我们可以看到, Frog 对象的所有部分都是按照创建的逆序进行销毁。

在这个例子中可以看到, 尽管我们通常不必执行清除处理, 但是一旦你选择要执行, 就必须谨慎和小心。

构造器内部的多态方法的行为

构造器调用的层次结构带来了一个有趣的两难问题。如果在一个构造器的内部, 同时调用正在构造的那个对象的某个动态绑定方法, 那会发生什么情况呢? 在一般的方法内部, 我们可以想象会发生什么: 动态绑定的调用是在运行期才被决定, 因为对象无法知道它是属于方法所在的那个类, 还是属于那个类的导出类。为保持一致性, 大家也许会认为这应该发生在构造器内部。

但事情并非完全如此。如果要调用构造器内部的一个动态绑定方法, 就要用到那个方法的被重载的定义。然而, 产生的效果可能相当难于预料, 并且可能造成一些难于发现的隐藏错误。

从概念上讲, 构造器的工作实际上是创建对象(这并非是一件平常的工作)。在任何构造器内部, 整个对象可能只有部分形成——我们只知道基类对象已经进行初始化, 但却不知道哪些类是从我们这里继承而来的。然而, 一个动态绑定的方法调用却会向外深入到继承层次结构内部。它可以调用导出类里的方法。如果我們是在构造器内部这样做, 那么我们可能会调用某个方法, 而这个方法所操纵的成员可能还未进行初始化——这肯定是招惹灾难的倪端。

通过下面这个例子, 我们会看到问题所在:

```
//: c07:PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect.
import com.bruceeckel.simpletest.*;

abstract class Glyph {
    abstract void draw();
    Glyph() {
        System.out.println("Glyph() before draw()");
        draw();
        System.out.println("Glyph() after draw()");
    }
}

class RoundGlyph extends Glyph {
```

```

private int radius = 1;
RoundGlyph(int r) {
    radius = r;
    System.out.println(
        "RoundGlyph.RoundGlyph(), radius = " + radius);
}
void draw() {
    System.out.println(
        "RoundGlyph.draw(), radius = " + radius);
}
}

public class PolyConstructors {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        new RoundGlyph(5);
        monitor.expect(new String[] {
            "Glyph() before draw()",
            "RoundGlyph.draw(), radius = 0",
            "Glyph() after draw()",
            "RoundGlyph.RoundGlyph(), radius = 5"
        });
    }
} ///:~

```

在 **Glyph** 中，**draw()** 方法是抽象的，是为了让其他方法重载。事实上，我们在 **RoundGlyph** 中被迫对其进行重载。但是 **Glyph** 构造器会调用这个方法，而且调用会在 **RoundGlyph.draw()** 中结束，这看起来似乎是我们的目的。但是如果我们看到输出结果，我们会发现当 **Glyph** 的构造器调用 **draw()** 方法时，**radius** 不是默认初始值 1，而是 0。这可能导致在屏幕上只画了一个点，或是根本什么东西都没有；我们只能干瞪眼，试图找出程序无法运转的原因所在。

前一节讲述的初始化顺序并不十分完整，而这正是解决这一谜题的关键所在。初始化的实际过程是：

1. 在其他任何事物发生之前，将分配给对象的存储空间初始化成二进制的零。
2. 如前所述的那样，调用基类构造器。此时，调用被重载的 **draw()** 方法（是的，是在调用 **RoundGlyph** 构造器之前调用的），由于步骤(1)的缘故，我们此时会发现 **radius** 的值为 0。
3. 按照声明的顺序调用成员的初始化代码。
4. 调用导出类的构造器主体。

这样做有一个优点，那就是所有东西都至少初始化成零（或者是某些特殊数据类型中与“零”等价的值），而不是仅仅留作垃圾。其中包括通过“组合”而嵌入一个类内部的对象引用。其

值是 `null`。所以如果忘记为该引用进行初始化，就会在运行期间抛出异常。查看输出结果时，我们会发现其他所有东西的值都会是零，这通常也正是发现问题的证据。

另一方面，我们应该对这个程序的结果相当震惊。在逻辑方面，我们做的已经十分完美，而它的行为却不可思议地错了，并且编译器也没有报错。（在这种情况下，C++ 语言会出现更合理的行为）。诸如此类的错误会很容易地被人忽略，而且要花很长的时间才能发现。

因此，编写构造器时有一条有益的规则：“用尽可能简单的方法使对象进入正常状态；如果可以的话，避免调用其他方法”。在构造器内唯一能够安全调用的那些方法是基类中的 `final` 方法（也适用于 `private` 方法，它们自动属于 `final` 方法）。这些方法不能被重载，因此也就不会出现上述令人惊讶的问题。

用继承进行设计

学习了多态之后，看起来似乎所有东西都可以被继承，因为多态是一种如此巧妙的工具。事实上，当我们使用现成的类来建立新类时，如果首先考虑使用继承技术，那么这样做反倒会加重我们的设计负担，使事情变得不必要地复杂起来。

一个更好的方法是首先选择“组合”，尤其是当你自己不能十分确定应该使用哪一种方式时。组合不会强制我们的程序设计进入继承的层次结构中。同时，组合更加灵活，因为它可以动态选择类型（因此，也就是选择了行为），相反，继承在编译期间就需要知道确切类型。下面举例说明这一点：

```
//: c07:Transmogrify.java
// Dynamically changing the behavior of an object
// via composition (the "State" design pattern).
import com.bruceeckel.simpletest.*;

abstract class Actor {
    public abstract void act();
}

class HappyActor extends Actor {
    public void act() {
        System.out.println("HappyActor");
    }
}

class SadActor extends Actor {
    public void act() {
        System.out.println("SadActor");
    }
}
```

```

class Stage {
    private Actor actor = new HappyActor();
    public void change() { actor = new SadActor(); }
    public void performPlay() { actor.act(); }
}

public class Transmogrify {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Stage stage = new Stage();
        stage.performPlay();
        stage.change();
        stage.performPlay();
        monitor.expect(new String[] {
            "HappyActor",
            "SadActor"
        });
    }
} ///:~

```

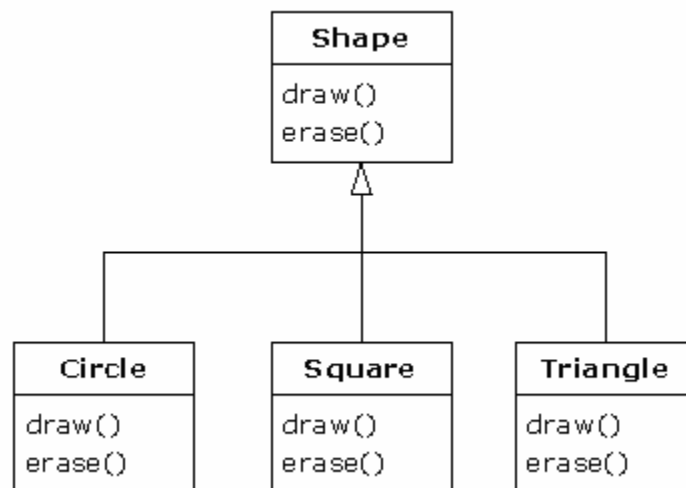
在这里，一个 `Stage` 对象包含了一个指向 `Actor` 的引用，且初始化为 `HappyActor` 对象。这意味着 `performPlay()` 会产生某种特定的行为。既然引用在运行期间可以与另一个不同的对象重新绑定起来，所以 `SadActor` 对象的引用可以在 `actor` 中被替代，然后由 `performPlay()` 产生的行为也随之改变。这样一来，我们在运行期间获得了动态灵活性。

（这也称作 `State Pattern`，请参阅 www.BruceEckel.com 上的 `Thinking in Patterns (with Java)`）与此相反，我们不能在运行期间换用不同的继承，因为它要求在编译期间完全确定下来。

一条通用的准则是：“用继承表达行为间的差异，并用属性表达状态上的变化”。在上述例子中，两者都用到了：通过继承得到了两个不同的类，用于表达 `act()` 方法的差异；而 `Stage` 通过运用组合使它自己的状态发生变化。在这种情况下，这种状态的改变也就产生了行为的改变。

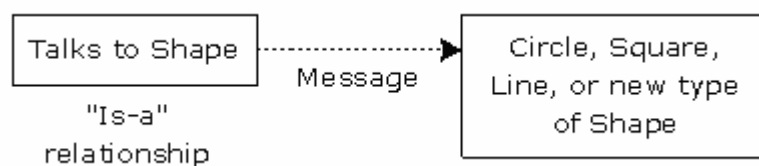
纯继承与扩展

在学习继承时，采取某种“纯粹”的方式来创建继承层次结构似乎是最清楚易懂的方法了。也就是说，只有在基类或接口中已经建立的方法才可以在导出类中被重载，如下图所示：



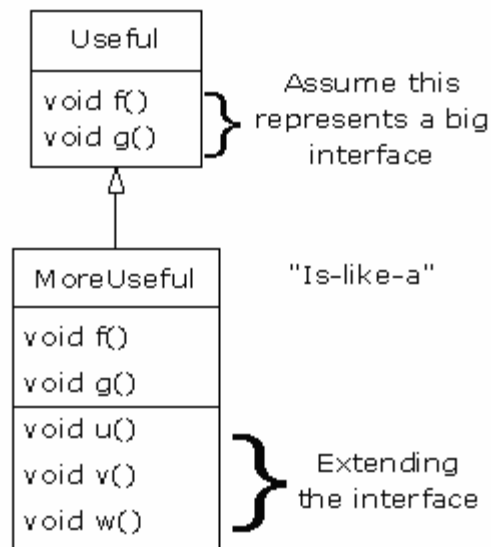
之所以被称作是纯粹的“is-a”(是一种)关系，是因为一个类的接口已经确定了它应该是什么。继承可以确保所有的导出类具有基类的所有接口，且绝对不会少。按上图那么做，导出类只是具有和基类一样的接口。

也可以认为这是一种纯替代，因为导出类可以完全地代替基类，那在我们使用时，就永远不需要知道关于子类的任何额外信息了。

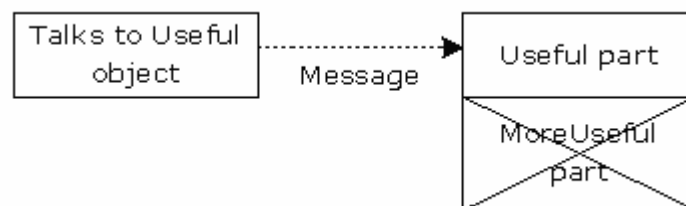


也就是说，基类可以接受我们发送给导出类的任何消息，因为二者有着完全相同的接口。我们只需从导出类向上转型，永远不需知道正在处理的对象的确切类型。所有这一切，都是通过多态进行处理的。

按这种方式考虑，似乎只有纯粹的 is-a 关系才是唯一明智的做法。所有其他的设计都只会导致混乱，当然也就错误百出。这仍是一个陷阱，因为一旦我们深入研究，就会改变主意，并发现扩展接口（遗憾的是，**extends** 关键字似乎在怂恿我们这样做）才是解决特定问题的完美方案。可将其定义为“is-like-a”（像一个）关系，因为导出类就像是一个基类——它有着相同的基本接口——但是它还具有由额外方法实现的其他特性。



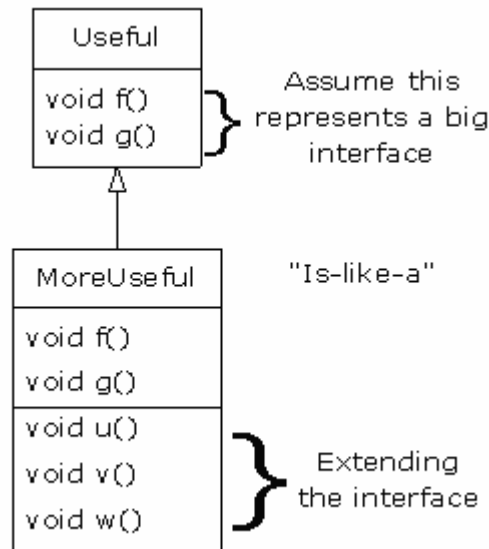
虽然这是一种有用且明智的方法（视情况而定），但是它也有缺点。导出类中扩展的接口不能被基类访问，因此，一旦我们向上转型，就不能调用那些新增方法：



在这种情况下，如果我们不进行向上转型，这样的问题也就不会出现。但是通常情况下，我们需要重新查明对象的确切类型，以便能够访问该类型所扩充的方法。随后的一节将展示该具体怎么做。

向下转型与运行期类型标识

由于我们在向上转型（在继承层次中向上移动）过程中丢失了具体的类型信息，所以我们就可以用向下转型——也就是在继承层次中向下移动——从而检索到类型信息。然而，我们知道向上转型是安全的；因为基类不会具有大于导出类的接口。因此，我们通过基类接口发送的消息都能够确保被接收到。但是对于向下转型，例如，我们无法知道一个几何形状它确实就是一个圆。它可以是一个三角形、方形或其他一些类型。



为了能够解决这个问题，必须有某种方法来确保向下转型的正确性，使我们不致于冒然转型到一种错误类型，进而发出该对象无法接受的消息。这样做是极其不安全的。

在某些程序设计语言（如 C++）中，我们必须执行一个特殊的操作以便获得安全的向下转型。但是在 Java 语言中，所有转型都会得到检查！所以即使我们只是进行一次普通的加括号形式的强制转换，在进入运行期时，仍然会对其进行检查，以便保证它的确是我们希望的那种类型。如果不是，就会返回一个 `ClassCastException`（转型异常）。这种在运行期间对类型进行检查的行为称作“运行期类型识别”（RTTI）。下面这个例子显示了 RTTI 的行为：

```

//: c07:RTTI.java
// Downcasting & Run-Time Type Identification (RTTI).
// {ThrowsException}

class Useful {
    public void f() {}
    public void g() {}
}

class MoreUseful extends Useful {
    public void f() {}
    public void g() {}
    public void u() {}
    public void v() {}
    public void w() {}
}

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

```

```

Useful[] x = {
    new Useful(),
    new MoreUseful()
};
x[0].f();
x[1].g();
// Compile time: method not found in Useful:
//! x[1].u();
((MoreUseful)x[1]).u(); // Downcast/RTTI
((MoreUseful)x[0]).u(); // Exception thrown
}
} ///:~

```

正如示意图中所示，**MoreUseful**（更有用的）扩展了 **Useful**（有用的）的接口。但是由于它是继承而来的，所以它也可以向上转型到 **Useful** 类型。我们在 **main()** 方法中对数组 **x** 进行初始化时可以看到这种情况的发生。既然数组中的两个对象都属于 **Useful** 类，所以我们可以调用 **f()** 和 **g()** 这两个方法。如果我们试图调用 **u()** 方法（它只存在于 **MoreUseful**），就会返回一条编译期出错信息。

如果我们想访问一个 **MoreUseful** 对象的扩展接口，就可以尝试进行向下转型。如果所转型类型它是正确的，这么转型成功。否则，就会返回一个 **ClassCastException** 异常。我们不必为这个异常编写任何特殊的代码，因为它指出的是程序员在程序中任何地方都可能会犯的错误。

RTTI 的内容不仅仅包括转型处理。例如他还提供了一种方法，使你可以在试图向下转型之前，查看你要所要处理的类型。整个第 10 章都是在研究 Java 运行时类型标识的所有不同方面。

总结

“多态”意味着“不同的形式”。在面向对象的程序设计中，我们持有相同的外观（基类的通用接口）以及使用该外观的不同形式：不同版本的动态绑定方法。

在本章中我们已经知道，如果不运用数据抽象和继承，就不可能去理解，进而也不可能创建一个多态例子。多态是一种不能单独来看待的特性（例如，像 **switch** 语句是可以的），相反它只能作为类关系“全景”中的一部分，与其它特性协同工作。人们经常被 Java 语言中其他的非面向对象的特性所困扰，比如方法重载等，人们有时会被认为这些是面向对象的特性。但是不要被愚弄：如果不是后期绑定，就不是多态。

为了在自己的程序中有效地运用多态乃至面向对象的技术，必须扩展自己的编程视野，使其不仅包括单个类的成员和消息，而且也包括类与类之间的共同特性以及它们之间的关系。尽

管这需要极大的努力，但是这样做是非常值得的，因为它可以带来很多成效：更快的程序开发过程、更好的代码组织、更好的代码扩展以及更容易的代码维护等。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 在基类 **Shapes.java** 中添加一个新方法，用于打印一条信息，但导出类中不要重载这个方法。请解释发生了什么。现在，将其中一个导出类的该方法重载，而其他的不变，观察又有什么发生。最后，重载所有的导出类中的这个方法。
 2. 向 **Shapes.java** 中添加一个新的 **shap** 类型，并在 **main()** 方法中验证：多态作用与新类型是否和作用与旧类型一样。
 3. 修改 **Music3.java**，使 **what()** 方法成为根 **Object** 的 **toString()** 方法。使用 **System.out.println()** 方法打印出 **Instrument** 对象（不用向上转型）。
 4. 向 **Music3.java** 添加一个新类 **Instrument**，并验证多态性是否作用与所添加的新类。
 5. 修改 **Music3.java**，使其可以像 **shapes.java** 中的方式那样随机创建 **Instrument** 对象。
 6. 创建一个 **Rodent**（啮齿动物）：**Mouse**（老鼠），**Gerbil**（鼯鼠），**Hamster**（大颊鼠）这样的继承层次结构。在基类中，提供对所有的 **Rodent** 都通用的方法，在基类中，根据特定的 **Rodent** 类型重载这些方法，以便执行不同的行为。创建一个 **Rodent** 数组，填充不同的 **Rodent** 类型，然后调用基类方法，观察发生什么情况。
 7. 修改练习 6 中的 **Rodent**，使其成为一个抽象类。只要有可能，就将 **Rodent** 的方法声明为抽象方法。
 8. 创建一个不包含任何抽象方法的抽象类，并验证我们不能为该类的创建任何对象。
 9. 向 **sandwich.java** 中添加 **pickle** 类。
 10. 修改练习 6，使其能够演示基类和导出类的初始化顺序。然后向基类和导出类中添加成员对象，并显示在构建期间初始化发生的顺序。
 11. 创建一个包含两个方法的基类。在第一个方法中可以调用第二个方法。然后产生一个继承自该基类的导出类，且重载基类第二个方法。为该导出类创建一个对象，向上转型到基类型并调用第一个方法，解释发生的情况。
 12. 创建一个基类，包含抽象方法 **print()**，并在导出类中将其重载。重载的版本中会打印基类中定义的某个整型变量的值。在定义改变量时，请赋予非零值。在基类的构造器中，可以调用这个方法。现在，在 **main()** 方法中，创建一个导出类对象，然后调用它的 **print()** 方法。请解释发生的情形。
 13. 遵循 **Transmogrify.java** 这个例子，创建一个 **Starship** 类，包含一个 **AlertStatus** 引用，此引用可以指示三种不同的状态。纳入一些可以改变这些状态的方法。
 14. 创建一个不包含任何方法的抽象类，从它那里导出出一个类，并添加一个方法。创建一个静态方法，可以接受一个指向基类的引用，将其向下转型到导出类，然后再调用该静态方法。在 **main()** 方法中，证实它的可行性。然后，将基类加上 **abstract** 声明，这样就不再需要进行向下转型。
-

第八章 接口与内部类

接口（**interface**）和内部类（**inner class**）为我们提供了一种用来组织和控制系统中的对象的更加精致的方法。

C++就不包含这些机制，尽管聪明的程序员可以自己去模拟实现它们。在 **Java** 中存在着这些机制的事实表明它们是如此重要，以至于要通过语言中的关键字直接提供对它们的支持。

在第 7 章中，你已经学习过了有关 **abstract** 关键字的知识，它允许你在一个类中创建一个或多个没有任何定义的方法——你提供了接口部分，但是没有提供任何相应的具体实现，这些实现是由此类继承者创建的。**interface** 这个关键字产生了一个完全抽象的类，它根本就没有提供任何具体实现。你将学习到接口不仅仅只是一个极度抽象的类，因为它允许你通过创建一个能够被向上转型为不止一种基类型的类，来实现一种 C++ 多重继承（**multiple inheritance**）的变种。

乍看起来，内部类就像是一种简单的代码隐藏机制：你将某些类置于另外一些类的内部。然而，你将认识到内部类能够比这做得更多——它了解它的外围类（**surrounding class**），并且能够与之通信。尽管对大多数人来说，内部类还是个新概念，但是那种你可以编写的带有内部类的代码仍然显得更加雅致和清晰。当然，要想让使用内部类进行设计使人觉得轻松自在，还是得花些功夫的。

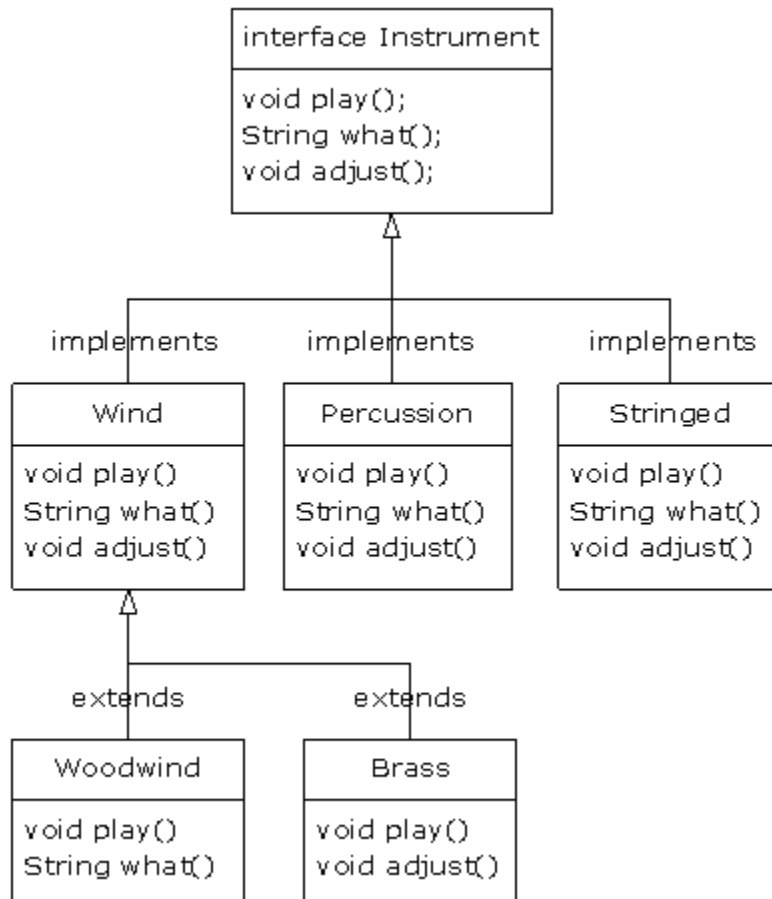
接口

interface 关键字比 **abstract** 的概念向前更迈进了一步。你可以将它看作是“纯粹的”抽象类。它允许类的创建者为类建立其形式：有方法名、参数列表和返回类型，但是没有任何方法体。接口也可以包含有数据成员，但是它们隐含都是 **static** 和 **final** 的。接口只提供了形式，而未提供任何具体实现。

一个接口表示：“所有实现了该特定接口的类看起来都像它”。因此，任何使用某特定接口的代码都知道可能会调用该接口的那些方法，而且仅需知道这些。因此，接口被用来建立类与类之间的“协议（**protocol**）”。（某些面向对象编程语言使用关键字 **protocol** 来完成这一功能。）

要想创建一个接口，需要用 **interface** 关键字来替代 **class** 关键字。就像类一样，你可以在 **interface** 关键字前面添加一个 **public** 关键字（但仅限于该接口在于其同名的文件中被定义），或者不添加它而使其只具有包访问权限，这样它就只能在同一个包内可用。

为了使一个类遵循某个特定接口（或者是一组接口），需要使用 **implements** 关键字，它表示：“该接口是这个类的外貌，但是现在我要声明他是如何运作的。”除此之外，它看起来还像是继承。有关乐器示例的图说明了这一点：



你可以从 **Woodwind** 和 **Brass** 类中看到，一旦你实现了某个接口，其实现就变成了一个普通的类，你可以按常规方式扩展它。

你可以选择在接口中显式地将方法声明为 **public** 的，但即使你不这么做，它们也是 **public** 的。因此，当你要实现一个接口时，在接口中被定义的方法必须要被定义为是 **public** 的。否则，它们将只能得到缺省的包访问权限，这样在方法被继承的过程中，其可访问权限就被降低了，这是 Java 编译器所不允许的。

你可以在修改过的 **Instrument** 的例子中看到这一点。要注意的是，在接口中的每一个方法确实都只是一个声明，这是编译器所允许的在接口中唯一能够存在的事物。此外，在 **Instrument** 中没有任何方法被声明为是 **public** 的，但是它们自动就是 **public** 的：

```
//: c08:music5:Music5.java
// Interfaces.
package c08.music5;
import com.bruceeckel.simpletest.*;
import c07.music.Note;

interface Instrument {
    // Compile-time constant:
```

```

    int I = 5; // static & final
    // Cannot have method definitions:
    void play(Note n); // Automatically public
    String what();
    void adjust();
}

class Wind implements Instrument {
    public void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}

class Percussion implements Instrument {
    public void play(Note n) {
        System.out.println("Percussion.play() " + n);
    }
    public String what() { return "Percussion"; }
    public void adjust() {}
}

class Stringed implements Instrument {
    public void play(Note n) {
        System.out.println("Stringed.play() " + n);
    }
    public String what() { return "Stringed"; }
    public void adjust() {}
}

class Brass extends Wind {
    public void play(Note n) {
        System.out.println("Brass.play() " + n);
    }
    public void adjust() {
        System.out.println("Brass.adjust()");
    }
}

class Woodwind extends Wind {
    public void play(Note n) {
        System.out.println("Woodwind.play() " + n);
    }
}

```

```

    public String what() { return "Woodwind"; }
}

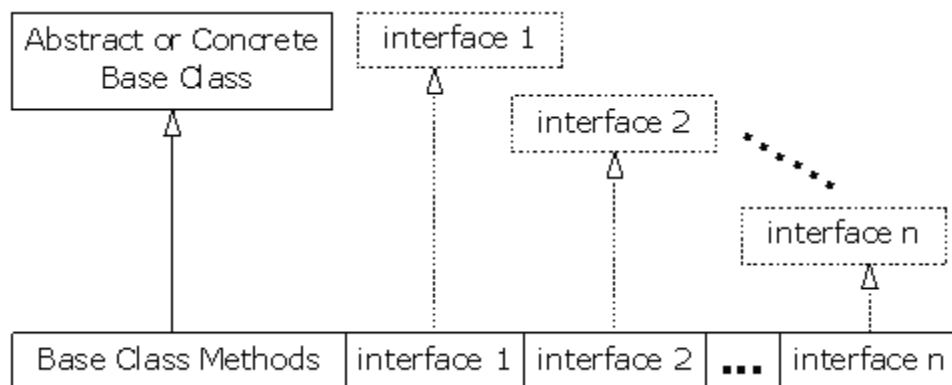
public class Music5 {
    private static Test monitor = new Test();
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
        monitor.expect(new String[] {
            "Wind.play() Middle C",
            "Percussion.play() Middle C",
            "Stringed.play() Middle C",
            "Brass.play() Middle C",
            "Woodwind.play() Middle C"
        });
    }
} ///:~

```

余下的代码其工作方式都是相同的。无论你是要将其向上转型为一个被称为 **Instrument** 的常规类，或是一个被称为 **Instrument** 的抽象类，还是一个被称为 **Instrument** 的接口，都不会有问题。它的行为都是相同的。事实上，你可以在 `tune()` 方法中看到，没有任何依据来证明 **Instrument** 是一个常规类、一个抽象类，还是一个接口。这样做是有目的的：每种方式都为程序员提供了不同的用来控制对象的创建与使用的方法。

Java 中的多重继承

接口不仅仅只是一种形式更纯粹的抽象类，它的目标比这要高。因为接口是根本没有任何具体实现的——也就是说，没有任何与接口相关的存储——因此，也就无法阻止多个接口的组合。这一点是很有价值的，因为你有时需要去表示“一个 **x** 是一个 **a**、一个 **b** 以及一个 **c**”。在 C++ 中，组合多个类的接口的行为被称作“多重继承（multiple inheritance）”。它可能会使你背负很棘手的包袱，因为每个类都有一个具体实现。在 Java 中，你可以执行相同的行为，但是只有一个类可以有具体实现，因此，通过组合多个接口，在 C++ 中看到的问题是不会在 Java 中发生的：



在一个导出类中，强制要求你必须有一个基类，它要么是抽象的，要么是“具体的”（没有任何抽象方法的类）。如果你要从一个非接口的类继承，那么你能只能从一个这样的类中去继承。其余的基元素都必须是接口。你需要将所有的接口名都置于 **implements** 关键字之后，用逗号将它们一一隔开。你可以继承任意多个接口，每一个都会成为一个你可以向上转型的独立类型。下面的例子展示的是：一个具体类组合数个接口之后产生了一个新类：

```
//: c08:Adventure.java
// Multiple interfaces.
```

```
interface CanFight {
    void fight();
}
```

```
interface CanSwim {
    void swim();
}
```

```
interface CanFly {
    void fly();
}
```

```
class ActionCharacter {
```



```

    public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}

public class Adventure {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
} ///:~

```

你可以看到，**Hero** 组合了具体类 **ActionCharacter** 和接口 **CanFight**、**CanSwim** 和 **CanFly**。当你要通过这种方式将一个具体类和多个接口组合到一起时，这个具体类必须是先行，后面跟着的才是接口。（否则编译器会报错。）

注意，**CanFight** 接口与 **ActionCharacter** 类中的 **fight()** 方法的签名是一样的，而且，在 **Hero** 中并没有提供 **fight()** 的定义。接口的规则是：你可以从接口中继承（就像稍后你会看到的那样），但是你得到的只是另一个接口。如果你想创建该新类型的对象，就必须有一个提供了其全部定义的类。即使 **Hero** 没有显式地提供 **fight()** 的定义，其定义也随 **ActionCharacter** 而存在，因此它是被自动提供的，这使得创建 **Hero** 对象成为了可能。

在 **Adventure** 类中，你可以看到有四个方法把上述各种接口和具体类作为参数。当 **Hero** 对象被创建时，它可以被传递给这些方法中的任何一个，这意味着它依次被向上转型为每一个接口。由于 Java 中这种接口设计的方式，使得这项工作并不需要程序员一方付出任何特别的努力。

一定要记住，前面的例子所展示的就是使用接口的核心原因：为了能够向上转型为不止一个的基类型。然而，使用接口的第二个原因却是与使用抽象基类相同：防止客户端程序员创建该类的对象，并确保这仅仅是建立一个接口。这就带来了一个问题：我们应该使用接口还是抽象类？接口为你带来了使用抽象类的好处，并且还带来了使用接口的好处，所以如果你要创建不带任何方法定义和成员变量的基类，那么你应该选择接口而不是抽象类。事实上，如果你知道某事物应该成为一个基类，那么你的第一选择应该是使它成为一个接口，只有在强

制你必须要有方法定义和成员变量的时候，你才应该改而选择抽象类，或者在必要时使其成为一个具体类。

组合接口时的名字冲突

在实现多重继承时，你可能会碰到一个小缺陷。在前面的例子中，**CanFight** 和 **ActionCharacter** 都有一个相同的 **void fight()** 方法。这并不是问题，因为该方法在二者中是相同的。但是如果它们不相同又会怎么样呢？这有一个例子：

```
//: c08:InterfaceCollision.java

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // overloaded
}

class C3 extends C implements I2 {
    public int f(int i) { return 1; } // overloaded
}

class C4 extends C implements I3 {
    // Identical, no problem:
    public int f() { return 1; }
}

// Methods differ only by return type:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

此时困难发生了，因为重载、实现和覆写令人不快地搅在了一起，而且被重载的方法仅通过返回类型是不能区分开的。当撤销最后两行的注释时，错误消息就说明了这一切：

```
InterfaceCollision.java:23: f( ) in C cannot implement f( ) in I1; attempting
to use incompatible return type
found : int
required: void
InterfaceCollision.java:24: interfaces I3 and I1 are incompatible; both
define f( ), but with different return type
```

在想要组合的不同接口中使用相同的方法名通常会造成代码可读性的混乱,请尽量避免这种情况。

通过继承来扩展接口

通过继承,你可以很容易地在接口中添加新的方法声明,你还可以通过继承在新接口中组合数个接口。在两种情况下,你都可以获得新的接口,就像在下例中所看到的:

```
//: c08:HorrorShow.java
// Extending an interface with inheritance.

interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
    void drinkBlood();
}

class VeryBadVampire implements Vampire {
    public void menace() {}
    public void destroy() {}
    public void kill() {}
    public void drinkBlood() {}
}

public class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
```

```

        d.menace();
        d.destroy();
    }
    static void w(Lethal l) { l.kill(); }
    public static void main(String[] args) {
        DangerousMonster barney = new DragonZilla();
        u(barney);
        v(barney);
        Vampire vlad = new VeryBadVampire();
        u(vlad);
        v(vlad);
        w(vlad);
    }
} ///:~

```

DangerousMonster 是 **Monster** 的直接扩展，它产生了一个新接口。**DragonZilla** 实现了这个接口。

在 **Vampire** 中使用的语法仅适用于接口继承。一般情况下，你只可以将 **extends** 用于一个单一的类，但是既然接口可以由多个其它接口产生，那么在创建一个新接口时，**extends** 当然可以引用多个基类接口。就像你可以看到的，只需用逗号将接口名一一分隔开即可。

群组常量

因为你放入接口中的任何数据成员都自动是 **static** 和 **final** 的，所以接口就成为了一种很便捷的用来创建常量组的工具，它非常象 C 或 C++ 中的 **enum**（枚举类型）。例如：

```

///: c08:Months.java
/// Using interfaces to create groups of constants.
package c08;

public interface Months {
    int
        JANUARY = 1, FEBRUARY = 2, MARCH = 3,
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
        NOVEMBER = 11, DECEMBER = 12;
} ///:~

```

请注意 Java 中用来标识具有初始化常量值的 **static final** 时，使用大写字母的风格（在一个单一的标识符中用下划线来分隔多个单词）。

接口中的数据成员自动是 **public** 的，所以它不需要特别指明这一点。

你可以使用包外部的常量，其方式是先导入 `c08.*` 或者是 `c08.Months`，就像使用其它的包一样，然后用诸如 `Months.JANUARY` 这样的表达式来引用它的值。当然，你所获得的只是一个 `int`，所以不存在像 C++ 的 `enum` 所具备的额外的类型安全，但是这项被广泛应用的技术相对于在程序中硬编码数字自然是一种提高。（硬编码数字的方式通常被称为“魔幻数字（magic number）”，它会产生难以维护的代码。）

如果你想要额外的类型安全，那么你可以象下面这样构建你的类¹：

```
//: c08:Month.java
// A more robust enumeration system.
package c08;
import com.bruceeckel.simpletest.*;

public final class Month {
    private static Test monitor = new Test();
    private String name;
    private Month(String nm) { name = nm; }
    public String toString() { return name; }
    public static final Month
        JAN = new Month("January"),
        FEB = new Month("February"),
        MAR = new Month("March"),
        APR = new Month("April"),
        MAY = new Month("May"),
        JUN = new Month("June"),
        JUL = new Month("July"),
        AUG = new Month("August"),
        SEP = new Month("September"),
        OCT = new Month("October"),
        NOV = new Month("November"),
        DEC = new Month("December");
    public static final Month[] month = {
        JAN, FEB, MAR, APR, MAY, JUN,
        JUL, AUG, SEP, OCT, NOV, DEC
    };
    public static final Month number(int ord) {
        return month[ord - 1];
    }
    public static void main(String[] args) {
        Month m = Month.JAN;
        System.out.println(m);
    }
}
```

¹这个方法的灵感来自于 Rich Hoffarth 的一封 e-mail。在 Joshua Bloch 的 *Effective Java* (Addison-Wesley, 2001) 中的第 21 条更详细地讨论了这个主题。

```

        m = Month.number(12);
        System.out.println(m);
        System.out.println(m == Month.DEC);
        System.out.println(m.equals(Month.DEC));
        System.out.println(Month.month[3]);
        monitor.expect(new String[] {
            "January",
            "December",
            "true",
            "true",
            "April"
        });
    }
} ///:~

```

Month 是一个具有一个 **private** 构造器的 **final** 类，所以无法继承它，也无法创建它的任何实例。所有的实例都是在该类内部由其自身创建的 **final static** 实例：**JAN**、**FEB**、**MAR** 等等。这些对象也会在 **month** 数组中被用到，该数组允许你遍历一个由 **Month2** 对象构成的数组。**number()**方法允许你通过给定表示月份的数字来选取相应的 **Month** 对象。在 **main()** 中，你可以看到类型安全性：**m** 是一个 **Month** 对象，因此它仅可以被赋予一个 **Month** 值。而前面的例子 **Months.java** 仅提供了 **int** 值，因此用来表示月份的 **int** 变量实际上可以被赋予任何整数值，这就显得不是很安全了。

这种方式同样允许你可互换地使用 **==** 或 **equals()**，就像在 **main()** 结尾处所展示的那样。之所以能这样做是因为 **Month** 的每一个值都仅有一个实例。在第 11 章中，你将学习到另一种创建类以使其对象可以互相彼此比较的方法。

在 **java.util.Calendar** 中也有一个表示月份的数据成员。

Apache 的 Jakarta Commons 的项目包含了用来创建枚举类型的工具，它与前面所示的例子相似，但是所需花费的精力更少。请查看 <http://jakarta.apache.org/commons>，在“lang”下面，**org.apache.commons.lang.enum** 包中。这个项目也包含了其它许多有用的类库。

初始化接口中的数据成员

在接口中定义的数据成员自动是 **static** 和 **final** 的。它们不能是“空 **final**”，但是可以被非常量表达式初始化。例如：

```

///: c08:RandVals.java
// Initializing interface fields with
// non-constant initializers.
import java.util.*;

```

```

public interface RandVals {
    Random rand = new Random();
    int randomInt = rand.nextInt(10);
    long randomLong = rand.nextLong() * 10;
    float randomFloat = rand.nextLong() * 10;
    double randomDouble = rand.nextDouble() * 10;
} ///:~

```

既然数据成员是 **static** 的，它们就可以在类第一次被加载时初始化，这发生在任何数据成员首次被访问时。这里给出一个简单的测试：

```

///: c08:TestRandVals.java
import com.bruceeckel.simpletest.*;

public class TestRandVals {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        System.out.println(RandVals.randomInt);
        System.out.println(RandVals.randomLong);
        System.out.println(RandVals.randomFloat);
        System.out.println(RandVals.randomDouble);
        monitor.expect(new String[] {
            "%% -?\\d+",
            "%% -?\\d+",
            "%% -?\\d\\.\\d+E?-?\\d+",
            "%% -?\\d\\.\\d+E?-?\\d+"
        });
    }
} ///:~

```

当然，这些数据成员不是接口的一部分，只是被存储在该接口的静态存储区域内。

嵌套接口

接口可以嵌套在类或其它接口中²。这揭示了许多非常有趣的特性：

```

///: c08:nesting:NestingInterfaces.java
package c08.nesting;

class A {
    interface B {

```

²感谢Martin Danner在研讨会上提出这个问题。

```

        void f();
    }
    public class BImp implements B {
        public void f() {}
    }
    private class BImp2 implements B {
        public void f() {}
    }
    public interface C {
        void f();
    }
    class CImp implements C {
        public void f() {}
    }
    private class CImp2 implements C {
        public void f() {}
    }
    private interface D {
        void f();
    }
    private class DImp implements D {
        public void f() {}
    }
    public class DImp2 implements D {
        public void f() {}
    }
    public D getD() { return new DImp2(); }
    private D dRef;
    public void receiveD(D d) {
        dRef = d;
        dRef.f();
    }
}

interface E {
    interface G {
        void f();
    }
    // Redundant "public":
    public interface H {
        void f();
    }
    void g();
    // Cannot be private within an interface:

```



```

    ///! private interface I {}
}

public class NestingInterfaces {
    public class BImp implements A.B {
        public void f() {}
    }
    class CImp implements A.C {
        public void f() {}
    }
    // Cannot implement a private interface except
    // within that interface's defining class:
    ///! class DImp implements A.D {
    ///! public void f() {}
    ///! }
    class EImp implements E {
        public void g() {}
    }
    class EGImp implements E.G {
        public void f() {}
    }
    class EImp2 implements E {
        public void g() {}
        class EG implements E.G {
            public void f() {}
        }
    }
}

public static void main(String[] args) {
    A a = new A();
    // Can't access A.D:
    ///! A.D ad = a.getD();
    // Doesn't return anything but A.D:
    ///! A.DImp2 di2 = a.getD();
    // Cannot access a member of the interface:
    ///! a.getD().f();
    // Only another A can do anything with getD():
    A a2 = new A();
    a2.receiveD(a.getD());
}
} ///:~

```

在类中嵌套接口的语法是相当显而易见的，就像非嵌套接口一样，可以拥有 **public** 和包内访问两种可视性。你可以看到，**public** 和包内访问的嵌套接口都可以被实现为 **public**、包内访问以及 **private** 的嵌套类。

作为一种新的方式，接口也可以被实现为是 **private** 的，就像在 **A.D** 中所看到的一样（相同的语法既适用于嵌套接口，也适用于嵌套类）。那么 **private** 的嵌套类能带来什么好处呢？你可能会猜想，它只能够被实现为在 **DImp** 中的一个 **private** 的内部类，但是 **A.Dimp2** 展示了它同样可以被实现为 **public** 类。但是，**A.DImp2** 只能被其自身所使用。你无法提及这样的事实，它实现了一个 **private** 接口。因此，实现一个 **private** 接口只是一种方法，它可以强制该接口中的方法不能够添加任何类型信息（也就是说，不允许向上转型）。

getD()方法使我们陷入了与 **private** 接口相关的一个更加窘迫的境地：它是一个返回对 **private** 接口的引用的 **public** 方法。你对这个方法的返回值能做些什么呢？在 **main()**中，你可以看到数次尝试使用返回值的行为都失败了。只有一种方式可成功，那就是将返回值交给有权使用它的对象。在本例中，是另外一个 **A** 通过 **receiveD()**方法来实现的。

接口 **E** 展示了接口彼此之间也可以嵌套。然而，作用于接口的各种规则，特别是所有的接口元素必须是 **public** 的这一条，在此都会被严格执行。

NestingInterfaces 展示了嵌套类的各种实现方式。特别要注意的是，当你实现某个接口时，并不需要实现嵌套在其内部的任何接口。而且，**private** 接口不能在定义它的类之外被实现。

添加这些特性的最初原因可能是出于对严格的语法一致性的考虑，但是我总认为，一旦你了解了某种特性，你就总能够找到它的用武之地。

内部类

可以将一个类的定义放在另一个类的定义内部，这就是内部类(inner class)。内部类是一种非常有用的特性，因为它允许你把一些逻辑相关的类组织在一起，并控制在内部的类的可视性。然而必须要了解，内部类与组合是完全不同的概念，这一点很重要。

在你学习内部类的过程中，并不一定清楚什么时候需要使用内部类。在这一节末尾，在学习了内部类所有的语法和语义之后，会给你一些例子，以帮助你清楚地理解内部类所带来的好处。

创建一个内部类就同你想的一样——把类的定义置于环绕类(surrounding class)的里面：

```
//: c08:Parcell.java
// Creating inner classes.

public class Parcell {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
```

```

        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    // Using inner classes looks just like
    // using any other class, within Parcel1:
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
} ///:~

```

当我们在 ship() 方法里面使用内部类的时候，与使用其他类没什么不同。在这里，实际的区别只是内部类的名字是嵌套在 Parcel1 里面的。不过你将会看到，这并不是唯一的区别。

典型的情况是，外部类会有一个方法，返回一个指向内部类的引用，就像这样：

```

///: c08:Parcel2.java
// Returning a reference to an inner class.

public class Parcel2 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        return new Destination(s);
    }
    public Contents cont() {
        return new Contents();
    }
}

```

```

    }
    public void ship(String dest) {
        Contents c = cont();
        Destination d = to(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel2 p = new Parcel2();
        p.ship("Tanzania");
        Parcel2 q = new Parcel2();
        // Defining references to inner classes:
        Parcel2.Contents c = q.cont();
        Parcel2.Destination d = q.to("Borneo");
    }
} ///:~

```

如果你想从外部类的非静态方法之外的任意位置创建某个内部类的对象，那么你必须像 `main()` 方法中那样，具体地指明这个对象的类型：*OuterClassName.InnerClassName*。

内部类与向上转型

到目前为止，内部类似乎并没有什么了不起的。毕竟，如果你只是想用于隐藏，Java 已经有了很好的隐藏机制——只给予某个类“包访问权”（仅在同一个包内可见），而用不着创建为内部类。

然而，当你将内部类向上转型为其基类，尤其是转型为一个接口的时候，内部类就有了用武之地。（从实现了某个接口的对象，得到对此接口的引用，与向上转型为这个对象的基类，实质上效果是一样的。）这是因为此内部类——某个接口的实现——对于其他人来说能够完全不可见，并且不可用。你所得到的只是指向基类或接口的一个引用，所以能够很方便地隐藏实现细节。

首先，我们将通用接口定义在独立的文件中，这样就可以在所有的例子中使用它们：

```

///: c08:Destination.java
public interface Destination {
    String readLabel();
} ///:~

///: c08:Contents.java
public interface Contents {
    int value();
} ///:~

```

现在 `Contents` 和 `Destination` 表示客户端程序员可用的接口。（记住，接口的所有成员自动被设置为 `public`。）

当你取得了一个指向基类或接口的引用时，你可能无法找出它确切的型别，看下面的例子：

```
///  
// Returning a reference to an inner class.  
  
class Parcel3 {  
    private class PContents implements Contents {  
        private int i = 11;  
        public int value() { return i; }  
    }  
    protected class PDestination implements Destination {  
        private String label;  
        private PDestination(String whereTo) {  
            label = whereTo;  
        }  
        public String readLabel() { return label; }  
    }  
    public Destination dest(String s) {  
        return new PDestination(s);  
    }  
    public Contents cont() {  
        return new PContents();  
    }  
}  
  
public class TestParcel {  
    public static void main(String[] args) {  
        Parcel3 p = new Parcel3();  
        Contents c = p.cont();  
        Destination d = p.dest("Tanzania");  
        // Illegal -- can't access private class:  
        //! Parcel3.PContents pc = p.new PContents();  
    }  
} ///:~
```

在上例中，`main()` 方法必须在一个单独的类中，才能演示出内部类 `PContents` 是私有的这个性质。

`Parcel3` 中增加了一些新东西：内部类 `PContents` 是 `private`，所以除了 `Parcel3`，没有人能访问它。`PDestination` 是 `protected`，所以只有 `Parcel3` 以及其子类，还有与 `Parcel3` 同一个包中的类（因为 `protected` 也给予了包访问权）能访问 `PDestination`，其他类都不

能访问 `PDestination`。这意味着，如果客户端程序员想了解，或访问这些成员，那是要受到限制的。实际上，你甚至不能向下转型成 `private` 内部类（或 `protected` 内部类，除非你有继承自它的子类），因为你不能访问其名字，就像你在 `TestParcel` 类中看到的那样。于是，`private` 内部类给类的设计者提供了一种途径，以完全阻止任何依赖于类型的编码，并且完全隐藏了实现的细节。此外，从客户端程序员的角度来看，由于不能访问任何新增加的、原本不属于公共接口的方法，所以扩展接口是没有价值的。这也给 Java 编译器提供了生成更高效代码的机会。

普通的（非内部的）类，不能声明为 `private` 或 `protected`；它们只可以被赋予 `public` 或者包访问权。

在方法和作用域内的内部类

目前为止，你所看到的只是内部类典型的用途。通常，如果你要读写的代码包含了内部类，那么它们都是“平凡的”内部类，简单并且容易理解。然而，内部类的设计却是相当完备的。如果你选择使用内部类，它还有许多难以理解的使用方式。例如，可以在一个方法里面或者在任意的作用域内定义内部类。这么做有两个理由：

1. 如前所示，你实现了某类型的接口，于是可以创建并返回对其的引用。
2. 你要解决一个复杂的问题，想创建一个类来辅助你的解决方案，但是又不希望这个类是公共可用的。

在后面的例子中，先前的代码将被修改以用来实现：

1. 一个定义在方法中的类
2. 一个定义在作用域内的类，此作用域在方法的内部
3. 一个实现了接口的匿名类
4. 一个匿名类，扩展了有非缺省构造器的类
5. 一个匿名类，执行成员初始化
6. 一个匿名类，通过实例初始化实现构造（匿名类不可能有构造器）

虽然 `Wrapping` 只是一个普通的类，但同时也被其导出类作为通用“接口”使用。

```
//: c08:Wrapping.java
public class Wrapping {
    private int i;
    public Wrapping(int x) { i = x; }
    public int value() { return i; }
} ///:~
```

请注意 `Wrapping` 的构造器，它需要一个参数，这使得事情更有趣了。

第一个例子展示了在方法的作用域内（而不是在其它类的作用域内）创建一个完整的类。这被称作局部内部类(local inner class)：

```
//: c08:Parcel4.java
// Nesting a class within a method.

public class Parcel4 {
    public Destination dest(String s) {
        class PDestination implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public String readLabel() { return label; }
        }
        return new PDestination(s);
    }
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Destination d = p.dest("Tanzania");
    }
} ///:~
```

与其说 PDestination 类是 Parcel4 的一部分，不如说是 dest() 方法的一部分。（注意到，你可以在同一个子目录下的任意类中定义名为 PDestination 的内部类，这并不会命名冲突。）所以，在 dest() 之外不能访问 PDestination。注意出现在 return 语句中的向上转型——返回的是 Destination 的引用，它是 PDestination 的基类。当然，在 dest() 中定义了内部类 PDestination，并不意味着一旦 dest() 方法执行完毕，PDestination 就不可用了。

下一个例子展示了如何在任意的作用域内嵌入一个内部类：

```
//: c08:Parcel5.java
// Nesting a class within a scope.

public class Parcel5 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) {
                    id = s;
                }
            }
            String getSlip() { return id; }
        }
    }
}
```

```

    }
    TrackingSlip ts = new TrackingSlip("slip");
    String s = ts.getSlip();
}
// Can't use it here! Out of scope:
//! TrackingSlip ts = new TrackingSlip("x");
}
public void track() { internalTracking(true); }
public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    p.track();
}
} ///:~

```

TrackingSlip 类被嵌入在 if 语句的作用域内，这并不是说它的创建是有条件的，它其实与别的类一样都经过编译了。然而，在定义 TrackingSlip 的作用域之外，它是不可用的。除此之外，它与普通的类一样。

匿名内部类

下面的例子看起来有点奇怪：

```

///: c08:Parcel6.java
// A method that returns an anonymous inner class.

public class Parcel6 {
    public Contents cont() {
        return new Contents() {
            private int i = 11;
            public int value() { return i; }
        }; // Semicolon required in this case
    }
    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        Contents c = p.cont();
    }
} ///:~

```

cont() 方法将下面两个动作合并在一起：返回值的生成，与表示这个返回值的类的定义！进一步说，这个类是匿名的，它没有名字。更糟的是，看起来是你正要创建一个 Contents 对象：

```

return new Contents()

```


但是，在到达语句结束的分号之前，你却说：“等一等，我想在这里插入一个类的定义”：

```
return new Contents() {  
    private int i = 11;  
    public int value() { return i; }  
};
```

这种奇怪的语法指的是：“创建一个继承自 `Contents` 的匿名类的对象。”通过 `new` 表达式返回的引用被自动向上转型为对 `Contents` 的引用。匿名内部类的语法是下面例子的简略形式：

```
class MyContents implements Contents {  
    private int i = 11;  
    public int value() { return i; }  
}  
return new MyContents();
```

在这个匿名内部类中，使用了缺省的构造器来生成 `Contents`。下面的代码展示的是，如果你的基类需要一个有参数的构造器，应该怎么办：

```
///  
// An anonymous inner class that calls  
// the base-class constructor.  
  
public class Parcel7 {  
    public Wrapping wrap(int x) {  
        // Base constructor call:  
        return new Wrapping(x) { // Pass constructor argument.  
            public int value() {  
                return super.value() * 47;  
            }  
        }; // Semicolon required  
    }  
    public static void main(String[] args) {  
        Parcel7 p = new Parcel7();  
        Wrapping w = p.wrap(10);  
    }  
} ///  
~
```

只需简单地传递合适的参数给基类的构造器即可，这里是将 `x` 传进 `new Wrapping(x)`。

在匿名内部类末尾的分号，并不是用来标记此内部类结束（C++中是那样）。实际上，它标记的是表达式的结束，只不过这个表达式正巧包含了内部类罢了。因此，这与别的地方使用的分号是一致的。

如果在匿名类中定义成员变量，你同样能够对其执行初始化操作：

```
//: c08:Parcel8.java
// An anonymous inner class that performs
// initialization. A briefer version of Parcel4.java.

public class Parcel8 {
    // Argument must be final to use inside
    // anonymous inner class:
    public Destination dest(final String dest) {
        return new Destination() {
            private String label = dest;
            public String readLabel() { return label; }
        };
    }

    public static void main(String[] args) {
        Parcel8 p = new Parcel8();
        Destination d = p.dest("Tanzania");
    }
} ///:~
```

如果你有一个匿名内部类，它要使用一个在它的外部定义的对象，编译器会要求其参数引用是 **final** 型的，就像 **dest()** 中的参数。如果你忘记了，会得到一个编译期错误信息。

如果只是简单地给一个成员变量赋值，那么此例中的方法就可以了。但是，如果你想做一些类似构造器的行为，该怎么办呢？在匿名类中不可能有已命名的构造器（因为它根本没有名字！），但通过实例初始化，你就能够达到为匿名内部类“制作”一个构造器的效果。像这样做：

```
//: c08:AnonymousConstructor.java
// Creating a constructor for an anonymous inner class.
import com.bruceeckel.simpletest.*;

abstract class Base {
    public Base(int i) {
        System.out.println("Base constructor, i = " + i);
    }

    public abstract void f();
}
```

```

public class AnonymousConstructor {
    private static Test monitor = new Test();
    public static Base getBase(int i) {
        return new Base(i) {
            {
                System.out.println("Inside instance initializer");
            }
            public void f() {
                System.out.println("In anonymous f()");
            }
        };
    }
    public static void main(String[] args) {
        Base base = getBase(47);
        base.f();
        monitor.expect(new String[] {
            "Base constructor, i = 47",
            "Inside instance initializer",
            "In anonymous f()"
        });
    }
} ///:~

```

在此例中，不要求变量 **i** 一定是 **final** 的。因为 **i** 被传递给匿名类的基类的构造器，它并不会在匿名类内部被直接使用。

下例是带实例初始化的“**parcel**”形式。注意 **dest()** 的参数必须是 **final**，因为它们是在匿名类内被使用的。

```

///: c08:Parcel9.java
// Using "instance initialization" to perform
// construction on an anonymous inner class.
import com.bruceeckel.simpletest.*;

public class Parcel9 {
    private static Test monitor = new Test();
    public Destination
    dest(final String dest, final float price) {
        return new Destination() {
            private int cost;
            // Instance initialization for each object:
            {
                cost = Math.round(price);
                if(cost > 100)

```

```

        System.out.println("Over budget!");
    }
    private String label = dest;
    public String readLabel() { return label; }
};
}
public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.dest("Tanzania", 101.395F);
    monitor.expect(new String[] {
        "Over budget!"
    });
}
} ///:~

```

在实例初始化的部分，你可以看到有一段代码，那原本是不能作为成员变量初始化的一部分而执行的（就是 if 语句）。所以对于匿名类而言，实例初始化的实际效果就是构造器。当然它受到了限制：你不能重载实例初始化，所以你只能有一个构造器。

链接到外部类

到目前为止，似乎内部类还只是一种隐藏命名和组织代码的模式，这些是很有用，但还不是最引人注目的。然而，它还有其他的用途。当你生成一个内部类的对象时，此对象与制造它的外围类的对象（**enclosing object**）之间就有了一种联系，所以它能访问其外围对象的所有成员，而不需要任何特殊条件。此外，内部类还拥有其外围类的所有元素的访问权³。下面的例子展示了这点：

```

///: c08:Sequence.java
// Holds a sequence of Objects.
import com.bruceeckel.simpletest.*;

interface Selector {
    boolean end();
    Object current();
    void next();
}

public class Sequence {
    private static Test monitor = new Test();
    private Object[] objects;

```

³这与C++嵌套类的设计非常不同，在C++中只是单纯的名称隐藏机制，与外围类的对象没有联系，也没有隐含的访问权。

```

private int next = 0;
public Sequence(int size) { objects = new Object[size]; }
public void add(Object x) {
    if(next < objects.length)
        objects[next++] = x;
}
private class SSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == objects.length; }
    public Object current() { return objects[i]; }
    public void next() { if(i < objects.length) i++; }
}
public Selector getSelector() { return new SSelector(); }
public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
        sequence.add(Integer.toString(i));
    Selector selector = sequence.getSelector();
    while(!selector.end()) {
        System.out.println(selector.current());
        selector.next();
    }
    monitor.expect(new String[] {
        "0",
        "1",
        "2",
        "3",
        "4",
        "5",
        "6",
        "7",
        "8",
        "9"
    });
}
} ///:~

```

`Sequence` 类只是一个固定大小的 `Object` 数组，以类的形式包装了起来。你可以调用 `add()`，在序列的尾端增加一个新的 `Object`（只要还有空间）。要获取 `Sequence` 中的每一个对象，可以使用 `Selector` 接口，方法 `end()` 允许你检查序列是否到末尾了，`current()` 查看当前对象，`next()` 移到序列中的下一个对象。因为 `Selector` 是一个接口，所以别的类可以按它们自己的方式来实现这个接口，并且许多方法能以此接口为参数，来生成通用的代码。

这里，SSelector 是提供 Selector 功能的 private 类。可以看到，在 main() 中创建了一个 Sequence，并向其中添加了一些 String 对象。然后通过调用 getSelector() 获取一个 Selector，并用它在 Sequence 中移动和选择每一个元素。

最初看到 SSelector，可能会觉得它只不过是另一个内部类罢了。但请仔细观察它。注意方法 end(), current(), 和 next() 都用到了 objects，这是一个引用，它并不是 SSelector 的一部分，而是外围类的一个 private 成员。所以内部类可以访问其外围类的方法和属性，就像自己拥有它们似的。这带来了很大的方便，就如前面的例子所示。

内部类自动拥有对其外围类所有成员的访问权。这是如何做到的呢？当某个外围类的对象创建了一个内部类对象时，此内部类对象必定会保存一个指向那个外围类对象的引用。然后，在你访问此外围类的成员时，就是用那个“隐藏的”引用来选择外围类的成员。幸运的是，编译器会帮你处理所有的细节，但你现在也知道了：内部类的对象只能在其外围类的对象相关联的情况下才能被创建。构建内部类对象时，需要一个指向其外围类对象的引用，如果编译器访问不到这个引用就会报错。不过绝大多数时候这都无需程序员操心。

嵌套类

如果你不需要内部类对象与其外围类对象之间的联系，那你可以将内部类声明为 static。这通常称为嵌套类（nested class）⁴。想要理解 static 应用于内部类时的含义，你就必须记住，普通的内部类对象隐含地保存了一个引用，指向创建它的外围类对象。然而，当内部类是 static 的时，就不是这样了。嵌套类意味着：

1. 要创建嵌套类的对象，并不需要其外围类的对象。
2. 不能从嵌套类的对象中访问非静态的外围类对象。

嵌套类与普通的内部类还有一个区别。普通内部类的属性与方法，只能放在类的外部层次上，所以普通的内部类不能有 static 数据和 static 属性，也不能包含嵌套类。但是嵌套类可以包含所有这些东西：

```
//: c08:Parcel10.java
// Nested classes (static inner classes).

public class Parcel10 {
    private static class ParcelContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }

    protected static class ParcelDestination
        implements Destination {
        private String label;
```

⁴与C++嵌套类大致相似，除了在C++中那些类不能访问私有成员，而在Java中可以访问。

```

private ParcelDestination(String whereTo) {
    label = whereTo;
}
public String readLabel() { return label; }
// Nested classes can contain other static elements:
public static void f() {}
static int x = 10;
static class AnotherLevel {
    public static void f() {}
    static int x = 10;
}
}
public static Destination dest(String s) {
    return new ParcelDestination(s);
}
public static Contents cont() {
    return new ParcelContents();
}
}
public static void main(String[] args) {
    Contents c = cont();
    Destination d = dest("Tanzania");
}
} ///:~

```

在 `main()` 中，没有任何 `Parcel10` 对象是必需的；取而代之的是，你要使用选取一个 `static` 成员的普通语法，来调用返回 `Contents` 和 `Destination` 引用的方法。

你将会看到，在一个普通的（非静态）内部类中，通过一个特殊的 `this` 引用，可以链接到其外围类对象。嵌套类就没有这个特殊的 `this` 引用，这使得它类似于一个 `static` 方法。

正常情况下，你不能在接口内部放置任何代码，但嵌套类可以作为接口的一部分，因为它是 `static` 的。只是将嵌套类置于接口的命名空间内，这并不违反接口的规则：

```

///: c08:IInterface.java
// Nested classes inside interfaces.

public interface IInterface {
    static class Inner {
        int i, j, k;
        public Inner() {}
        void f() {}
    }
} ///:~

```

我曾在本书中建议过，在每个类中都写一个 `main()` 方法，以用来测试这个类。这样做有一个缺点，你必须带着那些编译了的额外代码。如果这对你是个麻烦，你可以使用嵌套类来放置测试代码。

```
//: c08:TestBed.java
// Putting test code in a nested class.

public class TestBed {
    public TestBed() {}
    public void f() { System.out.println("f()"); }
    public static class Tester {
        public static void main(String[] args) {
            TestBed t = new TestBed();
            t.f();
        }
    }
} ///:~
```

这生成了一个独立的类 `TestBed$Tester`（要运行这个程序，执行 `java TestBed$Tester` 即可）。你可以使用这个类来做测试，但是你不必在发布的产品中包含它，只需简单地在将产品打包前删除 `TestBed$Tester.class`。

引用外围类的对象

如果你需要对外围类对象的引用，可以通过在此外围类名称后面跟一个句点和 `this` 关键字来获得。例如在类 `Sequence.SSelector` 中，通过 `Sequence.this`，任何方法都能够获取那个保存的指向外围类 `Sequence` 的引用。而作为结果获取的这个引用自动就是类型正确的。（它在编译期被获知并检查过，所以没有运行期的开销。）

有时候你会对某个对象说，我要生成你内部的某个内部类的一个对象。要想如此，你必须在 `new` 表达式中提供一个引用，指向那个外围类的对象，就像这样：

```
//: c08:Parcel11.java
// Creating instances of inner classes.

public class Parcel11 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) { label = whereTo; }
    }
}
```



```

        String readLabel() { return label; }
    }
    public static void main(String[] args) {
        Parcel11 p = new Parcel11();
        // Must use instance of outer class
        // to create an instances of the inner class:
        Parcel11.Contents c = p.new Contents();
        Parcel11.Destination d = p.new Destination("Tanzania");
    }
} ///:~

```

如果要直接创建内部类的对象，不能想当然地认为只需加上外围类 **Parcel11** 的名字，就可以按照通常的样子生成内部类的对象，而是必须使用此外围类的一个对象来创建其内部类的一个对象：

```
Parcel11.Contents c = p.new Contents();
```

因此，除非你已经有了外围类的一个对象，否则不可能生成内部类的对象。因为此内部类的对象会悄悄地链接到创建它的外围类的对象。如果你用的是嵌套类（静态的内部类），那就不需要对其外围类对象的引用。

从多层嵌套类中访问外部

⁵一个内部类被嵌套多少层并不重要——它能透明地访问所有它所嵌入的外围类的所有成员，如下所示：

```

///: c08:MultiNestingAccess.java
// Nested classes can access all members of all
// levels of the classes they are nested within.

class MNA {
    private void f() {}
    class A {
        private void g() {}
        public class B {
            void h() {
                g();
                f();
            }
        }
    }
}

```

⁵再次感谢Martin Danner。

```

    }

    public class MultiNestingAccess {
        public static void main(String[] args) {
            MNA mna = new MNA();
            MNA.A mnaa = mna.new A();
            MNA.A.B mnaab = mnaa.new B();
            mnaab.h();
        }
    } ///:~

```

可以看到在 **MNA.A.B** 中，调用方法 **g()** 和 **f()** 不需要任何条件（即使它们被定义为 **private**）。这个例子同时展示了如何从不同的类里面创建多层嵌套的内部类对象的基本语法。“**.new**”语法能产生正确的作用域，所以你不必在调用构造器时限定类名。

内部类的继承

因为内部类的构造器要用到其外围类对象的引用，所以在你继承一个内部类的时候，事情变得有点复杂。问题在于，那个“秘密的”外围类对象的引用必须被初始化，而在被继承的类中并不存在要联接的缺省对象。要解决这个问题，需使用专门的语法来明确说清它们之间的关联：

```

///: c08:InheritInner.java
// Inheriting an inner class.

class WithInner {
    class Inner {}
}

public class InheritInner extends WithInner.Inner {
    ///! InheritInner() {} // Won't compile
    InheritInner(WithInner wi) {
        wi.super();
    }
    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
} ///:~

```

可以看到，**InheritInner** 只继承自内部类，而不是外围类。但是当要生成一个构造器时，缺省的构造器并不算好，而且你不能只是传递一个指向外围类对象的引用。此外，你必须在构造器内使用如下语法：

```
enclosingClassReference.super();
```

这样才提供了必要的引用，然后程序才能编译通过。

内部类可以被重载吗？

如果你创建了一个内部类，然后继承其外围类并重新定义此内部类时，会发生什么呢？也就是说，内部类可以被重载吗？这看起来似乎是个很有用的点子，但是“重载”内部类就好像它是外围类的一个方法，其实并不起什么作用：

```
//: c08:BigEgg.java
// An inner class cannot be overridden like a method.
import com.bruceeckel.simpletest.*;

class Egg {
    private Yolk y;
    protected class Yolk {
        public Yolk() { System.out.println("Egg.Yolk()"); }
    }
    public Egg() {
        System.out.println("New Egg()");
        y = new Yolk();
    }
}

public class BigEgg extends Egg {
    private static Test monitor = new Test();
    public class Yolk {
        public Yolk() { System.out.println("BigEgg.Yolk()"); }
    }
    public static void main(String[] args) {
        new BigEgg();
        monitor.expect(new String[] {
            "New Egg()",
            "Egg.Yolk()"
        });
    }
} ///:~
```

缺省的构造器是编译器自动生成的，这里是调用基类的缺省构造器。你可能认为既然创建了 `BigEgg` 的对象，那么所使用的应该被“重载”过的 `Yolk`，但你可以从输出中看到实际情况并不是这样的。

这个例子说明，当你继承了某个外围类的时候，内部类并没有发生什么特别神奇的变化。这两个内部类是完全独立的两个实体，各自在自己的命名空间内。当然，明确地继承某个内部类也是可以的：

```
//: c08:BigEgg2.java
// Proper inheritance of an inner class.
import com.bruceeckel.simpletest.*;

class Egg2 {
    protected class Yolk {
        public Yolk() { System.out.println("Egg2.Yolk()"); }
        public void f() { System.out.println("Egg2.Yolk.f()"); }
    }

    private Yolk y = new Yolk();
    public Egg2() { System.out.println("New Egg2()"); }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
    private static Test monitor = new Test();
    public class Yolk extends Egg2.Yolk {
        public Yolk() { System.out.println("BigEgg2.Yolk()"); }
        public void f() {
            System.out.println("BigEgg2.Yolk.f()");
        }
    }

    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
        monitor.expect(new String[] {
            "Egg2.Yolk()",
            "New Egg2()",
            "Egg2.Yolk()",
            "BigEgg2.Yolk()",
            "BigEgg2.Yolk.f()"
        });
    }
} ///:~
```

现在 BigEgg2.Yolk 通过 extends Egg2.Yolk 明确地继承了此内部类，并且重载了其中的方法。Egg2 的 insertYolk() 方法使得 BigEgg2 将它自己的 Yolk 对象向上转型，然后传递给引用 y。所以当 g() 调用 y.f() 时，重载后的新版 f() 被执行。第二次调用 Egg2.Yolk()

是 BigEgg2.Yolk 的构造器调用了其基类的构造器。可以看到在调用 g() 的时候，新版的 f() 被调用了。

局部内部类（Local inner classes）

前面提到过，可以在代码块里面创建内部类，典型的方式是在一个方法体的里面创建。局部内部类不能有访问说明符，因为它不是外围类的一部分，但是它可以访问当前代码块内的常量，和此外围类所有的成员。下面的例子是对局部内部类与匿名内部类的创建作比较。

```
//: c08:LocalInnerClass.java
// Holds a sequence of Objects.
import com.bruceeckel.simpletest.*;

interface Counter {
    int next();
}

public class LocalInnerClass {
    private static Test monitor = new Test();
    private int count = 0;
    Counter getCounter(final String name) {
        // A local inner class:
        class LocalCounter implements Counter {
            public LocalCounter() {
                // Local inner class can have a constructor
                System.out.println("LocalCounter()");
            }
            public int next() {
                System.out.print(name); // Access local final
                return count++;
            }
        }
        return new LocalCounter();
    }
    // The same thing with an anonymous inner class:
    Counter getCounter2(final String name) {
        return new Counter() {
            // Anonymous inner class cannot have a named
            // constructor, only an instance initializer:
            {
                System.out.println("Counter()");
            }
            public int next() {
                System.out.print(name); // Access local final
            }
        };
    }
}
```

```

        return count++;
    }
};
}

public static void main(String[] args) {
    LocalInnerClass lic = new LocalInnerClass();
    Counter
        c1 = lic.getCounter("Local inner "),
        c2 = lic.getCounter2("Anonymous inner ");
    for(int i = 0; i < 5; i++)
        System.out.println(c1.next());
    for(int i = 0; i < 5; i++)
        System.out.println(c2.next());
    monitor.expect(new String[] {
        "LocalCounter()",
        "Counter()",
        "Local inner 0",
        "Local inner 1",
        "Local inner 2",
        "Local inner 3",
        "Local inner 4",
        "Anonymous inner 5",
        "Anonymous inner 6",
        "Anonymous inner 7",
        "Anonymous inner 8",
        "Anonymous inner 9"
    });
}
} ///:~

```

Counter 返回的是序列中的下一个值。我们分别使用局部内部类和匿名内部类实现了这个功能，它们有一样的行为和能力。既然局部内部类的名字在方法外是不可见的，那我们为什么仍然使用局部内部类而不是匿名类呢？唯一理由是，你需要一个已命名的构造器，或者需要重载构造器，而匿名类只能用于实例初始化。

所以使用局部内部类而不使用匿名内部类唯一的原因就是，你需要不止一个此内部类的对象。

内部类标识符

由于每个类都会产生一个.class 文件，其中包含了如何创建该类对象的全部信息（此信息产生一个“meta-class”，叫作 Class 对象），你可能猜到了，内部类也必须生成一个.class

文件以包含它们的 Class 对象信息。这些类文件的命名有严格的规则：外围类的名字，加上 ‘\$’，再加上内部类的名字。例如，LocalInnerClass.java 生成的.class 文件包括：

```
Counter.class  
LocalInnerClass$2.class  
LocalInnerClass$1LocalCounter.class  
LocalInnerClass.class
```

如果内部类是匿名的，编译器会简单地产生一个数字作为其标识符。如果内部类是嵌套在别的内部类之中，只需直接将它们的名字加在其外围类标识符与‘\$’的后面。

虽然这种命名格式简单而直接，但它还是很健壮的，足以应对绝大多数情况⁶。因为这是Java的标准命名方式，所以产生的文件自动都是平台无关的。（注意，为了保证你的内部类能起作用，Java编译器会尽可能地转换它们。）

为什么需要内部类？

你已经看到了许多描述内部类的语法和语义，但是这并不能说明“为什么需要内部类？”。那么，Sun 公司为什么会如此费心地增加这项基本的语言特性呢？

典型的情况是，内部类继承自某个类或实现某个接口，内部类的代码操作创建其的外围类的对象。所以你可以认为内部类提供了某种进入其外围类的窗口。

内部类必须要回答一个问题是：如果我只是需要一个对接口的引用，为什么我不通过外围类实现那个接口呢？答案是：“如果这能满足你的需求，那么你就应该这样做。”那么内部类实现一个接口与外围类实现这个接口有什么区别呢？答案是你不是总能享用到接口带来的方便，有时你需要与接口的实现进行交互，所以使用内部类最吸引人的原因是：

每个内部类都能独立地继承自一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。

如果没有内部类提供的可以继承多个具体的或抽象的类的能力，一些设计与编程问题就很难解决。从这个角度看，内部类使得多重继承的解决方案变得完整。接口解决了部分问题，而内部类有效地实现了“多重继承”。也就是说，内部类允许你继承多个非接口类型（译注：类或抽象类）。

为了看到更多的细节，考虑下面这种情形，你必须在一个类中以某种方式实现两个接口。由于接口的灵活性，你有两种选择：只使用单一的一个类，或者使用一个内部类：

```
//: c08:MultiInterfaces.java
```

⁶而在另一边，对于Unix shell而言，‘\$’是一个meta-character，所以在列出.class文件的时候，有时会有问题。这对于基于Unix的Sun公司而言，真是有点奇怪。我猜这是因为他们没有考虑这个问题，他们认为你自然是应该专注于源码文件的。

```

// Two ways that a class can implement multiple interfaces.

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
    B makeB() {
        // Anonymous inner class:
        return new B() {};
    }
}

public class MultiInterfaces {
    static void takesA(A a) {}
    static void takesB(B b) {}
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();
        takesA(x);
        takesA(y);
        takesB(x);
        takesB(y.makeB());
    }
} ///:~

```

当然，这里假设在两种方式下的代码结构都确实有逻辑意义。通常遇到问题的时候，问题本身就能给出某些指引，告诉你应该使用单独一个类，或是使用内部类。但如果没有任何其它限制，从实现的观点来看，前面的例子并没有什么区别。它们都能正常运作。

如果要继承的是抽象的类或具体的类，而不是接口，那你就只能使用内部类才能实现多重继承。

```

//: c08:MultiImplementation.java
// With concrete or abstract classes, inner
// classes are the only way to produce the effect
// of "multiple implementation inheritance."
package c08;

class D {}
abstract class E {}

class Z extends D {

```



```

        E makeE() { return new E() {};}
    }

    public class MultiImplementation {
        static void takesD(D d) {}
        static void takesE(E e) {}
        public static void main(String[] args) {
            Z z = new Z();
            takesD(z);
            takesE(z.makeE());
        }
    } ///:~

```

如果不需要解决“多重继承”的问题，你自然可以用别的方式编码，而不需要使用内部类。但如果使用内部类，你还可以获得其他一些特性：

1. 内部类可以有多个实例，每个实例都有自己的状态信息，并且与其外围类对象的信息相互独立。
2. 在单个外围类中，你可以让多个内部类以不同的方式实现同一个接口，或继承同一个类。稍后就会展示一个这样的例子。
3. 创建内部类对象的时刻并不依赖于外围类对象的创建。
4. 内部类并没有令人迷惑的“**is-a**”关系；它就是一个独立的实体。

举个例子，如果 `Sequence.java` 不使用内部类，你就必须要声明“`Sequence` 是一个 `Selector`”，对于一个特定的 `Sequence` 只能有一个 `Selector`。然而使用内部类你可以很容易地就拥有另一个方法 `getRSelector()`，用来生成一个反方向遍历序列的 `Selector`。只有内部类才有这种灵活性。

闭包与回调

闭包（closure）是一个可调用的对象，它记录了一些信息，这些信息来自于创建它的作用域。通过这个定义，可以看出内部类是面向对象的闭包，因为它不仅包含外围类对象（“创建内部类的作用域”）的信息，还自动拥有一个指向此外围类对象的引用，在此作用域内，内部类有权操作所有的成员，包括 `private` 成员。

Java 最引人争议的问题之一就是，人们认为 Java 应该包含某种类似指针的机制，以允许回调（callback）。通过回调，对象能够携带一些信息，这些信息允许它在稍后的某个时刻调用初始的对象。稍后你会看到这是一个非常有用的概念。如果回调是通过指针实现的，那么你就只能依赖于程序员不会误用该指针。然而，你应该已经了解到，Java 更小心仔细，所以没有在语言中包括指针。

通过内部类提供闭包的功能是完美的解决方案，它比指针更灵活，更安全。见下例：

```

//: c08:Callbacks.java
// Using inner classes for callbacks
import com.bruceeckel.simpletest.*;

interface Incrementable {
    void increment();
}

// Very simple to just implement the interface:
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        System.out.println(i);
    }
}

class MyIncrement {
    void increment() {
        System.out.println("Other operation");
    }
    static void f(MyIncrement mi) { mi.increment(); }
}

// If your class must implement increment() in
// some other way, you must use an inner class:
class Callee2 extends MyIncrement {
    private int i = 0;
    private void incr() {
        i++;
        System.out.println(i);
    }
    private class Closure implements Incrementable {
        public void increment() { incr(); }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
}

class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) { callbackReference = cbh; }
    void go() { callbackReference.increment(); }
}

```

```

    }

    public class Callbacks {
        private static Test monitor = new Test();
        public static void main(String[] args) {
            Callee1 c1 = new Callee1();
            Callee2 c2 = new Callee2();
            MyIncrement.f(c2);
            Caller caller1 = new Caller(c1);
            Caller caller2 = new Caller(c2.getCallbackReference());
            caller1.go();
            caller1.go();
            caller2.go();
            caller2.go();
            monitor.expect(new String[] {
                "Other operation",
                "1",
                "2",
                "1",
                "2"
            });
        }
    } ///:~

```

这个例子进一步揭示了外围类实现一个接口与内部类实现此接口之间的区别。就代码而言，Callee1 是简单的解决方式。Callee2 继承自 MyIncrement，后者已经有了一个不同的 increment() 方法，并且与 Incrementable 接口期望的 increment() 方法完全不相关。所以如果 Callee2 继承了 MyIncrement，就不能为了使用 Incrementable 而重载 increment() 方法，于是你只能使用内部类独立地实现 Incrementable。还要注意，当你创建了一个内部类时，你并没有把什么东西加入到外围类的接口中，也没有修改外围类的接口。

注意，在 Callee2 中除了 getCallbackReference() 以外，其它成员都是 private 的。要想建立与外部世界的任何连接，接口 Incrementable 都是必需的。在这里你可以看到，接口是如何与接口的实现完全独立的。

内部类 Closure 实现了 Incrementable，以提供一个返回 Callee2 的“钩子（hook）”。无论谁获得此 Incrementable 的引用，都只能调用 increment()，除此之外没有其它功能（不像指针那样，允许你做很多事情）

Caller 的构造器需要一个 Incrementable 的引用作为参数（虽然可以在任意时刻捕获回调引用），然后在以后的某个时刻，Caller 对象可以使用此引用回调 Callee 类。

回调的价值在于它的灵活性：你可以在运行期动态地决定需要调用什么方法。这样做的好处在第十四章更明显，在那里实现 GUI 功能的时候，到处都用到了回调。

内部类与控制框架

在我将要介绍的控制框架（control framework）中，可以看到更多使用内部类的具体例子。

应用程序框架（application framework）就是被设计用以解决某类特定问题的一个类或一组类。要使用某个应用程序框架，通常是继承一个或多个类，并重载某些方法。在重载的方法中，你的代码将应用程序框架提供的通用解决方案特殊化，以解决你的特定问题（这是设计模式中 Template Method 的一个例子；参考 www.BruceEckel.com 上的 Thinking in Patterns(with Java)）。控制框架是一类特殊的应用程序框架，它用来解决响应事件的需求。主要是用来响应事件的系统被称作事件驱动系统（event-driven system）。应用程序最重要的问题之一是图形用户接口（GUI），它几乎完全是事件驱动的系统。在第十四章你会看到，Java Swing 库就是一个控制框架，它优雅地解决了 GUI 的问题，并使用了大量的内部类。

要理解内部类是如何允许简单的创建过程以及如何使用控制框架的，就请你考虑这样一个控制框架的例子，它的工作就是在事件“就绪（ready）”的时候执行事件。虽然“就绪”可以指任何事，但在本例中缺省的是基于时间触发的。接下来的问题就是，对于要控制什么，控制框架并不包含任何具体的信息。那些信息是在实现“模板方法（template method）”的时候，通过继承来提供的。

首先，接口描述要控制的事件。因为其缺省的行为是基于时间去执行控制，所以使用抽象类代替实际的接口。下面的例子包含了某些实现：

```
//: c08:controller:Event.java
// The common methods for any control event.
package c08.controller;

public abstract class Event {
    private long eventTime;
    protected final long delayTime;
    public Event(long delayTime) {
        this.delayTime = delayTime;
        start();
    }
    public void start() { // Allows restarting
        eventTime = System.currentTimeMillis() + delayTime;
    }
    public boolean ready() {
        return System.currentTimeMillis() >= eventTime;
    }
}
```

```

    }
    public abstract void action();
} ///:~

```

如果你希望运行 **Event**，然后调用 **start()**，那么构造器就会捕获（从对象创建的时刻开始的）时间，此时间是这样得来的：**start()** 获取当前时间，然后加上一个延迟时间后，生成触发事件的时间。**start()** 是一个独立的方法，而没有包含在构造器内，因为这样你就可以在事件运行以后重新启动计时器，也就是能够重复使用 **Event** 对象。例如，如果你要重复一个事件，你只需简单地在 **action()** 中调用 **start()** 方法。

ready() 告诉你现在可以运行 **action()** 方法了。当然，可以在导出类中重载 **ready()**，使得 **Event** 能够基于时间以外的其它因素而触发。

下面的文件包含了一个实际的管理并触发事件的控制框架。**Event** 对象被保存在 **ArrayList** 类型的容器对象中，容器会在第十一章中详细介绍。目前你只需要知道 **add()** 方法将一个 **Object** 添加到 **ArrayList** 的尾端，**size()** 方法得到 **ArrayList** 中元素的个数，**get()** 通过索引从 **ArrayList** 中获取一个元素，**remove()** 方法从 **ArrayList** 中移除一个指定的元素。

```

///: c08:controller:Controller.java
// With Event, the generic framework for control systems.
package c08.controller;
import java.util.*;

public class Controller {
    // An object from java.util to hold Event objects:
    private List eventList = new ArrayList();
    public void addEvent(Event c) { eventList.add(c); }
    public void run() {
        while(eventList.size() > 0) {
            for(int i = 0; i < eventList.size(); i++) {
                Event e = (Event)eventList.get(i);
                if(e.ready()) {
                    System.out.println(e);
                    e.action();
                    eventList.remove(i);
                }
            }
        }
    }
} ///:~

```

run() 方法循环遍历 **eventList**，通过 **ready()** 寻找就绪的 **Event**。对找到的每一个就绪的事件，使用 **toString()** 打印其信息，调用其 **action()** 方法，然后从队列中移除此 **Event**。

注意，在目前的设计中你并不知道 **Event** 到底做了什么。这正是此设计的关键之处，“使变化的事物与不变的事物相互分离”。用我的话说，“变化向量（vector of change）”就是各种不同的 **Event** 对象所具有的不同行为，而你通过创建不同的 **Event** 子类来表现不同的行为。

这正是内部类要做的事情，你可以：

1. 用一个单独的类完整地实现一个控制框架，从而将实现的细节封装起来。内部类用来表示解决问题所必需的各种不同的 **action()**。
2. 内部类能够轻易的访问外围类的任意成员，所以可以避免这种实现变得很笨拙。如果没有这种能力，代码将变得很令人讨厌，以至于你肯定会选择别的方法。

考虑此控制框架的一个特殊实现，用以控制温室的运作⁷：控制灯光、水、温度调节器的开关，以及响铃和重新启动系统，每个行为都是完全不同的。控制框架的设计使得分离这些不同的代码变得非常容易。使用内部类，可以在单一类里面产生对同一个基类 **Event** 的多种继承版本。对于温室系统的每一种行为，都继承一个新的 **Event** 内部类，并在要实现的 **action()** 中编写控制代码。

作为典型的应用程序框架，**GreenhouseControls** 类继承自 **Controller**：

```
//: c08:GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import com.bruceeckel.simpletest.*;
import c08.controller.*;

public class GreenhouseControls extends Controller {
    private static Test monitor = new Test();
    private boolean light = false;
    public class LightOn extends Event {
        public LightOn(long delayTime) { super(delayTime); }
        public void action() {
            // Put hardware control code here to
            // physically turn on the light.
            light = true;
        }
        public String toString() { return "Light is on"; }
    }
    public class LightOff extends Event {
```

⁷基于某些原因，我一直很乐意解决这个问题；这出自我早前的书《C++ Inside & Out》，但是Java提供了更优雅的解决方案。

```

    public LightOff(long delayTime) { super(delayTime); }
    public void action() {
        // Put hardware control code here to
        // physically turn off the light.
        light = false;
    }
    public String toString() { return "Light is off"; }
}

private boolean water = false;
public class WaterOn extends Event {
    public WaterOn(long delayTime) { super(delayTime); }
    public void action() {
        // Put hardware control code here.
        water = true;
    }
    public String toString() {
        return "Greenhouse water is on";
    }
}

public class WaterOff extends Event {
    public WaterOff(long delayTime) { super(delayTime); }
    public void action() {
        // Put hardware control code here.
        water = false;
    }
    public String toString() {
        return "Greenhouse water is off";
    }
}

private String thermostat = "Day";
public class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
        super(delayTime);
    }
    public void action() {
        // Put hardware control code here.
        thermostat = "Night";
    }
    public String toString() {
        return "Thermostat on night setting";
    }
}

public class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {

```

```

        super(delayTime);
    }
    public void action() {
        // Put hardware control code here.
        thermostat = "Day";
    }
    public String toString() {
        return "Thermostat on day setting";
    }
}

// An example of an action() that inserts a
// new one of itself into the event list:
public class Bell extends Event {
    public Bell(long delayTime) { super(delayTime); }
    public void action() {
        addEvent(new Bell(delayTime));
    }
    public String toString() { return "Bing!"; }
}

public class Restart extends Event {
    private Event[] eventList;
    public Restart(long delayTime, Event[] eventList) {
        super(delayTime);
        this.eventList = eventList;
        for(int i = 0; i < eventList.length; i++)
            addEvent(eventList[i]);
    }
    public void action() {
        for(int i = 0; i < eventList.length; i++) {
            eventList[i].start(); // Rerun each event
            addEvent(eventList[i]);
        }
        start(); // Rerun this Event
        addEvent(this);
    }
    public String toString() {
        return "Restarting system";
    }
}

public class Terminate extends Event {
    public Terminate(long delayTime) { super(delayTime); }
    public void action() { System.exit(0); }
    public String toString() { return "Terminating"; }
}

```



```
} ///:~
```

注意，`light`、`water` 和 `thermostat` 都属于外围类，而这些内部类能够自由地访问那些成员变量，无需限定条件或特殊许可。而且，大多数 `action()` 方法都涉及对某种硬件的控制。

大多数 `Event` 类看起来都很相似，但是 `Bell` 和 `Restart` 则比较特别。`Bell` 控制响铃，然后在事件列表中增加一个 `Bell` 对象，于是过一会儿它可以再次响铃。你可能注意到了内部类是多么像的多重继承：`Bell` 和 `Restart` 有 `Event` 的所有方法，并且似乎也拥有外围类 `GreenhouseContrlos` 所有的方法。

一个 `Event` 对象的数组被地交给 `Restart`，该数组要加到控制器上的。由于 `Restart()` 也是一个 `Event` 对象，所以你同样可以将 `Restart` 对象添加到 `Restart.action()` 中，以使系统能够有规律的重新启动它自己。

下面的类通过创建一个 `GreenhouseControls` 对象，并添加各种不同的 `Event` 对象来配置该系统。这是命令（Command）设计模式的一个例子：

```
///: c08:GreenhouseController.java
// Configure and execute the greenhouse system.
// {Args: 5000}
import c08.controller.*;

public class GreenhouseController {
    public static void main(String[] args) {
        GreenhouseControls gc = new GreenhouseControls();
        // Instead of hard-wiring, you could parse
        // configuration information from a text file here:
        gc.addEvent(gc.new Bell(900));
        Event[] eventList = {
            gc.new ThermostatNight(0),
            gc.new LightOn(200),
            gc.new LightOff(400),
            gc.new WaterOn(600),
            gc.new WaterOff(800),
            gc.new ThermostatDay(1400)
        };
        gc.addEvent(gc.new Restart(2000, eventList));
        if(args.length == 1)
            gc.addEvent(
                gc.new Terminate(Integer.parseInt(args[0])));
        gc.run();
    }
} ///:~
```

这个类的作用是初始化系统，所以它添加了所有相应的事件。当然，更灵活的方法是避免对事件进行硬编码，取而代之的是从文件中读取需要的事件。（第十二章的练习会要求你照此方法修改这个例子。）如果你提供了命令行参数，系统会以它作为毫秒数，决定什么时候终止程序（这是测试程序时使用的）。

这个例子应该能使你更了解内部类的价值，特别是在控制框架中使用内部类的时候。而在第十四章中，你将看到内部类如何优雅地描述图形用户界面的行为。到那时，你应该就完全信服内部类的价值了。

总结

比起面向对象编程中其他的概念来，接口和内部类更深奥复杂；比如 C++ 就没有这些。将两者结合起来，能够解决 C++ 试图用多重继承解决的问题。然而，多重继承在 C++ 中被证明是相当难以使用的，相比较而言，Java 的接口和内部类就容易理解多了。

虽然这些特性本身是相当直观的，但是就像多态机制一样，这些特性的使用应该是设计阶段考虑的问题。随着时间的推移，你将能够更好地识别什么情况下应该使用接口，或是内部类，或者两者同时使用。但在此刻，你至少应该已经完全理解了它们的语法和语义。当你见到这些语言特性的确派上了用场时，你就最终理解它们了。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 证明一个接口的属性在缺省情况下是 **static** 和 **final** 的。
2. 在某个包内创建一个接口，内含三个方法，然后在另一个包中实现此接口。
3. 证明接口内所有的方法都自动是 **public** 的。
4. 在 `c07:Sandwich.java` 中，创建接口 **FastFood** 并添加合适的方法，然后修改 **Sandwich** 以实现 **FastFood** 接口。
5. 创建三个接口，使之各有两个方法。再创建一个新的接口，继承三者，并添加一个新方法。然后创建一个类，在实现此新接口的同时，继承一个实际的类。并为这个类写四个方法，每个方法分别以四个接口中的一个作为参数。在 `main()` 中，创建这个类的对象，然后将它作为参数传递给那四个方法。
6. 修改练习 5，创建一个抽象类，继承它以产生其导出类。
7. 修改 `Music5.java` 以实现 **Playable** 接口。将 `play()` 的声明从 **Instrument** 中移到 **Playable** 中。通过将 **Playable** 包括在 **implements** 列表中，把 **Playable** 添加到导出类中。修改 `tune()`，使它接受 **Playable** 作为参数，取代 **Instrument**。
8. 将第七章中练习 6 的 **Rodent** 修改成一个接口。
9. 在 `Adventure.java` 中，按照其它接口的样式，增加一个 **CanClimb** 接口。

10. 写一个程序，在其中导入并使用 **Month.java**。
11. 模仿 **Month.java** 中的例子，创建一周中每一天的一个枚举。
12. 在第一个包中创建一个至少有一个方法的接口。然后在第二个包内创建一个类，在其中增加一个 **protected** 的内部类以实现那个接口。在第三个包中，继承这个类，并在一个方法中返回该内部类的对象，在返回的时候向上转型为第一个包中的接口的类型。
13. 创建一个至少有一个方法的接口。在某个方法内定义一个内部类，以实现此接口，这个方法返回此接口的引用。
14. 重复练习 13，但将内部类定义在某个方法的一个作用域内。
15. 重复练习 13，这次使用匿名内部类。
16. 修改 **HorrorShow.java**，用匿名类实现 **DangerousMonster** 和 **Vampire**。
17. 创建一个 **private** 内部类，实现一个 **public** 接口。写一个方法，它返回一个指向此 **private** 内部类的实例的引用，并将此引用向上转型为该接口类型。通过尝试向下转型，说明此内部类被完全隐藏了。
18. 创建一个有非缺省（需要一个参数）的构造器，并且没有缺省构造器（没有不需要参数的构造器）的类。创建第二个类，它包含一个方法，能够返回第一个类的引用。通过写一个继承自第一个类的匿名内部类，而创建一个用以返回的对象。
19. 创建一个含有 **private** 属性和 **private** 方法的类。创建一个内部类，它有一个方法用来修改外围类的属性，并调用外围类的方法。在外围类的另一方法中，创建此内部类的对象，并且调用它的方法，然后说明对外围类对象的影响。
20. 使用匿名内部类重做练习 19。
21. 创建一个包含嵌套类的类。在 **main()** 中创建其内部类的实例。
22. 创建一个包含嵌套类的接口。实现此接口并创建嵌套类的实例。
23. 创建一个包含了内部类的类，而此内部类又包含有内部类。使用嵌套类重复这个过程。注意编译器生成的 **.class** 文件的名字。
24. 创建一个包含内部类的类，在另一个独立的类中，创建此内部类的实例。
25. 创建一个包含内部类的类，此内部类有一个非缺省的构造器（需要一个参数）。创建另一个也包含内部类的类，此内部类继承自第一个内部类。
26. 纠正 **WindError.java** 中的错误。
27. 修改 **Sequence.java**，增加一个 **getRSelector()** 方法，此方法是 **Selector** 接口的另一种实现，它在序列中从末尾之开头反方向移动。
28. 创建一个接口 **U**，包含三个方法。创建第一个类 **A**，它包含一个方法，在此方法中写一个匿名内部类，生成 **U** 的引用。创建第二个类 **B**，它包含 **U** 的数组。**B** 应该有几个方法，第一个方法可以接受一个 **U** 的引用并存储到数组中；第二个方法将数组中的引用设为 **null**；第三个方法遍历此数组。在 **U** 中调用这些方法，在其 **main()** 中创建一组 **A** 的对象，和单独一个 **B** 的对象。用那些 **A** 类对象所产生的 **U** 类型的引用填满 **B** 对象的数组。使用 **B** 回调所有 **A** 的对象。再从 **B** 中移除某些 **U** 的引用。
29. 在 **GreenhouseControls.java** 中增加一个 **Event** 内部类，用以打开关闭风扇。在 **GreenhouseController.java** 中使用这些新的 **Event** 对象。
30. 在 **GreenhouseControls.java** 中继承 **GreenhouseControls**，增加 **Event** 内部类，用以开启/关闭喷水机。写一个新版的 **GreenhouseController.java** 以使用这些新的 **Event** 对象。
31. 证明内部类有权访问其外围类的私有元素。并验证颠倒过来是否可行。

第九章 异常与错误处理

Java 的基本理念是“结构不佳的代码将不能运行”。

发现错误的理想时机是在编译阶段，也就是在你试图运行程序之前。然而，编译期间并不能找出所有的错误，余下的问题必须在运行期间得到解决。这就需要错误源能通过某种方式，把适当的信息传递给某个接收者，后者将知道如何正确处理这个问题。

C以及其它早期语言常常具有多种错误处理模式，这些模式往往建立在约定俗成的基础之上，而并不属于语言的一部分。通常：你会返回某个特殊值或者设置某个标志，并且假定接收者将对这个返回值或标志进行检查，以判定是否发生了错误。然而，随着时间的推移，人们发现，高傲的程序员们在使用程序库的时候更倾向于认为：“对，错误也许会发生，但那是别人造成的，不关我的事”。所以，程序员不去检查错误条件，也就不足为奇了（何况对某些错误条件的检查确实显得很无聊）¹。如果你在每次调用方法的时候都彻底地进行错误检查，代码很可能会变得难以阅读。正是由于程序员还能用这些方式拼凑系统，所以他们拒绝承认这样一个事实：对于构造大型、健壮、可维护的程序而言，这种错误处理模式已经成为了主要障碍。

解决的办法是，用强制规定的形式来消除错误处理过程中随心所欲的因素。这种作法由来已久，对“异常处理”（exception handling）的实现可以追溯到六十年代的操作系统，甚至于 BASIC 语言中的“on error goto”语句。C++的异常处理机制基于 Ada，Java 中的异常处理则建立在 C++的基础之上（尽管看上去更像 Object Pascal）。

“异常”（exception）这个词有“我对此感到意外”的意思。问题出现了，你也许不清楚该如何处理，但你的确知道不应该置之不理；你要停下来，看看是不是有别人或是在别的地方，能够处理这个问题。只是你在当前的环境（current context）中没有足够的信息来解决这个问题，所以你就把这个问题提交到一个更高级别的环境中，这里将有人作出正确的决定（有点像军队里的指挥系统）。

使用异常所带来的另一个相当明显的好处是，它能使错误处理代码变得更有条理。与原先“对于同一个错误，要在多个地方进行检查和处理”相比，你不必在方法调用处进行检查（因为异常机制将保证捕获这个错误）。并且，你只需在一个地方处理错误，既所谓的“异常处理程序”（exception handler）。这种方式不仅节省代码，而且把“描述做什么事”的代码和“出了问题怎么办”的代码相分离。总之，与以前的错误处理方法相比，异常机制使代码的阅读、编写和调试工作更加井井有条。

因为异常处理是Java中唯一正式的错误报告机制，并且通过编译器强制执行，所以不学习异常处理的话，也就只能对付前面学习过的那些例子了。本章将向你介绍如何编写正确的异常处理程序，以及当你的方法出问题的时候，如何产生自定义的异常。

¹比如，C程序员不妨去检查一下printf()的返回值。

基本异常

“异常情形”（**exceptional condition**）是指引发阻止当前方法或作用域继续执行的问题。把异常情形与普通问题相区分很重要，这里的普通问题是指，你在当前环境下能得到足够的信息，总能处理这个错误。而对于异常情形，你就不能继续下去了，因为你在当前环境下无法获得必要的信息来解决问题。你能做的就是从当前的环境中跳出，并且把问题提交给上一级别的环境。这就是抛出异常时所发生的事情。

除法就是个简单的例子。除数有可能为 0，所以先进行检查很有必要。但除数为 0 代表的究竟是什么意思呢？你通过当前正在解决的问题的环境，或许能知道该如何处理除数为 0 的情况。但如果这是一个意料之外的值，你也不清楚该如何处理，那就要抛出异常，而不是顺着原来的路径继续执行下去。

当你抛出异常后，有几件事会随之发生。首先，同 Java 中其它对象的创建一样，将使用 **new** 在堆上创建异常对象。然后，当前的执行路径（你不能继续下去了）被终止，并且从当前环境中弹出异常对象的引用。此时，异常处理机制接管程序，并开始寻找一个恰当的地方来继续执行程序。这个恰当的地方就是“异常处理程序”（**exception handler**），它的任务是将程序从错误状态中恢复：以使程序能要么换一种方式运行，要么继续运行下去。

举一个抛出异常的简单例子。对于对象引用 **t**，传给你的时候可能尚未被初始化。所以在使用这个引用调用其方法之前，你会先对引用进行检查。你可以创建一个代表错误信息的对象，并且将它从当前环境中“抛出”，这样就把错误信息传播到了“更大”的环境中。这被称为“抛出一个异常”（**throwing an exception**），看起来像这样：

```
if(t == null)
    throw new NullPointerException();
```

这就抛出了异常，于是你在当前的环境下就不必为这个问题而操心了，它将在别的地方得到处理。具体是哪个“地方”后面很快就会介绍。

异常形式参数

与 Java 中的其它对象一样，你总是用 **new** 在堆上创建异常对象，这也伴随着存储空间的分配和构造器的调用。所有标准异常类都有两个构造器：一个是缺省构造器；另一个是接受字符串作为参数，用来把相关信息放入异常对象的构造器：

```
throw new NullPointerException("t = null");
```

你将看到，有多种不同的方法可以把这个字符串的内容提取出来。

关键字 **throw** 将触发许多十分奇妙的事情。通常，你首先使用 **new** 来创建对象，用以表示错误情况，此对象的引用将传给 **throw**。尽管返回的异常对象其类型通常与方法设

计的返回类型不同，但从效果上看，它就像是方法“返回”的。可以简单地把异常处理看成是一种能返回不同类型的机制，当然你过分强调这种类比的话，就会有麻烦了。你也能用抛出异常的方式从当前的作用域退出。一旦返回了一个值，就会退出方法或作用域。

抛出异常与方法正常返回值的相似之处到此为止。因为异常返回的“地点”与普通方法调用返回的“地点”完全不同。（异常将在一个恰当的异常处理程序中得到解决，它的位置可能离异常被抛出的地方很远，也可能会跨越方法调用栈的许多层次。）

此外，你能抛出任意类型的`Throwable`（它是异常类型的根类）对象。通常，对于不同类型的错误，你要抛出相应的异常。错误信息可以保存在异常对象内部或者用异常类型的名称来暗示。上一层的环境通过这些信息得以决定如何处理你的异常。（通常，异常类型的名称就是唯一的信息，而异常对象本身则不包含任何有意义的内容。）

捕获异常

如果方法要抛出异常，它必须假定异常将被“捕获”并得到处理。异常处理的好处之一就是，使你得以先在一个地方专注于正在解决的问题，然后在别的地方处理这些代码中可能发生的错误。

要明白异常是如何被捕获的，你必须首先理解监控区域（`guarded region`）的概念。它是一段可能产生异常的代码，并且后面跟着针对这些异常的处理程序。

Try 块

如果你在方法内部抛出了异常（或者在方法内部调用的其它方法抛出了异常），这个方法将在抛出异常的过程中结束。要是你不希望方法就此结束，你可以在方法内设置一个特殊的块来捕获异常。因为你在这个块里“尝试”调用了一些（可能产生异常的）方法，所以称为 `try` 区块。它是跟在 `try` 关键字之后的普通程序块：

```
try {  
    // Code that might generate exceptions  
}
```

对于不支持异常处理的程序语言，要想仔细检查错误，你就得在每个方法调用的前后加上设置和错误检查的代码，甚至你每次调用同一方法时也得这么做。有了异常处理机制，你可以把所有动作都放在 `try` 区块里，然后只需在一个地方就可以捕获所有异常。这意味着代码将更容易被编写和阅读，因为完成任务的代码没有与错误检查的代码混在一起。

异常处理程序（Exception handler）

当然，抛出的异常必须在某处得到处理。这个“地点”就是“异常处理程序”（exception handler），针对每个要捕获的异常，你得准备相应的处理程序。异常处理程序紧跟在 try 区块之后，以关键字 catch 表示：

```
try {  
    // Code that might generate exceptions  
} catch (Type1 id1) {  
    // Handle exceptions of Type1  
} catch (Type2 id2) {  
    // Handle exceptions of Type2  
} catch (Type3 id3) {  
    // Handle exceptions of Type3  
}  
  
// etc...
```

每个 catch 子句（异常处理程序）看起来就像是仅仅接受一个特定参数的方法。可以在处理程序的内部使用标识符（id1, id2 等等），这与方法参数的使用很相似。有时你可能用不到标识符，因为异常的类型已经给了你足够的信息来对异常进行处理，但标识符并不可以省略。

异常处理程序必须紧跟在 try 块之后。当异常被抛出时，异常处理机制将负责搜寻参数与异常类型相匹配的第一个处理程序。然后进入 catch 子句执行，此时认为异常得到了处理。一旦 catch 子句结束，则处理程序的查找过程结束。注意，只有匹配的 catch 子句才能得到执行；这与 switch 语句不同，switch 语句需要你在每一个 case 后面跟一个 break，以避免执行后续的 case 子句。

注意在 try 块的内部，不同的方法调用可能会产生类型相同的异常，你只需要提供一个针对此类型的异常处理程序。

终止与恢复（Termination vs. Resumption）

异常处理理论上有两种基本模型。一种称为“终止模型”（它是 Java 和 C++ 所支持的模型）。在这种模型中，将假设错误非常关键，以至于程序无法返回到异常发生的地方继续执行。一旦异常被抛出，就表明错误已无法挽回，也不能回来继续执行。

另一种称为“恢复模型”。意思是异常处理程序的工作是修正错误，然后重新尝试调用出问题的方法，并认为第二次能成功。对于恢复模型，你希望异常被处理之后，能继续执行程序。在这种情况下，抛出异常更像是对方法的调用----你可以在 Java 里用这种方法进行配置，以得到类似“恢复”的行为。（换句话说，不是抛出异常，而是调用方法

来修正错误。)或者,把 `try` 块放在 `while` 循环里,这样就不断地进入 `try` 块,直到得到满意的结果。

长久以来,尽管程序员们使用的操作系统支持恢复模型的异常处理,但他们最终还是转向使用类似终结模型的代码,并且忽略恢复行为。所以虽然恢复模型开始显得很吸引人,但不是很实用。其中的主要原因可能是它所导致的“耦合”:你的处理程序必须关注异常抛出的地点,这势必要包含依赖于抛出位置的非一般性代码。这增加了代码编写和维护的困难,对于异常可能会从许多地方抛出的大型程序来说,更是如此。

创建自定义异常

你不必拘泥于 Java 中已有的异常类型。JDK 提供的异常体系不能预见你想报告的所有错误,所以你可以自己定义异常类来表示程序中可能遇到的特定问题。

要自己定义异常类,你必须从已有的异常类继承,最好是选择意思相近的异常类继承(不过这样的异常并不容易找)。建立新的异常类型最简单的方法就是让编译器为你产生缺省构造器,所以这几乎不用写多少代码:

```
//: c09:SimpleExceptionDemo.java
// Inheriting your own exceptions.
import com.bruceeckel.simpletest.*;

class SimpleException extends Exception {}

public class SimpleExceptionDemo {
    private static Test monitor = new Test();
    public void f() throws SimpleException {
        System.out.println("Throw SimpleException from f()");
        throw new SimpleException();
    }
    public static void main(String[] args) {
        SimpleExceptionDemo sed = new SimpleExceptionDemo();
        try {
            sed.f();
        } catch(SimpleException e) {
            System.err.println("Caught it!");
        }
        monitor.expect(new String[] {
            "Throw SimpleException from f()",
            "Caught it!"
        });
    }
} ///:~
```


编译器创建了缺省构造器，它将自动调用基类的缺省构造器。本例中你不会得到像 **SimpleException(String)** 这样的构造器，这种构造器也不实用。你将看到，对异常来说，最重要的部分就是类型的名称，所以本例中建立的异常类在大多数情况下已经够用了。

本例的结果通过 `System.err` 打印到控制台的标准错误流。通常这比把错误信息输出到 `System.out` 要好，因为 `System.out` 也许会被重定向。但是把结果送到 `System.err`，它就不会随 `System.out` 一起被重定向，这样更容易被用户注意。

你也可以为异常类定义一个接受字符串作为参数的构造器：

```
//: c09:FullConstructors.java
import com.bruceeckel.simpletest.*;

class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) { super(msg); }
}

public class FullConstructors {
    private static Test monitor = new Test();
    public static void f() throws MyException {
        System.out.println("Throwing MyException from f()");
        throw new MyException();
    }
    public static void g() throws MyException {
        System.out.println("Throwing MyException from g()");
        throw new MyException("Originated in g()");
    }
    public static void main(String[] args) {
        try {
            f();
        } catch (MyException e) {
            e.printStackTrace();
        }
        try {
            g();
        } catch (MyException e) {
            e.printStackTrace();
        }
        monitor.expect(new String[] {
            "Throwing MyException from f()",
            "MyException",
            "%% \tat FullConstructors.f\\(.*\\)",
        });
    }
}
```

```

        "%% \tat FullConstructors.main\\(.*\)",
        "Throwing MyException from g()",
        "MyException: Originated in g()",
        "%% \tat FullConstructors.g\\(.*\)",
        "%% \tat FullConstructors.main\\(.*)"
    });
}
} ///:~

```

新增的代码不长：两个构造器定义了 `MyException` 类型对象的创建方式。对于第二个构造器，使用 `super` 关键字明确调用了其基类构造器，它接受一个字符串作为参数。

在异常处理程序中，调用了在 `Throwable` 类声明（`Exception` 即从此类继承）的 `printStackTrace()` 方法。它将打印“从方法调用处直到异常抛出处”的方法调用序列。在缺省情况下，信息将被输出到标准错误流，但你也可以使用重载的版本，把信息输出到任意的流中。

还可以更进一步定义你的异常，比如加入额外的构造器和成员：

```

///: c09:ExtraFeatures.java
/// Further embellishment of exception classes.
import com.bruceeckel.simpletest.*;

class MyException2 extends Exception {
    private int x;
    public MyException2() {}
    public MyException2(String msg) { super(msg); }
    public MyException2(String msg, int x) {
        super(msg);
        this.x = x;
    }
    public int val() { return x; }
    public String getMessage() {
        return "Detail Message: " + x + " " + super.getMessage();
    }
}

public class ExtraFeatures {
    private static Test monitor = new Test();
    public static void f() throws MyException2 {
        System.out.println("Throwing MyException2 from f()");
        throw new MyException2();
    }
    public static void g() throws MyException2 {

```

```

        System.out.println("Throwing MyException2 from g()");
        throw new MyException2("Originated in g()");
    }
    public static void h() throws MyException2 {
        System.out.println("Throwing MyException2 from h()");
        throw new MyException2("Originated in h()", 47);
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException2 e) {
            e.printStackTrace();
        }
        try {
            g();
        } catch(MyException2 e) {
            e.printStackTrace();
        }
        try {
            h();
        } catch(MyException2 e) {
            e.printStackTrace();
            System.err.println("e.val() = " + e.val());
        }
        monitor.expect(new String[] {
            "Throwing MyException2 from f()",
            "MyException2: Detail Message: 0 null",
            "% \tat ExtraFeatures.f\\(.*)",
            "% \tat ExtraFeatures.main\\(.*)",
            "Throwing MyException2 from g()",
            "MyException2: Detail Message: 0 Originated in g()",
            "% \tat ExtraFeatures.g\\(.*)",
            "% \tat ExtraFeatures.main\\(.*)",
            "Throwing MyException2 from h()",
            "MyException2: Detail Message: 47 Originated in h()",
            "% \tat ExtraFeatures.h\\(.*)",
            "% \tat ExtraFeatures.main\\(.*)",
            "e.val() = 47"
        });
    }
} ///:~

```

新的异常添加了一个字段 `i`、设定 `i` 值的构造器和读取数据的方法。此外，还重载了 `Throwable.getMessage()` 方法，以产生更详细的信息。对于异常类来说，`getMessage()` 方法有点类似于 `toString()` 方法。

既然异常也是对象的一种，所以你可以继续修改这个异常类，以得到更强的功能。但要记住，使用程序包的客户端程序员可能仅仅只是查看一下抛出的异常类型，其它的就不管了（大多数 Java 库里的异常都是这么用的），所以你对异常添加的其它功能也许根本用不上。

异常说明

Java 鼓励你把方法可能会抛出的异常类型，告知使用此方法的客户端程序员。这是种优雅的做法，它使得调用者能确切知道写什么样的代码可以捕获所有潜在的异常。当然，如果提供了源代码，客户端程序员可以在源代码中查找 `throw` 语句来获知相关信息，然而程序库通常并不与源代码一起发布。为了预防这样的问题，Java 提供了相应的语法（并强制你使用这个语法），使你能以礼貌的方式告知客户端程序员某个方法可能会抛出的异常类型，然后客户端程序员就可以进行相应的处理。这就是“异常说明”（`exception specification`），它属于方法声明的一部分，紧跟在形式参数列表之后。

异常说明使用了附加的关键字 `throws`，后面接一个所有潜在异常类型的列表，所以方法定义可能看起来像这样：

```
void f() throws TooBig, TooSmall, DivZero { //...
```

要是你这么写：

```
void f() { // ...
```

就表示此方法不会抛出任何异常（除了从 `RuntimeException` 继承的异常，它们可以在没有异常说明的情况下被抛出，我们将在后面进行讨论）。

你的代码必须与异常说明保持一致。如果方法里的代码产生了异常却没有进行处理，编译器会发现这个问题并提醒你：要么处理这个异常，要么就在异常说明中表明此方法将产生异常。通过这种自顶向下强制执行的异常说明机制，Java 在编译期就可以保证相当程度的异常一致性。

不过还是有个能“作弊”的地方：你可以声明方法将抛出异常，实际上却不抛出。编译器相信了你的声明，并强制此方法的用户像真的抛出异常那样使用这个方法。这样做的好处是，为异常先占了个位子，以后就可以抛出这种异常而不用修改已有的代码。在定义抽象基类和接口时这种能力很重要，这样派生类或接口实现就能够抛出这些预先声明的异常。

这种在编译期被强制检查的异常称为“被检查的异常”（`checked exception`）。

捕获所有异常

你可以只写一个异常处理程序来捕获所有类型的异常。通过捕获异常类型的基类 `Exception`，就可以做到这一点（事实上还有其它的基类，但 `Exception` 是同编程活动相关的基类。）：

```
catch(Exception e) {  
    System.err.println("Caught an exception");  
}
```

这将捕获所有异常，所以你最好把它放在处理程序列表的末尾，以防止它抢在其它处理程序之前先把异常捕获了。

因为 `Exception` 是与编程有关的所有异常类的基类，所以它不会含有太多特定的信息，不过你可以调用它从 `Throwable` 继承的方法：

`String getMessage()`

`String getLocalizedMessage()`

用来获取详细信息，或用本地语言表示的详细信息。

`String toString()`

返回对 `Throwable` 的简单描述，要是有详细信息的话，也会把它包含在内。

`void printStackTrace()`

`void printStackTrace(PrintStream)`

`void printStackTrace(java.io.PrintWriter)`

打印 `Throwable` 和 `Throwable` 的调用栈轨迹（call stack trace）。调用栈显示了“把你带到异常抛出地点”的方法调用序列。此方法第一个版本输出到标准输出流，对后两个版本你可以选择要输出的流（在第 12 章，你将学习这两种流的不同之处）。

`Throwable fillInStackTrace()`

用于在 `Throwable` 对象的内部记录栈框架（stack frame）的当前状态。这在程序重新抛出错误或异常（很快就会讲到）时很有用。

此外，你也可以使用 `Throwable` 从其基类 `Object`（也是所有类的基类）继承的方法。对于异常来说，`getClass()` 也许是个很好用的方法，它将返回一个表示此对象类型的对象。然后你可以使用 `getName()` 方法查询这个 `Class` 对象的名称。你还可以用这个 `Class` 对象做更多复杂的操作，不过对于异常处理而言，这已经足够了。

下面的例子演示了如何使用 `Exception` 类型的方法：

```
//: c09:ExceptionMethods.java
```

```

// Demonstrating the Exception Methods.
import com.bruceeckel.simpletest.*;

public class ExceptionMethods {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        try {
            throw new Exception("My Exception");
        } catch (Exception e) {
            System.err.println("Caught Exception");
            System.err.println("getMessage(): " + e.getMessage());
            System.err.println("getLocalizedMessage(): " +
                e.getLocalizedMessage());
            System.err.println("toString(): " + e);
            System.err.println("printStackTrace():");
            e.printStackTrace();
        }
        monitor.expect(new String[] {
            "Caught Exception",
            "getMessage(): My Exception",
            "getLocalizedMessage(): My Exception",
            "toString(): java.lang.Exception: My Exception",
            "printStackTrace():",
            "java.lang.Exception: My Exception",
            "% \tat ExceptionMethods.main\\(.*\\"
        });
    }
} ///:~

```

你会发现每个方法都比前一个提供了更多的信息----实际上它们每一个都是前一个的超集。

重新抛出异常

有时你希望把刚捕获的异常重新抛出，尤其是在使用 `Exception` 捕获所有异常的时候。既然你已经得到了当前异常对象的引用，你可以直接把它重新抛出：

```

catch (Exception e) {
    System.err.println("An exception was thrown");
    throw e;
}

```

重抛异常会把异常抛给上一级环境中的异常处理程序。同一个 `try` 块的后续 `catch` 子句将被忽略。此外，异常对象的所有信息都得以保持，所以高级环境中捕获此异常的处理程序可以从这个异常对象中得到所有信息。

如果你只是把当前异常对象重新抛出，那么 `printStackTrace()` 方法显示的将是原来异常抛出点的调用栈信息，而并非重新抛出点的信息。要想更新这个信息，你可以调用 `fillInStackTrace()` 方法，这将返回一个 `Throwable` 对象，它是通过把当前调用栈信息填入原来那个异常对象而建立的。就像这样：

```
//: c09:Rethrowing.java
// Demonstrating fillInStackTrace()
import com.bruceeckel.simpletest.*;

public class Rethrowing {
    private static Test monitor = new Test();
    public static void f() throws Exception {
        System.out.println("originating the exception in f()");
        throw new Exception("thrown from f()");
    }
    public static void g() throws Throwable {
        try {
            f();
        } catch (Exception e) {
            System.err.println("Inside g(), e.printStackTrace()");
            e.printStackTrace();
            throw e; // 17
            // throw e.fillInStackTrace(); // 18
        }
    }
    public static void
    main(String[] args) throws Throwable {
        try {
            g();
        } catch (Exception e) {
            System.err.println(
                "Caught in main, e.printStackTrace()");
            e.printStackTrace();
        }
        monitor.expect(new String[] {
            "originating the exception in f()",
            "Inside g(), e.printStackTrace()",
            "java.lang.Exception: thrown from f()",
            "%% \tat Rethrowing.f(.*)",
            "%% \tat Rethrowing.g(.*)",
        });
    }
}
```

```

        "%% \tat Rethrowing.main(.*)",
        "Caught in main, e.printStackTrace()",
        "java.lang.Exception: thrown from f()",
        "%% \tat Rethrowing.f(.*)",
        "%% \tat Rethrowing.g(.*)",
        "%% \tat Rethrowing.main(.*)"
    });
}
} ///:~

```

重要的几行用数字注释出来了。如果第 17 行没有注释掉（所示情况），那么无论异常对象被重新抛出多少次，其调用栈信息始终是原始抛出地点的信息。

如果把第 17 行注释掉，第 18 行的注释解除，并使用 `fillInStackTrace()`，那么运行结果就会是：

```

originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:9)
    at Rethrowing.g(Rethrowing.java:12)
    at Rethrowing.main(Rethrowing.java:23)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.g(Rethrowing.java:18)
    at Rethrowing.main(Rethrowing.java:23)

```

（此外还有 `Test.expect()` 输出的出错信息。）因为有了 `fillInStackTrace()`，第 18 行就成了异常的新发生地了。

因为 `fillInStackTrace()` 返回的是对 `Throwable` 对象的引用，所以 `g()` 和 `main()` 的异常说明中必须要有 `Throwable` 类的名称。既然 `Throwable` 是 `Exception` 的基类，你就有可能抛出一个 `Throwable` 而非 `Exception` 的对象，所以 `main()` 中的捕获 `Exception` 的处理程序可能捕获不到这个对象。为了确保一切都能正常运行，编译器将强制在异常说明里使用 `Throwable`。例如，下面程序中的异常将不能在 `main()` 里被捕获：

```

///: c09:ThrowOut.java
/// {ThrowsException}
public class ThrowOut {
    public static void
    main(String[] args) throws Throwable {
        try {
            throw new Throwable();

```



```

    } catch(Exception e) {
        System.err.println("Caught in main()");
    }
}
} ///:~

```

你有可能在捕获异常之后抛出另一种异常。这么做的话，将得到类似使用 `fillInStackTrace()` 的效果，有关原来异常发生地点的信息会丢失，剩下的是与新的抛出地点有关的信息：

```

///: c09:RethrowNew.java
// Rethrow a different object from the one that was caught.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class OneException extends Exception {
    public OneException(String s) { super(s); }
}

class TwoException extends Exception {
    public TwoException(String s) { super(s); }
}

public class RethrowNew {
    private static Test monitor = new Test();
    public static void f() throws OneException {
        System.out.println("originating the exception in f()");
        throw new OneException("thrown from f()");
    }

    public static void
    main(String[] args) throws TwoException {
        try {
            f();
        } catch(OneException e) {
            System.err.println(
                "Caught in main, e.printStackTrace()");
            e.printStackTrace();
            throw new TwoException("from main()");
        }
        monitor.expect(new String[] {
            "originating the exception in f()",
            "Caught in main, e.printStackTrace()",
            "OneException: thrown from f()",
            "\tat RethrowNew.f(RethrowNew.java:18)",

```

```

        "\tat RethrowNew.main(RethrowNew.java:22)",
        "Exception in thread \""main\" " +
        "TwoException: from main()",
        "\tat RethrowNew.main(RethrowNew.java:28)"
    });
}
} ///:~

```

最后那个异常仅知道自己来自 `main()`，而对 `f()` 一无所知。

你永远不用为清理前一个异常对象而担心，或者说为异常对象的清理担心。它们都是用 `new` 在堆上创建的对象，所以垃圾回收器会自动把它们清理掉。

异常链

你常常会想要在捕获一个异常然后抛出另一个异常，并且希望把原始异常的信息保存下来，这被称为“异常链”（Exception chaining）。在 JDK 1.4 以前，程序员必须自己编写代码来保持原始异常的信息。现在所有 `Throwable` 的子类在构造器中都可以接受一个 `cause` 对象作为参数。这个 `cause` 就用来表示原始异常，这样通过把原始异常传递给新的异常，使得即使你在当前位置创建并抛出了新的异常，你也能通过这个异常链追踪到异常最初发生的位置。

有趣的是，在 `Throwable` 的子类中，只有三种基本的异常类提供了带 `cause` 的构造器。它们是 `Error`（用于 Java 虚拟机报告系统错误）、`Exception`，以及 `RuntimeException`。如果你要把其它类型的异常链接起来，你应该使用 `initCause()` 方法而不是构造器。

下面的例子能让你在运行时动态地向 `DynamicFields` 对象添加字段：

```

///: c09:DynamicFields.java
// A Class that dynamically adds fields to itself.
// Demonstrates exception chaining.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class DynamicFieldsException extends Exception {}

public class DynamicFields {
    private static Test monitor = new Test();
    private Object[][] fields;
    public DynamicFields(int initialSize) {
        fields = new Object[initialSize][2];
        for(int i = 0; i < initialSize; i++)
            fields[i] = new Object[] { null, null };
    }
}

```

```

public String toString() {
    StringBuffer result = new StringBuffer();
    for(int i = 0; i < fields.length; i++) {
        result.append(fields[i][0]);
        result.append(": ");
        result.append(fields[i][1]);
        result.append("\n");
    }
    return result.toString();
}

private int hasField(String id) {
    for(int i = 0; i < fields.length; i++)
        if(id.equals(fields[i][0]))
            return i;
    return -1;
}

private int
getFieldNumber(String id) throws NoSuchFieldException {
    int fieldNum = hasField(id);
    if(fieldNum == -1)
        throw new NoSuchFieldException();
    return fieldNum;
}

private int makeField(String id) {
    for(int i = 0; i < fields.length; i++)
        if(fields[i][0] == null) {
            fields[i][0] = id;
            return i;
        }
    // No empty fields. Add one:
    Object[][] tmp = new Object[fields.length + 1][2];
    for(int i = 0; i < fields.length; i++)
        tmp[i] = fields[i];
    for(int i = fields.length; i < tmp.length; i++)
        tmp[i] = new Object[] { null, null };
    fields = tmp;
    // Reursive call with expanded fields:
    return makeField(id);
}

public Object
getField(String id) throws NoSuchFieldException {
    return fields[getFieldNumber(id)][1];
}

public Object setField(String id, Object value)

```

```

throws DynamicFieldsException {
    if(value == null) {
        // Most exceptions don't have a "cause" constructor.
        // In these cases you must use initCause(),
        // available in all Throwable subclasses.
        DynamicFieldsException dfe =
            new DynamicFieldsException();
        dfe.initCause(new NullPointerException());
        throw dfe;
    }
    int fieldNumber = hasField(id);
    if(fieldNumber == -1)
        fieldNumber = makeField(id);
    Object result = null;
    try {
        result = getField(id); // Get old value
    } catch(NoSuchFieldException e) {
        // Use constructor that takes "cause":
        throw new RuntimeException(e);
    }
    fields[fieldNumber][1] = value;
    return result;
}

public static void main(String[] args) {
    DynamicFields df = new DynamicFields(3);
    System.out.println(df);
    try {
        df.setField("d", "A value for d");
        df.setField("number", new Integer(47));
        df.setField("number2", new Integer(48));
        System.out.println(df);
        df.setField("d", "A new value for d");
        df.setField("number3", new Integer(11));
        System.out.println(df);
        System.out.println(df.getField("d"));
        Object field = df.getField("a3"); // Exception
    } catch(NoSuchFieldException e) {
        throw new RuntimeException(e);
    } catch(DynamicFieldsException e) {
        throw new RuntimeException(e);
    }
}

monitor.expect(new String[] {
    "null: null",
    "null: null",

```

```

        "null: null",
        "",
        "d: A value for d",
        "number: 47",
        "number2: 48",
        "",
        "d: A new value for d",
        "number: 47",
        "number2: 48",
        "number3: 11",
        "",
        "A value for d",
        "Exception in thread \"main\" " +
        "java.lang.RuntimeException: " +
        "java.lang.NoSuchFieldException",
        "\tat DynamicFields.main(DynamicFields.java:98)",
        "Caused by: java.lang.NoSuchFieldException",
        "\tat DynamicFields.getFieldNumber(" +
        "DynamicFields.java:37)",
        "\tat DynamicFields.getField(DynamicFields.java:58)",
        "\tat DynamicFields.main(DynamicFields.java:96)"
    });
}
} ///:~

```

每个 `DynamicFields` 对象都含有一个数组，其元素是“成对的对象”。第一个对象表示字段标识（一个字符串），第二个表示字段值，值的类型可以是除基本类型外的任意类型。当创建对象的时候，你要合理估计一下需要多少字段。当调用 `setField()` 方法的时候，它将试图通过标识修改已有字段值，否则就建一个新的字段，并把值放入。如果空间不够了，将建立一个更长的数组，并把原来数组的元素复制进去。如果你试图为字段设置一个空值，将抛出一个 `DynamicFieldsException` 异常，它是通过使用 `initCause()` 方法把 `NullPointerException` 对象插入而建立的。

至于返回值，`setField()` 将用 `getField()` 方法把此位置的旧值取出，这个操作可能会抛出 `NoSuchFieldException` 异常。如果客户端程序员调用了 `getField()` 方法，那么他就有责任处理这个可能抛出的 `NoSuchFieldException` 异常，但如果异常是从 `setField()` 方法里抛出的，这种情况将被视为编程错误，所以就使用接受 `cause` 参数的构造器把 `NoSuchFieldException` 异常转换为 `RuntimeException` 异常。

Java 标准异常

`Throwable` 这个 Java 类被用来表示任何可以作为异常被抛出的类。`Throwable` 对象可分为两种类型（指从 `Throwable` 继承而得到的类型）：`Error` 用来表示你不用关心的编译期和系统错误（除了特殊情况）；`Exception` 是可以被抛出的基本类型，在 Java

类库、用户方法以及运行时故障中都可能抛出 `Exception` 型异常。所以 Java 程序员关心的主要是 `Exception`。

要想对异常有全面的了解，最好去浏览一下 HTML 格式的 Java 文档，你可以从 java.sun.com 下载。为了对不同的异常有个感性的认识，这么做是值得的。但很快你就会发现这些异常除了名称外其实都差不多。同时，Java 中异常的数目在持续增加，所以在书中简单罗列它们毫无意义。你使用的第三方类库也可能会有自己的异常。对异常来说，关键是理解概念以及如何使用。

异常的基本的概念是用名称代表发生的问题，并且异常的名称应该可以望文知意。异常并非全是在 `java.lang` 包里定义的；有些异常是用来支持其它像 `util`、`net` 和 `io` 这样的程序包，这些异常可以通过它们的完整名称或者从它们的父类中看出端倪。比如，所有的输入/输出异常都是从 `java.io.IOException` 继承而来的。

运行期异常（`RuntimeException`）的特例

在本章的第一个例子中：

```
if(t == null)
    throw new NullPointerException();
```

如果必须对传递给方法的每个引用都检查其是否为 `null`（因为你无法确定调用者是否传入了非法引用），这听起来着实吓人。幸运的是，这不必由你亲自来做，它属于 Java 的标准运行期检测的一部分。如果在 `null` 引用上调用方法，Java 会自动抛出 `NullPointerException` 异常。所以上述代码是多余的。

属于运行期异常的类型有很多。它们会自动被 Java 虚拟机抛出，所以你不必在异常说明中把它们列出来。这些异常都是从 `RuntimeException` 类继承而来，所以既体现了继承的优点，使用起来也很方便。这构成了一组具有相同特征和行为的异常类型。并且，你也不再需要在异常说明中声明方法将抛出 `RuntimeException` 类型的异常（或者任何从 `RuntimeException` 继承的异常），它们也被称为“未被检查的异常”（`unchecked exception`）。这种异常属于错误，将被自动捕获，就不用你亲自动手了。要是你自己去检查 `RuntimeException` 的话，代码就显得太混乱了。不过尽管你通常不用捕获 `RuntimeException` 异常，但还是可以在代码中抛出 `RuntimeException` 类型的异常。

如果你不捕获这种类型的异常会发生什么事呢？因为编译器没有在这个问题上对异常说明进行强制检查，`RuntimeException` 类型的异常也许会穿越所有的执行路径直达 `main()` 方法，而不会被捕获。要明白到底发生了什么，可以试试下面的例子：

```
//: c09:NeverCaught.java
// Ignoring RuntimeExceptions.
// {ThrowsException}
import com.bruceeckel.simpletest.*;
```

```

public class NeverCaught {
    private static Test monitor = new Test();
    static void f() {
        throw new RuntimeException("From f()");
    }
    static void g() {
        f();
    }
    public static void main(String[] args) {
        g();
        monitor.expect(new String[] {
            "Exception in thread \"main\" " +
            "java.lang.RuntimeException: From f()",
            "    at NeverCaught.f(NeverCaught.java:7)",
            "    at NeverCaught.g(NeverCaught.java:10)",
            "    at NeverCaught.main(NeverCaught.java:13)"
        });
    }
} ///:~

```

你能发现 `RuntimeException`（或任何从它继承的异常）是一个特例。对于这种异常类型，编译器不需要异常说明。

所以答案是：如果 `RuntimeException` 没有被捕获而直达 `main()`，那么在程序退出前将调用异常的 `printStackTrace()` 方法。

请务必记住：你只能在代码中忽略 `RuntimeException`（及其子类）类型的异常，其它类型异常的处理都是由编译器强制实施的。究其原因，`RuntimeException` 代表的是编程错误：

1. 你无法预料的错误。比如从你控制范围之外传递进来的 `null` 引用。
2. 作为程序员，你应该在代码中进行检查的错误。（比如对于 `ArrayIndexOutOfBoundsException`，你就得注意一下数组的大小了。）在一个地方发生的异常，常常会在另一个地方导致错误。

你会发现在这些情况下使用异常很有好处，它们能给调试带来便利。

值得注意的是：你不应该把Java的异常处理机制当成是单一用途的工具。它确实是被设计用来处理一些烦人的运行期错误，这些错误往往是由你的代码控制能力之外的因素导致的，然而，它对于发现某些编译器无法检测到的编程错误，也是非常重要的。

使用 finally 进行清理

对于一些代码，你可能会希望无论try块中的异常是否抛出，它们都能得到执行。这通常适用于内存回收（由垃圾回收器完成）之外的情况。为了达到这个效果，你可以在异常处理程序后面加上finally子句²。完整的异常处理程序看起来像这样：

```
try {
    // The guarded region: Dangerous activities
    // that might throw A, B, or C
} catch(A a1) {
    // Handler for situation A
} catch(B b1) {
    // Handler for situation B
} catch(C c1) {
    // Handler for situation C
} finally {
    // Activities that happen every time
}
```

为了证明 finally 总能运行，可以试一下这个程序：

```
//: c09:FinallyWorks.java
// The finally clause is always executed.
import com.bruceeckel.simpletest.*;

class ThreeException extends Exception {}

public class FinallyWorks {
    private static Test monitor = new Test();
    static int count = 0;
    public static void main(String[] args) {
        while(true) {
            try {
                // Post-increment is zero first time:
                if(count++ == 0)
                    throw new ThreeException();
                System.out.println("No exception");
            } catch(ThreeException e) {
                System.err.println("ThreeException");
            } finally {
                System.err.println("In finally clause");
                if(count == 2) break; // out of "while"
            }
        }
    }
}
```

² C++异常处理没有finally子句，它依赖析构函数来达到清理的目的。


```

    }
}
monitor.expect(new String[] {
    "ThreeException",
    "In finally clause",
    "No exception",
    "In finally clause"
});
}
} ///:~

```

你可以从输出中发现，无论异常是否被抛出，**finally** 子句总能被执行。

这个程序也给了你一些思路，当 Java 中的异常（与 C++ 类似）不允许你回到异常抛出的地点时，你该如何应对呢？如果你把 **try** 块放在循环里，你就建立了一个“程序继续执行之前必须要达到”的条件。你还可以加入一个静态类型的计数器或者别的装置使循环在放弃以前能尝试一定的次数。这将使程序的健壮性更上一个台阶。

finally 用来做什么？

对于没有垃圾回收和析构函数自动调用机制³的语言来说，**finally** 非常重要。它能使程序员保证：无论 **try** 块里发生了什么，内存总能得到释放。但 Java 有垃圾回收机制，所以内存释放不再是问题。况且，Java 也没有析构函数可供调用。那么，Java 在什么情况下才能用到 **finally** 呢？

当你要把除内存之外的资源恢复到它们的初始状态时，就要用到 **finally** 子句。这种需要清理的资源包括：已经打开的文件或网络连接，你在屏幕上画的图形，甚至可以是外部世界的某个开关，如下面例子所示：

```

///: c09:Switch.java
public class Switch {
    private boolean state = false;
    public boolean read() { return state; }
    public void on() { state = true; }
    public void off() { state = false; }
} ///:~

///: c09:OnOffException1.java
public class OnOffException1 extends Exception {} ///:~

```

³析构函数是“当对象不再被使用的时候”会被调用的函数。你总能确切地知道析构函数被调用的时间和地点。C++ 能自动调用析构函数，而 C#(它更像 Java) 里面会有自动进行清理的机制。

```

//: c09:OnOffException2.java
public class OnOffException2 extends Exception {} ///:~

//: c09:OnOffSwitch.java
// Why use finally?

public class OnOffSwitch {
    private static Switch sw = new Switch();
    public static void f()
    throws OnOffException1, OnOffException2 {}
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            f();
            sw.off();
        } catch (OnOffException1 e) {
            System.err.println("OnOffException1");
            sw.off();
        } catch (OnOffException2 e) {
            System.err.println("OnOffException2");
            sw.off();
        }
    }
} ///:~

```

程序的目的是要确保main()结束的时候开关必须是关闭的，所以在每个try块和异常处理程序的末尾都加入了对sw.off()方法的调用。但也可能有这种情况：异常被抛出，但没被处理程序捕获，这时sw.off()就得不到调用。但是有了finally，你只要针对try块把清理代码放在一处即可：

```

//: c09:WithFinally.java
// Finally Guarantees cleanup.

public class WithFinally {
    static Switch sw = new Switch();
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            OnOffSwitch.f();
        } catch (OnOffException1 e) {
            System.err.println("OnOffException1");
        } catch (OnOffException2 e) {

```

```

        System.err.println("OnOffException2");
    } finally {
        sw.off();
    }
}
} ///:~

```

这里 `sw.off()` 被移到一处，并且保证在任何情况下都能得到执行。

甚至在异常没有被当前的异常处理程序捕获的情况下，异常处理机制也会在跳到更高一层的异常处理程序之前，执行 **finally** 子句：

```

//: c09:AlwaysFinally.java
// Finally is always executed.
import com.bruceeckel.simpletest.*;

class FourException extends Exception {}

public class AlwaysFinally {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        System.out.println("Entering first try block");
        try {
            System.out.println("Entering second try block");
            try {
                throw new FourException();
            } finally {
                System.out.println("finally in 2nd try block");
            }
        } catch (FourException e) {
            System.err.println(
                "Caught FourException in 1st try block");
        } finally {
            System.err.println("finally in 1st try block");
        }
        monitor.expect(new String[] {
            "Entering first try block",
            "Entering second try block",
            "finally in 2nd try block",
            "Caught FourException in 1st try block",
            "finally in 1st try block"
        });
    }
} ///:~

```

当涉及到break和continue语句的时候，finally子句也会得到执行。请注意，如果把finally子句和带标记的break及continue配合使用，在Java里就没必要使用goto语句了。

缺憾：异常丢失

遗憾的是，Java 的异常实现也有瑕疵。异常作为程序出错的标志，决不应该被忽略，但它还是有可能轻易地被忽略。用某些特殊的方式使用 finally，就会发生这种情况：

```
//: c09:LostMessage.java
// How an exception can be lost.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class VeryImportantException extends Exception {
    public String toString() {
        return "A very important exception!";
    }
}

class HoHumException extends Exception {
    public String toString() {
        return "A trivial exception";
    }
}

public class LostMessage {
    private static Test monitor = new Test();
    void f() throws VeryImportantException {
        throw new VeryImportantException();
    }
    void dispose() throws HoHumException {
        throw new HoHumException();
    }
    public static void main(String[] args) throws Exception {
        LostMessage lm = new LostMessage();
        try {
            lm.f();
        } finally {
            lm.dispose();
        }
        monitor.expect(new String[] {
            "Exception in thread \"main\" A trivial exception",
```

```

        "\tat LostMessage.dispose(LostMessage.java:24)",
        "\tat LostMessage.main(LostMessage.java:31)"
    }); }
} ///:~

```

你能看到，`VeryImportantException`不见了，它被`finally`里的 `HoHumException`所取代。这是相当严重的缺陷，因为异常可能会以一种比前面例子所示更微妙和难以察觉的方式完全丢失。相比之下，C++把“前一个异常还没处理就抛出下一个异常”的情形看成是糟糕的编程错误。也许在Java的未来版本中会修正这个问题（同时，你要把所有类似`dispose()`的方法全部放到`try-catch`语句里面）。

异常的限制

当你重载方法的时候，你只能抛出在父类方法的异常说明里列出的那些异常。这个限制很有用，因为这样的话，对父类能工作的代码应用到子类对象的时候，一样能够工作（当然，这是面向对象的基本概念），异常也不例外。

下面例子演示了这种（在编译期）施加在异常上面的限制：

```

///: c09:StormyInning.java
// Overridden methods may throw only the exceptions
// specified in their base-class versions, or exceptions
// derived from the base-class exceptions.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
    public Inning() throws BaseballException {}
    public void event() throws BaseballException {
        // Doesn't actually have to throw anything
    }
    public abstract void atBat() throws Strike, Foul;
    public void walk() {} // Throws no checked exceptions
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
    public void event() throws RainedOut;
    public void rainHard() throws RainedOut;
}

```

```

}

public class StormyInning extends Inning implements Storm {
    // OK to add new exceptions for constructors, but you
    // must deal with the base constructor exceptions:
    public StormyInning()
        throws RainedOut, BaseballException {}
    public StormyInning(String s)
        throws Foul, BaseballException {}
    // Regular methods must conform to base class:
    //! void walk() throws PopFoul {} //Compile error
    // Interface CANNOT add exceptions to existing
    // methods from the base class:
    //! public void event() throws RainedOut {}
    // If the method doesn't already exist in the
    // base class, the exception is OK:
    public void rainHard() throws RainedOut {}
    // You can choose to not throw any exceptions,
    // even if the base version does:
    public void event() {}
    // Overridden methods can throw inherited exceptions:
    public void atBat() throws PopFoul {}
    public static void main(String[] args) {
        try {
            StormyInning si = new StormyInning();
            si.atBat();
        } catch(PopFoul e) {
            System.err.println("Pop foul");
        } catch(RainedOut e) {
            System.err.println("Rained out");
        } catch(BaseballException e) {
            System.err.println("Generic baseball exception");
        }
        // Strike not thrown in derived version.
        try {
            // What happens if you upcast?
            Inning i = new StormyInning();
            i.atBat();
            // You must catch the exceptions from the
            // base-class version of the method:
        } catch(Strike e) {
            System.err.println("Strike");
        } catch(Foul e) {
            System.err.println("Foul");
        }
    }
}

```

```

    } catch(RainedOut e) {
        System.err.println("Rained out");
    } catch(BaseballException e) {
        System.err.println("Generic baseball exception");
    }
}
} ///:~

```

在 `Inning` 类中，你能发现构造器和 `event()` 方法都声明将抛出异常，但实际上没有抛出。这种方式使你能强制用户去捕获可能在重载版本的 `event()` 方法中抛出的异常，所以它很合理。这对于抽象方法同样成立，比如 `atBat()`。

接口 `Storm` 值得注意，因为它包含了一个在 `Inning` 中定义的方法 `event()` 和一个不是在 `Inning` 中定义的方法 `rainHard()`。这两个方法都抛出新的异常 `RainedOut`。如果 `StormyInning` 类在继承 `Inning` 类的同时又实现了 `Storm` 接口，那么 `Storm` 里的 `event()` 方法就不能改变在 `Inning` 中的 `event()` 方法的异常说明。否则的话，你在使用基类的时候就不能判断是否捕获了正确的异常，所以这也很合理。当然，如果接口里定义的方法不是来自于基类，比如 `rainHard()`，那么此方法抛出什么样的异常都没有问题。

异常限制对构造器不起作用。你会发现 `StormInning` 的构造器可以抛出任何异常，而不必理会基类构造器的异常说明。然而，因为基类构造器必须以这样或那样的方式被调用（这里缺省构造器将自动被调用），派生类构造器的异常说明必须包含基类构造器的异常说明。注意，派生类构造器不能捕获基类构造器抛出的异常。

`StormyInning.walk()` 不能通过编译的原因是因为它抛出了异常，而 `Inning.walk()` 并没有声明此异常。如果编译器允许你这么说的话，你就可以在调用 `Inning.walk()` 的时候不用做异常处理了，而你把它替换成 `Inning` 派生类的对象时，这个方法就有可能抛出异常，于是程序就失灵了。通过强制派生类遵守基类方法的异常说明，对象的可替换性得到了保证。

重载后的 `event()` 方法显示，派生类方法可以不抛出任何异常，即使它是基类所定义的异常。这是因为，即使基类的方法会抛出异常，这样做也不会破坏已有的程序，所以也没有问题。类似的情况出现在 `atBat()` 身上，它抛出的是 `Popfoul`，这个异常是继承自“会被基类的 `atBat()` 抛出”的 `Foul`。这样，如果你写的代码是同 `Inning` 打交道的，并且调用了它的 `atBat()` 的话，你就肯定能捕获 `Foul`。而 `PopFoul` 是由 `Foul` 派生出来的，因此异常处理程序也能捕获 `PopFoul`。

最后一个值得注意的地方是 `main()`。这里你能看到，如果你处理的刚好是 `StormyInning` 对象的话，编译器只会强制要求你捕获这个类所抛出的异常。但是如果你将它转型成基类型，那么编译器就会（正确地）要求你捕获基类的异常。所有这些限制都是为了能产生更为强壮的异常处理程序。⁴

⁴ ISO C++ 加上了类似的约束，要求派生类的方法所抛出的异常要与基类方法相同，或者是基类方法抛出的异常的派生类。这是 C++ 真正能够在编译期对异常说明进行检查的情况之一。

尽管在继承过程中，编译器会对异常说明做强制要求，但异常说明本身并不属于方法原型的一部分，方法原型是由方法的名字与参数的类型组成的，理解这一点非常有用。因此，你不能根据异常说明的不同来重载方法。此外，一个出现在基类方法的异常说明中的异常，不一定会出现在派生类方法的异常说明里。这点同继承的规则明显不同，在继承中，基类的方法必须出现在派生类里，换一句话说，在继承和重载的过程中，方法的“异常说明的接口”不是变大了而是变小了——这恰好和类接口在继承时的情形相反。

构造器（Constructor）

当你编写的代码涉及到异常的时候，要时刻提醒自己“如果异常发生了，它能被正确地清除吗？”尽管大多数情况下你是非常安全的，但涉及到构造器时，问题就出现了。构造器会把对象设置成安全的初始状态，但还会有别的动作，比如打开一个文件，这样的动作只有在对象使用完毕并且用户调用了特殊的清除方法之后才能得到清除。如果在构造器内抛出了异常，这些清除行为也许就不能正常工作了。这意味着你在编写构造器时要格外细心。

鉴于刚学习过 **finally**，你也许会认为使用它就可以解决问题。但问题并非如此简单，因为 **finally** 会每次都执行清除代码，甚至是你还不需要的时候它也会执行。因此，如果你真的要用 **finally** 来进行清除的话，你可以在构造器正常结束时设置一个标志，然后在 **finally** 里判断，如果设定了这个标志，就不必做任何清除。这种方法不是特别优雅（这两处代码形成了耦合），因此除非没有别的办法，最好还是避免这样在 **finally** 里进行清除。

在下面的例子中，建立了一个 **InputFile** 类，它能打开一个文件并且每次读取其中的一行（转换成字符串）。这里使用了 Java 标准输入/输出库中的 **FileReader** 和 **BufferedReader**（将在 12 章讨论），不过它们的基本用法很简单，你应该很容易明白：

```
//: c09:Cleanup.java
// Paying attention to exceptions in constructors.
import com.bruceeckel.simpletest.*;
import java.io.*;

class InputFile {
    private BufferedReader in;
    public InputFile(String fname) throws Exception {
        try {
            in = new BufferedReader(new FileReader(fname));
            // Other code that might throw exceptions
        } catch (FileNotFoundException e) {
            System.err.println("Could not open " + fname);
            // Wasn't open, so don't close it
            throw e;
        }
    }
}
```



```

    } catch(Exception e) {
        // All other exceptions must close it
        try {
            in.close();
        } catch(IOException e2) {
            System.err.println("in.close() unsuccessful");
        }
        throw e; // Rethrow
    } finally {
        // Don't close it here!!!
    }
}

public String getLine() {
    String s;
    try {
        s = in.readLine();
    } catch(IOException e) {
        throw new RuntimeException("readLine() failed");
    }
    return s;
}

public void dispose() {
    try {
        in.close();
        System.out.println("dispose() successful");
    } catch(IOException e2) {
        throw new RuntimeException("in.close() failed");
    }
}

}

public class Cleanup {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        try {
            InputFile in = new InputFile("Cleanup.java");
            String s;
            int i = 1;
            while((s = in.getLine()) != null)
                ; // Perform line-by-line processing here...
            in.dispose();
        } catch(Exception e) {
            System.err.println("Caught Exception in main");
            e.printStackTrace();
        }
    }
}

```

```

    }
    monitor.expect(new String[] {
        "dispose() successful"
    });
}
} ///:~

```

`InputFile` 的构造器接受字符串作为参数，该字符串表示你要打开的文件名。在 `try` 块中，使用文件名建立了 `FileReader` 对象。`FileReader` 对象本身用处并不大，但可以用它来建立 `BufferedReader` 对象，而这正是你实际使用的对象。注意，使用 `InputFile` 的好处就能是把两步操作合而为一。

如果 `FileReader` 的构造器失败了，将抛出 `FileNotFoundException` 异常，它必须被捕获。对于这个异常，并不需要关闭文件，因为这个文件还没有被打开。而捕获其它异常的 `catch` 子句必须关闭文件，因为在它们捕获到异常之时，文件已经打开了。（当然，如果还有其它方法能抛出 `FileNotFoundException`，这个方法就显得有些投机取巧了。这时，你最好还是把这些方法分别放到各自的 `try` 块里。）`close()` 方法也可能会抛出异常，所以尽管是已经在 `try` 块里面了，还是要再用一层 `try-catch`。对 Java 编译器而言，这只不过是又多了一对花括号。在本地做完处理之后，异常被重新抛出，对于构造器而言这么做是很合适的，因为你总不希望去误导调用方，让他认为“这个对象已经创建完毕，可以使用了”。

这个例子没有使用前面提到的设立标志的技巧，由于 `finally` 会在每次完成构造器之后都执行一遍，因此它实在不该是调用 `close()` 以关闭文件的地方。我们希望文件在 `InputFile` 对象的整个生命周期内都处于打开状态，所以这么做肯定不合适。

`getLine()` 方法会返回表示文件下一行内容的字符串。它调用了能抛出异常的 `readLine()`，但是这个异常已经在方法内得到处理，因此 `getLine()` 不会抛出任何异常。在设计异常的时候存在一个问题：应该把异常全部放在这一层处理；还是先处理一部分，然后再向上层抛出相同的（或是新的）异常；又或是不做任何处理直接向上面抛。如果用法恰当的话，直接向上面抛的确能简化编程。在这里，`getLine()` 方法将异常转换为 `RuntimeException`，表示一个编程错误。

用户在用完 `InputFile` 对象之后必须调用 `dispose()` 方法。这将释放 `BufferedReader` 和 / 或 `FileReader` 对象所占用的系统资源（比如文件句柄）。在你用完 `InputFile` 对象之前是不会调用它的，所以现在你不需要这个。可能你会考虑把上述功能放到 `finalize()` 里面，但我在第 4 章讲过，你不知道 `finalize()` 会不会被调用（即使被调用，你也不知道在什么时候）。这也是 Java 的缺陷：除了内存的清除之外，所有的清除都不会自动发生。所以你必须告诉客户端程序员，他们也是有责任的，并且要在使用 `finalize()` 的时候确保清除动作的发生。

`Cleanup.java` 所创建的 `InputFile` 对象打开了这个程序的源文件，然后逐行读取文件，再加上行号。虽然我们也可以用更精巧的方法，但这里所有的异常都在 `main()` 中用 `Exception` 捕获。

这个例子还有一个好处，它向你解释了为什么本书要在这里介绍异常----如果不懂异常如何处理的话，很多类库（像前面提到的输入/输出库）根本没法使用。异常同 Java 程序的编写结合得如此紧密，特别是编译器也会做强制要求，因此不知道如何使用异常的话，你也就只能到此为止了。

异常匹配

抛出异常的时候，异常处理系统会按照你书写代码的顺序找出“最近”的处理程序。找到匹配的处理程序之后，它就认为异常将得到处理，然后就不再继续查找。

查找的时候并不要求抛出的异常同处理程序所声明的异常完全匹配。派生类的对象也可以匹配处理程序中声明的基类，就像这样：

```
//: c09:Human.java
// Catching exception hierarchies.
import com.bruceeckel.simpletest.*;

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        try {
            throw new Sneeze();
        } catch(Sneeze s) {
            System.err.println("Caught Sneeze");
        } catch(Annoyance a) {
            System.err.println("Caught Annoyance");
        }
        monitor.expect(new String[] {
            "Caught Sneeze"
        });
    }
} ///:~
```

Sneeze 会被第一个匹配的 catch 子句捕获，也就是程序里的第一个。然而如果你将这个 catch 删掉，只留下：

```
try {
    throw new Sneeze();
} catch(Annoyance a) {
    System.err.println("Caught Annoyance");
}
```

```
}
```

该程序仍然能运行，因为这次捕获的是 `Sneeze` 的基类。换句话说，`catch(Annoyance e)` 会捕获 `Annoyance` 以及所有从它派生的异常。这一点非常有用，因为如果你要在方法里加上更多派生异常的话，只要客户程序员捕获的是基类异常，那么它们的代码就无需更改。

如果你把捕获基类的 `catch` 子句放在最前面，就会把派生类的异常全给“屏蔽”掉，像这样：

```
try {
    throw new Sneeze();
} catch(Annoyance a) {
    System.err.println("Caught Annoyance");
} catch(Sneeze s) {
    System.err.println("Caught Sneeze");
}
```

这样编译器就会发现 `Sneeze` 的 `catch` 子句永远也得不到执行，因此它会向你报告错误。

其它可选方式

异常处理系统就像一个“活门（trap door）”，使你能放弃程序的正常执行序列。当“异常情形”发生的时候，正常的执行已变得不可能或者不需要了，这时就要用到这个“活门”。异常代表了当前方法不能继续执行的情形。开发异常处理系统的原因是，如果为每个方法所有可能发生的错误都进行处理的话，任务就显得过于繁重了，程序员也不愿意这么做。结果常常是，他们将错误忽略。应该注意到，开发异常处理的初衷是为了方便程序员处理错误。

异常处理的一个重要原则是“只有在你知道如何处理的情况下才捕获异常”。实际上，异常处理的一个重要目标就是把错误处理的代码同错误发生的地点相分离。这使你能在一段代码中专注于要完成的事情，至于如何处理错误，则放在另一段代码中完成。结果，你的主干代码就不会与错误处理逻辑混在一起，也更容易理解和维护。

“被检查的异常”（checked exception）使这个问题变得有些复杂，因为你可能在还没准备好处理错误的时候被迫加上 `catch` 语句。这就导致了“吞咽则有害（harmful if swallowed）”的问题：

```
try {
    // ... to do something useful
} catch(ObligatoryException e) {} // Gulp!
```

程序员们（也包括我写本书第一版的时候）常常是无意中“吞下”了异常。一旦这么做，确实能通过编译。然而除非你复查并改正代码，否则异常将会永远丢失。异常确实发生了，但又被完全“吞没”了。尽管是编译器强迫你写代码时立刻处理异常，但这种看起来最简单的方法，却可能是最糟糕的作法。

当我意识到犯了这么大一个错误时，简直吓了一跳。在本书第二版中，我在处理程序里用打印栈轨迹（**stack trace**）的方法来“修补”这个问题（本章中的很多例子还是使用了这个方法，看起来还是比较合适的）。虽然这样可以跟踪异常的行为，但是你仍旧不知道该如何处理异常。这一节，我们来研究一下“被检查的异常”及其并发症，以及采用什么方法来解决这些问题。

这个话题看起来简单，但实际上它不仅复杂，更重要的是还非常多变。总有人会顽固地坚持自己的立场，声称正确（也是他们的）答案是显而易见的。我觉得之所以会有这种观点，是因为我们使用的工具已经不是“ANSI 标准出台前的 C 那样的”弱类型语言（**poorly-typed language**），而是像 C++ 和 Java 这样的“强静态类型语言”（**strong statically-typed language**，也就是编译时就做类型检查的语言），这是前者无法比拟的。当你刚开始这个转变的时候（就像我一样），会发现它带来的好处是那样的明显，好像强类型检查总能解决所有的问题。在此，我想结合我自己的认识过程，告诉你我是怎样从对类型检查的绝对迷信变成怀疑的；当然，很多时候它还是非常有用的，但是当它挡住我们的去路并成为障碍的时候，我们就得跨过去。只是这条界限往往并不是很清晰。（我最喜欢的一句格言是：“所有模型都是错误的。但有些是能用的。”）

历史

异常处理起源于 PL/1 和 Mesa 之类的系统中，后来又出现在 CLU，SmallTalk，Modula-3，Ada，Eiffel，C++，Python，Java 以及 Java 后面的 Ruby 和 C# 中。Java 的设计和 C++ 很相似，只是 Java 的设计者去掉了一些他们认为 C++ 设计得不好的东西。

为了能向程序员提供一个他们更愿意使用的错误处理和恢复的框架，异常处理机制很晚才被加入 C++ 标准化过程中，这个倡议是由 C++ 设计者 Bjarne Stroustrup 所发起的。C++ 的异常模型主要借鉴了 CLU 的作法。然而，当时其他语言已经支持异常处理了：包括 Ada，Smalltalk（两者都有异常处理，但是都没有异常说明），以及 Modula-3（它既有异常处理也有异常说明）。

Liskov 和 Snyder 在他们有关异常的开创性论文⁵中指出，用 C 的瞬时风格（**transient fashion**）报告错误的语言有一个主要缺陷，就是：

“...每次调用的时候都必须执行条件测试，以确定会产生何种结果。这使程序难以阅读，并且有可能降低运行效率，因此程序员们既不愿意指出，也不愿意处理意外情况。”

⁵ Barbara Liskov 和 Alan Snyder: CLU 的异常处理 (*Exception Handling in CLU*), IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, 1979 年 11 月。这篇论文在网上是找不到的，只有印刷版本，所以你得去图书馆找一个副本。

注意，异常处理的初衷是要消除这种限制，但是我们又从 Java 的“被检查的异常”上看到了这种代码。他们继续写道：

“...在调用会引发异常的函数的同时，还要求程序员给出异常处理程序，这会降低程序的可读性，使得程序的正常思路被异常处理给破坏了。”

C++异常的设计参考了CLU方式。Stroustrup声称其目标是减少恢复错误所需的代码。我想他这话是说给那些“通常情况下都不写C的错误处理”的程序员们听的，因为要把那么多代码放到那么多地方实在不是什么好差事。所以他们写C程序的习惯是，忽略所有的错误，然后使用调试器来跟踪错误。这些程序员知道，使用异常就意味着他们要写一些通常不用写的“多出来的”代码。因此，要把他们拉到“使用错误处理”的正轨上，“多出来的”代码决不能太多。我认为，评价Java的“被检查的异常”的时候，这一点是很重要的。

C++还从CLU那里还带来另一种思想：异常说明。这样就可以用编程的方式在方法签名中声明这个方法可能会抛出哪些异常。异常说明可能有两种意思。一个是“我的代码会产生这种异常，你得处理”。另一个是“我的代码忽略了这些异常，它要由你来处理”。学习异常处理的机制和语法的时候，我们一直在关注“你来处理”的部分，但这里特别值得注意的事实是，我们通常都忽略了异常说明所表达的含义。

C++的异常说明不属于函数的类型信息。编译时唯一要检查的是异常说明是不是前后一致；比如，如果一个函数或方法会抛出某些异常，那么它的重载版本或者派生版本也必须抛出同样的异常。与Java不同，C++不会在编译时进行检查以确定函数或方法是不是真的抛出异常，或者异常说明是不是完整（也就是说，异常说明有没有精确描述所有可能被抛出的异常）。这样的检查只发生在运行期间。如果抛出的异常与异常说明不符，C++会调用标准类库的unexpected()函数。

值得注意的是，由于使用了模板（template），C++的标准类库实现里根本没有使用异常说明。由此看来，异常说明会对Java的泛型（generics）产生非常重大的影响（Java里面对应C++模板的功能，可望在JDK 1.5中出现）。

观点

首先，Java无谓地发明了“被检查的异常”（很明显是受C++异常说明的启发，以及C++程序员们一般对此无动于衷的事实）。这还只是一次尝试，目前为止还没有别的语言采用这种做法。

第二，仅从示意性的例子和小程序来看，“被检查的异常”好处很明显。但是当程序开始变大的时候，就会带来一些微妙的问题。当然，程序不是一下就变大的，这有个过程。如果把不适用于大项目的语言用于不断膨胀的小项目，突然有一天你会发现，原来可以管理的東西，现在已经变得无法管理了。这就是我所说的过多的类型检查，特别是“被检查的异常”所造成的问题。

看来程序的规模是个重要的因素。由于很多讨论都用小程序来做演示，因此这并不足以说明问题。有个C#的设计人员发现：

“仅从小程序来看，你会认为异常说明能增加开发人员的效率，并提高代码的质量；但考察大项目的时候，结论就不同了：开发效率下降了，而代码质量只有微不足道的提高，甚至毫无提高”。⁶

谈到未被捕获的异常的时候，CLU的设计师们认为：

“我们觉得强迫程序员在不知道该采取什么措施的时候提供处理程序，是不现实的。”⁷

在解释为什么“函数没有异常说明就表示可以抛出任何异常”的时候，Stroustrup这样认为：

“但是，这样一来几乎所有的函数都得提供异常说明了，也就都得重新编译，而且还会妨碍它同其它语言的交互。这样会迫使程序员违反异常处理机制的约束，他们会写欺骗程序来掩盖异常。这将给没有注意到这些异常的人造成一种虚假的安全感。”⁸

我们已经看到这种破坏异常机制的行为了，就在Java的“被检查的异常”里。

Martin Fowler (《UML Distilled》，《Refactoring》和《Analysis Patterns》的作者)给我写了下面这段：

“...总体来说，我觉得异常很不错，但是Java的“被检查的异常”带来的麻烦比好处要多。”

我觉得Java的当务之急应该是统一其报告错误的模型，这样所有的错误都能通过异常来报告。C++不这么做的原因是它要考虑向后兼容，要照顾那些直接忽略所有错误的C代码。但是如果你一致地用异常来报告错误，那么只要愿意，随时可以抛出异常，如果不愿意，这些错误会被传播到最上层（控制台或其它容器程序（container program））。只有当Java修改了它那类似C++的模型，使异常成为报告错误的唯一方式，那时“被检查的异常”的额外限制也许就会变得没有那么必要了。

过去，我曾坚定地认为“被检查的异常”和强静态类型检查对开发健壮的程序是非常必要的。但是，我看到的和我使用一些动态（类型检查）语言的亲身经历⁹告诉我，这些好处实际上是来自于：

1. 不在于编译器是否会强制程序员去处理错误，而是要有一致的使用异常来报告错误的模型。

⁶ <http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820>

⁷ 同上

⁸ Bjarne Stroustrup, *The C++ Programming Language*, 3rd edition, Addison-Wesley 1997, 第 376 页。

⁹ 间接经验来自于与很多资深Smalltalk程序员的谈话；直接的则是得自于Python（www.Python.org）

2. 不在于什么时候进行检查，而是一定要有类型检查。也就是说，必须强制程序使用正确的类型，至于这种强制施加于编译期还是运行期，那到没关系。

此外，减少编译期施加的约束能显著提高程序员的编程效率。事实上，反射（reflection）（还有泛型generics）都是用来补偿强静态类型检查所带来的过多限制，你会在下一章及本书很多例子中见到这种情形。

我已经听到有人在指责了，他们认为这种言论会令我名誉扫地，会让文明堕落，会导致更高比例的项目失败。他们的信念是应该在编译期指出所有错误，这样才能挽救项目，这种信念可以说是无比坚定的；其实更重要的是要理解编译器的能力限制；在第 15 章，我强调了自动构建过程（automated build process）和单元测试（unit testing）的重要性。比起把所有的东西都说成是语法错误，它们的效果可以说是事半功倍。下面这段话是至理名言：

好的程序设计语言能帮助程序员写出好程序，但无论哪种语言都挡不住程序员去写坏程序。¹⁰

不管怎么说，要让 Java 把“被检查的异常”从语言中去除，这种可能性看来非常渺茫。对语言来说，这个变化可能太激进了点，况且 Sun 的支持者们也非常强大。Sun 有完全向后兼容的历史和策略，实际上所有 Sun 的软件都能在 Sun 的硬件上运行，无论它们有多么古老。然而，如果你发现有些“被检查的异常”挡住了你的路，尤其是你发现你不得不去对付那些不知道该如何处理的异常，还是有些办法的。

把异常传递给控制台

对于简单的程序，比如本书中的许多例子，最简单而又不用写多少代码就能保持异常信息的方法，就是把它们从 `main()` 传递到控制台。例如，为了读取信息而打开一个文件（在第 12 章将详细介绍），你必须对 `FileInputStream` 进行打开和关闭操作，这就可能会产生异常。对于简单的程序，你可以像这样做（本书中很多地方采用了这种方法）：

```
//: c09:MainException.java
import java.io.*;

public class MainException {
    // Pass all exceptions to the console:
    public static void main(String[] args) throws Exception {
        // Open the file:
        FileInputStream file =
            new FileInputStream("MainException.java");
        // Use the file ...
    }
}
```

¹⁰ (Kees Koster, CDL语言的设计者，引自Eiffel语言的设计者Bertrand Meyer)。
<http://www.elj.com/elj/v1/n1/bm/right/>。


```

        // Close the file:
        file.close();
    }
} ///:~

```

注意，`main()` 作为一个方法也可以有异常说明，这里异常的类型是 `Exception`，它也是所有“被检查的异常”的基类。通过把它传递到控制台，你就不必在 `main()` 里写 `try-catch` 子句了。（不过，实际的文件输入/输出操作比这个例子显示的要复杂得多，所以在学习第 12 章之前，别高兴得太早）。

把“被检查的异常”转换为“不检查的异常”

写 `main()` 的时候，抛出异常是很方便的，但这不是通用的方法。问题的实质是，当你在一个普通方法里调用别的方法时，你要考虑到“我不知道该这样处理这个异常，但是也不能把它‘吞’了，或者打印一些无用的消息”。JDK 1.4 的异常链提供了一种新的思路来解决这个问题。你直接把“被检查的异常”包装进 `RuntimeException` 里面，就像这样：

```

try {
    // ... to do something useful
} catch (IDontKnowWhatToDoWithThisCheckedException e) {
    throw new RuntimeException(e);
}

```

如果你想把“被检查的异常”的功能“屏蔽”掉的话，这看上去像是一个好办法。你不用吞没异常，也不必把它放到方法的异常说明里面，而异常链还能保证你不会丢失任何原始异常的信息。

这种技巧给了你一种选择，你可以不写 `try-catch` 子句和/或异常说明，直接忽略异常，让它自己沿着调用栈往上跑。同时，你还可以用 `getCause()` 捕获并处理特定的异常，就像这样：

```

//: c09:TurnOffChecking.java
// "Turning off" Checked exceptions.
import com.bruceeckel.simpletest.*;
import java.io.*;

class WrapCheckedException {
    void throwRuntimeException(int type) {
        try {
            switch(type) {
                case 0: throw new FileNotFoundException();
                case 1: throw new IOException();
            }
        }
    }
}

```

```

        case 2: throw new RuntimeException("Where am I?");
        default: return;
    }
} catch (Exception e) { // Adapt to unchecked:
    throw new RuntimeException(e);
}
}
}

```

```

class SomeOtherException extends Exception {}

```

```

public class TurnOffChecking {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        WrapCheckedException wce = new WrapCheckedException();
        // You can call f() without a try 区块, and let
        // RuntimeExceptions go out of the method:
        wce.throwRuntimeException(3);
        // Or you can choose to catch exceptions:
        for(int i = 0; i < 4; i++)
            try {
                if(i < 3)
                    wce.throwRuntimeException(i);
                else
                    throw new SomeOtherException();
            } catch (SomeOtherException e) {
                System.out.println("SomeOtherException: " + e);
            } catch (RuntimeException re) {
                try {
                    throw re.getCause();
                } catch (FileNotFoundException e) {
                    System.out.println(
                        "FileNotFoundException: " + e);
                } catch (IOException e) {
                    System.out.println("IOException: " + e);
                } catch (Throwable e) {
                    System.out.println("Throwable: " + e);
                }
            }
        }
    }

    monitor.expect(new String[] {
        "FileNotFoundException: " +
        "java.io.FileNotFoundException",
        "IOException: java.io.IOException",
        "Throwable: java.lang.RuntimeException: Where am I?",
    });
}

```

```
        "SomeOtherException: SomeOtherException"
    });
}
} ///:~
```

`WrapCheckedException.throwRuntimeException()` 的代码可以生成不同类型的异常。这些异常被捕获并包装进了 `RuntimeException` 对象，所以它们成了这些运行期异常的“cause”了。

在 `TurnOffChecking` 里，你可以不用 `try` 块就调用 `throwRuntimeException()`，因为它没有抛出“被检查的异常”。但是，当你准备好去捕获异常的时候，你还是可以用 `try` 块来捕获任何你想捕获的异常。你应该捕获 `try` 块肯定会抛出的异常，这里就是 `SomeOtherException`。`RuntimeException` 要放到最后去捕获。然后把 `getCause()` 的结果（也就是包装在里面的那个原始异常）抛出来。这样就把原先的那个异常给提取出来了，然后就可以用它们自己的 `catch` 子句进行处理。

本书的其余部分将会在合适的时候使用这种“用 `RuntimeException` 来包装‘被检查的异常’”的技术。

异常使用指南

你应该在下列情况下使用异常：

1. 在恰当的级别处理问题。（在你知道该如何处理的情况下才捕获异常）。
2. 解决问题并且重新调用产生异常的方法。
3. 进行少许修补，然后绕过异常发生的地方继续执行。
4. 用别的数据进行计算，以代替方法返回的期望值。
5. 把当前运行环境下能做的事情尽量作完，然后把相同的异常重抛（`rethrow`）到更高层。
6. 把当前运行环境下能做的事情尽量作完，然后把不同的异常抛（`throw`）到更高层。
7. 终止程序。
8. 进行简化。（如果异常把问题搞得太复杂，那用起来会非常痛苦也很烦人。）
9. 让类库和程序更安全。（这既是在为调试做短期投资，也是在为程序的健壮性做长期投资。）

总结

改良的错误恢复机制是增强代码健壮性的最强有力的方式之一。对每个程序来说，错误恢复都是要考量的基本问题，在 `Java` 中尤其如此，因为 `Java` 的最初目的就是用来建立给别人使用的构件。“要构造健壮的系统，组成系统的每个构件也必须是健壮的”。通过使用异常来提供了一致的错误报告模型，`Java` 使构件能把错误信息可靠地通知给客户代码。

Java 异常处理的目的是尽可能用比现在更少的代码，更容易开发出大型、可靠的程序。并且在开发过程中你更有信心，因为程序中所有错误都将得到处理。异常这种语言功能并非很难学习，而且能为你的项目带来立竿见影的效果。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 编写一个类，在 `main()` 的 `try` 块里抛出一个 `Exception` 对象。传递一个字符串参数给 `Exception` 的构造器。在 `catch` 子句里捕获此异常对象，并且打印字符串参数。添加一个 `finally` 子句，打印一条信息以证明这里确实得到了执行。
2. 使用 `extends` 关键字建立一个自定义异常类。为这个类写一个接受字符串参数的构造器，把此参数保存在对象内部的字符串引用中。写一个方法打印此字符串。写一个 `try-catch` 子句，对这个新异常进行测试。
3. 定义一个类，令其方法抛出在练习 2 里定义的异常。不用异常说明，看看能否通过编译。然后加上异常说明，用 `try-catch` 子句测试你的类和异常。
4. 定义一个对象引用并初始化为 `null`，尝试用此引用调用方法。把这个调用放在 `try-catch` 子句里以捕获异常。
5. 为一个类定义两个方法，`f()` 和 `g()`。在 `g()` 里，抛出一个你定义的新异常。在 `f()` 里，调用 `g()`，捕获它抛出的异常，并且在 `catch` 子句里抛出另一个异常（你要定义的第二种异常）。在 `main()` 里测试你的代码。
6. 重复上一个练习，但是在 `catch` 子句里把 `g()` 要抛出的异常包装成一个 `RuntimeException`。
7. 定义三种新的异常，写一个类，在一个方法中抛出这三种异常。在 `main()` 里调用这个方法，仅用一个 `catch` 子句捕获这三种异常。
8. 编写能产生 `ArrayIndexOutOfBoundsException` 异常的代码，并将其捕获。
9. 使用 `while` 循环建立类似“恢复模型”的异常处理行为，它将不断重试，直到异常不再抛出。
10. 建立一个三层的异常继承体系，然后创建基类 `A`，它的一个方法能抛出异常体系的基类异常。用 `B` 继承 `A`，并且重载这个方法，让它抛出第二层的异常。让 `C` 继承 `B`，再次重载这个方法，让它抛出第三层的异常。在 `main()` 里面创建一个 `C` 类型的对象，把它向上转型为 `A`，然后调用这个方法。
11. 试证明，派生类的构造器不能捕获它的基类构造器所抛出的异常。
12. 试说明，如果在 `OnOffSwitch.java` 的 `try` 块抛出 `RuntimeException` 之后程序会出现错误。
13. 试说明，如果在 `WithFinally.java` 的 `try` 块抛出 `RuntimeException` 之后程序不会出现错误。
14. 修改练习 7，加一个 `finally` 子句。验证一下，即便是抛出 `NullPointerException` 的情况下，`finally` 子句也会得到执行。
15. 用“构造器”这一节第 2 段介绍的方法，写一个示例，用标志来控制是否调用清理代码。

16. 修改 StormyInning.java, 加一个 UmpireArgument 异常, 和一个能抛出此异常的方法。测试一下修改后的异常继承体系。
17. 删除 Human.java 的第一个 catch 子句, 并验证代码仍然能正确编译和运行。
18. 让 LostMessage.java 再多丢失一次异常, 即用第三个异常来掩盖 HoHumException 异常。
19. 为第 8 章的 GreenhouseControls.java 添加一组合适的异常。
20. 为第 8 章的 Sequence.java 添加一组合适的异常。
21. 用一个不存在的文件名来替换 MainException.java 中的文件名。运行程序并观察结果。

第十章 类型检查

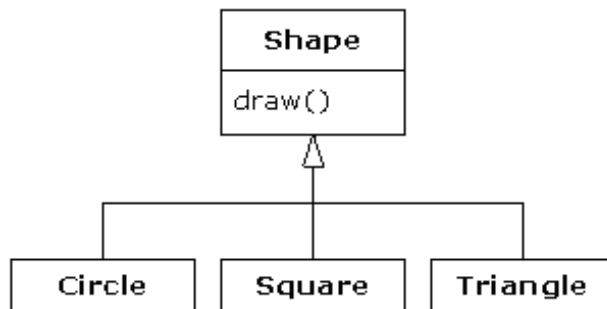
运行期类型识别（RTTI，run-time type identification）的概念初看起来非常简单：当你只有一个指向对象的基类的引用时，RTTI 机制可以让你找出这个对象确切的类型。

对 RTTI 的需要，揭示了面向对象设计中许多有趣（并且复杂）的问题，同时也提出了如何组织程序的问题。

本章将讨论 Java 是如何允许我们在运行期识别对象和类的信息。主要有两种方式：一种是传统的 RTTI，它假定我们在编译期和运行期已经知道了所有的类型；另一种是“反射机制（reflection）”，它允许我们在运行期获得类的信息。我们先讨论“传统”的 RTTI，再讨论反射。

为什么需要 RTTI

让我们来思考已经很熟悉了的一个使用了多态的类层次结构的例子。最一般化的类型是基类 Shape，而派生出的具体类有 Circle，Square 和 Triangle。



这是一个典型的类层次结构图，基类位于顶部，派生类向下扩展。面向对象编程基本的目的是：你的代码只操纵对基类（这里是 Shape）的引用。这样，如果你要添加一个新类（比如从 Shape 派生 Rhomboid）来扩展程序，就不会影响到原来的代码。在这个例子的 Shape 接口中动态绑定了 draw() 方法，目的就是让客户端程序员使用一般化的 Shape 的引用来调用 draw()。draw() 在所有派生类里都会被重载，并且由于它是被动态绑定的，所以即使是通过通用的 Shape 引用来调用，也能产生正确行为。这就是多态（polymorphism）。

因此，我们通常会创建一个特定的对象（Circle，Square，或者 Triangle），把它向上转型成 Shape（忽略对象的特定类型），并在后面的程序中使用匿名（译注：即不知道具体类型）的 Shape 引用。

简要复习一下多态和向上类型转换，并为上面的例子编码：

```
//: c10:Shapes.java
import com.bruceeckel.simpletest.*;
```

```

class Shape {
    void draw() { System.out.println(this + ".draw()"); }
}

class Circle extends Shape {
    public String toString() { return "Circle"; }
}

class Square extends Shape {
    public String toString() { return "Square"; }
}

class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}

public class Shapes {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        // Array of Object, not Shape:
        Object[] shapeList = {
            new Circle(),
            new Square(),
            new Triangle()
        };
        for(int i = 0; i < shapeList.length; i++)
            ((Shape)shapeList[i]).draw(); // Must cast
        monitor.expect(new String[] {
            "Circle.draw()",
            "Square.draw()",
            "Triangle.draw()"
        });
    }
} ///:~

```

基类中包含 draw() 方法，它通过传递 this 参数给 System.out.println(), 间接地使用 toString() 打印类标识符。如果是某个对象调用这个方法，它会自动调用 toString() 生成字符串。每个派生类都要重载(从 Object 类继承来的)toString() 方法，这样 draw() 在不同情况下就打印出不同的消息。

在 main() 中，生成了各种特定类型的 Shape，并加入到数组中。这个数组有点特别，因为它不是一个 Shape 的数组（虽然它可以是），而是根类 Object 类的对象的数组。这样做的原因是为第十一章作准备，我们将学习 collection 工具（也被称作容器

container)，它唯一的工作是保存与管理对象。基于通用性的考虑，这些 collection 应该能保存任何类型的对象，因此它们保存根类 Object 类的对象。Object 类的数组引出了我们将在第十一章 Collection 中学习的一个重要的问题。

在这个例子中，当把 Shape 对象放入 Object 类的数组时会向上转型。由于在 Java 中所有的对象都是根类 Object 类的对象（除了基本类型），所以一个 Object 的数组自然能保存 Shape 类的对象。但在向上类型转换为 Object 的时候也失去了作为 Shape 对象特定的信息。对于这个数组，它们就只是 Object 的对象。

当你通过索引操作符从数组中取出一个元素时，就要多做些事情了。由于数组保存的只能是 Object 对象，因此通过索引获得的也只是 Object 对象的引用。但我们知道那其实是 Shape 对象的引用，而且我们想给这个对象发送 Shape 对象能够接收的消息。所以必须使用传统的“(Shape)”方式显式地将 Object 对象的引用转换成 Shape 对象的引用。这是 RTTI 最基本的使用形式，因为在 Java 中，所有的类型转换都是在运行期检查的。这也是 RTTI 名字的来源：在运行期间，识别一个对象的类型。

在这个例子中，RTTI 类型转换并不彻底：Object 被转型为 Shape，而不是转型为 Circle, Square, 或者 Triangle。这是因为目前我们只知道这个数组保存的都是 Shape。在编译时刻，这只能由你自己设定的规则来强制确保这一点，而在运行时刻，由类型转换操作来确保这一点。

接下来就是多态机制的事情了，Shape 对象实际上执行什么样的代码，是由引用指向的具体对象是 Circle, Square 或者 Triangle 而决定的。通常这正是它应该执行的行为；你希望你的大部分代码尽可能少的了解对象特定的类型，而是只与一个对象家族中通用表示打交道（在这个例子中是 Shape）。这样你的代码会更容易写，更容易读，并更便于维护，你的设计也更容易实现、理解和改变。所以“多态”是面向对象编程的基本目标。

但是，假如你碰到了特殊的编程问题，如果你能够知道某个引用得确切类型，就可以使用最简单的方式去解决它，那么此时你又该怎么办呢？例如，假设我们允许用户将某一具体类型的几何形状全都变成紫色，以突出显示它们。通过这种方法，用户就能找出屏幕上所有被突出显示的三角形。或者，你的方法可能被用来旋转列表中的所有图形，但你想跳过圆形，因为对圆形作旋转没有意义。使用 RTTI，你可以查询某个 Shape 引用所指向的对象的确切类型，然后选择或者剔除特例。

Class 对象

要理解 RTTI 在 Java 中是如何工作的，首先必须要知道类型信息在运行期是如何表示的。这项工作是由被称为“Class 对象”的特殊对象完成的，它包含了与类有关的信息。事实上，Class 对象正是被用来创建类的“常规”对象的。

作为程序一部分，每个类都有一个 Class 对象。换言之，每当你编写并且编译了一个新类，就会产生一个 Class 对象（更恰当地说，是被保存在一个同名的.class 文件中）。在运行期，一旦我们想生成这个类的一个对象，运行这个程序的 Java 虚拟机（JVM）首先检查这个类

的 Class 对象是否已经加载。如果尚未加载, JVM 就会根据类名查找.class 文件, 并将其载入。所以 Java 程序并不是一开始执行, 就被完全加载的, 这一点与许多传统语言都不同。

一旦某个类的 Class 对象被载入内存, 它就被用来创建这个类的所有对象。如果这么解释仍然不清楚, 或者你并不相信, 下面的示范程序可以证明我的说法:

```
//: c10:SweetShop.java
// Examination of the way the class loader works.
import com.bruceeckel.simpletest.*;

class Candy {
    static {
        System.out.println("Loading Candy");
    }
}

class Gum {
    static {
        System.out.println("Loading Gum");
    }
}

class Cookie {
    static {
        System.out.println("Loading Cookie");
    }
}

public class SweetShop {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        System.out.println("inside main");
        new Candy();
        System.out.println("After creating Candy");
        try {
            Class.forName("Gum");
        } catch(ClassNotFoundException e) {
            System.out.println("Couldn't find Gum");
        }
        System.out.println("After Class.forName(\"Gum\")");
        new Cookie();
        System.out.println("After creating Cookie");
        monitor.expect(new String[] {
            "inside main",
```

```

        "Loading Candy",
        "After creating Candy",
        "Loading Gum",
        "After Class.forName(\"Gum\")",
        "Loading Cookie",
        "After creating Cookie"
    });
}
} ///:~

```

这里的每个类 Candy, Gum 和 Cookie 中, 都有一个 static 语句, 在类第一次被加载时执行, 这是会有相应的信息打印出来, 告诉我们这个类什么时候被加载了。在 main() 中, 创建对象的代码被置于打印语句之间, 以帮助我们判断加载的时间点。

你可以从输出中看到, Class 对象仅在需要的时候才被加载, static 语句块是在类加载时被执行的。

特别有趣的一行是:

```
Class.forName("Gum");
```

这是 Class 类 (所有 Class 对象都属于这个类型) 的一个 static 成员。Class 对象就和其他对象一样, 我们可以获取并操作它的引用 (这也就是类加载器的工作)。forName() 是取得 Class 对象的引用的一种方法。它是用一个包含目标类的文本名 (注意拼写和大小写) 的 String 作输入参数, 返回的是一个 Class 对象的引用, 上面的代码忽略了返回值。对 forName() 的调用是为了它产生的“副作用”: 如果类 Gum 还没有被加载就加载它。在加载的过程中, Gum 的 static 语句被执行。

在前面的例子里, 如果 Class.forName() 找不到你要加载的类, 它会抛出异常 ClassNotFoundException (理想状况下, 异常的名字就能告诉你出了什么问题)。这里我们只需简单报告问题, 但在更严密的程序里, 你可能要在异常处理程序中解决这个问题。

类字面常量 (Class literal)

Java 还提供了另一种方法来生成 Class 对象的引用: 使用“类字面常量 (class literal)”。对上述程序来说, 看起来就象下面这样:

```
Gum.class;
```

这样做不仅更简单, 而且更安全, 因为它在编译期就会受到检查。并且它无需方法调用, 所以也更高效。

类字面常量不仅可以应用于普通的类，也可以应用于接口、数组以及基本数据类型。以外，对于基本数据类型的包装类，还有一个标准域 `TYPE`。`TYPE` 域是一个引用，指向对应的基本数据类型的 `Class` 对象，如下所示：

... 等价于 ...	
<code>boolean.class</code>	<code>Boolean.TYPE</code>
<code>char.class</code>	<code>Character.TYPE</code>
<code>byte.class</code>	<code>Byte.TYPE</code>
<code>short.class</code>	<code>Short.TYPE</code>
<code>int.class</code>	<code>Integer.TYPE</code>
<code>long.class</code>	<code>Long.TYPE</code>
<code>float.class</code>	<code>Float.TYPE</code>
<code>double.class</code>	<code>Double.TYPE</code>
<code>void.class</code>	<code>Void.TYPE</code>

我建议使用“`.class`”的形式，以保持它们与常规类的一致性。

类型转换前先作检查

迄今为止，我们已知的 RTTI 形式包括：

1. 经典的类型转换，如“`(Shape)`”，由 RTTI 确保类型转换的正确性，如果你执行了一个错误的类型转换，就会抛出一个 `ClassCastException` 异常。
2. 代表对象类型的 `Class` 对象。通过查询 `Class` 对象可以获取运行期所需的信息。

在 C++ 中，经典的类型转换“`(Shape)`”并不使用 RTTI。它只是简单地告诉编译器将这个对象作为新的类型对待。而 Java 要执行类型检查，这通常被称为“类型安全的向下转型（`type safe downcast`）”。之所以叫“向下转型”，是由于类层次结构图从来就是这么排列的。如果将 `Circle` 类型转换为 `Shape` 被称作向上转型，那将 `Shape` 转型为 `Circle`，就被称为向下转型。你知道 `Circle` 肯定是一个 `Shape`，所以编译器允许自由地做向上转型的赋值操作；然而，你却不能保证一个 `Shape` 肯定是 `Circle`，因此，如果不显式做类型转换，编译器是不会自动做向下转型操作的。

RTTI 在 Java 中还有第三种形式，就是关键字 `instanceof`。它返回一个布尔值，告诉我们对象是不是某个特定类型的实例。你可以用提问的方式使用它，就象这样：

```
if(x instanceof Dog)
    ((Dog)x).bark();
```

在将 `x` 转型成一个 `Dog` 前，上面的 `if` 语句会检查对象 `x` 是否从属于 `Dog` 类。进行向下转型前，如果没有其他信息可以告诉你这个对象是什么类型，那么使用 `instanceof` 是非常重要的，否则会得到一个 `ClassCastException` 异常。

通常，你可能要查找一种类型（比如要找三角形，并填充成紫色），这时你可以轻松的使用 `instanceof` 来查找。假设你有一个 `Pet` 类的继承体系：

```
//: c10:Pet.java
package c10;
public class Pet {} ///:~

//: c10:Dog.java
package c10;
public class Dog extends Pet {} ///:~

//: c10:Pug.java
package c10;
public class Pug extends Dog {} ///:~

//: c10:Cat.java
package c10;
public class Cat extends Pet {} ///:~

//: c10:Rodent.java
package c10;
public class Rodent extends Pet {} ///:~

//: c10:Gerbil.java
package c10;
public class Gerbil extends Rodent {} ///:~

//: c10:Hamster.java
package c10;
public class Hamster extends Rodent {} ///:~
```

接下来的例子中，我们想跟踪每种特定类型的 `Pet` 的数量，因此我们需要一个类用来将这个数量存储为一个 `Int` 类型，你可以把它看作是一个可修改的 `Integer` 对象：

```
//: c10:Counter.java
package c10;

public class Counter {
    int i;
    public String toString() { return Integer.toString(i); }
```

```
} ///:~
```

然后我们需要一个工具，以同时保存两件东西：一个是 **Pet** 具体类型的指示器，另一个是当前 **pet** 具体类型的对象数量的计数器。也就是说，我们希望能够表达：“这里有多少个 **Gerbil** 对象”。普通的数组在这里是不行的，因为在数组里我们只能通过索引来查询对象。而这里我们需要通过具体的 **Pet** 类型名来查找对象。还要把计数器对象与表示 **Pet** 具体类型的对象关联起来。我们将使用一种标准的数据结构，“关联型数组 (associative array)”。下面是一个非常简单的版本：

```
//: c10:AssociativeArray.java
// Associates keys with values.
package c10;
import com.bruceeckel.simpletest.*;

public class AssociativeArray {
    private static Test monitor = new Test();
    private Object[] [] pairs;
    private int index;
    public AssociativeArray(int length) {
        pairs = new Object[length][2];
    }
    public void put(Object key, Object value) {
        if(index >= pairs.length)
            throw new ArrayIndexOutOfBoundsException();
        pairs[index++] = new Object[] { key, value };
    }
    public Object get(Object key) {
        for(int i = 0; i < index; i++)
            if(key.equals(pairs[i][0]))
                return pairs[i][1];
        throw new RuntimeException("Failed to find key");
    }
    public String toString() {
        String result = "";
        for(int i = 0; i < index; i++) {
            result += pairs[i][0] + " : " + pairs[i][1];
            if(i < index - 1) result += "\n";
        }
        return result;
    }
    public static void main(String[] args) {
        AssociativeArray map = new AssociativeArray(6);
        map.put("sky", "blue");
        map.put("grass", "green");
    }
}
```

```

        map.put("ocean", "dancing");
        map.put("tree", "tall");
        map.put("earth", "brown");
        map.put("sun", "warm");
        try {
            map.put("extra", "object"); // Past the end
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Too many objects!");
        }
        System.out.println(map);
        System.out.println(map.get("ocean"));
        monitor.expect(new String[] {
            "Too many objects!",
            "sky : blue",
            "grass : green",
            "ocean : dancing",
            "tree : tall",
            "earth : brown",
            "sun : warm",
            "dancing"
        });
    }
} ///:~

```

首先你可能会发现，关联型数组看起来像是个通用型的工具，为什么不把它放入 `com.bruceeckel.tools` 包中呢？确实，它是一个很有用的通用型的工具。实际上，`java.util` 中包含了好几个关联型数组（也称作映射 `map`），它们可比我们用的这个数组功能还要强大得多，并且执行速度更快。第十一章中将用很大的篇幅来讲解关联型数组，它们相当复杂。所以我们只使用这个简单的版本，同时也使你熟悉关联型数组的用处。

在一个关联型数组中，索引也称作“键（key）”，关联的对象称作“值（value）”。我们通过把键与值作为“键值对”存入一个二元素数组构成的二维数组中，来将它们关联在一起，这也就是你在程序中看到的 `pairs`。这是一个固定长度的数组，所以我们使用 `index` 以保证不会越界。当你用 `put()` 方法添加键值对时，就生成一个新的只有两个元素的数组，并添加在 `pairs` 中的下一个可用的位置。如果 `index` 大于或等于 `pairs` 的长度，就会抛出异常。

使用 `get()` 方法，你只需将要查找的键传给它，它返回与键相关联的值作为结果，或者在没有找到相关联的值的条件下抛出异常。`get()` 所使用的定位值的方法可能是你所能想象得到的最低效的一种：从数组顶端开始，依次调用 `equals()` 来比较键。但现在我们所关心的是简单性，而不是高效性，而且第十一章中的真正的映射（`map`）已经解决了执行效率的问题，所以现在不需要为此而担心。

关联型数组中基本的方法就是 `put()` 和 `get()`，但是为了便于显示，`toString()` 被重载

用来打印键值对。为了表明它是如何工作的，main() 先向 AssociativeArray 中加载一些键值对，然后把映射打印出来，最后还使用了 get() 获取一个特定值。

现在所有的工具都备齐了，我们能使用 instanceof 来给 Pet 计数了。

```
//: c10:PetCount.java
// Using instanceof.
package c10;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class PetCount {
    private static Test monitor = new Test();
    private static Random rand = new Random();
    static String[] typenames = {
        "Pet", "Dog", "Pug", "Cat",
        "Rodent", "Gerbil", "Hamster",
    };
    // Exceptions thrown to console:
    public static void main(String[] args) {
        Object[] pets = new Object[15];
        try {
            Class[] petTypes = {
                Class.forName("c10.Dog"),
                Class.forName("c10.Pug"),
                Class.forName("c10.Cat"),
                Class.forName("c10.Rodent"),
                Class.forName("c10.Gerbil"),
                Class.forName("c10.Hamster"),
            };
            for(int i = 0; i < pets.length; i++)
                pets[i] = petTypes[rand.nextInt(petTypes.length)]
                    .newInstance();
        } catch(InstantiationException e) {
            System.out.println("Cannot instantiate");
            System.exit(1);
        } catch(IllegalAccessException e) {
            System.out.println("Cannot access");
            System.exit(1);
        } catch(ClassNotFoundException e) {
            System.out.println("Cannot find class");
            System.exit(1);
        }
        AssociativeArray map =
```

```

        new AssociativeArray(typhenames.length);
for(int i = 0; i < typhenames.length; i++)
    map.put(typhenames[i], new Counter());
for(int i = 0; i < pets.length; i++) {
    Object o = pets[i];
    if(o instanceof Pet)
        ((Counter)map.get("Pet")).i++;
    if(o instanceof Dog)
        ((Counter)map.get("Dog")).i++;
    if(o instanceof Pug)
        ((Counter)map.get("Pug")).i++;
    if(o instanceof Cat)
        ((Counter)map.get("Cat")).i++;
    if(o instanceof Rodent)
        ((Counter)map.get("Rodent")).i++;
    if(o instanceof Gerbil)
        ((Counter)map.get("Gerbil")).i++;
    if(o instanceof Hamster)
        ((Counter)map.get("Hamster")).i++;
}
// List each individual pet:
for(int i = 0; i < pets.length; i++)
    System.out.println(pets[i].getClass());
// Show the counts:
System.out.println(map);
monitor.expect(new Object[] {
    new TestExpression("% class c10\\. "+
        "(Dog|Pug|Cat|Rodent|Gerbil|Hamster)",
        pets.length),
    new TestExpression(
        "% (Pet|Dog|Pug|Cat|Rodent|Gerbil|Hamster)" +
        " : \\d+", typhenames.length)
});
}
} ///:~

```

在 main() 中，用 Class.forName() 创建了一个名为 petTypes 的 Class 对象的数组。由于 Pet 对象属于 c10 这个包 (package)，因此命名的时候要把 package 的名字也包括进去。

接下来是填充 pets 数组。从 petTypes 中随机选择 Class 对象，并通过 Class.newInstance() 调用缺省的（无参数）构造器生成新的对象，加入到 pets 数组中。

`forName()` 与 `newInstance()` 都可能会抛出异常，你可以在 `try` 语句块后紧跟着的 `catch` 子句中看到是如何处理这些异常的。记住，异常的名字对发生的错误是相当有用的解释（例如 `IllegalAccessException` 表示违背了 Java 的安全机制）。

在创建了 `AssociativeArray` 之后，它就被填充满了由 `pet` 名字和数量组成的键值对。在这个随机生成的数组中每个 `Pet` 都是通过使用 `instanceof` 来检测并计数的。这个数组和 `AssociativeArray` 都被打印了出来，这样你就可以比较结果了。

对 `instanceof` 有比较严格的限制：你只可将其与类型的名字进行比较，而不能与 `Class` 对象作比较。在前面的例子中，你可能觉得写出那一堆 `instanceof` 表达式是很乏味的，的确如此。但是你也无法让 `instanceof` 聪明起来，能够自动地创建一个 `Class` 对象的数组，然后将目标对象与这个数组中的对象进行逐一的比较。（稍后你会看到一个替代方案）。其实这并非十分严格的限制，渐渐地你就会理解，如果程序中编写了许多的 `instanceof` 表达式，就说明你的设计可能存在瑕疵。

这个例子当然是刻意制作的，真要跟踪计数，最可能的做法是在每个类里添加一个 `static` 域，在构造器中逐渐累加此域。如果你掌握了类的源代码，并且能够改变它，你可能就会这样做。但实际情况并非总是这样，这时就需要使用 **RTTI**。

使用类字面常量

学习如何使用类字面常量重写 `PetCount.java` 示例是很有趣的。改写后的结果在各方面都显得更加清晰：

```
//: c10:PetCount2.java
// Using class literals.
package c10;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class PetCount2 {
    private static Test monitor = new Test();
    private static Random rand = new Random();
    public static void main(String[] args) {
        Object[] pets = new Object[15];
        Class[] petTypes = {
            // Class literals:
            Pet.class,
            Dog.class,
            Pug.class,
            Cat.class,
            Rodent.class,
            Gerbil.class,
        }
    }
}
```

```

        Hamster.class,
    };
    try {
        for(int i = 0; i < pets.length; i++) {
            // Offset by one to eliminate Pet.class:
            int rnd = 1 + rand.nextInt(petTypes.length - 1);
            pets[i] = petTypes[rnd].newInstance();
        }
    } catch(InstantiationException e) {
        System.out.println("Cannot instantiate");
        System.exit(1);
    } catch(IllegalAccessException e) {
        System.out.println("Cannot access");
        System.exit(1);
    }
    AssociativeArray map =
        new AssociativeArray(petTypes.length);
    for(int i = 0; i < petTypes.length; i++)
        map.put(petTypes[i].toString(), new Counter());
    for(int i = 0; i < pets.length; i++) {
        Object o = pets[i];
        if(o instanceof Pet)
            ((Counter)map.get("class c10.Pet")).i++;
        if(o instanceof Dog)
            ((Counter)map.get("class c10.Dog")).i++;
        if(o instanceof Pug)
            ((Counter)map.get("class c10.Pug")).i++;
        if(o instanceof Cat)
            ((Counter)map.get("class c10.Cat")).i++;
        if(o instanceof Rodent)
            ((Counter)map.get("class c10.Rodent")).i++;
        if(o instanceof Gerbil)
            ((Counter)map.get("class c10.Gerbil")).i++;
        if(o instanceof Hamster)
            ((Counter)map.get("class c10.Hamster")).i++;
    }
    // List each individual pet:
    for(int i = 0; i < pets.length; i++)
        System.out.println(pets[i].getClass());
    // Show the counts:
    System.out.println(map);
    monitor.expect(new Object[] {
        new TestExpression("% class c10\\" +
            "(Dog|Pug|Cat|Rodent|Gerbil|Hamster)",

```

```

        pets.length),
    new TestExpression("% class c10\\. " +
        "(Pet|Dog|Pug|Cat|Rodent|Gerbil|Hamster) : \\d+",
        petTypes.length)
    });
}
} ///:~

```

在这个例子中，`typenames`（类型名）数组已被移除，改为从 `Class` 对象获取类型名称。注意，系统能够区分类和接口的不同。

你同时也可以看到，生成 `petTypes` 的代码不需要放在 `try` 语句块内，因为它会在编译期得到检查，因此，它不会抛出任何异常，这与 `Class.forName()` 不一样。

`Pet` 对象被动态创建以后，可以看到随机数字已经被限制在 1 至 `petTypes.length` 但不包括零的范围内。因为零代表 `Pet.class`，而通用的 `Pet` 对象可能不会有人感兴趣。然而，由于 `Pet.class` 也是 `petTypes` 的一部分，因此所有 `Pet` 都被计算在内。

动态的 instanceof

`Class.isInstance` 方法提供了一种动态地调用 `instanceof` 运算符的途径。于是所有那些单调的 `instanceof` 语句都可以从 `PetCount` 例子中移除了。如下所示：

```

//: c10:PetCount3.java
// Using isInstance()
package c10;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class PetCount3 {
    private static Test monitor = new Test();
    private static Random rand = new Random();
    public static void main(String[] args) {
        Object[] pets = new Object[15];
        Class[] petTypes = {
            // Class literals:
            Pet.class,
            Dog.class,
            Pug.class,
            Cat.class,
            Rodent.class,
            Gerbil.class,
            Hamster.class,

```

```

};
try {
    for(int i = 0; i < pets.length; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + rand.nextInt(petTypes.length - 1);
        pets[i] = petTypes[rnd].newInstance();
    }
} catch(InstantiationException e) {
    System.out.println("Cannot instantiate");
    System.exit(1);
} catch(IllegalAccessException e) {
    System.out.println("Cannot access");
    System.exit(1);
}
}
AssociativeArray map =
    new AssociativeArray(petTypes.length);
for(int i = 0; i < petTypes.length; i++)
    map.put(petTypes[i].toString(), new Counter());
for(int i = 0; i < pets.length; i++) {
    Object o = pets[i];
    // Using Class.isInstance() to eliminate
    // individual instanceof expressions:
    for(int j = 0; j < petTypes.length; ++j)
        if(petTypes[j].isInstance(o))
            ((Counter)map.get(petTypes[j].toString())).i++;
}
// List each individual pet:
for(int i = 0; i < pets.length; i++)
    System.out.println(pets[i].getClass());
// Show the counts:
System.out.println(map);
monitor.expect(new Object[] {
    new TestExpression("% class c10\\\" +
        "(Dog|Pug|Cat|Rodent|Gerbil|Hamster)",
        pets.length),
    new TestExpression("% class c10\\\" +
        "(Pet|Dog|Pug|Cat|Rodent|Gerbil|Hamster) : \\d+",
        petTypes.length)
});
}
} ///:~

```

可以看到 `isInstance()` 方法使我们不再需要 `instanceof` 表达式。此外，这意味着如果要求添加新类型的宠物，只需简单地改变 `petTypes` 数组即可；而毋需改动程序其他的部

分（但是在使用 instanceof 时这却是必需的）。

等价性: instanceof vs. Class

在查询类型信息时，以 instanceof 的形式（或者是以 isInstance() 的形式，它们产生相同的结果）与直接比较 Class 对象有一个很重要的差别。下面的例子向你展示了这种差别：

```
//: c10:FamilyVsExactType.java
// The difference between instanceof and class
package c10;
import com.bruceeckel.simpletest.*;

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
    private static Test monitor = new Test();
    static void test(Object x) {
        System.out.println("Testing x of type " +
            x.getClass());
        System.out.println("x instanceof Base " +
            (x instanceof Base));
        System.out.println("x instanceof Derived " +
            (x instanceof Derived));
        System.out.println("Base.isInstance(x) " +
            Base.class.isInstance(x));
        System.out.println("Derived.isInstance(x) " +
            Derived.class.isInstance(x));
        System.out.println("x.getClass() == Base.class " +
            (x.getClass() == Base.class));
        System.out.println("x.getClass() == Derived.class " +
            (x.getClass() == Derived.class));
        System.out.println("x.getClass().equals(Base.class) " +
            (x.getClass().equals(Base.class)));
        System.out.println(
            "x.getClass().equals(Derived.class) " +
            (x.getClass().equals(Derived.class)));
    }
    public static void main(String[] args) {
        test(new Base());
        test(new Derived());
        monitor.expect(new String[] {
```

```

        "Testing x of type class c10.Base",
        "x instanceof Base true",
        "x instanceof Derived false",
        "Base.isInstance(x) true",
        "Derived.isInstance(x) false",
        "x.getClass() == Base.class true",
        "x.getClass() == Derived.class false",
        "x.getClass().equals(Base.class) true",
        "x.getClass().equals(Derived.class) false",
        "Testing x of type class c10.Derived",
        "x instanceof Base true",
        "x instanceof Derived true",
        "Base.isInstance(x) true",
        "Derived.isInstance(x) true",
        "x.getClass() == Base.class false",
        "x.getClass() == Derived.class true",
        "x.getClass().equals(Base.class) false",
        "x.getClass().equals(Derived.class) true"
    });
}
} ///:~

```

`test()` 方法使用了两种形式的 `instanceof` 作为参数，用以执行类型检查。然后获取 `Class` 的引用，并用 `==` 和 `equals()` 来检查 `Class` 对象是否相等。使人放心的是，`instanceof` 和 `isInstance()` 生成的结果完全一样，`equals()` 和 `==` 也一样。但是这两组测试得出的结论却不相同。`instanceof` 保持了类型的概念，它指的是“你是这个类吗，或者你是这个类的派生类吗？”而另一种情况是，如果你用 `==` 比较实际的 `Class` 对象，就不包含继承关系，——它或者恰好是这个确切的类型，或者不是。

RTTI 语法

Java 是通过 `Class` 对象来实现 RTTI 机制的，即使我们只是要做些诸如类型转换这类的事情。`Class` 类也提供了许多其他途径，以方便我们使用 RTTI。

首先，你需要获得指向适当的 `Class` 对象的引用。一种办法是用字符串以及 `Class.forName()` 方法，就象前例演示的那样。这种做法很方便，因为你在获取 `Class` 的引用事，并不需要生成该 `Class` 类型的对象。然而，如果你已经有了一个你感兴趣的类型的对象，那么你就可以通过调用 `getClass()` 来获取 `Class` 的引用，这是根类 `Object` 提供的方法。它返回 `Class` 的引用，用来表示对象的实际类型。`Class` 提供了一些有趣的方法，下面的例子为你展示这些方法：

```

//: c10:ToyTest.java
// Testing class Class.

```

```

import com.bruceeckel.simpletest.*;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}
class Toy {
    // Comment out the following default constructor
    // to see NoSuchMethodError from (*1*)
    Toy() {}
    Toy(int i) {}
}

class FancyToy extends Toy
implements HasBatteries, Waterproof, Shoots {
    FancyToy() { super(1); }
}

public class ToyTest {
    private static Test monitor = new Test();
    static void printInfo(Class cc) {
        System.out.println("Class name: " + cc.getName() +
            " is interface? [" + cc.isInterface() + "]");
    }
    public static void main(String[] args) {
        Class c = null;
        try {
            c = Class.forName("FancyToy");
        } catch(ClassNotFoundException e) {
            System.out.println("Can't find FancyToy");
            System.exit(1);
        }
        printInfo(c);
        Class[] faces = c.getInterfaces();
        for(int i = 0; i < faces.length; i++)
            printInfo(faces[i]);
        Class cy = c.getSuperclass();
        Object o = null;
        try {
            // Requires default constructor:
            o = cy.newInstance(); // (*1*)
        } catch(InstantiationException e) {
            System.out.println("Cannot instantiate");
            System.exit(1);
        } catch(IllegalAccessException e) {

```

```

        System.out.println("Cannot access");
        System.exit(1);
    }
    printInfo(o.getClass());
    monitor.expect(new String[] {
        "Class name: FancyToy is interface? [false]",
        "Class name: HasBatteries is interface? [true]",
        "Class name: Waterproof is interface? [true]",
        "Class name: Shoots is interface? [true]",
        "Class name: Toy is interface? [false]"
    });
}
} ///:~

```

从中可以看出，FancyToy 类相当复杂，因为它不但继承了 Toy 类，而且还实现了 HasBatteries, Waterproof 以及 Shoots 接口。在 main() 方法中，通过在适当的 try 语句块中使用 forName(), 创建并初始化了一个指向 FancyToy Class 的引用。

Class.getInterfaces() 方法返回 Class 对象的数组，这些对象代表的是某个 Class 对象所包含的接口。

如果你有一个 Class 对象，那么你就可以通过 getSuperclass() 获取它的直接基类。这个方法自然也是返回一个 Class 的引用，所以你可以进一步查询其基类。这意味着在运行期，你可以找到一个对象完整的类层次结构。

乍看起来，Class 的 newInstance() 方法似乎只是另一种克隆 clone() 对象的方法。然而，即使原先没有任何对象存在，你也可以用 newInstance() 创建一个新的对象，就像在上例中并没有任何 Toy 对象，只有 y 的 Class 对象的引用 cy。这是一种实现“虚拟构造器 (virtual constructor)”的途径，这使得你可以说：“尽管我不知道你的准确类型是什么，但无论如何，还是请正确地创建你自己。”在上面的例子中，cy 只是一个 Class 的引用，在编译期并不知道更多的类型信息。当你创建一个新实例时，你将得到一个 Object 引用，但这个引用实际指向的是一个 Toy 对象。所以如果向 Object 发送它不能接收的消息，你必须先对其进行深入了解，进行类型转换。以外，使用 newInstance() 的类必须要有一个缺省构造器。在下一节，通过使用 Java 反射 API (Java reflection API)，你会看到如何使用任意的构造器来动态地创建对象。

最后一个要介绍的方法是 printInfo(), 它以一个 Class 引用为参数，通过 getName() 获取其名字，并通过 isInterface() 查看它是否是一个接口。因此，通过使用 Class 对象，我们可以找出一个对象的任何信息。

反射 (Reflection): 运行期的类信息

如果你不知道某个对象的确切类型，RTTI 可以告诉你。但是有一个限制：这个类型在编译

期必须已知，才能使用 RTTI 识别它，并利用这些信息做一些有用的事。换句话说，在编译期，编译器必须知道你要通过 RTTI 来处理的所有类。

初看起来这似乎不是个限制，但是假设你获取了一个指向某个并不在你的程序空间中的对象的引用。事实上，在编译期你的程序根本没法获知这个对象所属的类。例如，假设你从磁盘文件，或者网络连接中获取了一串字节，并且你被告知这些字节代表了一个类。可是编译器在编译你的程序代码的时候不可能了解有关这个在后面才会出现的类的信息。你怎样才能使用这样的类呢？

在传统的编程环境中不太可能出现这种情况。但当我们置身于更大规模的编程世界中，在许多重要情况下就会发生上面的事情。首先就是“基于构件的编程（component-based programming）”，在此种编程方式中，你将使用某种基于“快速应用开发（RAD, Rapid Application Development）”的应用构建工具来构建项目。这是一种可视化编程方法，你可以通过将代表不同组件的图标拖曳到表单中来创建程序（在屏幕上看到的“表单（form）”就是你所创建的程序）。然后在编程时通过设置构件的属性值来配置它们。这种设计期的配置，要求构件都是可实例化的，并且要暴露其部分信息，以允许程序员读取和设置构件的值。此外，处理 GUI 事件的构件还必须暴露相关信息，以便 RAD 环境帮助程序员重载这些处理事件的方法。反射提供了一种机制，用来检查可用的方法，并返回方法名。Java 通过 `JavaBean`（第十四章将详细介绍）提供了基于构件的编程架构。

另一个让人们想要在运行期获取类的信息的动机，便是希望提供在跨网络的远程平台上创建和运行对象的能力。这被称为“远程方法调用（RMI, Remote Method Invocation）”，它允许一个 Java 程序将对象分布到多台机器上。需要这种分布能力是有许多原因的，例如，你可能正在执行一项计算密集型的任务，为了提高运算速度，你想将计算划分为许多小的计算单元，分布到空闲的机器上运行。另一种情况是，你可能希望将处理特定类型任务的代码（例如多层的 C/S 客户服务器架构中的“业务规则 Business Rules”），置于特定的机器上，于是这台机器就成为了描述这些动作的公共场所，你可以很容易地通过改动它就达到影响系统中所有人的效果（这是一种有趣的开发方式，因为机器的存在仅仅是为了方便软件的改动！）。同时，分布式计算也支持执行特殊任务的专有硬件，例如矩阵转置，而这对于通用型程序就显得不太合适或者太昂贵了。

`Class` 类（本章前面已有论述）支持反射的概念，Java 附带的库 `java.lang.reflect` 包含了 `Field`、`Method` 以及 `Constructor` 类（每个类都实现了 `Member` 接口）。这些类型的对象是由 JVM 在运行期创建的，用以表示未知类里对应的成员。这样你就可以使用 `Constructor` 创建新的对象，用 `get()` 和 `set()` 方法读取和修改与 `Field` 对象关联的属性，用 `invoke()` 方法调用与 `Method` 对象关联的方法。另外，你还可以调用 `getFields()`、`getMethods()`、`getConstructors()` 等等很便利的方法，以返回表示属性、方法以及构造器的对象数组，这些对象（在 JDK 文档中，可找到与 `Class` 类相关的更多的资料）。这样，匿名对象的类信息就能在运行期被完全确定下来，而在编译期不需要知道任何事情。

重要的是，反射机制并没有什么魔法。当你通过反射与一个未知类型的对象打交道时，JVM 只是简单地检查这个对象，看它属于哪个特定的类（就象 RTTI 那样）。但在这之后，在做其它事情之前，必须加载那个类的 `Class` 对象。因此，那个类的 `.class` 文件对于 JVM 来说必须是可获取的，要么在本地机器上，要么可以通过网络取得。所以 RTTI 和反射之间真

正的区别只在于，对 RTTI 来说，编译器在编译期打开和检查.class 文件。（换句话说，我们可以用“普通”方式调用一个对象的所有方法。）而对于反射机制来说.class 文件在编译期间是不可获取的，所以是在运行期打开和检查.class 文件。

类方法提取器

你很少直接使用反射机制；它在Java中是用来支持其他特性的，例如第十二章的对象序列化(serialization),第十四章的JavaBean。但是有的时候如果能动态地提取某个类的信息还是很有用的。类方法提取器就是一个非常有用的工具。前面已经说过，阅读实现了类定义的源代码或是其JDK文档，你只能找到在这个类定义中被定义或被重载的方法。但对你来说可能有数十个更有用的方法是继承自基类的。要找出这些方法可能是很乏味而且耗费时间的¹。幸运的是，反射机制提供了一种方法，使我们能够编写可以自动展示完整接口的简单工具。下面就是其工作方式：

```
//: c10:ShowMethods.java
// Using reflection to show all the methods of a class,
// even if the methods are defined in the base class.
// {Args: ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;

public class ShowMethods {
    private static final String usage =
        "usage: \n" +
        "ShowMethods qualified.class.name\n" +
        "To show all methods in class or: \n" +
        "ShowMethods qualified.class.name word\n" +
        "To search for methods involving 'word'";
    private static Pattern p = Pattern.compile("\\w+\\.");
    public static void main(String[] args) {
        if(args.length < 1) {
            System.out.println(usage);
            System.exit(0);
        }
        int lines = 0;
        try {
            Class c = Class.forName(args[0]);
            Method[] m = c.getMethods();
            Constructor[] ctor = c.getConstructors();
            if(args.length == 1) {
                for(int i = 0; i < m.length; i++)
```

¹特别是在过去，现在Sun改进了HTML Java 文档，所以查找基类的方法已经简单多了。

```

        System.out.println(
            p.matcher(m[i].toString()).replaceAll("");
    for(int i = 0; i < ctor.length; i++)
        System.out.println(
            p.matcher(ctor[i].toString()).replaceAll("");
    lines = m.length + ctor.length;
} else {
    for(int i = 0; i < m.length; i++)
        if(m[i].toString().indexOf(args[1]) != -1) {
            System.out.println(
                p.matcher(m[i].toString()).replaceAll("");
            lines++;
        }
    for(int i = 0; i < ctor.length; i++)
        if(ctor[i].toString().indexOf(args[1]) != -1) {
            System.out.println(p.matcher(
                ctor[i].toString()).replaceAll("");
            lines++;
        }
    }
} catch(ClassNotFoundException e) {
    System.out.println("No such class: " + e);
}
}
} ///:~

```

Class 的 `getMethods()` 和 `getConstructors()` 方法分别返回 `Method` 数组和 `Constructor` 数组。这两个类都提供了深层方法，用以解析其对象所代表的方法，并获取其名字、输入参数以及返回值。但你也可以象这里一样，只使用 `toString()` 生成一个含有完整方法签名的字符串。代码其他部分用于提取命令行信息，判断某个特定的签名是否与我们的目标字符串相符（使用 `indexOf()`），并去掉了命名修饰词。

为了从“`Java.lang.String`”中去掉“`java.lang.`”这样的命名修饰词，可以使用 Java JDK 1.4 提供的强大而简洁的工具：正则表达式（regular expression），这在某些语言中数年前就已经有了。你已经在 `com.bruceeckel.simpletest.Test` 类中的 `expect()` 语句中看到了正则表达式的简单用法。在前面的例子中，已经展示了在你自己的程序中使用正则表达式所必需的基本编码步骤。

在导入 `java.util.regex` 这个库之后，你要使用 `static Pattern.compile()` 方法先编译正则表达式，它根据输入的字符串参数生成一个 `Pattern` 对象。在本例中，输入的参数是：

```
"\\w+\\. "
```

想要理解这个以及其他的正则表达式，你可以查看 JDK 文档中的 `java.util.regex.Pattern`。这里的 `'\w'` 代表“一个构成词的字符：`[a-zA-Z_0-9]`”。`'+'` 代表“一个或多个前述的表达式”，所以这个例子表示，一个或多个构成词的字符。`'\.'` 产生一个字面常量的点号（不能直接用点号，那在正则表达式中代表“任何字符”）。所以这个表达式会匹配任何“由一个或多个字符构成的单词，并且其后紧跟一个点号”，这正是我们要去剔除的修饰词。

在你编译了一个 `Pattern` 对象之后，可以通过调用 `matcher()` 方法使用它，只需把你想要查找的字符串传递给它。`matcher()` 方法生成一个 `Matcher` 对象，这个对象包含一组用来从目标对象中做选择的操作（你可以在 JDK 中看到与 `java.util.regex.Matcher` 有关的所有内容）。这里，`replaceAll()` 方法用来将所有的匹配部分用空字符串替代掉，也就是说，删除所有的匹配部分。

要想写得更简洁紧凑，你可以在 `String` 类中嵌入使用正则表达式。例如，在前面的程序中最后用到的 `replaceAll()` 可以从下面的形式：

```
p.matcher(ctor[i].toString()).replaceAll("")
```

改写为：

```
ctor[i].toString().replaceAll("\\w+\\.","")
```

这是不需要预编译的形式。这种形式适合正则表达式只使用一次的情况，如果你需要多次重复使用某个正则表达式的话，就像上例中所展示的情形，预编译形式的效率明显更高。

这个例子展示了反射机制的用法。由于 `Class.forName()` 生成的对象在编译期是不可知的，因此所有的方法签名信息都是在执行期被提取出来的。如果你研究一下 JDK 文档关于反射的部分，你会看到，反射机制提供了足够的支持，使得你能够创建一个在编译期完全未知的对象，并调用此对象的方法。（在本书后面会有示例）虽然开始的时候，你可能认为永远也不需要用到这些功能，但是反射机制的价值会令你非常吃惊的。

运行下面这个演示性的示例：

```
java ShowMethods ShowMethods
```

它将生成一份列表，其中包含了 `public` 的缺省构造器，即便你可能看到在代码中根本没有定义任何构造器，在这个列表中也会包含一个 `public` 的缺省构造器。你看到的这个包含在列表中的构造器是编译器自动生成的。如果你将 `ShowMethods` 作为一个非 `public` 的类（也就是 `friendly`），就不会再显示出这个自动生成的构造器了。这样的构造器会自动被赋予与类一样的访问权限。

还有一个有趣的例子，用一个额外的 `char`, `int` 或 `String` 参数来调用 `java ShowMethods java.lang.String`。

在编程时，特别是如果你不记得一个类是否有某个的方法，或者你不知道一个类究竟能做什么，例如 `Color` 对象，而你又不想通过索引或类的层次结构去查找 JDK 文档，这时这个工具确实能为你节省很多时间。

第十四章包含这个程序的 GUI 版本（专为 Swing 构件定制的），使你在编写代码的同时能够通过运行它来快速查询有用的信息。

总结

RTTI 允许你通过匿名基类的引用来发现类型信息。初学者极易误用它，因为在学会使用多态调用方法之前，这么做也很有效。对许多有过程化编程背景的人来说，很难让他们不把程序组织成一系列 `switch` 语句。他们可能会用 RTTI 做到这一点，但是这样就在代码开发和维护过程中损失了多态机制的重要价值。Java 希望我们始终使用多态机制，只在必需的时候使用 RTTI。

然而使用多态机制的方法调用，要求我们拥有基类定义的控制权，因为在你扩展程序的时候，可能会发现基类并未包含我们想要的方法。如果基类来自一个库，或者由别人控制，这时候 RTTI 便是一种解决之道：可继承一个新类，然后添加你需要的方法。在代码的其他地方，你可以识别自己特定的类，并调用你自己的方法。这样做不会破坏多态性以及程序的扩展能力，因为这样添加一个新的类并不需要你在程序中搜索 `switch` 语句。但如果你在程序主体中添加你需要的新特性的代码，就必须使用 RTTI 来检查你的特定的类。

如果只是为了某一个特定类的利益，而将某个功能放进基类里，这意味着从那个基类派生出的所有其他子类都带有一些无意义的东西。这会使得接口更不清晰，因为我们必须重载由基类继承而来的所有抽象方法，这是很恼人的。例如，考虑一个表示乐器 `Instrument` 的类层次结构。假设我们想清洁管弦乐队中某些乐器残留的口水。一种办法是在基类 `Instrument` 中放入 `ClearSpitValve()` 方法。但这样做会造成混淆，因为它意味着打击乐器 `Percussion` 和电子乐器 `Electronic` 也需要清除口水。在这个例子中，RTTI 可以提供了一种更合理的解决方案。你可以将 `clearSpitValve()` 至于适当的特定类中，在这里，是 `Wind`（管乐器）。更恰当的解决方法是将 `prepareInstrument()` 置于基类中，但是初次面对这个问题时你可能看不出来，而误认为你必须使用 RTTI。

最后一点，RTTI 有时能解决效率问题。也许你的程序漂亮地运用了多态，但其中某个对象是以极端缺乏效率的方式到达这个目的的。你可以挑出这个类，使用 RTTI，并且为其编写一段特别的代码以提高效率。然而必须要注意，不要太早地关注程序的效率问题，这是个诱人的陷阱。最好首先让程序运作起来，然后再考虑它的速度，如果要解决效率问题可以使用 `profiler`（见第十五章）。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文

档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 将 Rhomboid (菱形) 加入 Shapes.java 中。创建一个 Rhomboid，将其向上转型为 Shape，然后向下转型回 Rhomboid。试着将其向下转型成 Circle，看看会发生什么。
2. 修改练习 1，让你的程序在执行向下转型之前先运用 instanceof 检查类型。
3. 修改 Shapes.java，使这个程序有能力将所有隶属于特定类的 Shape 对象都“标示”起来(通过设标志)。每一个 Shape 的继承类的 toString 方法应该更够指出它是否被标示。
4. 修改 SweetShop.java，使每种类型对象的创建由命令行参数控制。即，如果你的命令行是“java SweetShop Candy”，那么只有 Candy 对象被创建。注意你是如何通过命令行来控制加载哪个 Class 对象。
5. 将新的 Pet 类型加到 PetCount3.java 中。检查 main() 里面是否正确产生该类型的对象，并正确计算了个数。
6. 写一个方法，令它接受任意对象作为参数，并能够递归打印出该对象所在的继承体系中的所有类。
7. 修改练习 6，让这个使用方法使用 Class.getDeclaredFields() 来打印各个类中的数据成员的相关信息。
8. 请将 ToyTest.java 里头的 Toy 缺省构造器注释掉，并解释发生的现象。
9. 将新的 interface 加到 ToyTest.java 中，看看这个程序是否能够正确检测出来并加以显示。
10. 写一个程序，使它能判断 char 数组究竟是个基本类型，还是一个对象。
11. 实现本章小结中所描述的 clearSpitValve()。
12. 实现本章描述的 rotate(shape) 方法，让它能判断它所旋转的是不是 Circle (如果是，就不执行)。
13. 在 ToyTest.java 中，使用反射机制，通过非缺省构造器创建 Toy 对象。
14. 请在 java.sun.com 提供的 Java HTML 文档中找出 java.lang.Class 的接口。写一个程序，使它能够接受命令行参数所指定的类名称。然后使用 Class 的方法打印该类所有可以获得的信息。用标准程序库的类和你自己的写的类，分别测试这个程序。
15. 修改 ShowMethods.java 中的正则表达式，以去掉 native 和 final 关键字(提示：使用“或”运算符'|')。

第十一章 对象的集合

如果程序只包含固定数量的对象，并且其生命周期都已知，那么这个程序就实在太简单了。

通常，你的程序会根据运行时才知道的条件创建新对象。不到运行期，不会知道所需对象的数量，甚至不知道确切的类型。为解决这个普遍的编程问题，需要能够在任意时刻，任意位置，创建任意数量的对象。所以，你就不能指望创建具名的引用来持有每一个对象：

```
MyObject myReference;
```

因为你不知道实际上会需要多少这样的引用。

大多数语言都提供了某种方法来解决这个基本问题。Java 有多种方式保存对象（应该说是对象的引用 *reference*）。例如前面曾经学习过的数组，它是语言内置的类型。Java 实用类库还提供了一套相当完整的容器类（也称为集合类，但由于 Java 2 的类库中使用了 *Collection* 来指代该类库的一个特殊子集，所以我使用“容器”称呼它们）。容器提供了近乎完美的方式来保存和操纵对象。

数组

第四章最后一节包含了对数组必要的简介，说明了如何定义并初始化一个数组。本章的主题是持有对象，而数组正是保存对象的方式之一。不过持有对象还有许多别的方式，那么数组在其中有什么特别之处呢？

数组与其它种类的容器之间的区别有三方面：效率、类型和持有基本类型的能力。在 Java 中，数组是一种效率最高的存储和随机访问对象引用序列的方式。数组就是个简单的线性序列，这使得元素访问非常快速，但也损失了其他一些特性。当你创建了一个数组对象（将数组本身作为对象看待），数组的大小就被固定了，并且这个数组的生命周期也是不可改变的。通常是创建一个特定大小的数组，在空间不足的时候再创建一个新的数组，然后把旧数组中所有的引用移到新数组中。这正是后面会学到的 *ArrayList* 类的行为方式。然而这种弹性带来的开销，使得 *ArrayList* 比数组效率低。

在 C++ 中，容器类 *vector* 知道自己保存的对象是何类型，不过与 Java 中的数组相比，它有一个缺点：C++ 中 *vector* 的操作符 `[]` 不做边界检查，所以你可能会越界操作。¹ 而在 Java 中，无论你使用数组或容器，都有边界检查。如果越界操作就会得到一个 *RuntimeException* 异常。这类异常通常说明是程序员的错误，因此你不必自己作越界检查。多说一句，为了速度，C++ 的 *vector* 存取访问不作边界检查；而 Java 的数组与容器会因为时刻存在的边界检查带来固定的性能开销。

本章还会学习其它通用的容器类 *List*, *Set*, 和 *Map*，它们不以具体的类型来处理对象。换句话说，它们将所有对象都按 *Object* 类型处理，即 Java 中所有类的基类。从某个角度来

¹ 要问 *vector* 它有多大是可能的，*at()* 方法会做边界检查。

说，这种方式很好：你只需要做一个容器，任意的 Java 对象都可以放入其中。（除了基本类型，可以使用 Java 包装类将其作为常量包装后存入容器，或者用你自己的类将其作为变量包装起来存入容器）这正是数组比通用容器优越的第二点：当你创建一个数组时，它只能保存特定类型的数据（这与第三点相关——数组可以保存基本类型，容器则不能）。这意味着会在编译期做类型检查，以防止你将错误的类型插入数组，或取出数据时弄错类型。当然，无论在编译期还是运行期，Java 都会阻止你向对象发送不恰当的消息。所以并不是说哪种方法更不安全，只是如果编译期就能够指出错误，那么程序可以运行得更快，也减少了程序的使用者被异常吓着的可能性。

考虑到效率与类型检查，应该尽可能使用数组。然而，如果要解决更一般化的问题，数组就太受限制了。在看过数组之后，本章余下的部分专门讨论 Java 提供的容器类。

数组是第一级对象

无论使用哪种数组，数组标识符其实只是一个引用，指向在堆（heap）中创建的一个真实对象，这个（数组）对象用以保存指向其他对象的引用。可以作为数组初始化语法的一部分隐式地创建此对象，或者用 `new` 表达式显式地创建。只读成员 `length` 是数组对象的一部分（事实上，这是唯一一个可以访问的属性或方法），表示此数组对象可以存储多少元素。‘[]’语法是访问数组对象唯一的方式。

下例将演示初始化数组的各种方式，以及如何对指向数组的引用赋值，使之指向另一个数组对象。此例也证明，对象数组和基本类型数组在使用上几乎是同样的。唯一的区别就是对象数组保存的是引用，基本类型数组直接保存基本类型的值。

```
//: c11:ArraySize.java
// Initialization & re-assignment of arrays.
import com.bruceeckel.simpletest.*;

class Weeble {} // A small mythical creature

public class ArraySize {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        // Arrays of objects:
        Weeble[] a; // Local uninitialized variable
        Weeble[] b = new Weeble[5]; // Null references
        Weeble[] c = new Weeble[4];
        for(int i = 0; i < c.length; i++)
            if(c[i] == null) // Can test for null reference
                c[i] = new Weeble();
        // Aggregate initialization:
        Weeble[] d = {
            new Weeble(), new Weeble(), new Weeble()
        };
    }
}
```



```

// Dynamic aggregate initialization:
a = new Weeble[] {
    new Weeble(), new Weeble()
};
System.out.println("a.length=" + a.length);
System.out.println("b.length = " + b.length);
// The references inside the array are
// automatically initialized to null:
for(int i = 0; i < b.length; i++)
    System.out.println("b[" + i + "]= " + b[i]);
System.out.println("c.length = " + c.length);
System.out.println("d.length = " + d.length);
a = d;
System.out.println("a.length = " + a.length);

// Arrays of primitives:
int[] e; // Null reference
int[] f = new int[5];
int[] g = new int[4];
for(int i = 0; i < g.length; i++)
    g[i] = i*i;
int[] h = { 11, 47, 93 };
// Compile error: variable e not initialized:
//!System.out.println("e.length=" + e.length);
System.out.println("f.length = " + f.length);
// The primitives inside the array are
// automatically initialized to zero:
for(int i = 0; i < f.length; i++)
    System.out.println("f[" + i + "]= " + f[i]);
System.out.println("g.length = " + g.length);
System.out.println("h.length = " + h.length);
e = h;
System.out.println("e.length = " + e.length);
e = new int[] { 1, 2 };
System.out.println("e.length = " + e.length);
monitor.expect(new String[] {
    "a.length=2",
    "b.length = 5",
    "b[0]=null",
    "b[1]=null",
    "b[2]=null",
    "b[3]=null",
    "b[4]=null",
    "c.length = 4",

```

```

        "d.length = 3",
        "a.length = 3",
        "f.length = 5",
        "f[0]=0",
        "f[1]=0",
        "f[2]=0",
        "f[3]=0",
        "f[4]=0",
        "g.length = 4",
        "h.length = 3",
        "e.length = 3",
        "e.length = 2"
    });
}
} ///:~

```

数组 **a** 是一个尚未初始化的局部变量，在将其恰当地初始化之前，编译器不允许你用此引用做任何事情。数组 **b** 初始化为指向一个 **Weeble** 引用的数组，但其实并没有 **Weeble** 对象置入数组中。你仍然可以询问数组的大小，因为 **b** 指向一个合法的对象。这样做有一个小缺点：你无法知道在此数组中确切地有多少元素，因为 **length** 只表示数组能够容纳多少元素。也就是说，**length** 是数组的大小，而不是实际保存的元素个数。新生成一个数组对象时，其中所有的引用被自动初始化为 **null**，所以检查其中的引用是否为 **null**，即可知道数组的某个位置是否存有对象。同样地，基本类型的数组如果是数值型的，就被自动初始化为 **0**，字符型 (**char**) 数组被初始化为 (**char**)**0**，布尔 (**boolean**) 数组被初始化为 **false**。

数组 **c** 在创建之后，随即将数组的各个位置都赋值为 **Weeble** 对象。数组 **d** 演示了使用“聚集初始化 (**aggregate initialization**)”语法创建数组对象（隐式地使用 **new** 在堆中创建，就像数组 **c** 一样），并且以 **Weeble** 对象将其初始化的过程，这些操作只用了一条语句。

下一个数组初始化可以看作“动态的聚集初始化”。数组 **d** 采用的聚集初始化动作必须在定义 **d** 的位置使用，但若使用第二种语法，你可以在任意位置创建和初始化数组对象。例如，假设方法 **hide()** 需要一个 **Weeble** 对象的数组作为输入参数。可以如下调用：

```
hide(d);
```

但你也可以动态地创建数组，将其作为参数传递：

```
hide(new Weeble[] { new Weeble(), new Weeble() });
```

在许多情况下，此语法使得代码书写变得更方便了。

表达式：

```
a = d;
```

演示了如何将指向某个数组对象的引用赋值指向另一个数组对象，这与其他类型对象的引用没什么区别。现在 **a** 与 **d** 都指向堆中的同一个数组对象。

`ArraySize.java` 的第二部分说明，基本类型的数组的工作方式与对象数组一样，不过基本类型的数组直接存储基本类型数据的值。

基本类型的容器

容器类只能保存对象的引用。而数组可以像保存对象的引用一样，直接保存基本类型。在容器类可以使用“包装”类，例如 `Integer`，`Double` 等，以代替基本类型的值。但是相对于基本类型，包装类使用起来很笨拙。此外，与包装过的基本类型的容器相比，创建与访问一个基本类型的数组效率更高。

当然，如果你在处理基本类型数据，而又需要容器的灵活性，以便在需要更多空间的时候能够自动扩展，就应该使用包装过的基本类型的容器，因为数组做不到这一点。你可能会认为，对于每一种基本类型都应该有对应类型的 `ArrayList`，但是Java并没有这么做。²

返回一个数组

假设你要写一个方法，而且希望它返回的不止一个值，而是一组值。这对于 C 和 C++ 这样的语言来说可有点困难，因为它们不能返回一个数组，而是只能返回指向数组的指针。这会造成一些问题，因为它使得控制数组的生命周期变得很困难，并且容易造成内存泄漏。

Java 采用类似的方法，但允许你直接“返回一个数组”。与 C++ 不同，使用 Java 你不需要担心要为数组负责——只要你需要它，它就会一直存在，当你使用完后，垃圾回收器会清理掉它。

下例演示了如何返回 `String` 数组：

```
//: c11:IceCream.java
// Returning arrays from methods.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class IceCream {
    private static Test monitor = new Test();
    private static Random rand = new Random();
    public static final String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
};
```

² 这正是C++比Java高明之处，因为C++有`template`关键字支持参数化的类型。

```

public static String[] flavorSet(int n) {
    String[] results = new String[n];
    boolean[] picked = new boolean[flavors.length];
    for(int i = 0; i < n; i++) {
        int t;
        do
            t = rand.nextInt(flavors.length);
        while(picked[t]);
        results[i] = flavors[t];
        picked[t] = true;
    }
    return results;
}

public static void main(String[] args) {
    for(int i = 0; i < 20; i++) {
        System.out.println(
            "flavorSet(" + i + ") = ");
        String[] fl = flavorSet(flavors.length);
        for(int j = 0; j < fl.length; j++)
            System.out.println("\t" + fl[j]);
        monitor.expect(new Object[] {
            "%% flavorSet\\((\\d+\\) = ",
            new TestExpression("%% \\t(Chocolate|Strawberry|"
                + "Vanilla Fudge Swirl|Mint Chip|Mocha Almond "
                + "Fudge|Rum Raisin|Praline Cream|Mud Pie)", 8)
        });
    }
}
} ///:~

```

方法 `flavorSet()` 创建了一个名为 `results` 的 `String` 数组。此数组容量为 `n`，由传入方法的参数决定。然后从数组 `flavors` 中随机地选择元素，存入 `results` 中，此方法最终返回 `results` 数组。返回一个数组与返回任何其他对象（本质是引用）没什么区别。数组是在 `flavorSet()` 中被创建的，还是在别的地方被创建的并不重要。当你使用完毕后，垃圾回收器负责清除数组，而只要你还需要它，此数组就会一直存在。

说句题外话，注意当 `flavorSet()` 随机选择各种 `flavor`（数组的元素）时，它确保不会重复选择。由一个 `do` 循环不断进行随机选择，直到找出一个在数组 `picked` 中还不存在的元素。（当然，还会比较 `String` 以检查随机选择的元素是否已经在数组 `results` 中。）如果成功，将此元素加入数组，然后查找下一个（`i` 递增）。

`main()` 打印 20 个完整的 `flavor` 的集合，因此可以看出 `flavorSet()` 每次确实是在随机选择 `flavor`。最容易看到执行效果的方式是将输出信息重定向到一个文件。当你查看这个文件时，要记住，你只是“想要”冰激凌，并不是“需要”冰激凌。

Arrays 类

在 `java.util` 类库中可以找到 `Arrays` 类，它有一套 `static` 方法，提供操作数组的实用功能。其中有四个基本方法：`equals()`，比较两个数组是否相等；`fill()`，用某个值填充整个数组；`sort()`，对数组排序；还有 `binarySearch()`，在已经排序的数组中查找元素。所有这些方法都被各种基本类型和 `Object` 类重载过。此外，方法 `asList()` 接受任意的数组为参数，并将其转变为 `List` 容器（稍后会学到 `List`）。

`Arrays` 类虽然很有用，但是它却仍然不具备完备的功能。例如，如果能够很容易地打印数组的所有元素，而不需要每次都自己去编写 `for` 循环的代码，那就太好了。而且你会看到，方法 `fill()` 只能以某个单一的值填充整个数组，如果你想用随机生成的若干数字填充数组，`fill()` 就无能为力了。

所以为 `Arrays` 类添加这样一些实用功能会很有帮助，我将它们打包放入 `com.bruceeckel.util` 包中，以便于使用。利用它们可以打印任意类型的数组，以不同的值或对象填充数组，这些对象由你定义的“生成器（generator）”对象创建。

因为要为每一种基本类型和对象都要编写代码，所以有许多重复的代码³。例如，要为每一种类型编写一个“生成器（generator）”接口，因为 `next()` 的返回类型各不相同：

```
//: com:bruceeckel:util:Generator.java
package com.bruceeckel.util;
public interface Generator { Object next(); } ///:~

//: com:bruceeckel:util:BooleanGenerator.java
package com.bruceeckel.util;
public interface BooleanGenerator { boolean next(); } ///:~

//: com:bruceeckel:util:ByteGenerator.java
package com.bruceeckel.util;
public interface ByteGenerator { byte next(); } ///:~

//: com:bruceeckel:util:CharGenerator.java
package com.bruceeckel.util;
public interface CharGenerator { char next(); } ///:~
```

³ C++程序员会注意到，使用默认参数和 `template` 可以省略很多代码。Python 程序员会注意到，在 Python 中此类库整个都是不必要的。

```

//: com:bruceeckel:util:ShortGenerator.java
package com.bruceeckel.util;
public interface ShortGenerator { short next(); } ///:~

```

```

//: com:bruceeckel:util:IntGenerator.java
package com.bruceeckel.util;
public interface IntGenerator { int next(); } ///:~

```

```

//: com:bruceeckel:util:LongGenerator.java
package com.bruceeckel.util;
public interface LongGenerator { long next(); } ///:~

```

```

//: com:bruceeckel:util:FloatGenerator.java
package com.bruceeckel.util;
public interface FloatGenerator { float next(); } ///:~

```

```

//: com:bruceeckel:util:DoubleGenerator.java
package com.bruceeckel.util;
public interface DoubleGenerator { double next(); } ///:~

```

Arrays2 为各种类型都重载了 `toString()` 方法。这些方法使得打印数组毫不费力。`toString()` 的代码展示了如何使用 `StringBuffer` 代替 `String` 对象，这是基于效率考虑；如果你多次调用一个方法，其中需要组装字符串，那么使用更高效的 `StringBuffer` 取代 `String` 就是明智之举。此处的 `StringBuffer`，在创建的时候就带有初始值，然后再向其中添加 `String`。最后，将 `result` 转换成 `String` 再返回：

```

//: com:bruceeckel:util:Arrays2.java
// A supplement to java.util.Arrays, to provide additional
// useful functionality when working with arrays. Allows
// any array to be converted to a String, and to be filled
// via a user-defined "generator" object.
package com.bruceeckel.util;
import java.util.*;

public class Arrays2 {

```

```

public static String toString(boolean[] a) {
    StringBuffer result = new StringBuffer("");
    for(int i = 0; i < a.length; i++) {
        result.append(a[i]);
        if(i < a.length - 1)
            result.append(", ");
    }
    result.append("]");
    return result.toString();
}

public static String toString(byte[] a) {
    StringBuffer result = new StringBuffer("");
    for(int i = 0; i < a.length; i++) {
        result.append(a[i]);
        if(i < a.length - 1)
            result.append(", ");
    }
    result.append("]");
    return result.toString();
}

public static String toString(char[] a) {
    StringBuffer result = new StringBuffer("");
    for(int i = 0; i < a.length; i++) {
        result.append(a[i]);
        if(i < a.length - 1)
            result.append(", ");
    }
    result.append("]");
    return result.toString();
}

public static String toString(short[] a) {
    StringBuffer result = new StringBuffer("");
    for(int i = 0; i < a.length; i++) {
        result.append(a[i]);
        if(i < a.length - 1)
            result.append(", ");
    }
    result.append("]");
    return result.toString();
}

public static String toString(int[] a) {
    StringBuffer result = new StringBuffer("");
    for(int i = 0; i < a.length; i++) {
        result.append(a[i]);

```

```

        if(i < a.length - 1)
            result.append(", ");
    }
    result.append("]");
    return result.toString();
}

public static String toString(long[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
        result.append(a[i]);
        if(i < a.length - 1)
            result.append(", ");
    }
    result.append("]");
    return result.toString();
}

public static String toString(float[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
        result.append(a[i]);
        if(i < a.length - 1)
            result.append(", ");
    }
    result.append("]");
    return result.toString();
}

public static String toString(double[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
        result.append(a[i]);
        if(i < a.length - 1)
            result.append(", ");
    }
    result.append("]");
    return result.toString();
}

// Fill an array using a generator:
public static void fill(Object[] a, Generator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(Object[] a, int from, int to, Generator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

```



```

    }
    public static void
    fill(boolean[] a, BooleanGenerator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void
    fill(boolean[] a, int from, int to, BooleanGenerator gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    public static void fill(byte[] a, ByteGenerator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void
    fill(byte[] a, int from, int to, ByteGenerator gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    public static void fill(char[] a, CharGenerator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void
    fill(char[] a, int from, int to, CharGenerator gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    public static void fill(short[] a, ShortGenerator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void
    fill(short[] a, int from, int to, ShortGenerator gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    public static void fill(int[] a, IntGenerator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void
    fill(int[] a, int from, int to, IntGenerator gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    public static void fill(long[] a, LongGenerator gen) {
        fill(a, 0, a.length, gen);
    }

```

```

    }
    public static void
    fill(long[] a, int from, int to, LongGenerator gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    public static void fill(float[] a, FloatGenerator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void
    fill(float[] a, int from, int to, FloatGenerator gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    public static void fill(double[] a, DoubleGenerator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void
    fill(double[] a, int from, int to, DoubleGenerator gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    private static Random r = new Random();
    public static class
    RandBooleanGenerator implements BooleanGenerator {
        public boolean next() { return r.nextBoolean(); }
    }
    public static class
    RandByteGenerator implements ByteGenerator {
        public byte next() { return (byte)r.nextInt(); }
    }
    private static String ssource =
        "ABCDEFGHJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    private static char[] src = ssource.toCharArray();
    public static class
    RandCharGenerator implements CharGenerator {
        public char next() {
            return src[r.nextInt(src.length)];
        }
    }
    public static class
    RandStringGenerator implements Generator {
        private int len;
        private RandCharGenerator cg = new RandCharGenerator();

```

```

    public RandStringGenerator(int length) {
        len = length;
    }
    public Object next() {
        char[] buf = new char[len];
        for(int i = 0; i < len; i++)
            buf[i] = cg.next();
        return new String(buf);
    }
}

public static class
RandShortGenerator implements ShortGenerator {
    public short next() { return (short)r.nextInt(); }
}

public static class
RandIntGenerator implements IntGenerator {
    private int mod = 10000;
    public RandIntGenerator() {}
    public RandIntGenerator(int modulo) { mod = modulo; }
    public int next() { return r.nextInt(mod); }
}

public static class
RandLongGenerator implements LongGenerator {
    public long next() { return r.nextLong(); }
}

public static class
RandFloatGenerator implements FloatGenerator {
    public float next() { return r.nextFloat(); }
}

public static class
RandDoubleGenerator implements DoubleGenerator {
    public double next() {return r.nextDouble();}
}
} ///:~

```

为了使用生成器生成的元素填充数组，`fill()`方法以恰当类型的生成器接口的引用作为参数，生成器的`next()`方法生成一个类型正确的对象（依赖于接口如何实现）。`fill()`方法直接调用`next()`，填充所需的范围。现在，实现恰当的接口即可制作一个生成器，并在`fill()`中使用自己的生成器。

随机数据生成器对于测试很有用。所以，这里使用了一组内部类，实现基本类型的生成器接口。同时，使用 `String` 生成器作为 `Object` 的代表。可以看到 `RandStringGenerator` 使用 `RandCharGenerator` 填充一个字符数组，然后将其转成 `String`。此数组的大小由构造器的参数决定。

`RandIntGenerator` 默认以 10,000 为模数，如果不需要生成太大的数字，可以重载构造器选择一个稍小的值。

下面的程序测试并演示了如何使用此类库：

```
//: c11:TestArrays2.java
// Test and demonstrate Arrays2 utilities.
import com.bruceeckel.util.*;

public class TestArrays2 {
    public static void main(String[] args) {
        int size = 6;
        // Or get the size from the command line:
        if(args.length != 0) {
            size = Integer.parseInt(args[0]);
            if(size < 3) {
                System.out.println("arg must be >= 3");
                System.exit(1);
            }
        }
        boolean[] a1 = new boolean[size];
        byte[] a2 = new byte[size];
        char[] a3 = new char[size];
        short[] a4 = new short[size];
        int[] a5 = new int[size];
        long[] a6 = new long[size];
        float[] a7 = new float[size];
        double[] a8 = new double[size];
        Arrays2.fill(a1, new Arrays2.RandBooleanGenerator());
        System.out.println("a1 = " + Arrays2.toString(a1));
        Arrays2.fill(a2, new Arrays2.RandByteGenerator());
        System.out.println("a2 = " + Arrays2.toString(a2));
        Arrays2.fill(a3, new Arrays2.RandCharGenerator());
        System.out.println("a3 = " + Arrays2.toString(a3));
        Arrays2.fill(a4, new Arrays2.RandShortGenerator());
        System.out.println("a4 = " + Arrays2.toString(a4));
        Arrays2.fill(a5, new Arrays2.RandIntGenerator());
        System.out.println("a5 = " + Arrays2.toString(a5));
        Arrays2.fill(a6, new Arrays2.RandLongGenerator());
        System.out.println("a6 = " + Arrays2.toString(a6));
        Arrays2.fill(a7, new Arrays2.RandFloatGenerator());
        System.out.println("a7 = " + Arrays2.toString(a7));
        Arrays2.fill(a8, new Arrays2.RandDoubleGenerator());
        System.out.println("a8 = " + Arrays2.toString(a8));
    }
}
```

```
} ///:~
```

参数 `size` 有默认值，不过你也可以通过命令行参数设置它。

填充数组

Java 标准类库 `Arrays` 也有 `fill()` 方法，但是它作用有限。只能用同一个值填充各个位置，对于保存对象的数组，就是复制同一个引用进行填充。使用 `Arrays2.toString()` 方法，很容易证明这一点：

```
///: c11:FillingArrays.java
// Using Arrays.fill()
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;
import java.util.*;

public class FillingArrays {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        int size = 6;
        // Or get the size from the command line:
        if(args.length != 0)
            size = Integer.parseInt(args[0]);
        boolean[] a1 = new boolean[size];
        byte[] a2 = new byte[size];
        char[] a3 = new char[size];
        short[] a4 = new short[size];
        int[] a5 = new int[size];
        long[] a6 = new long[size];
        float[] a7 = new float[size];
        double[] a8 = new double[size];
        String[] a9 = new String[size];
        Arrays.fill(a1, true);
        System.out.println("a1 = " + Arrays2.toString(a1));
        Arrays.fill(a2, (byte)11);
        System.out.println("a2 = " + Arrays2.toString(a2));
        Arrays.fill(a3, 'x');
        System.out.println("a3 = " + Arrays2.toString(a3));
        Arrays.fill(a4, (short)17);
        System.out.println("a4 = " + Arrays2.toString(a4));
        Arrays.fill(a5, 19);
        System.out.println("a5 = " + Arrays2.toString(a5));
        Arrays.fill(a6, 23);
        System.out.println("a6 = " + Arrays2.toString(a6));
```

```

Arrays.fill(a7, 29);
System.out.println("a7 = " + Arrays2.toString(a7));
Arrays.fill(a8, 47);
System.out.println("a8 = " + Arrays2.toString(a8));
Arrays.fill(a9, "Hello");
System.out.println("a9 = " + Arrays.asList(a9));
// Manipulating ranges:
Arrays.fill(a9, 3, 5, "World");
System.out.println("a9 = " + Arrays.asList(a9));
monitor.expect(new String[] {
    "a1 = [true, true, true, true, true, true]",
    "a2 = [11, 11, 11, 11, 11, 11]",
    "a3 = [x, x, x, x, x, x]",
    "a4 = [17, 17, 17, 17, 17, 17]",
    "a5 = [19, 19, 19, 19, 19, 19]",
    "a6 = [23, 23, 23, 23, 23, 23]",
    "a7 = [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]",
    "a8 = [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]",
    "a9 = [Hello, Hello, Hello, Hello, Hello, Hello]",
    "a9 = [Hello, Hello, Hello, World, World, Hello]"
});
}
} ///:~

```

使用 `Arrays.fill()` 可以填充整个数组，或者像最后两条语句所示，只填充数组的某个区域。不过使用 `Arrays.fill()` 你只能提供单个值用来填充，而使用 `Arrays2.fill()` 则可以生成更多有趣的结果。

复制数组

Java 标准类库提供有 `static` 方法 `System.arraycopy()`，用它复制数组比用 `for` 循环复制要快很多。`System.arraycopy()` 为所有类型作了重载。下面的例子就是用来处理 `int` 数组的：

```

//: c11:CopyingArrays.java
// Using System.arraycopy()
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;
import java.util.*;

public class CopyingArrays {
    private static Test monitor = new Test();

```

```

public static void main(String[] args) {
    int[] i = new int[7];
    int[] j = new int[10];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    System.out.println("i = " + Arrays2.toString(i));
    System.out.println("j = " + Arrays2.toString(j));
    System.arraycopy(i, 0, j, 0, i.length);
    System.out.println("j = " + Arrays2.toString(j));
    int[] k = new int[5];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    System.out.println("k = " + Arrays2.toString(k));
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    System.out.println("i = " + Arrays2.toString(i));
    // Objects:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    System.out.println("u = " + Arrays.asList(u));
    System.out.println("v = " + Arrays.asList(v));
    System.arraycopy(v, 0, u, u.length/2, v.length);
    System.out.println("u = " + Arrays.asList(u));
    monitor.expect(new String[] {
        "i = [47, 47, 47, 47, 47, 47, 47]",
        "j = [99, 99, 99, 99, 99, 99, 99, 99, 99, 99]",
        "j = [47, 47, 47, 47, 47, 47, 47, 99, 99, 99]",
        "k = [47, 47, 47, 47, 47]",
        "i = [103, 103, 103, 103, 103, 47, 47]",
        "u = [47, 47, 47, 47, 47, 47, 47, 47, 47, 47]",
        "v = [99, 99, 99, 99, 99]",
        "u = [47, 47, 47, 47, 47, 99, 99, 99, 99, 99]"
    });
}
} ///:~

```

`arraycopy()`需要的参数有：源数组、表示从源数组中的什么位置开始复制的偏移量、表示从目标数组的什么位置开始复制的偏移量、以及需要复制的元素个数。当然，对数组的任何越界操作都会导致异常。

这个例子说明基本类型数组与对象数组都可以复制。然而，如果你复制对象数组，那么只是复制了引用——不会出现两份对象的拷贝。这被称作浅复制（**shallow copy**）（参见附录 A）。

数组的比较

`Arrays` 类重载了 `equals()` 方法，用来比较整个数组。同样的，此方法被所有基本类型与 `Object` 都作了重载。数组相等的条件时元素个数必须相等，并且对应位置的元素也相等，这可以通过对每一个元素使用 `equals()` 做比较来判断。（对于基本类型，需要使用基本类型的包装类的 `equals()` 方法，例如，对于 `int` 类型使用 `Integer.equals()` 作比较）见下例：

```
//: c11:ComparingArrays.java
// Using Arrays.equals()
import com.bruceeckel.simpletest.*;
import java.util.*;

public class ComparingArrays {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        int[] a1 = new int[10];
        int[] a2 = new int[10];
        Arrays.fill(a1, 47);
        Arrays.fill(a2, 47);
        System.out.println(Arrays.equals(a1, a2));
        a2[3] = 11;
        System.out.println(Arrays.equals(a1, a2));
        String[] s1 = new String[5];
        Arrays.fill(s1, "Hi");
        String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
        System.out.println(Arrays.equals(s1, s2));
        monitor.expect(new String[] {
            "true",
            "false",
            "true"
        });
    }
} ///:~
```

最初，`a1` 与 `a2` 完全相等，所以输出为“true”，然后改变其中一个元素，使得结果为“false”。在最后一个例子中，`s1` 的所有元素都指向同一个对象，而数组 `s2` 持有五个相互独立的对象。然而，数组相等是基于内容的（通过 `Object.equals()` 比较），所以结果仍为“true”。

数组元素的比较

Java 1.0 和 1.1 的类库缺少许多特性，其中之一就是缺少算法操作——甚至是简单的排序操作都没有。对于盼望能得到一个无所不有的标准类库的人来说，缺少这些操作是很难理解的。幸好 Java 2 对此作了补救，至少解决了排序问题。

书写一般性的排序程序要先解决一个问题，排序必须根据对象的实际类型执行比较操作。一种自然的解决方案是为每种不同的类型各编写一个不同的排序方法，不过你应该能够认识到，这样的代码无法被新的类型所复用。

程序设计的基本目标是“将保持不变的事物与会发生改变的事物相分离”，而这里，不变的是通用的排序算法，变化的是各种对象相互比较的方式。因此，不是将进行比较的代码编写成为不同的子程序，而是使用回调技术（callback）。通过使用回调，可以将会发生变化的代码分离出来，然后由不会发生变化的代码回头调用会发生变化的代码。

Java 有两种方式提供比较功能。第一种是实现 `java.lang.Comparable` 接口，使你的类具有“天生”的比较能力。此接口很简单，只有 `compareTo()` 一个方法。此方法以要比较的 `Object` 为参数，如果当前对象小于参数则返回负值，如果相等则返回零，如果当前对象大于参数则返回正值。

下面的类实现了 `Comparable` 接口，并且使用 Java 标准类库的方法 `Arrays.sort()` 演示了比较的效果：

```
//: c11:CompType.java
// Implementing Comparable in a class.
import com.bruceeckel.util.*;
import java.util.*;

public class CompType implements Comparable {
    int i;
    int j;
    public CompType(int n1, int n2) {
        i = n1;
        j = n2;
    }
    public String toString() {
        return "[i = " + i + ", j = " + j + "]";
    }
    public int compareTo(Object rv) {
        int rvi = ((CompType)rv).i;
        return (i < rvi ? -1 : (i == rvi ? 0 : 1));
    }
    private static Random r = new Random();
    public static Generator generator() {
        return new Generator() {
            public Object next() {
                return new CompType(r.nextInt(100), r.nextInt(100));
            }
        };
    }
    public static void main(String[] args) {
```

```

CompType[] a = new CompType[10];
Arrays2.fill(a, generator());
System.out.println(
    "before sorting, a = " + Arrays.asList(a));
Arrays.sort(a);
System.out.println(
    "after sorting, a = " + Arrays.asList(a));
}
} ///:~

```

在定义作比较的方法时，你负责决定如何将你的对象与另一个对象作比较。这里在比较中只用到了 *i* 值，而忽略了 *j* 值。

方法 `static randInt()` 生成 0 到 100 之间的正值，方法 `generator()` 生成一个对象，此对象通过创建一个匿名内部类（见第八章）来实现 `Generator` 接口。该例中以使用随机数进行初始化的方式构建 `CompType` 对象。在 `main()` 中，使用生成器填充 `CompType` 的数组，然后对其排序。如果没有实现 `Comparable` 接口，调用 `sort()` 的时候会抛出 `ClassCastException` 的运行期异常。因为 `sort()` 需要把参数的类型转变为 `Comparable`。

假设有人给你一个并没有实现 `Comparable` 的类，或者给你的类实现了 `Comparable`，但是你不喜欢它的实现方式，你需要另外一种不同的比较方法。那么与将进行比较的代码硬编码进对象相比，下面采用了完全不同的方式解决此问题：使用策略（**strategy**）设计模式⁴。通过使用策略，将会发生变化的代码包装在类中（即所谓策略对象）。将策略对象交给保持不变的代码，后者使用此策略实现它的算法。也就是说，可以为不同的比较方式生成不同的对象，将它们用在同样的排序程序中。此处，通过定义一个实现了 `Comparable` 接口的类而创建了一个策略。这个类有 `compare()` 和 `equals()` 两个方法。然而，不是一定要实现 `equals()` 方法，除非是为了特别的性能需要。因为无论何时创建一个类，都是间接继承自 `Object`，而 `Object` 带有 `equals()` 方法。所以只用默认的 `Object` 的 `equals()` 方法就可以满足接口的要求了。

`Collections` 类（后面会详细地学习到）包含一个 `Comparator`，可以倒转自然的排序顺序。它很容易应用于 `CompType`：

```

///: c11:Reverse.java
// The Collections.reverseOrder() Comparator
import com.bruceeckel.util.*;
import java.util.*;

public class Reverse {
    public static void main(String[] args) {
        CompType[] a = new CompType[10];
        Arrays2.fill(a, CompType.generator());
        System.out.println(

```

⁴ 《设计模式》 *Design Patterns*, Erich Gamma *et al.*, Addison-Wesley 1995.

```

        "before sorting, a = " + Arrays.asList(a));
    Arrays.sort(a, Collections.reverseOrder());
    System.out.println(
        "after sorting, a = " + Arrays.asList(a));
    }
} ///:~

```

调用 `Collections.reverseOrder()` 即生成指向这个 `Comparator` 的引用。

在接下来的例子中，`Comparator` 基于 `j` 值比较 `CompType` 对象，而不是 `i` 值。

```

///: c11:ComparatorTest.java
// Implementing a Comparator for a class.
import com.bruceeckel.util.*;
import java.util.*;

class CompTypeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        int j1 = ((CompType)o1).j;
        int j2 = ((CompType)o2).j;
        return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
    }
}

public class ComparatorTest {
    public static void main(String[] args) {
        CompType[] a = new CompType[10];
        Arrays2.fill(a, CompType.generator());
        System.out.println(
            "before sorting, a = " + Arrays.asList(a));
        Arrays.sort(a, new CompTypeComparator());
        System.out.println(
            "after sorting, a = " + Arrays.asList(a));
    }
} ///:~

```

`compare()` 方法返回负整数、零、或正整数，分别对应于第一个参数小于、等于、或大于第二个参数。

数组排序

使用内置的排序方法，就可以对任意的基本类型数组排序，也可以对任意的对象数组进行排序，只要该对象实现了 `Comparable` 接口或具有相关联的 `Comparator`。这对 Java 类

库是很大的补充；你相信吗，Java 1.0 或 1.1 甚至不支持 `String` 的排序。下面的例子生成随机的 `String` 对象，并且对其排序：

```
//: c11:StringSorting.java
// Sorting an array of Strings.
import com.bruceeckel.util.*;
import java.util.*;

public class StringSorting {
    public static void main(String[] args) {
        String[] sa = new String[30];
        Arrays2.fill(sa, new Arrays2.RandStringGenerator(5));
        System.out.println(
            "Before sorting: " + Arrays.asList(sa));
        Arrays.sort(sa);
        System.out.println(
            "After sorting: " + Arrays.asList(sa));
    }
} ///:~
```

注意，`String` 排序算法依据词典顺序 (*lexicographic*) 排序，所以大写字母开头的词都放在前面，然后才是小写字母开头的词。（电话簿通常是这样排序的。）如果我想忽略大小写字母将单词都放在一起排序，可以定义自己的 `Comparator` 类，然后重载其默认的 `String Comparable`，改变作比较的方式。为了能够重用，将 `AlphabeticComparator` 加入我的“util”类库中：

```
//: com:bruceeckel:util:AlphabeticComparator.java
// Keeping upper and lowercase letters together.
package com.bruceeckel.util;
import java.util.*;

public class AlphabeticComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.toLowerCase().compareTo(s2.toLowerCase());
    }
} ///:~
```

程序首先将 `Object` 转型为 `String`，如果你使用了错误类型的对象，就会得到一个异常。每个 `String` 在比较前都被转换成小写字母，然后使用 `String` 内置的 `compareTo()` 方法作比较。

下面使用 `AlphabeticComparator` 做测试：

```

//: c11:AlphabeticSorting.java
// Keeping upper and lowercase letters together.
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSorting {
    public static void main(String[] args) {
        String[] sa = new String[30];
        Arrays2.fill(sa, new Arrays2.RandStringGenerator(5));
        System.out.println(
            "Before sorting: " + Arrays.asList(sa));
        Arrays.sort(sa, new AlphabeticComparator());
        System.out.println(
            "After sorting: " + Arrays.asList(sa));
    }
} ///:~

```

Java 标准类库中的排序算法为各种类型作了优化——针对基本类型设计的“快速排序（Quicksort）”，以及针对对象设计的“稳定归并排序（stable merge sort）”。所以你无须担心排序的性能，除非你可以证明排序部分的确是程序效率的瓶颈。

在已排序的数组中查找

如果数组已经排好序了，就可以使用 `Arrays.binarySearch()` 执行快速查找。千万不要对未排序的数组使用 `binarySearch()`，否则结果不可预料。下面的例子使用 `RandIntGenerator` 填充数组，再用此生成器生成一个值用来测试查找：

```

//: c11:ArraySearching.java
// Using Arrays.binarySearch().
import com.bruceeckel.util.*;
import java.util.*;

public class ArraySearching {
    public static void main(String[] args) {
        int[] a = new int[100];
        Arrays2.RandIntGenerator gen =
            new Arrays2.RandIntGenerator(1000);
        Arrays2.fill(a, gen);
        Arrays.sort(a);
        System.out.println(
            "Sorted array: " + Arrays2.toString(a));
        while(true) {
            int r = gen.next();
            int location = Arrays.binarySearch(a, r);

```

```

        if(location >= 0) {
            System.out.println("Location of " + r +
                " is " + location + ", a[" +
                location + "] = " + a[location]);
            break; // Out of while loop
        }
    }
}
} ///:~

```

在 **while** 循环中随机生成一些值，作为查找的对象，直到找到一个才停止循环。

如果找到了目标，**Arrays.binarySearch()**的返回值等于或大于 0。否则，返回负值，表示为了保持数组的排序状态，此目标元素应该插入的位置。这个负值的计算方式是：

- (插入点) - 1

“插入点”是指，第一个大于查找对象的元素在数组中的位置，如果数组所有的元素都小于要查找的对象，“插入点”就等于 **a.size()**。

如果数组包含重复的元素，则无法保证找到的是哪一个。此算法并不是为包含重复元素的数组专门设计的，不过仍然可用。如果你需要对没有重复元素的数组排序，可以使用（保持排序顺序的）**TreeSet**，或者（保持插入顺序的）**LinkedHashSet**，后面我们将会学习它们。这些类会自动处理所有的细节。除非它们成为程序性能的瓶颈，否则你都不应该自己维护数组。

如果使用 **Comparator** 排序某个对象数组（基本类型数组无法使用 **Comparator** 进行排序），在使用 **binarySearch()**时必须提供同样的 **Comparator**（使用此方法的重载版本）。例如，修改 **AlphabeticSorting.java** 程序做这种查询：

```

///: c11:AlphabeticSearch.java
// Searching with a Comparator.
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSearch {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        String[] sa = new String[30];
        Arrays2.fill(sa, new Arrays2.RandomStringGenerator(5));
        AlphabeticComparator comp = new AlphabeticComparator();
        Arrays.sort(sa, comp);
        int index = Arrays.binarySearch(sa, sa[10], comp);
        System.out.println("Index = " + index);
    }
}

```

```
monitor.expect(new String[] {  
    "Index = 10"  
});  
}  
} ///:~
```

这里的 `Comparator` 必须传递给重载过的 `binarySearch()`，作为其第三个参数。在这个例子中，由于要查找的目标就是从数组中选出来的元素，所以肯定能查找到。

对数组的小结

总结目前为止我们所学的知识：数组是效率最高的保存一组对象的方式，它是你的第一选择。而且，如果要保存基本类型，则只能用数组。本章接下来将介绍更通用的情况。例如，写程序的时候并不知道程序运行时会需要多少对象，或者，如果需要更复杂的方式存储对象。Java 提供了容器类库，以解决此问题。容器基本上分为：**List**，**Set**，和 **Map**。使用这些工具可以解决许多问题。

Java 的容器类各有特点——例如 **Set**，每个对象只保存一份，**Map** 是一种关联性数组，允许你将任意一个对象与另一个对象关联起来。同时，Java 所有的容器都能够自动调整容量，这与数组不同。可以将任意数量的对象放入其中，所以在写程序的时候不用操心容器需要多大。

容器简介

就我而言，容器类对于新的开发者是最强大的工具之一，可以大幅提高编程能力。由于 Java 1.0 和 1.1 中容器的可怜表现，在 Java 2 中，所有的容器都经过重新设计⁵。某些容器改变不大，但多数容器完全改变了。提供了类似链表（linked list），队列（queue）和双向队列（deque，发音同“decks”）的行为，丰富了容器类库的功能。

设计容器类库是很困难的事（多数类库的设计也都如此）。在 C++ 中，容器类是用许多不同的类组成了基础，这好过 C++ 早期根本没有容器类，但是 Java 并不是这样的。我还见过的另一种极端情况是，整个容器类库只有一个类，“container”，其行为既像是线性序列（linear sequence），又像是关联数组（associative array）。Java 2 的容器类库强调平衡：成熟的容器类库应具备你所期望得到的完整功能，但比 C++ 的容器和其他类似的容器更易于学习与使用。于是其结果看起来就有点奇怪。这与早期 Java 类库的某些决策不同，这种奇怪并不是偶然的，而是权衡复杂性，经过深思熟虑做出的决策。要花点时间才能掌握类库的某些方面，不过我认为你很快就能掌握这些新工具。

Java 2 容器类库的用途是“持有你的对象”，并将其划分为两个不同的概念：

1. **Collection**: 一组独立的元素，通常有某种规则应用于其上。**List** 必须保持元

⁵ 由 Sun 公司的 Joshua Bloch 负责设计。

素特定的顺序，而 **Set** 不能有重复元素。（Java 的容器没有实现 **bag**，因为 **List** 提供了足够的功能）

2.Map: 一组成对的键值对（**key-value**）对象。初看起来这似乎应该是一个 **Collection**，其元素是成对的对象，但是这样的设计实现起来太笨拙了，于是我们将 **Map** 明确提取出来形成一个独立的概念。另一方面，如果使用 **Collection** 表示 **Map** 的部分内容，会便于查看此部分内容。因此 **Map** 可以返回所有键组成的 **Set**，所有值组成的 **Collection**，或其键值对组成的 **Set**；并且象数组一样容易扩展成多维 **Map**，无需增加新的概念，只要让 **Map** 中键值对的每个“值”也是一个 **Map** 即可（此 **Map** 中的“值”还可以是 **Map**，依此类推）。

接下来我们先学习容器的一般特性，然后深入细节，最后学习为什么某些容器具有不同的版本，以及如何它们在它们之间进行选择。

容器的打印

与数组不同，打印容器无需任何帮助，下面是一个例子，同时将介绍一些基本类型的容器：

```
//: c11:PrintingContainers.java
// Containers print themselves automatically.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class PrintingContainers {
    private static Test monitor = new Test();
    static Collection fill(Collection c) {
        c.add("dog");
        c.add("dog");
        c.add("cat");
        return c;
    }
    static Map fill(Map m) {
        m.put("dog", "Bosco");
        m.put("dog", "Spot");
        m.put("cat", "Rags");
        return m;
    }
    public static void main(String[] args) {
        System.out.println(fill(new ArrayList()));
        System.out.println(fill(new HashSet()));
        System.out.println(fill(new HashMap()));
        monitor.expect(new String[] {
            "[dog, dog, cat]",
            "[dog, cat]",
        })
    }
}
```



```

        "{dog=Spot, cat=Rags}"
    });
}
} ///:~

```

前面说过，Java 的容器类有两种基本类型。区别在于容器中每个位置保存的元素个数。**Collection** 每个位置只能保存一个元素（这个名字有点误导，因为整个容器类库经常被称作集合“collections”）。此类容器包括 **List**，它以特定的顺序保存一组元素；**Set**，元素不能重复。**ArrayList** 是一种 **List**，而 **HashSet** 是一种 **Set**。将元素添入任意 **Collection** 都可使用 **add()** 方法。

Map 保存的是键值对，就像一个小型数据库。前面例子使用的 **HashMap** 是一种 **Map**。使用 **Map** 可以将美国州名与其首府联系起来，如果想知道 Ohio 的首府，只需像数组下标一样查找 Ohio 即可。（**Map** 也被称作关联数组）使用方法 **put()** 为 **Map** 添加元素，它需要一个键与一个值作为参数。这个例子只演示了添加元素，并没有查找元素。稍后会有演示。

重载的 **fill()** 方法可分别用以填充 **Collection** 与 **Map**。查看输出你会发现，默认的打印动作（使用容器提供的 **toString()** 方法）即可生成可读性很好的结果，所以不需要额外的操作，而数组则不行。**Collection** 打印出来的内容用方括号括住，每个元素由逗号分隔。**Map** 则用大括号括住，键与值由等号联系（键在等号左边，值在右边）。

通过前面的例子即可看到不同容器的基本行为。**List** 按对象进入的顺序保存对象，不做排序或编辑操作。**Set** 对每个对象只接受一次，并使用自己内部的排序方法（通常，你只关心某个元素是否属于 **Set**，而不关心它的顺序——否则应该使用 **List**）。**Map** 同样对每个元素只保存一份，但这是基于“键”的，**Map** 也有内置的排序，因而不关心元素添加的顺序。如果添加元素的顺序对你很重要，应该使用 **LinkedHashSet** 或者 **LinkedHashMap**。

填充容器

虽然容器帮你解决了打印的问题，填充容器的操作仍然同 **java.util.Arrays** 一样有缺陷。与 **Arrays** 一样，**Collections** 也有一个实用的 **static** 方法集，其中包括有 **fill()**。此 **fill()** 方法也是用同一个对象的引用来填充容器的，并且只对 **List** 对象有用，而对 **Set** 或 **Map** 并不起作用。

```

///: c11:FillingLists.java
// The Collections.fill() method.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class FillingLists {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        List list = new ArrayList();
        for(int i = 0; i < 10; i++)

```

```

        list.add("");
        Collections.fill(list, "Hello");
        System.out.println(list);
        monitor.expect(new String[] {
            "[Hello, Hello, Hello, Hello, Hello, " +
            "Hello, Hello, Hello, Hello, Hello]"
        });
    }
} ///:~

```

此方法的作用有限，只能替换已经在 List 中存在的元素，并不能增加新元素。

为了编写更有趣的例子，我补充了带有 fill() 方法的 Collections2 类（为便于使用，我将其作为 com.bruceeckel.util 的一部分），此方法是用一个生成器来添加元素，它使你订制需要 add() 的元素个数。Collection 可以使用先前定义的 Generator 接口，但是 Map 需要它自己的生成器接口，因为对于 Map 每次调用 next() 需要生成一对对象（一个“键”与一个“值”）。下面是 Pair 类：

```

///: com:bruceeckel:util:Pair.java
package com.bruceeckel.util;

public class Pair {
    public Object key, value;
    public Pair(Object k, Object v) {
        key = k;
        value = v;
    }
} ///:~

```

下面的生成器接口用来生成 Pair：

```

///: com:bruceeckel:util:MapGenerator.java
package com.bruceeckel.util;
public interface MapGenerator { Pair next(); } ///:~

```

有了这些类，就可以开发一套与容器类一起使用的实用类了：

```

///: com:bruceeckel:util:Collections2.java
// To fill any type of container using a generator object.
package com.bruceeckel.util;
import java.util.*;

public class Collections2 {
    // Fill an array using a generator:
    public static void

```

```

fill(Collection c, Generator gen, int count) {
    for(int i = 0; i < count; i++)
        c.add(gen.next());
}

public static void
fill(Map m, MapGenerator gen, int count) {
    for(int i = 0; i < count; i++) {
        Pair p = gen.next();
        m.put(p.key, p.value);
    }
}

public static class
RandStringPairGenerator implements MapGenerator {
    private Arrays2.RandStringGenerator gen;
    public RandStringPairGenerator(int len) {
        gen = new Arrays2.RandStringGenerator(len);
    }
    public Pair next() {
        return new Pair(gen.next(), gen.next());
    }
}

// Default object so you don't have to create your own:
public static RandStringPairGenerator rsp =
    new RandStringPairGenerator(10);

public static class
StringPairGenerator implements MapGenerator {
    private int index = -1;
    private String[][] d;
    public StringPairGenerator(String[][] data) {
        d = data;
    }
    public Pair next() {
        // Force the index to wrap:
        index = (index + 1) % d.length;
        return new Pair(d[index][0], d[index][1]);
    }
    public StringPairGenerator reset() {
        index = -1;
        return this;
    }
}

// Use a predefined dataset:
public static StringPairGenerator geography =
    new StringPairGenerator(CountryCapitals.pairs);

```

```

// Produce a sequence from a 2D array:
public static class StringGenerator implements Generator{
    private String[][] d;
    private int position;
    private int index = -1;
    public StringGenerator(String[][] data, int pos) {
        d = data;
        position = pos;
    }
    public Object next() {
        // Force the index to wrap:
        index = (index + 1) % d.length;
        return d[index][position];
    }
    public StringGenerator reset() {
        index = -1;
        return this;
    }
}

// Use a predefined dataset:
public static StringGenerator countries =
    new StringGenerator(CountryCapitals.pairs, 0);
public static StringGenerator capitals =
    new StringGenerator(CountryCapitals.pairs, 1);
} ///:~

```

两个版本的 `fill()` 都需要一个参数，以确定向容器添加的元素个数。此外，对于 `Map` 有两种生成器：`RandStringPairGenerator`，生成任意数量的成对的随机 `String`，其长度由构造器的参数决定；`StringPairGenerator`，使用外面传入的二维 `String` 数组产生成对的 `String`。`StringGenerator` 也接受二维 `String` 数组为参数，但是生成一个单独的元素而不是 `Pair` 对象。`static rsp, geography, countries, capitals` 等对象各自提供预建的生成器，最后三者用到了所有的国家与首都。注意，如果你尝试创建很多的键值对，超过了以上对象提供的可用数据，生成器会循环至数据集的起点，在将键值对存入 `Map` 时，重复的内容会被忽略。

下面是预先定义的数据集，由国家的名字与首都组成：

```

//: com:bruceeckel:util:CountryCapitals.java
package com.bruceeckel.util;

public class CountryCapitals {
    public static final String[][] pairs = {
        // Africa
        {"ALGERIA", "Algiers"}, {"ANGOLA", "Luanda"},
        {"BENIN", "Porto-Novo"}, {"BOTSWANA", "Gaberone"},

```

{"BURKINA FASO", "Ouagadougou"},
 {"BURUNDI", "Bujumbura"},
 {"CAMEROON", "Yaounde"}, {"CAPE VERDE", "Praia"},
 {"CENTRAL AFRICAN REPUBLIC", "Bangui"},
 {"CHAD", "N' djamena"}, {"COMOROS", "Moroni"},
 {"CONGO", "Brazzaville"}, {"DJIBOUTI", "Djibouti"},
 {"EGYPT", "Cairo"}, {"EQUATORIAL GUINEA", "Malabo"},
 {"ERITREA", "Asmara"}, {"ETHIOPIA", "Addis Ababa"},
 {"GABON", "Libreville"}, {"THE GAMBIA", "Banjul"},
 {"GHANA", "Accra"}, {"GUINEA", "Conakry"},
 {"GUINEA", "-"}, {"BISSAU", "Bissau"},
 {"COTE D' IVOIR (IVORY COAST)", "Yamoussoukro"},
 {"KENYA", "Nairobi"}, {"LESOTHO", "Maseru"},
 {"LIBERIA", "Monrovia"}, {"LIBYA", "Tripoli"},
 {"MADAGASCAR", "Antananarivo"}, {"MALAWI", "Lilongwe"},
 {"MALI", "Bamako"}, {"MAURITANIA", "Nouakchott"},
 {"MAURITIUS", "Port Louis"}, {"MOROCCO", "Rabat"},
 {"MOZAMBIQUE", "Maputo"}, {"NAMIBIA", "Windhoek"},
 {"NIGER", "Niamey"}, {"NIGERIA", "Abuja"},
 {"RWANDA", "Kigali"},
 {"SAO TOME E PRINCIPE", "Sao Tome"},
 {"SENEGAL", "Dakar"}, {"SEYCHELLES", "Victoria"},
 {"SIERRA LEONE", "Freetown"}, {"SOMALIA", "Mogadishu"},
 {"SOUTH AFRICA", "Pretoria/Cape Town"},
 {"SUDAN", "Khartoum"},
 {"SWAZILAND", "Mbabane"}, {"TANZANIA", "Dodoma"},
 {"TOGO", "Lome"}, {"TUNISIA", "Tunis"},
 {"UGANDA", "Kampala"},
 {"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)",
 "Kinshasa"},
 {"ZAMBIA", "Lusaka"}, {"ZIMBABWE", "Harare"},
 // Asia
 {"AFGHANISTAN", "Kabul"}, {"BAHRAIN", "Manama"},
 {"BANGLADESH", "Dhaka"}, {"BHUTAN", "Thimphu"},
 {"BRUNEI", "Bandar Seri Begawan"},
 {"CAMBODIA", "Phnom Penh"},
 {"CHINA", "Beijing"}, {"CYPRUS", "Nicosia"},
 {"INDIA", "New Delhi"}, {"INDONESIA", "Jakarta"},
 {"IRAN", "Tehran"}, {"IRAQ", "Baghdad"},
 {"ISRAEL", "Tel Aviv"}, {"JAPAN", "Tokyo"},
 {"JORDAN", "Amman"}, {"KUWAIT", "Kuwait City"},
 {"LAOS", "Vientiane"}, {"LEBANON", "Beirut"},
 {"MALAYSIA", "Kuala Lumpur"}, {"THE MALDIVES", "Male"},
 {"MONGOLIA", "Ulan Bator"},

```

{"MYANMAR (BURMA)", "Rangoon"},
{"NEPAL", "Katmandu"}, {"NORTH KOREA", "P'yongyang"},
{"OMAN", "Muscat"}, {"PAKISTAN", "Islamabad"},
{"PHILIPPINES", "Manila"}, {"QATAR", "Doha"},
{"SAUDI ARABIA", "Riyadh"}, {"SINGAPORE", "Singapore"},
{"SOUTH KOREA", "Seoul"}, {"SRI LANKA", "Colombo"},
{"SYRIA", "Damascus"},
{"TAIWAN (REPUBLIC OF CHINA)", "Taipei"},
{"THAILAND", "Bangkok"}, {"TURKEY", "Ankara"},
{"UNITED ARAB EMIRATES", "Abu Dhabi"},
{"VIETNAM", "Hanoi"}, {"YEMEN", "Sana'a"},
// Australia and Oceania
{"AUSTRALIA", "Canberra"}, {"FIJI", "Suva"},
{"KIRIBATI", "Bairiki"},
{"MARSHALL ISLANDS", "Dalap-Uliga-Darrit"},
{"MICRONESIA", "Palikir"}, {"NAURU", "Yaren"},
{"NEW ZEALAND", "Wellington"}, {"PALAU", "Koror"},
{"PAPUA NEW GUINEA", "Port Moresby"},
{"SOLOMON ISLANDS", "Honaira"}, {"TONGA", "Nuku'alofa"},
{"TUVALU", "Fongafale"}, {"VANUATU", "Port-Vila"},
{"WESTERN SAMOA", "Apia"},
// Eastern Europe and former USSR
{"ARMENIA", "Yerevan"}, {"AZERBAIJAN", "Baku"},
{"BELARUS (BYELORUSSIA)", "Minsk"},
{"GEORGIA", "Tbilisi"},
{"KAZAKSTAN", "Almaty"}, {"KYRGYZSTAN", "Alma-Ata"},
{"MOLDOVA", "Chisinau"}, {"RUSSIA", "Moscow"},
{"TAJIKISTAN", "Dushanbe"}, {"TURKMENISTAN", "Ashkabad"},
{"UKRAINE", "Kyiv"}, {"UZBEKISTAN", "Tashkent"},
// Europe
{"ALBANIA", "Tirana"}, {"ANDORRA", "Andorra la Vella"},
{"AUSTRIA", "Vienna"}, {"BELGIUM", "Brussels"},
{"BOSNIA", "--"}, {"HERZEGOVINA", "Sarajevo"},
{"CROATIA", "Zagreb"}, {"CZECH REPUBLIC", "Prague"},
{"DENMARK", "Copenhagen"}, {"ESTONIA", "Tallinn"},
{"FINLAND", "Helsinki"}, {"FRANCE", "Paris"},
{"GERMANY", "Berlin"}, {"GREECE", "Athens"},
{"HUNGARY", "Budapest"}, {"ICELAND", "Reykjavik"},
{"IRELAND", "Dublin"}, {"ITALY", "Rome"},
{"LATVIA", "Riga"}, {"LIECHTENSTEIN", "Vaduz"},
{"LITHUANIA", "Vilnius"}, {"LUXEMBOURG", "Luxembourg"},
{"MACEDONIA", "Skopje"}, {"MALTA", "Valletta"},
{"MONACO", "Monaco"}, {"MONTENEGRO", "Podgorica"},
{"THE NETHERLANDS", "Amsterdam"}, {"NORWAY", "Oslo"},

```

```

{"POLAND", "Warsaw"}, {"PORTUGAL", "Lisbon"},
{"ROMANIA", "Bucharest"}, {"SAN MARINO", "San Marino"},
{"SERBIA", "Belgrade"}, {"SLOVAKIA", "Bratislava"},
{"SLOVENIA", "Ljubljana"}, {"SPAIN", "Madrid"},
{"SWEDEN", "Stockholm"}, {"SWITZERLAND", "Berne"},
{"UNITED KINGDOM", "London"}, {"VATICAN CITY", "----"},
// North and Central America
{"ANTIGUA AND BARBUDA", "Saint John's"},
{"BAHAMAS", "Nassau"},
{"BARBADOS", "Bridgetown"}, {"BELIZE", "Belmopan"},
{"CANADA", "Ottawa"}, {"COSTA RICA", "San Jose"},
{"CUBA", "Havana"}, {"DOMINICA", "Roseau"},
{"DOMINICAN REPUBLIC", "Santo Domingo"},
{"EL SALVADOR", "San Salvador"},
{"GRENADA", "Saint George's"},
{"GUATEMALA", "Guatemala City"},
{"HAITI", "Port-au-Prince"},
{"HONDURAS", "Tegucigalpa"}, {"JAMAICA", "Kingston"},
{"MEXICO", "Mexico City"}, {"NICARAGUA", "Managua"},
{"PANAMA", "Panama City"}, {"ST. KITTS", "--"},
{"NEVIS", "Basseterre"}, {"ST. LUCIA", "Castries"},
{"ST. VINCENT AND THE GRENADINES", "Kingstown"},
{"UNITED STATES OF AMERICA", "Washington, D.C."},
// South America
{"ARGENTINA", "Buenos Aires"},
{"BOLIVIA", "Sucre (legal)/La Paz (administrative)"},
{"BRAZIL", "Brasilia"}, {"CHILE", "Santiago"},
{"COLOMBIA", "Bogota"}, {"ECUADOR", "Quito"},
{"GUYANA", "Georgetown"}, {"PARAGUAY", "Asuncion"},
{"PERU", "Lima"}, {"SURINAME", "Paramaribo"},
{"TRINIDAD AND TOBAGO", "Port of Spain"},
{"URUGUAY", "Montevideo"}, {"VENEZUELA", "Caracas"},
};
} ///:~

```

这是一个简单的String数组⁶。下面使用fill()与生成器做个简单的测试：

```

//: c11:FillTest.java
import com.bruceeckel.util.*;
import java.util.*;

public class FillTest {
    private static Generator sg =

```

⁶ 此数据可在互联网上找到，并用Python程序处理过（参见www.Python.org）。

```

        new Arrays2.RandStringGenerator(7);
public static void main(String[] args) {
    List list = new ArrayList();
    Collections2.fill(list, sg, 25);
    System.out.println(list + "\n");
    List list2 = new ArrayList();
    Collections2.fill(list2, Collections2.capitals, 25);
    System.out.println(list2 + "\n");
    Set set = new HashSet();
    Collections2.fill(set, sg, 25);
    System.out.println(set + "\n");
    Map m = new HashMap();
    Collections2.fill(m, Collections2.rsp, 25);
    System.out.println(m + "\n");
    Map m2 = new HashMap();
    Collections2.fill(m2, Collections2.geography, 25);
    System.out.println(m2);
}
} ///:~

```

有了这些工具就可以通过使用有趣的数据填充容器来测试各种容器了。

容器的缺点：未知类型

使用 Java 容器有个“缺点”，在将对象加入容器的时候就丢失了类型信息。因为使用容器的程序员不关心你想要添入容器的对象的具体类型。如果容器只能保存你自己的类型，就失去了作为通用工具的意义。所以容器只保存 **Object** 型的引用，这是所有类的基类，因此容器可以保存任何类型的对象。（当然不包括基本类型，因为它们不是真正的对象，没有继承任何东西。）这是了不起的解决方式，不过：

1. 因为在你将对象的引用加入容器时就丢失了类型的信息，所以对于添入容器的对象没有类型限制，即使你刻意保持容器的类型，例如类型“猫”的容器。别人还是可以轻易将“狗”放入容器。
2. 因为丢失了类型信息，容器只知道它保存的是 **Object** 类型的引用。在使用容器中的元素前必须要做类型转换操作。

好在 Java 并不会让你误用容器中的对象。如果你将“狗”丢入存放“猫”的容器，然后将其中的每件东西都作为“猫”，当你将指向“狗”的引用取出容器，类型转换为“猫”时会收到 **RuntimeException** 异常。

下面的例子用到了最基本、最常用的容器 **ArrayList**。对于初学者，你可以将 **ArrayList** 看作是“能够自动扩展的数组”。**ArrayList** 用起来很方便：创建一个 **ArrayList**，使用 **add()**

添加对象，使用`get()`和索引取出对象——就像数组那样，只是不使用方括号⁷。`ArrayList`也有`size()`方法，可以告诉你容器中有多少元素，使你不至于超过边界而引发异常。

首先，创建 `Cat` 类和 `Dog` 类：

```
//: c11:Cats.java
package c11;

public class Cat {
    private int catNumber;
    public Cat(int i) { catNumber = i; }
    public void id() {
        System.out.println("Cat #" + catNumber);
    }
} ///:~
```

```
//: c11:Dogs.java
package c11;

public class Dog {
    private int dogNumber;
    public Dog(int i) { dogNumber = i; }
    public void id() {
        System.out.println("Dog #" + dogNumber);
    }
} ///:~
```

`Cat` 和 `Dog` 被添入容器，然后再取出来：

```
//: c11:CatsAndDogs.java
// Simple container example.
// {ThrowsException}
package c11;
import java.util.*;

public class CatsAndDogs {
    public static void main(String[] args) {
        List cats = new ArrayList();
        for(int i = 0; i < 7; i++)
            cats.add(new Cat(i));
        // Not a problem to add a dog to cats:
```

⁷此处如果可用操作符重载就太好了。

```

cats.add(new Dog(7));
for(int i = 0; i < cats.size(); i++)
    ((Cat)cats.get(i)).id();
    // Dog is detected only at run time
}
} ///:~

```

Cat 与 Dog 是不同的类：除了都是 Object 的子类，它们之间没有相同之处。（如果不明确地指出你的类从何处继承而来，则自动继承自 Object）因为 ArrayList 保存 Object，所以 you 不仅可以用 ArrayList 的 add() 方法将 Cat 添入容器，也可以将 Dog 添入同一个容器而不会引起任何编译期或运行期错误。当你使用 get() 方法从 ArrayList 中取出你以为是 Cat 的元素时，得到的只是 Object 的引用，需要做类型转换为 Cat。于是使用圆括号将整个表达式括住，强迫做类型转换，才能使用 Cat 的方法 id()。否则是一个语法错误。在程序运行的时候，当你将 Dog 对象作类型转换为 Cat 时，会收到异常。

这不只是个小麻烦，它有可能造成很难发现的错误。如果程序在某处（或多处）向容器插入对象，而在程序的另一处捕获到异常，发现有错误的对象被置入容器，然后你必须找出这个错误的插入是在哪里发生的。多数时候这并不是严重问题，但你应该留意这种可能性。

有时候它也能工作

某些情况下，即使没有将对象转回原本的类型，仍能正常工作。有一种特殊情况：编译器对 String 类有特别的支持，使其运作自如。如果编译器需要的是 String 对象，而它并没有得到，编译器就自动调用 toString() 方法，这是定义在 Object 中的方法，可以被任意的 Java 类重载。此方法生成编译器需要的 String 对象，可以用在任何需要的地方。

因此要打印你的类的对象，只需重载类的 toString() 方法，如下例所示：

```

//: c11:Mouse.java
// Overriding toString().

public class Mouse {
    private int mouseNumber;
    public Mouse(int i) { mouseNumber = i; }
    // Override Object.toString():
    public String toString() {
        return "This is Mouse #" + mouseNumber;
    }
    public int getNumber() { return mouseNumber; }
} ///:~

```

```

//: c11:MouseTrap.java

```

```

public class MouseTrap {
    static void caughtYa(Object m) {
        Mouse mouse = (Mouse)m; // Cast from Object
        System.out.println("Mouse: " + mouse.getNumber());
    }
} ///:~

//: c11:WorksAnyway.java
// In special cases, things just seem to work correctly.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class WorksAnyway {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        List mice = new ArrayList();
        for(int i = 0; i < 3; i++)
            mice.add(new Mouse(i));
        for(int i = 0; i < mice.size(); i++) {
            // No cast necessary, automatic
            // call to Object.toString():
            System.out.println("Free mouse: " + mice.get(i));
            MouseTrap.caughtYa(mice.get(i));
        }
        monitor.expect(new String[] {
            "Free mouse: This is Mouse #0",
            "Mouse: 0",
            "Free mouse: This is Mouse #1",
            "Mouse: 1",
            "Free mouse: This is Mouse #2",
            "Mouse: 2"
        });
    }
} ///:~

```

可以看到 `Mouse` 重载了 `toString()` 方法。在 `main()` 的第二个 `for` 循环中有这样的语句：

```
System.out.println("Free mouse: " + mice.get(i));
```

编译器期待 '+' 号之后是一个 `String` 对象。而 `get()` 返回一个 `Object`，编译器为了得到所需的 `String` 会隐式地调用 `toString()`。可惜这种神奇的工作方式仅限于 `String`，对其他类型无效。

MouseTrap 中用到了第二种隐藏的类型转换。方法 caughtYa() 不接受 Mouse，只接受 Object，然后类型转换为 Mouse。此方法相当专横，由于是接受 Object，所以任何东西都可以传入此方法。然而如果类型转换不正确——即如果传递了错误的类型——会在运行期收到异常。这虽然没有编译期做类型检查那么好，但也算健壮。

```
MouseTrap.caughtYa(mice.get(i));
```

注意，使用此方法时不必做类型转换。

制作一个类型明确的 ArrayList

如果你对 ArrayList 不能保存类型信息还不死心。那么还有一个更牢靠的解决方式，使用 ArrayList 生成一个新的类，只接受和返回你指定的类型：

```
//: c11:MouseListener.java
// A type-conscious List.
import java.util.*;

public class MouseList {
    private List list = new ArrayList();
    public void add(Mouse m) { list.add(m); }
    public Mouse get(int index) {
        return (Mouse)list.get(index);
    }
    public int size() { return list.size(); }
} ///:~
```

下面测试此新容器：

```
//: c11:MouseListenerTest.java
import com.bruceeckel.simpletest.*;

public class MouseListTest {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        MouseList mice = new MouseList();
        for(int i = 0; i < 3; i++)
            mice.add(new Mouse(i));
        for(int i = 0; i < mice.size(); i++)
            MouseTrap.caughtYa(mice.get(i));
        monitor.expect(new String[] {
            "Mouse: 0",
            "Mouse: 1",
            "Mouse: 2"
        });
    }
}
```

```
    });  
    }  
} ///:~
```

这同前面的例子很相似，除了新的 `MouseListener` 类有一个 `private` 成员 `ArrayList`。此新容器的方法也与 `ArrayList` 相似，但是不接受也不返回通用的 `Object`，而其能够接受和返回的只能是 `Mouse` 对象。

注意，如果 `MouseListener` 是继承自 `ArrayList`，那么方法 `add(Mouse)` 只是简单地重载了 `add(Object)`，对于要添加的对象的类型仍然没有严格的限制，所以也就无法达到目的。而如果使用组合（composition），`MouseListener` 将直接使用 `ArrayList`，在将工作交给 `ArrayList` 之前，`MouseListener` 将做一些处理类型的操作。

因为 `MouseListener` 只接受 `Mouse`，所以如果你使用下面语句：

```
mice.add(new Pigeon());
```

会在编译期得到错误信息。从代码的角度来说，此方法虽然沉闷乏味，但如果你用了错误的类型，它立刻就能告诉你。

注意，这里使用 `get()` 时无需做类型转换，它总是返回 `Mouse`。

参数化类型

此类问题并不罕见。许多情况下会需要基于某种类型而生成新的类型，如果编译期可以获得特定类型的信息，将会很有用。这就是“参数化类型（*parameterized type*）”的概念。`C++` 使用 `template`（模板）为其直接提供语言级支持。而 `Java JDK 1.5` 可能会提供 `generic`——`Java` 版的参数化类型。

迭代器

任何容器都必须有方法可以将东西放进去，然后有方法将东西取出来。毕竟，存放事物是容器最基本的工作。对于 `ArrayList`，`add()` 是插入对象的方法，而 `get()` 是取出元素的方法之一。`ArrayList` 很灵活，可以随时选取任意元素，或使用不同的下标一次选取多个元素。

如果你从更高层的角度思考，会发现这里有个缺点：要使用容器必须知道其中元素确切的类型。起初看起来这没什么不好，但是考虑下面的情况：如果原本是使用 `ArrayList`，但是后来考虑到容器的特点，你想换用 `Set`，应该怎么做？或者你打算写通用的代码，它只是使用容器，不知道或不关心容器的类型，那么如何才能不重写代码就可以应用于不同类型的容器呢？

迭代器的概念（也是一种设计模式）可以用来达成此目的。迭代器是一个对象，它的工作是遍历并选择序列中的对象。客户端程序员不关心序列底层的结构。此外，迭代器通常被

称为“轻量级”对象：创建它的代价小。因此，经常可以见到对迭代器有些奇怪的限制。例如，某些迭代器只能单向移动。

Java 的 `Iterator` 就是迭代器受限制的例子，它只能用来：

1. 使用方法 `iterator()` 要求容器返回一个 `Iterator`。第一次调用 `Iterator` 的 `next()` 方法时，它返回序列的第一个元素。
2. 使用 `next()` 获得序列中的下一个元素。
3. 使用 `hasNext()` 检查序列中是否还有元素。
4. 使用 `remove()` 将上一次返回的元素从迭代器中移除。

`Iterator` 能做的也就是这些了。它是迭代器最简单的实现，但是已经很有用了（而且还有为 `List` 设计的更强大的 `ListIterator`）。为观察它的工作方式，我们先复习 `CatsAndDogs.java` 程序。原先的版本是用方法 `get()` 来选择每个元素，下面例子是使用 `Iterator`：

```
//: c11:CatsAndDogs2.java
// Simple container with Iterator.
package c11;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class CatsAndDogs2 {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        List cats = new ArrayList();
        for(int i = 0; i < 7; i++)
            cats.add(new Cat(i));
        Iterator e = cats.iterator();
        while(e.hasNext())
            ((Cat)e.next()).id();
    }
} ///:~
```

可以看到，程序最后几行不再使用 `for` 循环，而是使用 `Iterator` 遍历整个序列。有了 `Iterator` 就不必为容器中元素的数量操心，由 `hasNext()` 和 `next()` 为你照看着。

下面是创建一个通用的打印方法的例子：

```
//: c11:Printer.java
// Using an Iterator.
import java.util.*;

public class Printer {
    static void printAll(Iterator e) {
```

```

        while(e.hasNext())
            System.out.println(e.next());
    }
} ///:~

```

请仔细观察 `printAll()` 方法。注意，其中没有关于序列类型的信息，只有一个迭代器 `Iterator`，而且对于序列，知道此 `Iterator` 就足够了：通过它可以取得下一个对象，还可以知道是否到达了序列的底部。像这种“一次取出容器中的所有元素，然后一个一个进行单独处理”的思想是很有用的，并将贯穿全书。

这个例子间接地使用了 `Object.toString()` 方法，因而更具通用性。方法 `println()` 被所有基本类型和 `Object` 都重载过；对每种情况，都能自动调用相应的 `toString()` 方法生成 `String`。

虽然不必要，但你仍然可以明确地作类型转换，效果与调用 `toString()` 相同：

```

System.out.println((String)e.next());

```

通常 `Object` 提供的方法并不能满足需求，所以又要考虑类型转换的问题了。你得假设你已经取得了某个特定类型的序列的 `Iterator`，并将对结果对象作类型转换（如果类型错误会得到一个运行期异常）。

我们可以通过打印 `Hamster` 试试看：

```

///: c11:Hamster.java

public class Hamster {
    private int hamsterNumber;
    public Hamster(int hamsterNumber) {
        this.hamsterNumber = hamsterNumber;
    }
    public String toString() {
        return "This is Hamster #" + hamsterNumber;
    }
} ///:~

```

```

///: c11:HamsterMaze.java
// Using an Iterator.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class HamsterMaze {
    private static Test monitor = new Test();

```

```

public static void main(String[] args) {
    List list = new ArrayList();
    for(int i = 0; i < 3; i++)
        list.add(new Hamster(i));
    Printer.printAll(list.iterator());
    monitor.expect(new String[] {
        "This is Hamster #0",
        "This is Hamster #1",
        "This is Hamster #2"
    });
}
} ///:~

```

可以编写 `printAll()`，使它接受一个 `Collection` 对象为参数，以代替 `Iterator`，但是后者有更好的去耦性。

无意识中造成的递归

由于 Java 标准容器（与其他的类一样）继承自 `Object`，所以它们都有 `toString()` 方法。重载过的 `toString()` 生成可以代表容器自身的 `String`，以及容器持有的对象。例如，`ArrayList` 的 `toString()` 方法遍历 `ArrayList` 的所有元素，对每个元素都调用 `toString()`。假设你要打印类的地址，见下例，你可能会简单地使用 `this` 关键字（C++ 程序员会特别倾向于这个方法）：

```

///: c11:InfiniteRecursion.java
// Accidental recursion.
// {RunByHand}
import java.util.*;

public class InfiniteRecursion {
    public String toString() {
        return " InfiniteRecursion address: " + this + "\n";
    }

    public static void main(String[] args) {
        List v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new InfiniteRecursion());
        System.out.println(v);
    }
} ///:~

```


如果直接创建一个 `InfiniteRecursion` 对象，然后打印它，你会收到一个无穷无尽的异常序列。如果将 `InfiniteRecursion` 对象放入 `ArrayList`，然后打印 `ArrayList` 也会如此。问题在于 `String` 的自动类型转换，如果你这样写：

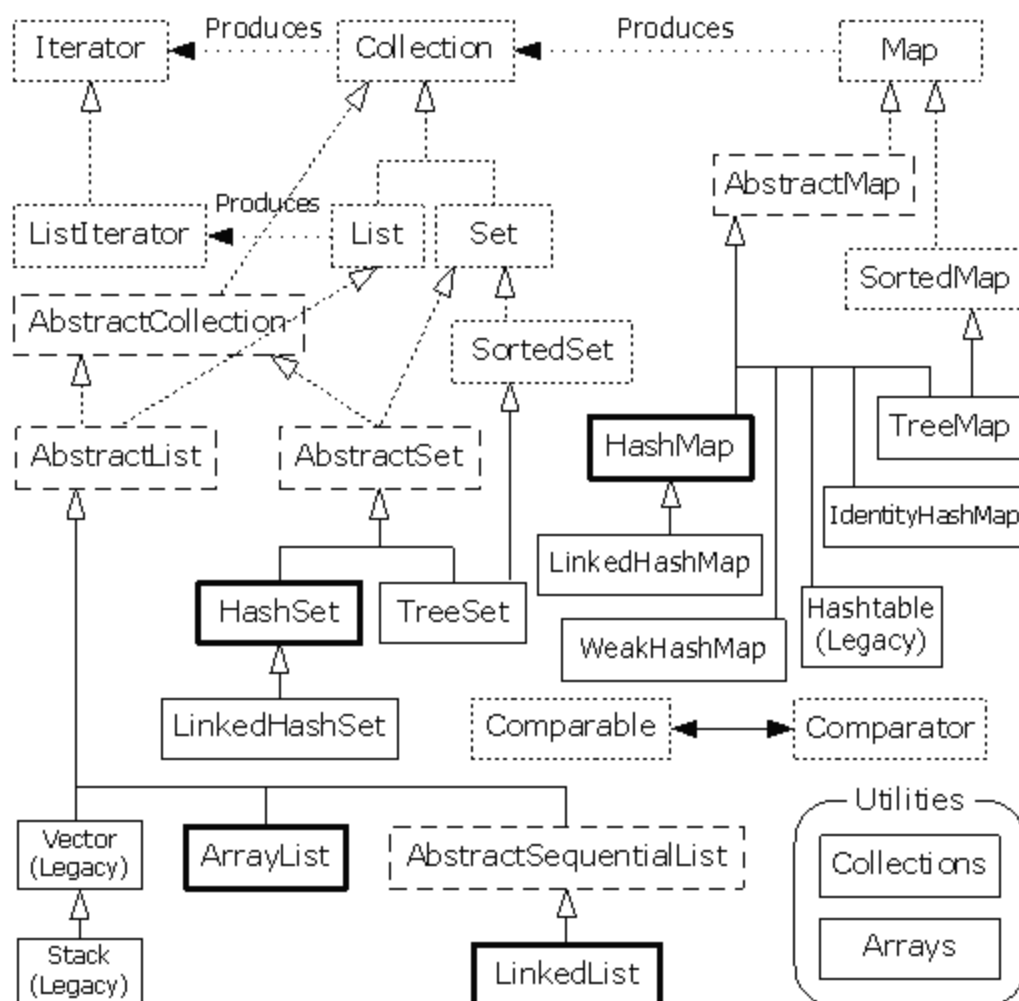
```
"InfiniteRecursion address: " + this
```

编译器见到 `String` 后跟着一个 '+' 号，而 '+' 后的对象却不是 `String`，于是编译器尝试将 `this` 转变成 `String` 类型。此类型转换操作调用的是 `toString()` 方法，于是产生递归调用。

在此例中，如果你确实想打印对象的地址，应该调用 `Object` 的 `toString()` 方法，它专门做此工作。因此使用 `super.toString()` 取代 `this` 即可。

容器的分类法

为满足编程的需求，`Collection` 和 `Map` 有多种不同的实现。下图对于掌握 Java（JDK 1.4）的众多容器将很有帮助：



这张图初看起来有点吓人，熟悉之后你会发现其实只有三种容器：**Map**，**List** 和 **Set**，它们各有两到三个实现版本。常用的容器用黑色粗线框表示。看到这里，容器应该没有那么可怕了吧。

点线方框表示接口，虚线方框表示抽象类，实线方框表示普通的（具体的）类。点线箭头代表特定的类实现一个接口（若是抽象类，则表示部分实现了接口）。实线箭头表示一个类可以生成箭头所指向类的对象。例如，任意的 **Collection** 可以生成 **Iterator**，而 **List** 可以生成 **ListIterator**（也能生成普通的 **Iterator**，因为 **List** 继承自 **Collection**）。

与持有对象有关的接口是 **Collection**，**List**，**Set** 和 **Map**。最理想的情况是，你的代码只与这些接口打交道，仅在创建容器的时候说明容器的特定类型。因此可以这样创建一个 **List**：

```
List x = new LinkedList();
```

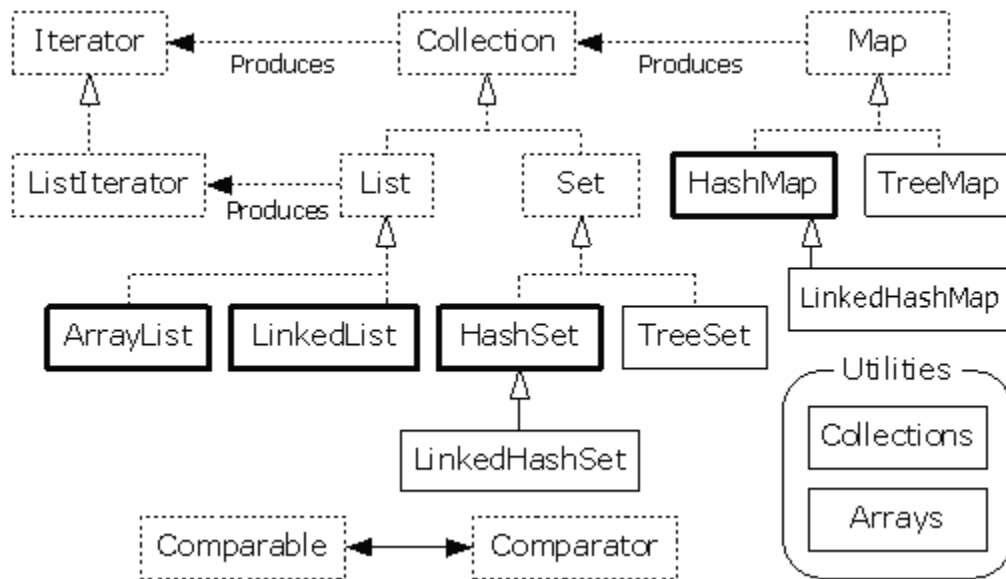
当然，你可以将 **x** 定义为 **LinkedList**（代替通用的 **List**），使 **x** 带有明确的类型信息。接口的优美之处（或者说目的）在于，如果你决定改变当前的实现，只需要在创建的位置做些修改即可，就像这样：

```
List x = new ArrayList();
```

而其它代码则无需改动（迭代器也能实现部分通用性）。

在容器类层次结构图中，可以看到有许多以“**Abstract**”开头的名字，这有点令人迷惑。其实它们只是部分实现了某个特定接口的简单工具而已。举个例子，如果要制作自己的 **Set**，一般不会直接继承 **Set** 接口，然后实现所有的方法。而是应该继承 **AbstractSet**，只为自己的新类作最必要的工作。不过，容器类库已经包含足够的功能来满足你的需要了。所以，对我们而言，可以忽略那些以“**Abstract**”开头的类。

因此，再看上图时，我们只关心顶层的接口和“具体类”（由实线方框表示）。典型情况是你生成一个“具体类”的对象，然后将它向上转型为对应的接口，在代码中使用接口操作它。此外，新的程序不需要使用过时的容器。于是可简化上图：



现在的图只包含通常会用到的接口与类，以及我们要学习的元素。注意，**WeakHashMap** 和 JDK 1.4 中的 **IdentityHashMap** 没有包含在图中，因为它们是有特殊用途的工具，很少会用到。

下面是个简单的例子，使用 **String** 对象填充 **Collection**（使用的是 **ArrayList**），并且打印 **Collection** 中的每个元素：

```

//: c11:SimpleCollection.java
// A simple example using Java 2 Collections.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class SimpleCollection {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        // Upcast because we just want to
        // work with Collection features
        Collection c = new ArrayList();
        for(int i = 0; i < 10; i++)
            c.add(Integer.toString(i));
        Iterator it = c.iterator();
        while(it.hasNext())
            System.out.println(it.next());
        monitor.expect(new String[] {
            "0",
            "1",
            "2",
            "3",
            "4",
            "5",
        })
    }
}

```

```

        "6",
        "7",
        "8",
        "9"
    });
}
} ///:~

```

`main()`中第一行代码生成一个 `ArrayList` 对象，并向上类型转换为 `Collection`。因为此例只用到了 `Collection` 的方法，所以任何 `Collection` 子类的对象都能工作，不过 `ArrayList` 是最常用的 `Collection`。

方法 `add()` 正如其名，其功能是将新元素添加到 `Collection` 中。不过说明文档更谨慎地对 `add()` 做出了定义：“确保容器包含特定的元素”。这是考虑到了 `Set` 的含义，它不添加重复元素。对于 `ArrayList`，或者任意的 `List`，方法 `add()` 都表示“将元素存入容器”，因为 `List` 不在乎元素是否重复。

所有的 `Collection` 都能通过 `iterator()` 方法生成 `Iterator`。此例生成了一个 `Iterator`，用来遍历 `Collection` 并打印每个元素。

Collection 的功能方法

下表列出了你可以通过 `Collection` 执行的所有操作（不包括自 `Object` 继承而来的方法），因此，它们也就是可以通过 `Set` 或 `List` 执行的所有操作（`List` 还有额外的功能。）`Map` 不是继承自 `Collection`，所以另行介绍。

boolean add(Object)	确保容器持有此参数。如果没有将此参数添加进容器则返回 <code>false</code> 。（这是个“可选”的方法，稍后会解释。）
boolean addAll(Collection)	添加参数中的所有元素。只要添加了任意元素就返回 <code>true</code> 。（“可选”）
void clear()	移除容器中的所有元素。（“可选”）
boolean contains(Object)	如果容器已经持有参数则返回 <code>true</code>
boolean containsAll(Collection)	如果容器持有参数中的所有元素则返回 <code>true</code>
boolean isEmpty()	容器中没有元素时返回 <code>true</code>
Iterator iterator()	返回一个 <code>Iterator</code> ，可以用来遍历容器中的元素。
boolean remove(Object)	如果参数在容器中，则移除此元素的一个实例。如果做了移除动作则返回 <code>true</code> 。（“可选”）

boolean removeAll(Collection)	移除参数中的所有元素。只要有移除动作发生就返回 true。（“可选”）
boolean retainAll(Collection)	只保存参数中的元素（应用集合论的“交集”概念）。只要 Collection 发生了改变就返回 true。（“可选”）
int size()	返回容器中元素的数目
Object[] toArray()	返回一个数组，包含容器中的所有元素。
Object[] toArray(Object[] a)	返回一个数组，包含容器中的所有元素，其类型与数组 a 的类型相同，而不是单纯的 Object（你必须对此数组做类型转换）。

请注意，其中不包括随机访问所选择元素的 `get()` 方法。因为 `Collection` 包括 `Set`，而 `Set` 是自己维护内部顺序的（这使得随机访问变得没有意义）。因此，如果你想检查 `Collection` 中的元素，必须使用迭代器。

下面的例子示范了所有这些方法。任何实现了 `Collection` 的类都可以使用这些方法，示例总是使用 `ArrayList`，作为各种 `Collection` 子类的“最小公分母（least-common denominator）”：

```

//: c11:Collection1.java
// Things you can do with all Collections.
import com.bruceeckel.simpletest.*;
import java.util.*;
import com.bruceeckel.util.*;

public class Collection1 {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Collection c = new ArrayList();
        Collections2.fill(c, Collections2.countries, 5);
        c.add("ten");
        c.add("eleven");
        System.out.println(c);
        // Make an array from the List:
        Object[] array = c.toArray();
        // Make a String array from the List:
        String[] str = (String[])c.toArray(new String[1]);
        // Find max and min elements; this means
        // different things depending on the way
        // the Comparable interface is implemented:
        System.out.println("Collections.max(c) = " +
            Collections.max(c));
        System.out.println("Collections.min(c) = " +

```

```

        Collections.min(c));
// Add a Collection to another Collection
Collection c2 = new ArrayList();
Collections2.fill(c2, Collections2.countries, 5);
c.addAll(c2);
System.out.println(c);
c.remove(CountryCapitals.pairs[0][0]);
System.out.println(c);
c.remove(CountryCapitals.pairs[1][0]);
System.out.println(c);
// Remove all components that are
// in the argument collection:
c.removeAll(c2);
System.out.println(c);
c.addAll(c2);
System.out.println(c);
// Is an element in this Collection?
String val = CountryCapitals.pairs[3][0];
System.out.println("c.contains(" + val + ") = "
    + c.contains(val));
// Is a Collection in this Collection?
System.out.println(
    "c.containsAll(c2) = " + c.containsAll(c2));
Collection c3 = ((List)c).subList(3, 5);
// Keep all the elements that are in both
// c2 and c3 (an intersection of sets):
c2.retainAll(c3);
System.out.println(c);
// Throw away all the elements
// in c2 that also appear in c3:
c2.removeAll(c3);
System.out.println("c.isEmpty() = " + c.isEmpty());
c = new ArrayList();
Collections2.fill(c, Collections2.countries, 5);
System.out.println(c);
c.clear(); // Remove all elements
System.out.println("after c.clear():");
System.out.println(c);
monitor.expect(new String[] {
    "[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, " +
    "ten, eleven]",
    "Collections.max(c) = ten",
    "Collections.min(c) = ALGERIA",
    "[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, " +

```

```

        "ten, eleven, BURUNDI, CAMEROON, CAPE VERDE, " +
        "CENTRAL AFRICAN REPUBLIC, CHAD]",
        "[ANGOLA, BENIN, BOTSWANA, BURKINA FASO, ten, " +
        "eleven, BURUNDI, CAMEROON, CAPE VERDE, " +
        "CENTRAL AFRICAN REPUBLIC, CHAD]",
        "[BENIN, BOTSWANA, BURKINA FASO, ten, eleven, " +
        "BURUNDI, CAMEROON, CAPE VERDE, " +
        "CENTRAL AFRICAN REPUBLIC, CHAD]",
        "[BENIN, BOTSWANA, BURKINA FASO, ten, eleven]",
        "[BENIN, BOTSWANA, BURKINA FASO, ten, eleven, " +
        "BURUNDI, CAMEROON, CAPE VERDE, " +
        "CENTRAL AFRICAN REPUBLIC, CHAD]",
        "c.contains(BOTSWANA) = true",
        "c.containsAll(c2) = true",
        "[BENIN, BOTSWANA, BURKINA FASO, ten, eleven, " +
        "BURUNDI, CAMEROON, CAPE VERDE, " +
        "CENTRAL AFRICAN REPUBLIC, CHAD]",
        "c.isEmpty() = false",
        "[COMOROS, CONGO, DJIBOUTI, EGYPT, " +
        "EQUATORIAL GUINEA]",
        "after c.clear():",
        "[]"
    });
}
} ///:~

```

`ArrayList` 被创建用来保存不同的数据集，然后向上类型转化为 `Collection`，所以很明显，代码只是在操作 `Collection` 接口。`main()` 用简单的练习展示了 `Collection` 中的所有方法。

以下几节将介绍 `List`、`Set` 和 `Map` 的各种实现，每种情况都会（以星号）标出默认的选择。你会发现我们没有讲到已经过时的 `Vector`、`Stack` 和 `Hashtable`，因为所有情况下都应该首选 Java 2 的容器类。

List 的功能方法

基本的 `List` 很容易使用，如同你见过的 `ArrayList`。虽然大多数时候只是使用 `add()` 添加对象，使用 `get()` 一次取出一个元素，或者使用 `iterator()` 获取 `Iterator`，不过 `List` 其实还有许多其他很有用的方法。

此外，实际上有两种 `List`：一种是基本的 `ArrayList`，其优点在于随机访问元素，另一种是更强大的 `LinkedList`，它并不是为快速随机访问设计的，而是具有一套更通用的方法。

List

次序是 `List` 最重要的特点；它保证维护元素特定的顺序。

(interface)	List 为 Collection 添加了许多方法，使得能够向 List 中间插入与移除元素。（这只推荐 LinkedList 使用。）一个 List 可以生成 ListIterator，使用它可以两个方向遍历 List，也可以从 List 中间插入和移除元素。
ArrayList*	由数组实现的 List。允许对元素进行快速随机访问，但是向 List 中间插入与移除元素的速度很慢。ListIterator 只应该用来由后向前遍历 ArrayList，而不是用来插入和移除元素，因为那比 LinkedList 开销要大很多。
LinkedList	对顺序访问进行了优化，向 List 中间插入与删除的开销并不大。随机访问则相对较慢。（使用 ArrayList 代替。）还具有下列方法：addFirst()，addLast()，getFirst()，getLast()，removeFirst()，和 removeLast()，这些方法（没有在任何接口或基类中定义过）使得 LinkedList 可以当作堆栈、队列和双向队列使用。

下面例子中的每个方法都涵盖了一组不同的动作：basicTest()中包含每个 List 都可以执行的操作；iterMotion()使用 Iterator 遍历元素；对应的 iterManipulation()使用 Iterator 修改元素；用以查看 List 操作效果的 testVisual()；还有一些 LinkedList 专用的操作。

```

//: c11:List1.java
// Things you can do with Lists.
import java.util.*;
import com.bruceeckel.util.*;

public class List1 {
    public static List fill(List a) {
        Collections2.countries.reset();
        Collections2.fill(a, Collections2.countries, 10);
        return a;
    }

    private static boolean b;
    private static Object o;
    private static int i;
    private static Iterator it;
    private static ListIterator lit;
    public static void basicTest(List a) {
        a.add(1, "x"); // Add at location 1
        a.add("x"); // Add at end
        // Add a collection:
        a.addAll(fill(new ArrayList()));
        // Add a collection starting at location 3:
        a.addAll(3, fill(new ArrayList()));
        b = a.contains("1"); // Is it in there?
    }
}

```



```

// Is the entire collection in there?
b = a.containsAll(fill(new ArrayList()));
// Lists allow random access, which is cheap
// for ArrayList, expensive for LinkedList:
o = a.get(1); // Get object at location 1
i = a.indexOf("1"); // Tell index of object
b = a.isEmpty(); // Any elements inside?
it = a.iterator(); // Ordinary Iterator
lit = a.listIterator(); // ListIterator
lit = a.listIterator(3); // Start at loc 3
i = a.lastIndexOf("1"); // Last match
a.remove(1); // Remove location 1
a.remove("3"); // Remove this object
a.set(1, "y"); // Set location 1 to "y"
// Keep everything that's in the argument
// (the intersection of the two sets):
a.retainAll(fill(new ArrayList()));
// Remove everything that's in the argument:
a.removeAll(fill(new ArrayList()));
i = a.size(); // How big is it?
a.clear(); // Remove all elements
}

public static void iterMotion(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
    i = it.previousIndex();
}

public static void iterManipulation(List a) {
    ListIterator it = a.listIterator();
    it.add("47");
    // Must move to an element after add():
    it.next();
    // Remove the element that was just produced:
    it.remove();
    // Must move to an element after remove():
    it.next();
    // Change the element that was just produced:
    it.set("47");
}

public static void testVisual(List a) {

```

```

System.out.println(a);
List b = new ArrayList();
fill(b);
System.out.print("b = ");
System.out.println(b);
a.addAll(b);
a.addAll(fill(new ArrayList()));
System.out.println(a);
// Insert, remove, and replace elements
// using a ListIterator:
ListIterator x = a.listIterator(a.size()/2);
x.add("one");
System.out.println(a);
System.out.println(x.next());
x.remove();
System.out.println(x.next());
x.set("47");
System.out.println(a);
// Traverse the list backwards:
x = a.listIterator(a.size());
while(x.hasPrevious())
    System.out.print(x.previous() + " ");
System.out.println();
System.out.println("testVisual finished");
}

// There are some things that only LinkedLists can do:
public static void testLinkedList() {
    LinkedList ll = new LinkedList();
    fill(ll);
    System.out.println(ll);
    // Treat it like a stack, pushing:
    ll.addFirst("one");
    ll.addFirst("two");
    System.out.println(ll);
    // Like "peeking" at the top of a stack:
    System.out.println(ll.getFirst());
    // Like popping a stack:
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());
    // Treat it like a queue, pulling elements
    // off the tail end:
    System.out.println(ll.removeLast());
    // With the above operations, it's a dequeue!
    System.out.println(ll);
}

```

```

    }
    public static void main(String[] args) {
        // Make and fill a new list each time:
        basicTest(fill(new LinkedList()));
        basicTest(fill(new ArrayList()));
        iterMotion(fill(new LinkedList()));
        iterMotion(fill(new ArrayList()));
        iterManipulation(fill(new LinkedList()));
        iterManipulation(fill(new ArrayList()));
        testVisual(fill(new LinkedList()));
        testLinkedList();
    }
} ///:~

```

`basicTest()` 和 `iterMotion()` 中的方法调用只是为了演示正确的语法，虽然取得了返回值，却没有使用。某些情况则根本没有捕获返回值。使用这些方法前，应该从 java.sun.com 查询 JDK 帮助文档，以充分了解各种方法的用途。

请记住，容器只是用来持有对象的储藏盒而已。如果此盒子能够满足所有的需要，就不必在意它是如何实现的（这也是使用各种类型对象的基本思想）。如果在开发环境中，其它的因素是性能开销的主要来源，那么 `ArrayList` 与 `LinkedList` 之间的开销差异就不重要了，无论使用哪一种都可以。你甚至可以想象它是一种“完美”的容器，能够根据不同的使用方式自动改变容器的底层实现。

使用 `LinkedList` 制作一个栈

“栈（stack）”通常是指“后进先出”（LIFO）的容器。最后“push”入栈的元素，第一个“pop”出栈。与 Java 中其他容器一样，进栈出栈的都是 `Object`，因此从栈中取出元素后必须做类型转换，除非你只是使用 `Object` 具有的操作。

`LinkedList` 具有能够直接实现栈的所有功能的方法，因此你可以直接将 `LinkedList` 作为栈使用。不过，有时一个真正的“栈”更能把事情讲清楚：

```

///: c11:StackL.java
// Making a stack from a LinkedList.
import com.bruceeckel.simpletest.*;
import java.util.*;
import com.bruceeckel.util.*;

public class StackL {
    private static Test monitor = new Test();
    private LinkedList list = new LinkedList();
    public void push(Object v) { list.addFirst(v); }
    public Object top() { return list.getFirst(); }
}

```

```

public Object pop() { return list.removeFirst(); }
public static void main(String[] args) {
    StackL stack = new StackL();
    for(int i = 0; i < 10; i++)
        stack.push(Collections2.countries.next());
    System.out.println(stack.top());
    System.out.println(stack.top());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    monitor.expect(new String[] {
        "CHAD",
        "CHAD",
        "CHAD",
        "CENTRAL AFRICAN REPUBLIC",
        "CAPE VERDE"
    });
}
} ///:~

```

如果只是需要“栈”的功能，此处继承 `LinkedList` 就不恰当了，因为它同时会具有 `LinkedList` 的其他方法（后面将会看到，Java 1.0 类库的设计师在设计 `Stack` 时就犯了这个错误）。

使用 `LinkedList` 制作一个队列

“队列（queue）”是一个“先进先出”（FIFO）容器。即从容器的一端放入事物，从另一端取出。因此事物放入容器的顺序与取出的顺序是相同的。`LinkedList` 提供了方法以支持“队列”的行为，因此可以用来制作 `Queue` 类：

```

///: c11:Queue.java
// Making a queue from a LinkedList.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class Queue {
    private static Test monitor = new Test();
    private LinkedList list = new LinkedList();
    public void put(Object v) { list.addFirst(v); }
    public Object get() { return list.removeLast(); }
    public boolean isEmpty() { return list.isEmpty(); }
    public static void main(String[] args) {
        Queue queue = new Queue();
        for(int i = 0; i < 10; i++)
            queue.put(Integer.toString(i));
    }
}

```

```

while(!queue.isEmpty())
    System.out.println(queue.get());
monitor.expect(new String[] {
    "0",
    "1",
    "2",
    "3",
    "4",
    "5",
    "6",
    "7",
    "8",
    "9"
});
}
} ///:~

```

你也可以很容易地通过 `LinkedList` 创建一个“双向队列”，它同“队列”一样，不过可以从两个方向添加和删除元素。

Set 的功能方法

`Set` 具有与 `Collection` 完全一样的接口，因此没有任何额外的功能，不像前面有两个不同的 `List`。实际上 `Set` 就是 `Collection`，只是行为不同。（这是继承与多态思想的典型应用：表现不同的行为。）`Set` 不保存重复的元素（至于如何判断元素相同则较为复杂，稍后便会看到）。

Set (interface)	存入 <code>Set</code> 的每个元素都必须是唯一的，因为 <code>Set</code> 不保存重复元素。加入 <code>Set</code> 的元素必须定义 <code>equals()</code> 方法以确保对象的唯一性。 <code>Set</code> 与 <code>Collection</code> 有完全一样的接口。 <code>Set</code> 接口不保证维护元素的次序。
HashSet *	为快速查找设计的 <code>Set</code> 。存入 <code>HashSet</code> 的对象必须定义 <code>hashCode()</code> 。
TreeSet	保持次序的 <code>Set</code> ，底层为树结构。使用它可以从 <code>Set</code> 中提取有序的序列。
LinkedHashSet (JDK 1.4)	具有 <code>HashSet</code> 的查询速度，且内部使用链表维护元素的顺序（插入的次序）。于是在使用迭代器遍历 <code>Set</code> 时，结果会按元素插入的次序显示。

下例并没有演示 `Set` 能够做的所有事情，因为它的接口与 `Collection` 相同，所以有些在前例中练习过了。这里演示的只是 `Set` 独有的行为：

```

///: c11:Set1.java

```

```

// Things you can do with Sets.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class Set1 {
    private static Test monitor = new Test();
    static void fill(Set s) {
        s.addAll(Arrays.asList(
            "one two three four five six seven".split(" ")));
    }
    public static void test(Set s) {
        // Strip qualifiers from class name:
        System.out.println(
            s.getClass().getName().replaceAll("\\w+\\. ", ""));
        fill(s); fill(s); fill(s);
        System.out.println(s); // No duplicates!
        // Add another set to this one:
        s.addAll(s);
        s.add("one");
        s.add("one");
        s.add("one");
        System.out.println(s);
        // Look something up:
        System.out.println("s.contains(\"one\"): " +
            s.contains("one"));
    }
    public static void main(String[] args) {
        test(new HashSet());
        test(new TreeSet());
        test(new LinkedHashSet());
        monitor.expect(new String[] {
            "HashSet",
            "[one, two, five, four, three, seven, six]",
            "[one, two, five, four, three, seven, six]",
            "s.contains(\"one\"): true",
            "TreeSet",
            "[five, four, one, seven, six, three, two]",
            "[five, four, one, seven, six, three, two]",
            "s.contains(\"one\"): true",
            "LinkedHashSet",
            "[one, two, three, four, five, six, seven]",
            "[one, two, three, four, five, six, seven]",
            "s.contains(\"one\"): true"
        });
    }
}

```

```
    }
} ///:~
```

例子中向 **Set** 添加了重复的值，但是打印的结果说明，对每一个值 **Set** 只接受一份实例。

运行此程序，你会注意到，**HashSet** 维护的元素次序不同于 **TreeSet** 和 **LinkedHashSet**，因为它们保存元素的方式各有不同，使得以后还能找到元素。（**TreeSet** 采用红黑树的数据结构排序元素，**HashSet** 则采用散列函数，这是专门为快速查询设计的。**LinkedHashSet** 内部使用散列以加快查询速度，同时使用链表维护元素的次序，使得看起来元素是以插入的顺序保存的。）生成自己的类时，注意 **Set** 需要维护元素的存储顺序，这意味着你必须实现 **Comparable** 接口，并且定义 **compareTo()** 方法。参见下例：

```
///: c11:Set2.java
// Putting your own type in a Set.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class Set2 {
    private static Test monitor = new Test();
    public static Set fill(Set a, int size) {
        for(int i = 0; i < size; i++)
            a.add(new MyType(i));
        return a;
    }
    public static void test(Set a) {
        fill(a, 10);
        fill(a, 10); // Try to add duplicates
        fill(a, 10);
        a.addAll(fill(new TreeSet(), 10));
        System.out.println(a);
    }
    public static void main(String[] args) {
        test(new HashSet());
        test(new TreeSet());
        test(new LinkedHashSet());
        monitor.expect(new String[] {
            "[2 , 4 , 9 , 8 , 6 , 1 , 3 , 7 , 5 , 0 ]",
            "[9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1 , 0 ]",
            "[0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ]"
        });
    }
} ///:~
```

本章稍后会介绍如何定义 **equals()** 和 **hashCode()**。使用以上两种 **Set** 都必须为你的类定义 **equals()**，而 **hashCode()**，只在你的类会被 **HashSet** 用到的情况下才是必要的（这

种可能性很大，因为 `HashSet` 通常是使用 `Set` 的第一选择）。无论如何，作为一种编程风格，当你重载 `equals()` 的时候，就应该同时重载 `hashCode()`。本章稍后会详细介绍这个过程。

注意在 `compareTo()` 中，我并没有使用“简单而明显”的 `return i-i2`。虽然这是一个常犯的编程错误，但是如果 `i` 与 `i2` 正巧是“无符号”的 `int` 时（如果 Java 有“无符号”`unsigned` 这个关键字的话，其实没有），此语句也能正常工作。如果是 Java 的有符号整数 `int` 则无法确保正确，因为 `int` 不够大，不足以表现两个有符号整数 `int` 的差。例如 `i` 是很大的正整数，而 `j` 是很大的负整数，`i-j` 就会溢出并且返回负值，这就不正确了。

SortedSet

使用 `SortedSet`（`TreeSet` 是其唯一的实现），可以确保元素处于排序状态，还可以使用 `SortedSet` 接口提供的附加功能：

Comparator comparator()：返回当前 `Set` 使用的 `Comparator`，或者返回 `null`，表示以自然方式排序。

Object first()：返回容器中的第一个元素。

Object last()：返回容器中的最末一个元素。

SortedSet subSet(fromElement, toElement)：生成此 `Set` 的子集，范围从 `fromElement`（包含）到 `toElement`（不包含）。

SortedSet headSet(toElement)：生成此 `Set` 的子集，由小于 `toElement` 的元素组成。

SortedSet tailSet(fromElement)：生成此 `Set` 的子集，由大于或等于 `fromElement` 的元素组成。

下面是一个简单的示例：

```
//: c11:SortedSetDemo.java
// What you can do with a TreeSet.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class SortedSetDemo {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        SortedSet sortedSet = new TreeSet(Arrays.asList(
            "one two three four five six seven eight".split(" ")));
        System.out.println(sortedSet);
    }
}
```



```

Object
    low = sortedSet.first(),
    high = sortedSet.last();
System.out.println(low);
System.out.println(high);
Iterator it = sortedSet.iterator();
for(int i = 0; i <= 6; i++) {
    if(i == 3) low = it.next();
    if(i == 6) high = it.next();
    else it.next();
}
System.out.println(low);
System.out.println(high);
System.out.println(sortedSet.subSet(low, high));
System.out.println(sortedSet.headSet(high));
System.out.println(sortedSet.tailSet(low));
monitor.expect(new String[] {
    "[eight, five, four, one, seven, six, three, two]",
    "eight",
    "two",
    "one",
    "two",
    "[one, seven, six, three]",
    "[eight, five, four, one, seven, six, three]",
    "[one, seven, six, three, two]"
});
}
} ///:~

```

注意，SortedSet 的意思是“按比较函数对元素排序”，而不是指“元素插入的次序”。

Map 的功能方法

ArrayList 允许你使用数字从对象序列中选择元素，因此它在数字与对象之间建立了关联。如果要使用其他条件来从序列中选择对象，应该怎么做呢？“栈”是一个例子。它的选择条件是“最后一个加入栈中的东西”。由“从序列中进行选择”这种思想发展出了 map 这种强大的工具，也称作字典，或关联数组（曾在前面的章节中见过的

AssociativeArray.java 就是一个简单的例子）。从概念上讲，它看起来就像是 ArrayList，只是不再用数字下标查找对象，而是以另一个对象来进行查找。这是一种至关重要的编程技术。

Map 接口即是此概念在 Java 中的体现。方法 put(Object key, Object value) 添加一个“值”（value）（想要的东西）和与“值”相关联的“键”（key）（使用它来查找）。

方法 `get(Object key)` 返回与给定“键”相关联的“值”。可以用 `containsKey()` 和 `containsValue()` 测试 `Map` 中是否包含某个“键”或“值”。

标准的 Java 类库中包含了几种不同的 `Map`: `HashMap`, `TreeMap`, `LinkedHashMap`, `WeakHashMap`, `IdentityHashMap`。它们都有同样的基本接口 `Map`，但是行为、效率、排序策略、保存对象的生命周期和判定“键”等价的策略等各不相同。

执行效率是 `Map` 的一个大问题。看看 `get()` 要做哪些事，就会明白为什么在 `ArrayList` 中搜索“键”是相当慢的。而这正是 `HashMap` 提高速度的地方。`HashMap` 使用了特殊的值，称作“散列码”（`hash code`），来取代对“键”的缓慢搜索。“散列码”是“相对唯一”用以代表对象的 `int` 值，它是通过将该对象的某些信息进行转换而生成的。所有 Java 对象都能产生散列码，因为 `hashCode()` 是定义在基类 `Object` 中的方法。`HashMap` 就是使用对象的 `hashCode()` 进行快速查询的。此方法能够显著提高性能。⁸

Map (interface)	维护“键值对”的关联性，使你可以通过“键”查找“值”。
HashMap *	<code>Map</code> 基于散列表的实现。（取代了 <code>Hashtable</code> 。）插入和查询“键值对”的开销是固定的。可以通过构造器设置容量 <code>capacity</code> 和负载因子 <code>load factor</code> ，以调整容器的性能。
LinkedHashMap (JDK 1.4)	类似于 <code>HashMap</code> ，但是迭代遍历它时，取得“键值对”的顺序是其插入次序，或者是最近最少使用（ <code>LRU</code> ）的次序。只比 <code>HashMap</code> 慢一点。而在迭代访问时反而更快，因为它使用链表维护内部次序。
TreeMap	基于红黑树数据结构的实现。查看“键”或“键值对”时，它们会被排序（次序由 <code>Comparable</code> 或 <code>Comparator</code> 决定，稍后会讨论）。 <code>TreeMap</code> 的特点在于，你得到的结果是经过排序的。 <code>TreeMap</code> 是唯一的带有 <code>subMap()</code> 方法的 <code>Map</code> ，它可以返回一个子树。
WeakHashMap	弱键（ <code>weak key</code> ） <code>Map</code> ， <code>Map</code> 中使用的对象也被允许释放；这是为解决特殊问题设计的。如果没有 <code>map</code> 之外的引用指向某个“键”，则此“键”可以被垃圾收集器回收。
IdentityHashMap (JDK 1.4)	使用 <code>==</code> 代替 <code>equals()</code> 对“键”作比较的 <code>hash map</code> 。专为解决特殊问题而设计。

⁸ 如果这仍不能满足你对性能的要求，那么你还可以通过创建自己的 `Map` 来进一步提高查询速度，并且令新的 `Map` 只针对你使用的特定类型，这样可以避免与 `Object` 之间的类型转换操作。要到达更高的性能，速度狂们可以参考 Donald Knuth 的 *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*。使用数组代替溢出桶，这有两个好处：第一，可以针对磁盘存储方式做优化；第二，在创建和回收单独的记录时，能节约很多时间。

散列是 `map` 存储元素时最常用的方式。有时，你需要了解散列函数是如何工作的，因此我们稍后会看看散列函数的细节。

下面的例子使用了 `Collections2.fill()` 方法和先前定义的测试用数据集：

```
//: c11:Map1.java
// Things you can do with Maps.
import java.util.*;
import com.bruceeckel.util.*;

public class Map1 {
    private static Collections2.StringPairGenerator geo =
        Collections2.geography;
    private static Collections2.RandStringPairGenerator
        rsp = Collections2.rsp;
    // Producing a Set of the keys:
    public static void printKeys(Map map) {
        System.out.print("Size = " + map.size() + ", ");
        System.out.print("Keys: ");
        System.out.println(map.keySet());
    }
    public static void test(Map map) {
        // Strip qualifiers from class name:
        System.out.println(
            map.getClass().getName().replaceAll("\\w+\\. ", ""));
        Collections2.fill(map, geo, 25);
        // Map has 'Set' behavior for keys:
        Collections2.fill(map, geo.reset(), 25);
        printKeys(map);
        // Producing a Collection of the values:
        System.out.print("Values: ");
        System.out.println(map.values());
        System.out.println(map);
        String key = CountryCapitals.pairs[4][0];
        String value = CountryCapitals.pairs[4][1];
        System.out.println("map.containsKey(\"" + key +
            "\"): " + map.containsKey(key));
        System.out.println("map.get(\"" + key + "\"): "
            + map.get(key));
        System.out.println("map.containsValue(\""
            + value + "\"): " + map.containsValue(value));
        Map map2 = new TreeMap();
        Collections2.fill(map2, rsp, 25);
        map.putAll(map2);
        printKeys(map);
    }
}
```

```

        key = map.keySet().iterator().next().toString();
        System.out.println("First key in map: " + key);
        map.remove(key);
        printKeys(map);
        map.clear();
        System.out.println("map.isEmpty(): " + map.isEmpty());
        Collections2.fill(map, geo.reset(), 25);
        // Operations on the Set change the Map:
        map.keySet().removeAll(map.keySet());
        System.out.println("map.isEmpty(): " + map.isEmpty());
    }
    public static void main(String[] args) {
        test(new HashMap());
        test(new TreeMap());
        test(new LinkedHashMap());
        test(new IdentityHashMap());
        test(new WeakHashMap());
    }
} ///:~

```

`printKeys()` 和 `printValues()` 方法不仅很实用，而且还展示了如何由 `Map` 生成 `Collection`。方法 `keySet()` 返回由 `Map` 的“键”组成的 `Set`。类似的有方法 `values()`，它返回一个 `Collection`，包含 `Map` 中所有的“值”。（注意，“键”必须是唯一的，而“值”可以有重复。）由于这些 `Collection` 背后是由 `Map` 支持的，所以对 `Collection` 的任何改动都会反映到与之相关联的 `Map`。

此程序接下来直接示范了 `Map` 的各种方法，并测试了各种 `Map`。

作为应用 `HashMap` 的例子，考虑写一个程序来检查 Java 中 `Random` 类的随机性。理想情况下，`Random` 类生成的随机数应该有很好的分布。作这个测试需要生成大量的随机数，然后计算落在不同区间内的随机数个数。`HashMap` 很适宜做这个工作，因为它将对象与对象关联起来（在此例中，是将 `Math.random()` 生成的值与其出现的次数关联起来）：

```

///: c11:Statistics.java
// Simple demonstration of HashMap.
import java.util.*;

class Counter {
    int i = 1;
    public String toString() { return Integer.toString(i); }
}

public class Statistics {
    private static Random rand = new Random();
    public static void main(String[] args) {

```

```

Map hm = new HashMap();
for(int i = 0; i < 10000; i++) {
    // Produce a number between 0 and 20:
    Integer r = new Integer(rand.nextInt(20));
    if(hm.containsKey(r))
        ((Counter)hm.get(r)).i++;
    else
        hm.put(r, new Counter());
}
System.out.println(hm);
}
} ///:~

```

在 `main()` 中，每生成一个随机数就用 `Integer` 对象包装起来，这样 `HashMap` 才能使用。（容器不能保存基本类型，只能保存对象的引用。）`containsKey()` 方法会检查此“键”是否已经在容器中了（也就是说，此随机数是否已经生成过了）。如果找到了，`get()` 方法返回当前与“键”相关联的“值”，就是 `Counter` 对象。`Counter` 中的 `i` 值增加时表示当前的随机数又出现了一次。

如果当前的“键”还没有生成过，方法 `put()` 就会将一个新的“键值对”放入 `HashMap`。因此创建 `Counter` 对象的时候它会自动初始化 `i` 值为 1，表示当前的随机数是第一次出现。

要显示 `HashMap`，只需直接打印。`HashMap` 的 `toString()` 方法会遍历所有的键值对，并对每一个键值对调用 `toString()`。`Integer.toString()` 是预先定义好的，还可以看到 `Counter` 的 `toString()` 方法。程序的某次输出（添加了一些换行符）如下：

```

{15=529, 4=488, 19=518, 8=487, 11=501, 16=487, 18=507, 3=524,
7=474, 12=485, 17=493, 2=490, 13=540, 9=453, 6=512, 1=466,
14=522, 10=471, 5=522, 0=531}

```

也许你会好奇为什么需要 `Counter` 类，它似乎没有 `Integer` 包装类的功能强。为什么不使用 `int` 或者 `Integer`？首先，不能使用 `int`，因为所有的容器都只能保存对 `Object` 的引用。了解容器之后，由于不能将基本类型的元素放入容器，所以你可能会选择使用包装类。然而，`Java` 包装类只能做一件事，就是将其初始化为某个值，然后读取这个值。也就是说，一旦创建了包装类的对象，就没有办法改变它的值。这使得 `Integer` 包装类对于解决当前问题无能为力，所以我们只能创建一个新的类来满足需要。

SortedMap

使用 `SortedMap`（`TreeMap` 是其唯一的实现），可以确保“键”处于排序状态，这使得它具有额外的功能，这些功能由 `SortedMap` 接口中的下列方法提供：

`Comparator comparator()`：返回当前 `Map` 使用的 `Comparator`，或者返回 `null`，表示以自然方式排序。

Object firstKey(): 返回 Map 中的第一个“键”。

Object lastKey(): 返回 Map 中的最末一个“键”。

SortedMap subMap(fromKey, toKey): 生成此 Map 的子集，范围由从 fromKey（包含）到 toKey（不包含）的“键”确定。

SortedMap headMap(toKey): 生成此 Map 的子集，由“键”小于 toKey 的所有“键值对”组成。

SortedMap tailMap(fromKey): 生成此 Map 的子集，由“键”大于或等于 fromKey 的所有“键值对”组成。

下面的例子与 SortedSetDemo.java 相似，演示了 TreeMap 新增的功能：

```
//: c11:SimplePairGenerator.java
import com.bruceeckel.util.*;
//import java.util.*;

public class SimplePairGenerator implements MapGenerator {
    public Pair[] items = {
        new Pair("one", "A"), new Pair("two", "B"),
        new Pair("three", "C"), new Pair("four", "D"),
        new Pair("five", "E"), new Pair("six", "F"),
        new Pair("seven", "G"), new Pair("eight", "H"),
        new Pair("nine", "I"), new Pair("ten", "J")
    };
    private int index = -1;
    public Pair next() {
        index = (index + 1) % items.length;
        return items[index];
    }
    public static SimplePairGenerator gen =
        new SimplePairGenerator();
} ///:~
```

```
//: c11:SortedMapDemo.java
// What you can do with a TreeMap.
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;
import java.util.*;

public class SortedMapDemo {
```

```

private static Test monitor = new Test();
public static void main(String[] args) {
    TreeMap sortedMap = new TreeMap();
    Collections2.fill(
        sortedMap, SimplePairGenerator.gen, 10);
    System.out.println(sortedMap);
    Object
        low = sortedMap.firstKey(),
        high = sortedMap.lastKey();
    System.out.println(low);
    System.out.println(high);
    Iterator it = sortedMap.keySet().iterator();
    for(int i = 0; i <= 6; i++) {
        if(i == 3) low = it.next();
        if(i == 6) high = it.next();
        else it.next();
    }
    System.out.println(low);
    System.out.println(high);
    System.out.println(sortedMap.subMap(low, high));
    System.out.println(sortedMap.headMap(high));
    System.out.println(sortedMap.tailMap(low));
    monitor.expect(new String[] {
        "{eight=H, five=E, four=D, nine=I, one=A, seven=G, " +
        " six=F, ten=J, three=C, two=B}",
        "eight",
        "two",
        "nine",
        "ten",
        "{nine=I, one=A, seven=G, six=F}",
        "{eight=H, five=E, four=D, nine=I, " +
        "one=A, seven=G, six=F}",
        "{nine=I, one=A, seven=G, six=F, " +
        "ten=J, three=C, two=B}"
    });
}
} ///:~

```

此处，“键值对”是按“键”的次序排列的。在 `TreeMap` 中“次序”是有意义的，因此才有“位置”的概念，所以你能取得第一个和最后一个元素，并且可以提取 `Map` 的子集。

LinkedHashMap

为了提高速度，`LinkedHashMap` 散列化所有的元素，但是在遍历“键值对”时，却又以元素的插入顺序返回“键值对”（`println()`会迭代遍历 `Map`，因此你可以看到遍历的结果）。此外，可以在构造器中设定 `LinkedHashMap`，使之采用基于访问的“最近最少使用”（LRU）算法，于是没有被访问过的（可被看作需要删除的）元素就会出现在队列的前面。对于需要定期按顺序清除元素以节省空间的程序来说，此功能使得程序很容易得以实现。下面就是一个简单的例子，演示了 `LinkedHashMap` 的这两种特点：

```
///  
// What you can do with a LinkedHashMap.  
import com.bruceeckel.simpletest.*;  
import com.bruceeckel.util.*;  
import java.util.*;  
  
public class LinkedHashMapDemo {  
    private static Test monitor = new Test();  
    public static void main(String[] args) {  
        LinkedHashMap linkedMap = new LinkedHashMap();  
        Collections2.fill(  
            linkedMap, SimplePairGenerator.gen, 10);  
        System.out.println(linkedMap);  
        // Least-recently used order:  
        linkedMap = new LinkedHashMap(16, 0.75f, true);  
        Collections2.fill(  
            linkedMap, SimplePairGenerator.gen, 10);  
        System.out.println(linkedMap);  
        for(int i = 0; i < 7; i++) // Cause accesses:  
            linkedMap.get(SimplePairGenerator.gen.items[i].key);  
        System.out.println(linkedMap);  
        linkedMap.get(SimplePairGenerator.gen.items[0].key);  
        System.out.println(linkedMap);  
        monitor.expect(new String[] {  
            "{one=A, two=B, three=C, four=D, five=E, " +  
            "six=F, seven=G, eight=H, nine=I, ten=J}",  
            "{one=A, two=B, three=C, four=D, five=E, " +  
            "six=F, seven=G, eight=H, nine=I, ten=J}",  
            "{eight=H, nine=I, ten=J, one=A, two=B, " +  
            "three=C, four=D, five=E, six=F, seven=G}",  
            "{eight=H, nine=I, ten=J, two=B, three=C, " +  
            "four=D, five=E, six=F, seven=G, one=A}"  
        });  
    }  
} ///  
~
```


在输出中可以看到，“键值对”是以插入的顺序进行遍历的，甚至 LRU 算法的版本也是如此。在（只）访问过前面七个元素后，最后三个元素移到了队列前面。然后再一次访问元素“one”，它就被移到队列后端了。

散列算法与散列码

在 Statistics.java 中，标准类库中的类（Integer）被用作 HashMap 的“键”。它运作得很好，因为它具备了“键”所需的全部性质。但是，如果你创建自己的类作为 HashMap 的“键”使用，通常会犯一个错误。例如，考虑一个天气预报系统，将 Groundhog（土拨鼠）对象与 Prediction（预测）对象联系起来。看起来相当简单，创建这两个类，使用 Groundhog 作为“键”，Prediction 作为“值”：

```
//: c11:Groundhog.java
// Looks plausible, but doesn't work as a HashMap key.
```

```
public class Groundhog {
    protected int number;
    public Groundhog(int n) { number = n; }
    public String toString() {
        return "Groundhog #" + number;
    }
} ///:~
```

```
//: c11:Prediction.java
// Predicting the weather with groundhogs.
```

```
public class Prediction {
    private boolean shadow = Math.random() > 0.5;
    public String toString() {
        if(shadow)
            return "Six more weeks of Winter!";
        else
            return "Early Spring!";
    }
} ///:~
```

```
//: c11:SpringDetector.java
// What will the weather be?
import com.bruceeckel.simpletest.*;
import java.util.*;
```

```

import java.lang.reflect.*;

public class SpringDetector {
    private static Test monitor = new Test();
    // Uses a Groundhog or class derived from Groundhog:
    public static void
    detectSpring(Class groundHogClass) throws Exception {
        Constructor ghog = groundHogClass.getConstructor(
            new Class[] {int.class});
        Map map = new HashMap();
        for(int i = 0; i < 10; i++)
            map.put(ghog.newInstance(
                new Object[] { new Integer(i) }), new Prediction());
        System.out.println("map = " + map + "\n");
        Groundhog gh = (Groundhog)
            ghog.newInstance(new Object[] { new Integer(3) });
        System.out.println("Looking up prediction for " + gh);
        if(map.containsKey(gh))
            System.out.println((Prediction)map.get(gh));
        else
            System.out.println("Key not found: " + gh);
    }
    public static void main(String[] args) throws Exception {
        detectSpring(Groundhog.class);
        monitor.expect(new String[] {
            "%% map = \{(Groundhog #\d=" +
            "(Early Spring!|Six more weeks of Winter!)" +
            "(, )?) {10}\}",
            "",
            "Looking up prediction for Groundhog #3",
            "Key not found: Groundhog #3"
        });
    }
} ///:~

```

每个 Groundhog 被给予一个标识数字，于是可以在 HashMap 中这样查找 Prediction：“给我与#3 号 Groundhog 相关的 Prediction。” Prediction 类包含一个 boolean 值，使用 Math.random() 对其初始化；而 toString() 方法则为你解释它的意义。detectSpring() 方法使用映射机制创建实例，可以使用 Class Groundhog 或其子类。如果我们为解决当前的问题，由 Groundhog 继承了一个新类的时候，detectSpring() 方法使用的这个技巧就变得很有用了。detectSpring() 首先会使用 Groundhog 和与之相关联的 Prediction 填充 HashMap。然后打印此 HashMap。所以你可以看到，它确实被填入了一些内容。然后使用标识数字为 3 的 Groundhog 作为“键”，查找与之相关的 Prediction（可以看到，它一定是在 Map 中）。

这看起来够简单了，但是它不工作。问题出在 `Groundhog` 继承自基类 `Object`（如果你不特别指定父类，任何类都会自动继承自 `Object`，因此所有的类最终都继承自 `Object`）。所以这里是使用 `Object` 的 `hashCode()` 方法生成散列码，而它默认是使用对象的地址计算散列码。因此，由 `Groundhog(3)` 生成的第一个实例的散列码与由 `Groundhog(3)` 生成的第二个实例的散列码是不同的，而我们正是使用后者进行查找的。

可能你会认为，你只需重载一份恰当的 `hashCode()` 方法即可。但是它仍然无法正常运行，除非你同时重载 `equals()` 方法，它也是 `Object` 的一部分。`HashMap` 使用 `equals()` 判断当前的“键”是否与表中存在的“键”相同。

正确的 `equals()` 方法必须满足下列 5 个条件：

1. 自反性：对任意 `x`，`x.equals(x)` 一定返回 `true`。
2. 对称性：对任意 `x` 和 `y`，如果 `y.equals(x)` 返回 `true`，则 `x.equals(y)` 也返回 `true`。
3. 传递性：对任意 `x`，`y`，`z`，如果有 `x.equals(y)` 返回 `true`，`y.equals(z)` 返回 `true`，则 `x.equals(z)` 一定返回 `true`。
4. 一致性：对任意 `x` 和 `y`，如果对象中用于等价比较的信息没有改变，那么无论调用 `x.equals(y)` 多少次，返回的结果应该保持一致，要么一直是 `true`，要么一直是 `false`。
5. 对任何不是 `null` 的 `x`，`x.equals(null)` 一定返回 `false`。

再说一次，默认的 `Object.equals()` 只是比较对象的地址，所以一个 `Groundhog(3)` 并不等于另一个 `Groundhog(3)`。因此，如果要使用自己的类作为 `HashMap` 的“键”，你必须同时重载 `hashCode()` 和 `equals()`，如下所示：

```
//: c11:Groundhog2.java
// A class that's used as a key in a HashMap
// must override hashCode() and equals().

public class Groundhog2 extends Groundhog {
    public Groundhog2(int n) { super(n); }
    public int hashCode() { return number; }
    public boolean equals(Object o) {
        return (o instanceof Groundhog2)
            && (number == ((Groundhog2)o).number);
    }
} ///:~
```

```
//: c11:SpringDetector2.java
// A working key.
import com.bruceeckel.simpletest.*;
import java.util.*;
```

```

public class SpringDetector2 {
    private static Test monitor = new Test();
    public static void main(String[] args) throws Exception {
        SpringDetector.detectSpring(Groundhog2.class);
        monitor.expect(new String[] {
            "% map = \{(Groundhog #\d=" +
            "(Early Spring!|Six more weeks of Winter!)" +
            "(, )?\{10\}\}",
            "",
            "Looking up prediction for Groundhog #3",
            "% Early Spring!|Six more weeks of Winter!"
        });
    }
} ///:~

```

Groundhog2.hashCode()返回 Groundhog 的标识数字（编号）作为散列码。在此例中，程序员负责确保不同的 Groundhog 具有不同的编号。hashCode()并不需要总是能够返回唯一的标识码（稍后你会更加理解其原因），但是 equals()方法必须能够严格地判断两个对象是否相同。此处的equals()是判断Groundhog的号码，所以作为HashMap中的“键”，如果两个 Groundhog2 对象具有相同的 Groundhog 编号，程序就出错了。

看起来 equals()方法只是检查其参数是否 Groundhog2 的实例（使用第十章学过的 instanceof 关键字），但是 instanceof 悄悄地检查了此对象是否为 null，因为如果 instanceof 左边的参数为 null，它会返回 false。如果 equals()的参数不为 null 且类型正确，则基于 ghNumber 进行比较。从输出中可以看到，现在的方式是正确的。

当你在 HashSet 中使用自己的类作为“键”时，必须注意这个问题。

理解 hashCode()

前面的例子只是正确解决问题的第一步。它只说明，如果不为你的“键”重载 hashCode() 和 equals()，那么使用散列的数据结构（HashSet, HashMap, LinkedHashSet, or LinkedHashMap）就无法正确处理你的“键”。然而，要很好地解决此问题，你必须了解这些数据结构的内部构造。

首先，使用散列的目的在于：想要使用一个对象来查找另一个对象。不过使用 TreeSet 或 TreeMap 也能实现此目的，还可以自己实现一个 Map。要达到此目的，必须提供 Map.entrySet()方法，以生成 Map.Entry 对象的 Set。MPair 被定义为一种新型的 Map.Entry，为了能够将其存入 TreeSet 中，MPair 必须实现 Comparable 接口，并要重载 equals()方法：

```

//: c11:MPair.java
// A new type of Map.Entry.
import java.util.*;

```

```

public class MPair implements Map.Entry, Comparable {
    private Object key, value;
    public MPair(Object k, Object v) {
        key = k;
        value = v;
    }
    public Object getKey() { return key; }
    public Object getValue() { return value; }
    public Object setValue(Object v) {
        Object result = value;
        value = v;
        return result;
    }
    public boolean equals(Object o) {
        return key.equals(((MPair)o).key);
    }
    public int compareTo(Object rv) {
        return ((Comparable)key).compareTo(((MPair)rv).key);
    }
} ///:~

```

注意，比较所感兴趣的只是“键”，所以重复的“值”是完全可以接受的。

下面的例子使用一对 **ArrayList** 实现了一个 **Map**：

```

//: c11:SlowMap.java
// A Map implemented with ArrayLists.
import com.bruceeckel.simpletest.*;
import java.util.*;
import com.bruceeckel.util.*;

public class SlowMap extends AbstractMap {
    private static Test monitor = new Test();
    private List
        keys = new ArrayList(),
        values = new ArrayList();
    public Object put(Object key, Object value) {
        Object result = get(key);
        if(!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else
            values.set(keys.indexOf(key), value);
        return result;
    }
}

```

```

    }
    public Object get(Object key) {
        if(!keys.contains(key))
            return null;
        return values.get(keys.indexOf(key));
    }
    public Set entrySet() {
        Set entries = new HashSet();
        Iterator
            ki = keys.iterator(),
            vi = values.iterator();
        while(ki.hasNext())
            entries.add(new MPair(ki.next(), vi.next()));
        return entries;
    }
    public String toString() {
        StringBuffer s = new StringBuffer("{}");
        Iterator
            ki = keys.iterator(),
            vi = values.iterator();
        while(ki.hasNext()) {
            s.append(ki.next() + "=" + vi.next());
            if(ki.hasNext()) s.append(", ");
        }
        s.append("{}");
        return s.toString();
    }
    public static void main(String[] args) {
        SlowMap m = new SlowMap();
        Collections2.fill(m, Collections2.geography, 15);
        System.out.println(m);
        monitor.expect(new String[] {
            "{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo,"+
            " BOTSWANA=Gaberone, BURKINA FASO=Ouagadougou, " +
            "BURUNDI=Bujumbura, CAMEROON=Yaounde, " +
            "CAPE VERDE=Praia, CENTRAL AFRICAN REPUBLIC=Bangui,"+
            " CHAD=N'djamena, COMOROS=Moroni, " +
            "CONGO=Brazzaville, DJIBOUTI=Djibouti, " +
            "EGYPT=Cairo, EQUATORIAL GUINEA=Malabo}"
        });
    }
} ///:~

```

`put()`方法只是将“键”与“值”放入相应的 `ArrayList`。在 `main()`中装载了一个 `SlowMap`，然后通过打印证明它能正常运作。

此例说明创建一种新的 **Map** 并不困难。但是正如其名 **SlowMap** 所示，它不会很快，所以如果有更好的选择，就应该放弃它。它的问题在于对“键”的查询，由于没有排序，所以只能使用简单的线性查询，而这是最慢的查询方式。

散列的价值在于速度：散列使得查询得以快速进行。由于速度的瓶颈是对“键”的查询，因此解决方案之一就是保持“键”的排序状态，然后使用 `Collections.binarySearch()` 进行查询（本章末尾会有一个练习，带着你走完整个过程）。

散列则更进一步，它将“键”保存在某处，使你能够很快速的找到。正如你在本章所看到的，存储一组元素最快的数据结构是数组，所以使用它来代表“键”的信息（请小心留意，我是说“键的信息”，而不是“键”本身）。本章也曾讲过，数组有一个特性：一旦分配，容量就不能改变。因此我们就有一个问题：我们需要能够在 **Map** 中保存任意数量的“值”，但是如果“键”的数量被数组的容量限制了，该怎么办呢？

答案就是：数组并不保存“键”本身。而是通过“键”对象生成一个数字，将其作为数组的下标索引。这个数字就是散列码，由定义在 **Object** 中的 `hashCode()` 生成（在计算机科学的术语中称为散列函数）。你的类总是应该重载 `hashCode()` 方法。为解决数组容量被固定的问题，不同的“键”可以产生相同的下标。也就是说，可能会有冲突（*collision*）。因此，数组多大就不重要了，每个“键”总能在数组中找到它的位置。

于是查询一个“值”的过程首先就是计算散列码，然后使用散列码查询数组。如果能够保证没有冲突（如果“值”的数量是固定的，那么就有可能），那你可就有了一个完美的散列函数，但是这种情况很特殊。通常，冲突是由“外部链接”（**external chaining**）处理：数组并不直接保存“值”，而是保存“值”的 **list**。然后对 **list** 中的“值”使用 `equals()` 方法进行线性的查询。这部分的查询自然会比较慢，但是，如果有好的散列函数，数组的每个位置就只有较少的“值”。因此，不是查询所有的 **list**，而是快速地跳到数组的某个位置，只对很少的元素进行比较。这便是 **HashMap** 会如此快的原因。

理解了散列的原理，就能够实现一个简单的散列 **Map** 了：

```
//: c11:SimpleHashMap.java
// A demonstration hashed Map.
import java.util.*;
import com.bruceeckel.util.*;

public class SimpleHashMap extends AbstractMap {
    // Choose a prime number for the hash table
    // size, to achieve a uniform distribution:
    private static final int SZ = 997;
    private LinkedList[] bucket = new LinkedList[SZ];
    public Object put(Object key, Object value) {
        Object result = null;
        int index = key.hashCode() % SZ;
        if(index < 0) index = -index;
        if(bucket[index] == null)
```

```

        bucket[index] = new LinkedList();
        LinkedList pairs = bucket[index];
        MPair pair = new MPair(key, value);
        ListIterator it = pairs.listIterator();
        boolean found = false;
        while(it.hasNext()) {
            Object iPair = it.next();
            if(iPair.equals(pair)) {
                result = ((MPair) iPair).getValue();
                it.set(pair); // Replace old with new
                found = true;
                break;
            }
        }
        if(!found)
            bucket[index].add(pair);
        return result;
    }

    public Object get(Object key) {
        int index = key.hashCode() % SZ;
        if(index < 0) index = -index;
        if(bucket[index] == null) return null;
        LinkedList pairs = bucket[index];
        MPair match = new MPair(key, null);
        ListIterator it = pairs.listIterator();
        while(it.hasNext()) {
            Object iPair = it.next();
            if(iPair.equals(match))
                return ((MPair) iPair).getValue();
        }
        return null;
    }

    public Set entrySet() {
        Set entries = new HashSet();
        for(int i = 0; i < bucket.length; i++) {
            if(bucket[i] == null) continue;
            Iterator it = bucket[i].iterator();
            while(it.hasNext())
                entries.add(it.next());
        }
        return entries;
    }

    public static void main(String[] args) {
        SimpleHashMap m = new SimpleHashMap();
    }

```



```

        Collections2.fill(m, Collections2.geography, 25);
        System.out.println(m);
    }
} ///:~

```

由于散列表中的“槽位（slot）”通常称为“桶”（bucket），因此我将作为散列表的数组命名为**bucket**。为使散列分布均匀，桶的数量通常使用质数⁹。注意，为了能够自动处理冲突，使用了一个**LinkedList**的数组；每一个新的元素只是直接添加到**list**的末尾。

如果指定的“键”已经存在于 **list** 中了，那么 **put()** 将返回与此“键”相关联的“旧”“值”，否则返回 **null**。本例中返回值是 **result**，它被初始化为 **null**，如果此“键值对”已经存在于 **list** 中，则 **result** 被赋值为 **list** 中此“键”对应的“值”。

方法 **put()** 和 **get()** 要做的第一件事情，是对“键”调用 **hashCode()**。结果被强制转换为正数，然后用数组容量对其取模，使它适合 **bucket** 数组的大小。如果数组的某个位置是 **null**，这表示还没有元素被散列至此，所以，为了保存定位于此的第一个对象，需要创建一个新的 **LinkedList**。一般的过程是，查看当前位置的 **list** 中是否有相同的元素，如果有，则将旧的“值”赋给 **result**，然后用新的“值”取代旧的“值”。标记 **found** 用来跟踪是否找到（相同的）旧的“键值对”，如果没有，则将新 **pair** 添加到 **list** 的末尾。

方法 **get()** 的代码与 **put()** 相似，不过更简单。首先计算 **bucket** 数组的下标，如果此位置有 **LinkedList** 存在，就对其进行查询。

entrySet() 遍历所有的 **list**，将其中的元素加入到作为结果的 **Set** 中。有了这个方法，便可以进行用“值”填充 **Map**，然后将它们打印出来的测试了。

HashMap 的性能因子

要理解这个问题，必须先解释一些术语：

容量（Capacity）：散列表中桶的数量。

初始化容量（Initial capacity）：创建散列表时桶的数量。**HashMap** 和 **HashSet** 都允许你在构造器中指定初始化容量。

尺寸（Size）：当前散列表中记录的数量。

负载因子（Load factor）：等于“size/capacity”。负载因子为 0，表示空的散列表，0.5 表示半满的散列表，依此类推。轻负载的散列表具有冲突少、适宜插入与查询的特点（但是使用迭代器遍历会变慢）。**HashMap** 与 **HashSet** 的构造器允许

⁹ 事实证明，质数实际上并不是散列桶的理想容量。近来，（经过广泛的测试）Java 的散列函数都使用 2 的整数次方。对现代的处理器的来说，除法与求余数是最慢的操作。使用 2 的整数次方长度的散列表，可用掩码代替除法。因为 **get()** 是使用最多的操作，求余数的 % 操作是其开销最大的部分，而使用 2 的整数次方可以消除此开销（也可能对 **hashCode()** 有些影响）。

你指定负载因子。这意味着，当负载达到指定值时，容器会自动成倍的增加容量（桶的数量），并将原有的对象重新分配，存入新的桶内（这称为“重散列” *rehashing*）。

HashMap 默认的负载因子为 0.75（除非此表的 3/4 已经被填满了，否则不会重散列）。这很好的权衡了时间与空间的成本。较高的负载因子在降低空间需求的同时，会提高查询的时间开销，而查询是你用得最多的操作（`get()`与 `put()`中都用到查询），因此负载因子是很重要的概念。

如果知道HashMap中会有很多记录，在创建的时候就使用较大的初始化容量，可以避免自动重散列rehashing的开销¹⁰。

重载 hashCode()

在明白了 HashMap 具有哪些功能之后，学习如何写一个 `hashCode()` 会更有意义。

首先，你无法控制 bucket 数组的索引值的产生。这个值依赖于具体的 HashMap 对象的容量，而容量的改变与负载因子和容器有多满有关。`hashCode()`生成的结果，经过处理后成为“桶”的索引（在 `SimpleHashMap` 中，只是对其取模，模数为 bucket 数组的大小）。

设计 `hashCode()`时最重要的因素就是：无论何时，对同一个对象调用 `hashCode()`都应该生成同样的值。如果在将一个对象用 `put()`添加进 HashMap 时，产生一个 `hashCode()`值，而用 `get()`取出时，却产生了另一个 `hashCode()`值，那么你就无法重新取得该对象了。所以，如果你的 `hashCode()`依赖于对象中易变的数据，用户就必须当心了，因为此数据发生变化时，`hashCode()`就会生成一个不同的散列码，相当于产生了一个不同的“键”。

此外，你也不应该使 `hashCode()`依赖于具有唯一性的对象信息。尤其是使用 `this` 的值，这只能作出很糟糕的 `hashCode()`。因为你无法生成一个新的“键”，使之与 `put()`中原始的“键值对”中的“键”相同。这正是 `SpringDetector.java` 的问题所在，因为它默认的 `hashCode()`使用的是对象的地址。所以，你应该使用对象内有意义的识别信息。

下面以 `String` 类为例。`String` 有个特点：如果程序中有多多个 `String` 对象，都包含相同的字符串序列，那么这些 `String` 对象都映射到同一块内存区域（附录 A 详细描述此机制）。所以 `new String("hello")`生成的两个实例，虽然是相互独立的，但是对它们使用 `hashCode()`应该生成同样的结果。通过下面的程序可以看到这种情况：

```
//: c11:StringHashCode.java
import com.bruceeckel.simpletest.*;

public class StringHashCode {
```

¹⁰ 私下里，Joshua Bloch曾写道：“...我认为，允许用户实现API底层的细节（例如散列表的容量和负载因子）这种做法是错误的。客户应该可以告诉我们，对一个集合所期望的最大容量，然后我们采用其意见。如果让客户自己去选择这些参数，将是弊大于利。举个偏激的例子，考虑Vector的capacityIncrement。任何人都该设定此值，我们也不该提供方法。如果你将其设定为任意非零值，则顺序添加操作的开销将由线性的变成平方级的。换言之，它破坏了性能。很久之后，我们才能够聪明的应对此类事情。如果你看看IdentityHashMap就会发现，它已经不提供底层的调节参数了。”

```

private static Test monitor = new Test();
public static void main(String[] args) {
    System.out.println("Hello".hashCode());
    System.out.println("Hello".hashCode());
    monitor.expect(new String[] {
        "69609650",
        "69609650"
    });
}
} ///:~

```

对于 String 而言，hashCode()明显是基于 String 的内容的。

因此，要想使 hashCode()实用，它必须速度快，并且必须有意义。也就是说，它必须基于对象的内容生成散列码。记得吗，散列码不必是独一无二的（应该更关注生成速度，而不是唯一性），但是通过 hashCode()和 equals()，必须能够完全确定对象的身份。

因为在生成 bucket 的索引前，hashCode()还需要进一步的处理，所以散列码的生成范围并不重要，只要是 int 即可。

还有另一个影响因素：好的 hashCode()应该产生分布均匀的散列码。如果散列码都集中在一块，那么 HashMap 或者 HashSet 在某些区域的负载会很重，就不会有分布均匀的散列函数那么快。

在 *Effective Java* (Addison-Wesley 2001) 这本书中，Joshua Bloch 为怎样写出一份像样的 hashCode()，给出了基本的指导：

1. 给 int 变量 **result** 赋予某个非零值常量，例如 17。
2. 为对象内每个有意义的属性 **f**（即每个可以作 equals()操作的属性）计算出一个 int 散列码 **c**：

域类型	计算
boolean	c = (f ? 0 : 1)
byte, char, short, or int	c = (int)f
long	c = (int)(f ^ (f >>> 32))
float	c = Float.floatToIntBits(f);
double	long l = Double.doubleToLongBits(f); c = (int)(l ^ (l >>> 32))
Object, 其 equals() 调用这个域的 equals()	c = f.hashCode()

数组	对每个元素应用上述规则
----	-------------

1. 合并计算得到的散列码:

result = 37 * result + c;

2. 返回 **result**

3. 检查 **hashCode()** 最后生成的结果，确保相同的对象有相同的散列码。

下面便是遵循这些指导的一个例子:

```

//: c11:CountedString.java
// Creating a good hashCode().
import com.bruceeckel.simpletest.*;
import java.util.*;

public class CountedString {
    private static Test monitor = new Test();
    private static List created = new ArrayList();
    private String s;
    private int id = 0;
    public CountedString(String str) {
        s = str;
        created.add(s);
        Iterator it = created.iterator();
        // Id is the total number of instances
        // of this string in use by CountedString:
        while(it.hasNext())
            if(it.next().equals(s))
                id++;
    }
    public String toString() {
        return "String: " + s + " id: " + id +
            " hashCode(): " + hashCode();
    }
    public int hashCode() {
        // Very simple approach:
        // return s.hashCode() * id;
        // Using Joshua Bloch's recipe:
        int result = 17;
        result = 37*result + s.hashCode();
        result = 37*result + id;
        return result;
    }
    public boolean equals(Object o) {
        return (o instanceof CountedString)

```

```

        && s.equals(((CountedString)o).s)
        && id == ((CountedString)o).id;
    }

    public static void main(String[] args) {
        Map map = new HashMap();
        CountedString[] cs = new CountedString[10];
        for(int i = 0; i < cs.length; i++) {
            cs[i] = new CountedString("hi");
            map.put(cs[i], new Integer(i));
        }
        System.out.println(map);
        for(int i = 0; i < cs.length; i++) {
            System.out.println("Looking up " + cs[i]);
            System.out.println(map.get(cs[i]));
        }
        monitor.expect(new String[] {
            "{String: hi id: 4 hashCode(): 146450=3," +
            " String: hi id: 10 hashCode(): 146456=9," +
            " String: hi id: 6 hashCode(): 146452=5," +
            " String: hi id: 1 hashCode(): 146447=0," +
            " String: hi id: 9 hashCode(): 146455=8," +
            " String: hi id: 8 hashCode(): 146454=7," +
            " String: hi id: 3 hashCode(): 146449=2," +
            " String: hi id: 5 hashCode(): 146451=4," +
            " String: hi id: 7 hashCode(): 146453=6," +
            " String: hi id: 2 hashCode(): 146448=1}",
            "Looking up String: hi id: 1 hashCode(): 146447",
            "0",
            "Looking up String: hi id: 2 hashCode(): 146448",
            "1",
            "Looking up String: hi id: 3 hashCode(): 146449",
            "2",
            "Looking up String: hi id: 4 hashCode(): 146450",
            "3",
            "Looking up String: hi id: 5 hashCode(): 146451",
            "4",
            "Looking up String: hi id: 6 hashCode(): 146452",
            "5",
            "Looking up String: hi id: 7 hashCode(): 146453",
            "6",
            "Looking up String: hi id: 8 hashCode(): 146454",
            "7",
            "Looking up String: hi id: 9 hashCode(): 146455",
            "8",

```

```

        "Looking up String: hi id: 10 hashCode(): 146456",
        "9"
    });
}
} ///:~

```

`CountedString` 由一个 `String` 和一个 `id` 组成，此 `id` 代表包含相同 `String` 的 `CountedString` 对象的编号。所有的 `String` 都被存储在 `static ArrayList` 中，构造器中使用迭代器遍历此 `ArrayList`，此时完成对 `id` 的计算。

`hashCode()` 和 `equals()` 都基于 `CountedString` 的这两个属性来生成结果；如果它们只基于 `String` 或者只基于 `id`，不同的对象就可能产生相同的值。

在 `main()` 中，使用相同的 `String` 创建了一串 `CountedString` 对象。以此说明，虽然 `String` 相同，但是由于 `id` 不同，所以使得它们的散列码并不相同。在程序中，`HashMap` 被打印了出来，因此你可以看到它内部是如何存储元素的（以无法辨别的次序），然后程序单独查询每一个“键”，以此证明查询机制工作正常。

为你的类编写正确的 `hashCode()` 和 `equals()` 是很需要技巧的。Apache 的“Jakarta Commons”项目中有许多工具可以帮助你，可在 jakarta.apache.org/commons 的“lang”下面找到。（此项目还包括许多其他有用的类库，而且它似乎是 Java 社区对 C++ 的 www.boost.org 作出的回应）。

持有引用

`java.lang.ref` 类库包含一组类，为垃圾回收提供了更大的灵活性。当存在可能会耗尽内存的大对象的时候，这些类显得特别有用。有三个继承自抽象类 `Reference` 的类：`SoftReference`、`WeakReference` 和 `PhantomReference`。如果垃圾回收器正在考察的对象只能通过某个 `Reference` 对象才“可获得”，那么不同的 `Reference` 对象为垃圾回收器提供了不同级别的间接性指示。

对象是“可获得的”（`reachable`），是指此对象可在程序中的某处找到。这意味着你在内存栈中有一个普通的引用，而它正指向此对象；也可能是你的引用指向某个对象，而那个对象含有另一个引用，指向正在讨论的对象；也可能有更多的中间链接。如果一个对象是“可获得的”，垃圾回收器就不能释放它，因为它仍然为你的程序所用。如果一个对象不是“可获得的”，那么你的程序将无法使用到它，所以将其回收是安全的。

如果你想继续持有对某个对象的引用，希望以后还能够访问到该对象，但是你也希望能够允许垃圾回收器释放它，这时就应该使用 `Reference` 对象。于是你可以继续使用该对象，而在内存消耗殆尽的时候，又允许释放该对象。

以 `Reference` 对象做为你和普通引用之间的媒介，就能达成此目的。同时一定不能有普通的引用指向那个对象。（普通的引用：指没有经 `Reference` 对象包装过的引用。）如果垃圾回收器发现某个对象通过普通引用是可获得的，该对象就不会被释放。

SoftReference、WeakReference 和 PhantomReference 由强到弱排列，对应不同级别的“可获得性”（reachability）。Softreference 用以实现内存敏感的高速缓存。Weak reference 是为实现“规范映射”（canonicalizing mappings）而设计的，它不妨碍垃圾回收器回收 map 的“键”（或“值”）。“规范映射”中使得对象的实例可以在程序的多处被同时使用，以节省存储空间。Phantomreference 用以调度回收前的清理工作，它比 Java 终止机制（finalization mechanism）更灵活。

使用 SoftReference 和 WeakReference 时，你可以选择是否要将它们放入 ReferenceQueue 中（用于“回收前清理工作”的工具）。而 PhantomReference 只能依赖于 ReferenceQueue。下面是一个简单的示例：

```
//: c11:References.java
// Demonstrates Reference objects
import java.lang.ref.*;

class VeryBig {
    private static final int SZ = 10000;
    private double[] d = new double[SZ];
    private String ident;
    public VeryBig(String id) { ident = id; }
    public String toString() { return ident; }
    public void finalize() {
        System.out.println("Finalizing " + ident);
    }
}

public class References {
    private static ReferenceQueue rq = new ReferenceQueue();
    public static void checkQueue() {
        Object inq = rq.poll();
        if(inq != null)
            System.out.println("In queue: " +
                (VeryBig)((Reference)inq).get());
    }

    public static void main(String[] args) {
        int size = 10;
        // Or, choose size via the command line:
        if(args.length > 0)
            size = Integer.parseInt(args[0]);
        SoftReference[] sa = new SoftReference[size];
        for(int i = 0; i < sa.length; i++) {
            sa[i] = new SoftReference(
                new VeryBig("Soft " + i), rq);
            System.out.println("Just created: " +
                (VeryBig)sa[i].get());
        }
    }
}
```



```

        checkQueue();
    }
    WeakReference[] wa = new WeakReference[size];
    for(int i = 0; i < wa.length; i++) {
        wa[i] = new WeakReference(
            new VeryBig("Weak " + i), rq);
        System.out.println("Just created: " +
            (VeryBig)wa[i].get());
        checkQueue();
    }
    SoftReference s =
        new SoftReference(new VeryBig("Soft"));
    WeakReference w =
        new WeakReference(new VeryBig("Weak"));
    System.gc();
    PhantomReference[] pa = new PhantomReference[size];
    for(int i = 0; i < pa.length; i++) {
        pa[i] = new PhantomReference(
            new VeryBig("Phantom " + i), rq);
        System.out.println("Just created: " +
            (VeryBig)pa[i].get());
        checkQueue();
    }
}
} ///:~

```

运行此程序可以看到（将输出定向到“more”工具，便可以查看分页的输出），尽管你还需要通过 **Reference** 访问那些对象（使用 `get()` 取得实际的对象引用），但对象还是被垃圾回收器回收了。你还可以看到，**ReferenceQueue** 总是生成一个包含 `null` 对象的 **Reference**。要利用此机制，可以继承某种你感兴趣的 **Reference**，然后为其添加一些更有用的方法。

WeakHashMap

容器类中有一种特殊的 **Map**: **WeakHashMap**，它被用来保存 **WeakReference**。它使得“规范映射” (canonicalized mappings) 更易于使用。在这种映射中，每个“值”只保存一份实例以节省存储空间。当程序需要那个“值”的时候，便在映射中查询现有的对象，然后使用它（而不是重新再创建）。可将“值”作为“规范映射”的一部分，一同初始化，不过通常是在需要的时候才生成“值”。

这是一种节约存储空间的技术，因为 **WeakHashMap** 允许垃圾回收器自动清理“键”和“值”，所以它显得十分便利。对于向 **WeakHashMap** 添加“键”和“值”的操作，则没有什么特殊要求。**WeakHashMap** 会自动使用 **WeakReference** 包装它们。允许清理元素的触发条件是，不再需要此“键”了，如下所示：


```

//: c11:CanonicalMapping.java
// Demonstrates WeakHashMap.
import java.util.*;
import java.lang.ref.*;

class Key {
    private String ident;
    public Key(String id) { ident = id; }
    public String toString() { return ident; }
    public int hashCode() { return ident.hashCode(); }
    public boolean equals(Object r) {
        return (r instanceof Key)
            && ident.equals(((Key)r).ident);
    }
    public void finalize() {
        System.out.println("Finalizing Key " + ident);
    }
}

class Value {
    private String ident;
    public Value(String id) { ident = id; }
    public String toString() { return ident; }
    public void finalize() {
        System.out.println("Finalizing Value " + ident);
    }
}

public class CanonicalMapping {
    public static void main(String[] args) {
        int size = 1000;
        // Or, choose size via the command line:
        if(args.length > 0)
            size = Integer.parseInt(args[0]);
        Key[] keys = new Key[size];
        WeakHashMap map = new WeakHashMap();
        for(int i = 0; i < size; i++) {
            Key k = new Key(Integer.toString(i));
            Value v = new Value(Integer.toString(i));
            if(i % 3 == 0)
                keys[i] = k; // Save as "real" references
            map.put(k, v);
        }
        System.gc();
    }
}

```

```
    }  
} ///:~
```

如同本章前面所述，**Key** 类必须有 `hashCode()` 和 `equals()`，因为在散列数据结构中，它被用作“键”。

运行此程序，你会看到垃圾回收器会每隔三个“键”，就跳过一个，因为指向那个“键”的普通引用被存入了 `keys` 数组，所以那些对象不能被垃圾回收器回收。

重访迭代器

现在我们可以展示 **Iterator** 真实的威力了：将遍历一个序列的操作与此序列底层结构相分离的能力。（本章前面定义的）`PrintData` 类使用 **Iterator** 来遍历一个序列，并且对每个对象调用 `toString()` 方法。在下面的例子中，创建了两种不同的容器，一个 `ArrayList` 和一个 `HashMap`，分别使用 `Mouse` 和 `Hamster` 对象进行填充。（本章先前定义了这些类。）因为 **Iterator** 隐藏了容器底层的结构，所以 `Printer.printAll()` 并不知道，也不关心此 **Iterator** 是从何容器而来：

```
///: c11:Iterators2.java  
// Revisiting Iterators.  
import com.bruceeckel.simpletest.*;  
import java.util.*;  
  
public class Iterators2 {  
    private static Test monitor = new Test();  
    public static void main(String[] args) {  
        List list = new ArrayList();  
        for(int i = 0; i < 5; i++)  
            list.add(new Mouse(i));  
        Map m = new HashMap();  
        for(int i = 0; i < 5; i++)  
            m.put(new Integer(i), new Hamster(i));  
        System.out.println("List");  
        Printer.printAll(list.iterator());  
        System.out.println("Map");  
        Printer.printAll(m.entrySet().iterator());  
        monitor.expect(new String[] {  
            "List",  
            "This is Mouse #0",  
            "This is Mouse #1",  
            "This is Mouse #2",  
            "This is Mouse #3",  
            "This is Mouse #4",  
            "Map",  
        });  
    }  
}
```

```

        "4=This is Hamster #4",
        "3=This is Hamster #3",
        "2=This is Hamster #2",
        "1=This is Hamster #1",
        "0=This is Hamster #0"
    }, Test.IGNORE_ORDER);
}
} ///:~

```

对 `HashMap` 而言, `entrySet()` 方法生成一个 `Set`, 由 `Map.Entry` 对象组成, 此对象包含映射中每个元素的“键”和“值”, 可以看到它们都被打印了出来。

注意, `PrintData.print()` 利用了容器的特点: 容器中的对象都是 `Object` 类型的, 所以 `System.out.println()` 会自动调用 `toString()` 方法。对你而言, 则必须假设你的 `Iterator` 要遍历的是某种特定类型的容器。例如, 你可能需要假设容器中的所有对象是 `Shape` 类型, 带有 `draw()` 方法。然后必须对 `Iterator.next()` 返回的 `Object` 向下类型转换成 `Shape`。

选择接口的不同实现

现在你已经知道了, 实际上只有三种容器: `Map`, `List`, 和 `Set`, 但是每种接口都有不止一个实现版本。如果需要使用某种接口的功能, 应该如何选择使用哪一个实现呢?

要了解问题的答案, 首先必须要理解每种实现各自的特征、优点和缺点。例如, 从容器分类图中可以看出, `Hashtable`、`Vector` 和 `Stack` 的“特征”是, 它们是过去遗留下来的类, 目的只是为了支持老的程序罢了。因此, 最好不要在新的程序中使用它们。

容器之间的区别, 通常归结为由什么在背后“支持”它们。也就是说, 你使用的接口是由什么样的数据结构实现的。例如, `ArrayList` 和 `LinkedList` 都实现了 `List` 接口, 因此无论选择哪一个, 基本操作都是相同的。然而, `ArrayList` 底层由数组支持; 而 `LinkedList` 是由双向链表实现的, 其中的每个对象包含数据的同时, 还包含指向链表中前一个与后一个元素的引用。因此, 如果要经常在 `list` 中插入或删除元素, `LinkedList` 就比较合适。

(`LinkedList` 还有建立在 `AbstractSequentialList` 基础上的其他功能。) 否则, 应该使用速度更快的 `ArrayList`。

再举个例子, `Set` 可被实现为 `TreeSet`, `HashSet`, 或 `LinkedHashSet`。每一种都有不同的行为: `HashSet` 是最常用的, 查询速度最快; `LinkedHashSet` 保持元素插入的次序; `TreeSet` 基于 `TreeMap`, 生成一个总是处于排序状态的 `Set`。由此, 你可以根据所需的行来来选择接口不同的实现。通常, 使用 `HashSet` 就足够了, 它是使用 `Set` 的默认首选。

对 List 的选择

要证明 `List` 不同实现之间的区别, 最有说服力的办法就是做性能测试。下面的程序使用内部类作为测试框架, 然后建立一个匿名内部类的数组, 每个匿名类对应一个不同的测试。

这些内部类都是通过 `test()` 方法调用的。使用这种方式，可以很方便地删除和添加新的测试。

```
//: c11:ListPerformance.java
// Demonstrates performance differences in Lists.
// {Args: 500}
import java.util.*;
import com.bruceeckel.util.*;

public class ListPerformance {
    private static int reps = 10000;
    private static int quantity = reps / 10;
    private abstract static class Tester {
        private String name;
        Tester(String name) { this.name = name; }
        abstract void test(List a);
    }
    private static Tester[] tests = {
        new Tester("get") {
            void test(List a) {
                for(int i = 0; i < reps; i++) {
                    for(int j = 0; j < quantity; j++)
                        a.get(j);
                }
            }
        },
        new Tester("iteration") {
            void test(List a) {
                for(int i = 0; i < reps; i++) {
                    Iterator it = a.iterator();
                    while(it.hasNext())
                        it.next();
                }
            }
        },
        new Tester("insert") {
            void test(List a) {
                int half = a.size()/2;
                String s = "test";
                ListIterator it = a.listIterator(half);
                for(int i = 0; i < reps * 10; i++)
                    it.add(s);
            }
        },
    }
```

```

new Tester("remove") {
    void test(List a) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext()) {
            it.next();
            it.remove();
        }
    }
},
};

public static void test(List a) {
    // Strip qualifiers from class name:
    System.out.println("Testing " +
        a.getClass().getName().replaceAll("\\w+\\. ", ""));
    for(int i = 0; i < tests.length; i++) {
        Collections2.fill(a, Collections2.countries.reset(),
            quantity);
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(a);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
    }
}

public static void testArrayAsList(int reps) {
    System.out.println("Testing array as List");
    // Can only do first two tests on an array:
    for(int i = 0; i < 2; i++) {
        String[] sa = new String[quantity];
        Arrays2.fill(sa, Collections2.countries.reset());
        List a = Arrays.asList(sa);
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(a);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
    }
}

public static void main(String[] args) {
    // Choose a different number of
    // repetitions via the command line:
    if(args.length > 0)
        reps = Integer.parseInt(args[0]);
    System.out.println(reps + " repetitions");
}

```

```

        testArrayAsList(reps);
        test(new ArrayList());
        test(new LinkedList());
        test(new Vector());
    }
} ///:~

```

因为要为各种测试提供一个基类，所以内部类 **Tester** 被设计成为抽象类。它包含一个 **String** 和抽象的 **test()** 方法，前者用以在测试开始的时候打印，后者做测试工作。所有不同的测试被集中存放在 **tests** 数组中，它使用各种继承自 **Tester** 的匿名内部类初始化。要添加或删除测试，只需直接在此数组中添加或删除一个内部类，其余的事情会被自动处理。

比较数组和容器（主要是与 **ArrayList** 作比较）的测试有点特殊，需要先将数组使用 **Arrays.asList()** 包装成一个 **List**。注意，这种情况下只能进行前两项测试，因为你不能在数组中插入或删除元素。

由 **test()** 处理的 **List**，首先会被填充元素，然后记录 **tests** 数组中每个测试的用时。结果会因机器而异，不过仍然可以反映不同容器之间的性能区别。下面是某此测试的小结：

Type	Get	Iteration	Insert	Remove
array	172	516	na	na
ArrayList	281	1375	328	30484
LinkedList	5828	1047	109	16
Vector	422	1890	360	30781

同预期的一样，对于随机查询与迭代遍历操作，数组比所有的容器都要快。可以看到，对于随机访问（**get()**），**ArrayList** 的开销小于 **LinkedList**。（奇怪的是，对于迭代遍历操作，**LinkedList** 比 **ArrayList** 要快，这有点有悖于直觉。）另一方面，从中间的位置插入和删除元素，**LinkedList** 要比 **ArrayList** 快，特别是删除操作。**Vector** 通常不如 **ArrayList** 快，而且应该避免使用；它目前仍然存在于类库中的原因是为了支持过去的代码（在此程序中它能正常工作，完全是因为在 **Java 2** 中它被转换成了 **List**）。因此，最佳的做法是将 **ArrayList** 作为默认首选，只有当程序的性能因为经常从 **list** 中间进行插入和删除而变差的时候，才去选择 **LinkedList**。当然了，如果只是使用固定数量的元素，就应该选择数组。

对 Set 的选择

根据你的需要，可以选择 **TreeSet**、**HashSet**，或者 **LinkedHashSet**。下面的测试在性能开销方面给出了指示：

```

///: c11:SetPerformance.java
// {Args: 500}
import java.util.*;

```

```

import com.bruceeckel.util.*;

public class SetPerformance {
    private static int reps = 50000;
    private abstract static class Tester {
        String name;
        Tester(String name) { this.name = name; }
        abstract void test(Set s, int size);
    }
    private static Tester[] tests = {
        new Tester("add") {
            void test(Set s, int size) {
                for(int i = 0; i < reps; i++) {
                    s.clear();
                    Collections2.fill(s,
                        Collections2.countries.reset(), size);
                }
            }
        },
        new Tester("contains") {
            void test(Set s, int size) {
                for(int i = 0; i < reps; i++)
                    for(int j = 0; j < size; j++)
                        s.contains(Integer.toString(j));
            }
        },
        new Tester("iteration") {
            void test(Set s, int size) {
                for(int i = 0; i < reps * 10; i++) {
                    Iterator it = s.iterator();
                    while(it.hasNext())
                        it.next();
                }
            }
        },
    };
    public static void test(Set s, int size) {
        // Strip qualifiers from class name:
        System.out.println("Testing " +
            s.getClass().getName().replaceAll("\\w+\\. ", "") +
            " size " + size);
        Collections2.fill(s,
            Collections2.countries.reset(), size);
        for(int i = 0; i < tests.length; i++) {

```

```

        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(s, size);
        long t2 = System.currentTimeMillis();
        System.out.println(": " +
            ((double) (t2 - t1)/(double) size));
    }
}

public static void main(String[] args) {
    // Choose a different number of
    // repetitions via the command line:
    if(args.length > 0)
        reps = Integer.parseInt(args[0]);
    System.out.println(reps + " repetitions");
    // Small:
    test(new TreeSet(), 10);
    test(new HashSet(), 10);
    test(new LinkedHashSet(), 10);
    // Medium:
    test(new TreeSet(), 100);
    test(new HashSet(), 100);
    test(new LinkedHashSet(), 100);
    // Large:
    test(new TreeSet(), 1000);
    test(new HashSet(), 1000);
    test(new LinkedHashSet(), 1000);
}
} ///:~

```

下表展示了程序运行的结果。（此数据与所使用的计算机和 JVM 有关，你应该自己运行程序去做测试）：

Type	Test size	Add	Contains	Iteration
	10	25.0	23.4	39.1
TreeSet	100	17.2	27.5	45.9
	1000	26.0	30.2	9.0
	10	18.7	17.2	64.1
HashSet	100	17.2	19.1	65.2
	1000	8.8	16.6	12.8
	10	20.3	18.7	64.1

LinkedHashSet	100	18.6	19.5	49.2
	1000	10.0	16.3	10.0

HashSet 的性能总是比 TreeSet 好（特别是最常用的添加和查询元素操作）。TreeSet 存在的唯一原因是，它可以维持元素的排序状态。所以，只有当你需要一个排好序的 Set 时，才应该使用 TreeSet。

注意，对于插入操作，LinkedHashSet 比 HashSet 略微慢一点；这是由维护链表所带来额外开销造成的。不过，因为有了链表，遍历 LinkedHashSet 会更快。

对 Map 的选择

对 Map 不同的实现做选择时，Map 的大小是影响性能最重要的因素，下面的测试程序，在性能开销方面给出了指示：

```

//: c11:MapPerformance.java
// Demonstrates performance differences in Maps.
// {Args: 500}
import java.util.*;
import com.bruceeckel.util.*;

public class MapPerformance {
    private static int reps = 50000;
    private abstract static class Tester {
        String name;
        Tester(String name) { this.name = name; }
        abstract void test(Map m, int size);
    }
    private static Tester[] tests = {
        new Tester("put") {
            void test(Map m, int size) {
                for(int i = 0; i < reps; i++) {
                    m.clear();
                    Collections2.fill(m,
                        Collections2.geography.reset(), size);
                }
            }
        },
        new Tester("get") {
            void test(Map m, int size) {
                for(int i = 0; i < reps; i++)
                    for(int j = 0; j < size; j++)
                        m.get(Integer.toString(j));
            }
        }
    };
}

```

```

    }
},
new Tester("iteration") {
    void test(Map m, int size) {
        for(int i = 0; i < reps * 10; i++) {
            Iterator it = m.entrySet().iterator();
            while(it.hasNext())
                it.next();
        }
    }
},
};

public static void test(Map m, int size) {
    // Strip qualifiers from class name:
    System.out.println("Testing " +
        m.getClass().getName().replaceAll("\\w+\\. ", "") +
        " size " + size);
    Collections2.fill(m,
        Collections2.geography.reset(), size);
    for(int i = 0; i < tests.length; i++) {
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(m, size);
        long t2 = System.currentTimeMillis();
        System.out.println(": " +
            ((double) (t2 - t1) / (double) size));
    }
}

public static void main(String[] args) {
    // Choose a different number of
    // repetitions via the command line:
    if(args.length > 0)
        reps = Integer.parseInt(args[0]);
    System.out.println(reps + " repetitions");
    // Small:
    test(new TreeMap(), 10);
    test(new HashMap(), 10);
    test(new LinkedHashMap(), 10);
    test(new IdentityHashMap(), 10);
    test(new WeakHashMap(), 10);
    test(new Hashtable(), 10);
    // Medium:
    test(new TreeMap(), 100);
    test(new HashMap(), 100);
}

```

```

test(new LinkedHashMap(), 100);
test(new IdentityHashMap(), 100);
test(new WeakHashMap(), 100);
test(new Hashtable(), 100);
// Large:
test(new TreeMap(), 1000);
test(new HashMap(), 1000);
test(new LinkedHashMap(), 1000);
test(new IdentityHashMap(), 1000);
test(new WeakHashMap(), 1000);
test(new Hashtable(), 1000);
}
} ///:~

```

因为 Map 的大小是关键，所以测试耗时的结果是按容器大小进行划分的。下面是某次测试的结果。（你的测试结果或许会不同。）

Type	Test size	Put	Get	Iteration
	10	26.6	20.3	43.7
TreeMap	100	34.1	27.2	45.8
	1000	27.8	29.3	8.8
	10	21.9	18.8	60.9
HashMap	100	21.9	18.6	63.3
	1000	11.5	18.8	12.3
	10	23.4	18.8	59.4
LinkedHashMap	100	24.2	19.5	47.8
	1000	12.3	19.0	9.2
	10	20.3	25.0	71.9
IdentityHashMap	100	19.7	25.9	56.7
	1000	13.1	24.3	10.9
	10	26.6	18.8	76.5
WeakHashMap	100	26.1	21.6	64.4
	1000	14.7	19.2	12.4
	10	18.8	18.7	65.7
Hashtable	100	19.4	20.9	55.3
	1000	13.1	19.9	10.8

正如你期待的，`Hashtable` 和 `HashMap` 的效率大致相同。（你也看到了，`HashMap` 通常更快一点，所以 `HashMap` 有意取代 `Hashtable`。）`TreeMap` 通常比 `HashMap` 慢，为什么还需要它？因为可以使用它生成一个排好序的队列。树的行为方式是：它总是处于排序状态，不需要专门进行排序操作。当 `TreeMap` 被填充之后，就可以调用 `keySet()`，取得由“键”组成的 `Set`，然后使用 `toArray()` 生成“键”的数组。接下来使用 `static Arrays.binarySearch()` 方法（稍后会讨论），在已排序的数组中快速地查询对象。当然，你应该只在因为某些原因而无法使用 `HashMap` 的时候，才去这么做。因为 `HashMap` 正是为快速查询而设计的。而且，你可以很方便地通过 `TreeMap` 生成 `HashMap`。所以，当你需要使用 `Map` 时，首选 `HashMap`，只有在你需要一个总是排好序的 `Map` 时，才使用 `TreeMap`。

`LinkedHashMap` 比 `HashMap` 慢一点，因为它维护散列数据结构的同时还要维护链表。`IdentityHashMap` 则具有完全不同的性能，因为它使用 `==` 而不是 `equals()` 来比较元素。

List 的排序和查询

List 排序与查询所使用的方法与对象数组所使用的方法有相同的名字与语法，只是用 `Collections` 的 `static` 方法代替 `Arrays` 的方法而已。下面的例子，由 `ArraySearching.java` 修改而来：

```
//: c11:ListSortSearch.java
// Sorting and searching Lists with 'Collections.'
import com.bruceeckel.util.*;
import java.util.*;

public class ListSortSearch {
    public static void main(String[] args) {
        List list = new ArrayList();
        Collections2.fill(list, Collections2.capitals, 25);
        System.out.println(list + "\n");
        Collections.shuffle(list);
        System.out.println("After shuffling: " + list);
        Collections.sort(list);
        System.out.println(list + "\n");
        Object key = list.get(12);
        int index = Collections.binarySearch(list, key);
        System.out.println("Location of " + key +
            " is " + index + ", list.get(" +
            index + ") = " + list.get(index));
        AlphabeticComparator comp = new AlphabeticComparator();
        Collections.sort(list, comp);
        System.out.println(list + "\n");
        key = list.get(12);
        index = Collections.binarySearch(list, key, comp);
    }
}
```

```

        System.out.println("Location of " + key +
            " is " + index + ", list.get(" +
            index + ") = " + list.get(index));
    }
} ///:~

```

这些方法的使用方式与 `Arrays` 中的一样，只不过 `List` 取代了数组。与数组相同的还有，如果使用 `Comparator` 进行排序，那么 `binarySearch()` 必须使用相同的 `Comparator`。

此程序还演示了 `Collections` 的 `shuffle()` 方法，它用来打乱 `List` 的顺序。

实用方法

`Collections` 类还有许多很实用方法：

max(Collection)	返回参数 <code>Collection</code> 中最大或最小的元素——采用 <code>Collection</code> 内含的自然比较法。
min(Collection)	
max(Collection, Comparator)	返回参数 <code>Collection</code> 中最大或最小的元素——采用 <code>Comparator</code> 进行比较。
min(Collection, Comparator)	
indexOfSubList(List source, List target)	返回 <code>target</code> 在 <code>source</code> 中第一次出现的位置。
lastIndexOfSubList(List source, List target)	返回 <code>target</code> 在 <code>source</code> 中最后一次出现的位置。
replaceAll(List list, Object oldVal, Object newVal)	使用 <code>newVal</code> 替换所有的 <code>oldVal</code> 。
reverse()	逆转所有元素的次序。
rotate(List list, int distance)	所有元素向后移动 <code>distance</code> 个位置，将末尾的元素循环到前面来。
copy(List dest, List src)	将 <code>src</code> 中的元素复制到 <code>dest</code>
swap(List list, int i, int j)	交换 <code>List</code> 中位置 <code>i</code> 与位置 <code>j</code> 的元素。通常比你自己的代码快。
fill(List list, Object o)	用对象 <code>o</code> 替换所有的元素。
nCopies(int n, Object o)	返回大小为 <code>n</code> 的 <code>List</code> ，此 <code>List</code> 不可改变，其中的引用都指向 <code>o</code> 。
enumeration(Collection)	由参数生成一个旧式的 <code>Enumeration</code> 。
list(Enumeration e)	返回由 <code>e</code> 生成的 <code>ArrayList</code> 。用来转换遗留的老代码。

注意，`min()`和`max()`只能作用于 `Collection` 对象，而不能作用于 `List`，所以你无需担心 `Collection` 是否应该被排序。（如前所述，只有在执行 `binarySearch()`之前，才确实需要对 `List` 或数组进行排序。）

```
//: c11:Utilities.java
// Simple demonstrations of the Collections utilities.
import com.bruceeckel.simpletest.*;
import java.util.*;
import com.bruceeckel.util.*;

public class Utilities {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        List list = Arrays.asList(
            "one Two three Four five six one".split(" "));
        System.out.println(list);
        System.out.println("max: " + Collections.max(list));
        System.out.println("min: " + Collections.min(list));
        AlphabeticComparator comp = new AlphabeticComparator();
        System.out.println("max w/ comparator: " +
            Collections.max(list, comp));
        System.out.println("min w/ comparator: " +
            Collections.min(list, comp));
        List sublist =
            Arrays.asList("Four five six".split(" "));
        System.out.println("indexOfSubList: " +
            Collections.indexOfSubList(list, sublist));
        System.out.println("lastIndexOfSubList: " +
            Collections.lastIndexOfSubList(list, sublist));
        Collections.replaceAll(list, "one", "Yo");
        System.out.println("replaceAll: " + list);
        Collections.reverse(list);
        System.out.println("reverse: " + list);
        Collections.rotate(list, 3);
        System.out.println("rotate: " + list);
        List source =
            Arrays.asList("in the matrix".split(" "));
        Collections.copy(list, source);
        System.out.println("copy: " + list);
        Collections.swap(list, 0, list.size() - 1);
        System.out.println("swap: " + list);
        Collections.fill(list, "pop");
        System.out.println("fill: " + list);
        List dups = Collections.nCopies(3, "snap");
```

```

System.out.println("dups: " + dups);
// Getting an old-style Enumeration:
Enumeration e = Collections.enumeration(dups);
Vector v = new Vector();
while(e.hasMoreElements())
    v.addElement(e.nextElement());
// Converting an old-style Vector
// to a List via an Enumeration:
ArrayList arrayList = Collections.list(v.elements());
System.out.println("arrayList: " + arrayList);
monitor.expect(new String[] {
    "[one, Two, three, Four, five, six, one]",
    "max: three",
    "min: Four",
    "max w/ comparator: Two",
    "min w/ comparator: five",
    "indexOfSubList: 3",
    "lastIndexOfSubList: 3",
    "replaceAll: [Yo, Two, three, Four, five, six, Yo]",
    "reverse: [Yo, six, five, Four, three, Two, Yo]",
    "rotate: [three, Two, Yo, Yo, six, five, Four]",
    "copy: [in, the, matrix, Yo, six, five, Four]",
    "swap: [Four, the, matrix, Yo, six, five, in]",
    "fill: [pop, pop, pop, pop, pop, pop, pop]",
    "dups: [snap, snap, snap]",
    "arrayList: [snap, snap, snap]"
});
}
} ///:~

```

该程序的输出可看作对每个实用方法的解释。请注意由于大小写的缘故而造成的使用 `AlphabeticComparator` 时 `min()` 和 `max()` 的差异。

设定 Collection 或 Map 为不可修改

创建一个“只读”的 `Collection` 或 `Map`，有时可以带来某些方便。`Collections` 类可以帮你达成此目的，它有一个方法，参数为原本的容器，返回值是容器的“只读”版本。此方法有四个变种，对应于 `Collection`（如果你只知道是 `Collection`）、`List`、`Set` 和 `Map`。下例将说明如何正确生成各种“只读”容器：

```

//: c11:ReadOnly.java
// Using the Collections.unmodifiable methods.
import java.util.*;
import com.bruceeckel.util.*;

```

```

public class ReadOnly {
    private static Collections2.StringGenerator gen =
        Collections2.countries;
    public static void main(String[] args) {
        Collection c = new ArrayList();
        Collections2.fill(c, gen, 25); // Insert data
        c = Collections.unmodifiableCollection(c);
        System.out.println(c); // Reading is OK
        //! c.add("one"); // Can't change it

        List a = new ArrayList();
        Collections2.fill(a, gen.reset(), 25);
        a = Collections.unmodifiableList(a);
        ListIterator lit = a.listIterator();
        System.out.println(lit.next()); // Reading is OK
        //! lit.add("one"); // Can't change it

        Set s = new HashSet();
        Collections2.fill(s, gen.reset(), 25);
        s = Collections.unmodifiableSet(s);
        System.out.println(s); // Reading is OK
        //! s.add("one"); // Can't change it

        Map m = new HashMap();
        Collections2.fill(m, Collections2.geography, 25);
        m = Collections.unmodifiableMap(m);
        System.out.println(m); // Reading is OK
        //! m.put("Ralph", "Howdy!");
    }
} ///:~

```

对特定类型的“unmodifiable”方法的调用并不产生编译期的检查，但是转换完成后，任何会改变容器内容的操作都会引起 `UnsupportedOperationException` 异常。

无论哪一种情况，在将容器设为“只读”之前，必须填入有意义的数据。装载数据后，就应该使用“unmodifiable”方法返回的引用去替换掉原本的引用。这样就不用担心无意中修改了“只读”的内容。另一方面，此方法允许你保留一份可修改的容器，作为类的 `private` 成员，然后通过某个方法，返回“只读”的引用。于是就只有你可以修改容器的内容，而别人只能读取。

Collection 或 Map 的同步控制

关键字 `synchronized` 是多线程课题中重要的一部分，第十三章将讨论此更为复杂的主题。这里，我只提醒你注意，`Collections` 类有办法能够自动同步整个容器。其语法与“`unmodifiable`”方法相似：

```
//: c11:Synchronization.java
// Using the Collections.synchronized methods.
import java.util.*;

public class Synchronization {
    public static void main(String[] args) {
        Collection c =
            Collections.synchronizedCollection(new ArrayList());
        List list =
            Collections.synchronizedList(new ArrayList());
        Set s = Collections.synchronizedSet(new HashSet());
        Map m = Collections.synchronizedMap(new HashMap());
    }
} ///:~
```

在此例中，直接将新生成的容器传递给了适当的“`synchronized`”方法；这样做就不会有任何机会暴露出不同步的版本。

快速报错（Fail fast）

Java 容器有一种保护机制，能够防止多个进程同时修改同一个容器的内容。如果你正在迭代遍历某容器，此时另一个进程介入其中，并且插入、删除或修改此容器内的某个对象，那么就会出现問題：也许迭代过程已经处理了此对象，也许还没处理，也许在调用 `size()` 之后容器的大小收缩了——还有许多可能的灾难。Java 容器类库采用“快速报错”

（fail-fast）机制。它会探查容器的任何变化，除了你的进程对容器进行的操作以外，一旦它发现其他进程修改了容器，就会立刻抛出 `ConcurrentModificationException` 异常。这就是“快速报错”的意思，它不是使用复杂的算法在事后来检查问题。

很容易就可以看到“快速报错”机制是如何工作的：你只需创建一个迭代器，然后向迭代器指向的 `Collection` 添加点什么，就像这样：

```
//: c11:FailFast.java
// Demonstrates the "fail fast" behavior.
// {ThrowsException}
import java.util.*;

public class FailFast {
```

```

public static void main(String[] args) {
    Collection c = new ArrayList();
    Iterator it = c.iterator();
    c.add("An object");
    // Causes an exception:
    String s = (String)it.next();
}
} ///:~

```

程序运行时发生了异常，因为在容器取得迭代器之后，又有东西被放入到了该容器中。当程序的不同部分修改同一个容器时，就可能导致容器的状态不一致，所以，此异常提醒你，应该修改代码。在此例中，应该在添加完所有的元素之后，再获取迭代器。

注意，对 List 使用 `get()` 取得元素时，我写的监视器就不起作用了。

未获支持的操作

使用 `Arrays.asList()`，能够将数组转换成 List：

```

//: c11:Unsupported.java
// Sometimes methods defined in the
// Collection interfaces don't work!
// {ThrowsException}
import java.util.*;

public class Unsupported {
    static List a = Arrays.asList(
        "one two three four five six seven eight".split(" "));
    static List a2 = a.subList(3, 6);
    public static void main(String[] args) {
        System.out.println(a);
        System.out.println(a2);
        System.out.println("a.contains(" + a.get(0) + ") = " +
            a.contains(a.get(0)));
        System.out.println("a.containsAll(a2) = " +
            a.containsAll(a2));
        System.out.println("a.isEmpty() = " + a.isEmpty());
        System.out.println("a.indexOf(" + a.get(5) + ") = " +
            a.indexOf(a.get(5)));
        // Traverse backwards:
        ListIterator lit = a.listIterator(a.size());
        while(lit.hasPrevious())
            System.out.print(lit.previous() + " ");
        System.out.println();
    }
}

```

```

        // Set the elements to different values:
        for(int i = 0; i < a.size(); i++)
            a.set(i, "47");
        System.out.println(a);
        // Compiles, but won't run:
        lit.add("X"); // Unsupported operation
        a.clear(); // Unsupported
        a.add("eleven"); // Unsupported
        a.addAll(a2); // Unsupported
        a.retainAll(a2); // Unsupported
        a.remove(a.get(0)); // Unsupported
        a.removeAll(a2); // Unsupported
    }
} ///:~

```

你会发现，它实际上只是部分实现了 `Collection` 和 `List` 接口。调用其他方法时导致了 `UnsupportedOperationException` 异常。`Collection` 接口包含有“可选择”的方法（Java 容器类的某些接口也是如此），而实现了容器接口的“具体类”可能“支持”，也可能不“支持”这种“可选择”的方法。调用不被支持的方法会引发 `UnsupportedOperationException` 异常，说明有编程错误。

“什么?!?”你可能会怀疑。“接口和基类的关键就在于，它们承诺这些方法将会做一些有意义的事情！此例说明它们没有遵守承诺；不但调用到了没有实现有意义行为的方法，而且它们导致程序终止了！类型安全被扔到了窗外！”

其实并没有那么糟糕。在使用 `Collection`、`List`、`Set` 或者 `Map` 时，编译器限制你只能调用当前接口中的方法，因此它与 `SmallTalk` 不同（这种语言允许你对任意对象调用任意方法，只在程序运行时才检查当前的调用是否有意义）。此外，大多数以 `Collection` 为参数的方法，只是读取 `Collection` 的内容，而 `Collection` 的“读”方法都不是“可选择”的。

这种设计方法可以避免接口数量的爆炸。其他容器类类库似乎总是对一个主题的各种变化都设计一个接口，最终形成过多的容易混淆的接口，导致人们难以学习。而由于人们总是可以发明新的接口，所以甚至不可能找出接口所有的特殊情况。“未获支持的操作”使得 Java 容器类类库达成一个重要目标：可以保证容器总是易于学习与使用；“未获支持的操作”是可以以后再学的特殊情况。要使这种思想可行，必须遵循两点：

1. `UnsupportedOperationException` 必须是很少出现的事件。也就是说，对大多数类而言，所有操作都应该能工作，只有在很特殊的情况下，才可以“不支持”某个操作。Java 容器类类库做到了这一点，因为 99% 的时间里，你使用的类不包含“未获支持的操作”——包括 `ArrayList`、`LinkedList`、`HashSet` 和 `HashMap`，以及其他具体的实现。如果你想创建一个新的 `Collection`，使之与当前类库相融，而不想定义 `Collection` 接口中的所有方法，那么这种设计为你提供了“后门”。
2. 如果一个操作“未获支持”，在实现接口的时候可能就会导致 `UnsupportedOperationException` 异常，而不是你将产品程序交给客户以后

才出现此异常，这种可能性是有道理的。毕竟，它表示编程上有错误：使用了不正确的接口实现。然而事实不见得总是如此，这正是此设计还处于实验性的地方。只有时间才能告诉我们它的效果如何。

在前面的例子中，`Arrays.asList()`会生成一个 `List`，它基于一个固定大小的数组。因此，仅支持那些不会改变数组大小的操作，对它而言是有道理的。但是换一种方式，如果采用新的接口来表现这种不同的行为（例如，“`FixedSizeList`”），就会使事情变得复杂，很快，当你再次尝试使用此类库时，就会不知从何着手了。

注意，你应该把 `Arrays.asList()`的结果，作为构造器的参数传递给 `List` 或者 `Set`，这样可以生成规范的容器，于是所有的方法都可以使用了。

对于以 `Collection`、`List`、`Set` 或者 `Map` 作为参数的方法，其文档都应该专门指出，哪些“可选择”的方法必须实现。例如，对于排序操作，需要实现 `set()`和 `Iterator.set()`方法，而用不着 `add()`和 `remove()`方法。

Java 1.0/1.1 的容器

很不幸，许多老的代码是使用 Java 1.0/1.1 的容器写成的，甚至有些新的程序也使用了这些类。因此，虽然在写新的程序时，决不应该使用旧的容器，但你仍然应该了解它们。不过旧容器功能有限，所以对它们也没太多可说的。（毕竟它们都过时了，所以我也不想强调某些设计有多糟糕。）

Vector & Enumeration

在 Java 1.0/1.1 中，`Vector` 是唯一可以自动扩展的序列，所以它被大量使用。它的缺点多到这里都难以描述（可以参见本书的第一版，可从 www.BruceEckel.com 免费下载）。基本上，可将其看作 `ArrayList`，但是具有又长又难记的方法名。在 Java 2 的容器类类库中，`Vector` 被改造过，可将其归类为 `Collection` 和 `List`，所以下面的例子可以使用 `Collection2.fill()`方法。这样做有点不妥当，可能会让人误会 `Vector` 变得好用了，实际上这样做只是为了支持 Java 2 之前的代码。

Java 1.0/1.1 版的迭代器，发明了一个新名字，“`enumeration`”，取代了为人熟知的术语。此 `Enumeration` 接口比 `Iterator` 小，只有两个名字很长的方法：`boolean hasMoreElements()`，如果此 `enumeration` 包含有元素则返回 `true`；`Object nextElement()`返回此 `enumeration` 中的下一个元素，如果还有的话（否则抛出异常）。

`Enumeration` 只是接口而不是实现，所以有时新的类库仍然使用了旧的 `Enumeration`，这令人十分遗憾，但通常不会造成伤害。虽然在你的代码中，应该尽量使用 `Iterator`，但也得有所准备，类库可能会返回给你一个 `Enumeration`。

此外，你还可以通过使用 `Collections.enumeration()`方法来从 `Collection` 生成一个 `Enumeration`，见下面的例子：

```

//: c11:Enumerations.java
// Java 1.0/1.1 Vector and Enumeration.
import java.util.*;
import com.bruceeckel.util.*;

public class Enumerations {
    public static void main(String[] args) {
        Vector v = new Vector();
        Collections2.fill(v, Collections2.countries, 100);
        Enumeration e = v.elements();
        while(e.hasMoreElements())
            System.out.println(e.nextElement());
        // Produce an Enumeration from a Collection:
        e = Collections.enumeration(new ArrayList());
    }
} ///:~

```

Java 1.0/1.1 的 `Vector` 只有 `addElement()` 一个方法，`fill()` 中用到的 `add()` 方法，是 `Vector` 转换成 `List` 时新增加的。可以使用 `elements()` 生成 `Enumeration`，然后使用它进行前序遍历。

最后一行代码创建了一个 `ArrayList`，并且使用 `enumeration()` 将 `ArrayList` 的 `Iterator` 转换成了 `Enumeration`。这样，即使有需要 `Enumeration` 的旧代码，你仍然可以使用新容器。

Hashtable

正如你在性能比较那一节看到的，基本的 `Hashtable` 与 `HashMap` 很相似，甚至方法名也相似。所以，在新的程序中，没有理由再使用 `Hashtable` 而不用 `HashMap`。

Stack

前面在使用 `LinkedList` 时，已经介绍过“栈”的概念。Java 1.0/1.1 的 `Stack` 很奇怪，竟然不是用 `Vector` 来构建 `Stack`，而是继承 `Vector`。所以它拥有 `Vector` 所有的特点和行为，再加上一些额外的 `Stack` 行为。很难了解是否设计者有意识的认为这样作特别有用，或者只是一个幼稚的设计。唯一清楚的是，在匆忙发布之前它没有经过仔细审查，因此这个糟糕的设计仍然挂在这里（但是你永远都不应该使用它）。

这里是 `Stack` 一个简单的示例，将 `String` 数组中的每个元素 `push` 压入 `Stack`：

```

//: c11:Stacks.java
// Demonstration of Stack Class.
import com.bruceeckel.simpletest.*;

```

```

import java.util.*;
import c08.Month;

public class Stacks {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Stack stack = new Stack();
        for(int i = 0; i < Month.month.length; i++)
            stack.push(Month.month[i] + " ");
        System.out.println("stack = " + stack);
        // Treating a stack as a Vector:
        stack.addElement("The last line");
        System.out.println("element 5 = " +
            stack.elementAt(5));
        System.out.println("popping elements:");
        while(!stack.empty())
            System.out.println(stack.pop());
        monitor.expect(new String[] {
            "stack = [January , February , March , April , May "+
            " , June , July , August , September , October , " +
            "November , December ]",
            "element 5 = June ",
            "popping elements:",
            "The last line",
            "December ",
            "November ",
            "October ",
            "September ",
            "August ",
            "July ",
            "June ",
            "May ",
            "April ",
            "March ",
            "February ",
            "January "
        });
    }
} ///:~

```

months 数组的每一行被 `push()` 压入 Stack，然后再从栈的顶端用 `pop()` 取出来。这里要特别强调：“可以在 Stack 对象上执行 Vector 的操作”，这不会有任何问题，因为继承带来的效果使得“Stack 是一个 Vector”，因此所有可以对 Vector 执行的操作，都可以对 Stack 执行，例如 `elementAt()`。

前面曾经说过，如果需要“栈”的行为，应该使用 `LinkedList`。

BitSet

如果想要高效率地存储大量“开-关”信息，`BitSet` 是很好的选择。不过它的效率仅是空间而言。如果你需要高效率的访问时间，`BitSet` 比数组稍慢一点。

此外，`BitSet` 的最小容量是 **long: 64 bit**。如果你存储的内容比较小，例如 8 bit，那么 `BitSet` 就浪费了一些空间。因此如果空间对你很重要，最好撰写自己的类，或者直接采用数组来存储你的标志信息。

普通的容器都会随着元素的加入而扩充其容量，`BitSet` 也是。以下示范 `BitSet` 是如何工作的：

```
//: c11:Bits.java
// Demonstration of BitSet.
import java.util.*;

public class Bits {
    public static void printBitSet(BitSet b) {
        System.out.println("bits: " + b);
        String bbits = new String();
        for(int j = 0; j < b.size() ; j++)
            bbits += (b.get(j) ? "1" : "0");
        System.out.println("bit pattern: " + bbits);
    }

    public static void main(String[] args) {
        Random rand = new Random();
        // Take the LSB of nextInt():
        byte bt = (byte)rand.nextInt();
        BitSet bb = new BitSet();
        for(int i = 7; i >= 0; i--)
            if(((1 << i) & bt) != 0)
                bb.set(i);
            else
                bb.clear(i);
        System.out.println("byte value: " + bt);
        printBitSet(bb);

        short st = (short)rand.nextInt();
        BitSet bs = new BitSet();
        for(int i = 15; i >= 0; i--)
            if(((1 << i) & st) != 0)
                bs.set(i);
    }
}
```

```

        else
            bs.clear(i);
System.out.println("short value: " + st);
printBitSet(bs);

int it = rand.nextInt();
BitSet bi = new BitSet();
for(int i = 31; i >= 0; i--)
    if(((1 << i) & it) != 0)
        bi.set(i);
    else
        bi.clear(i);
System.out.println("int value: " + it);
printBitSet(bi);

// Test bitsets >= 64 bits:
BitSet b127 = new BitSet();
b127.set(127);
System.out.println("set bit 127: " + b127);
BitSet b255 = new BitSet(65);
b255.set(255);
System.out.println("set bit 255: " + b255);
BitSet b1023 = new BitSet(512);
b1023.set(1023);
b1023.set(1024);
System.out.println("set bit 1023: " + b1023);
}
} ///:~

```

随机数发生器被用来生成随机的 `byte`、`short` 和 `int`，每一个都被转换为 `BitSet` 中相应的 `bit` 模式。因为 `BitSet` 是 64bit 的，所以没有任何生成的随机数会导致 `BitSet` 扩充容量。然后创建了一个 512bit 的 `BitSet`。构造器会分配比 `bit` 数大一倍的存储空间，不过你也可以设定为 1024bit（或更大）。

总结

回顾一遍 Java 标准类库提供的容器：

1. 数组将数字与对象联系起来。它保存类型明确的对象，查询对象时，不需要对结果做类型转换。它可以是多维的，可以保存基本类型的数据。但是，数组一旦生成，其容量就不能改变。
2. **Collection** 保存单个的元素，而 **Map** 保存相关联的键值对。
3. 像数组一样，**List** 也建立数字与对象的关联，可以认为数组和 **List** 都是排好序的容器。**List** 能够自动扩充容量。但是 **List** 不能保存基本类型，只能保存 **Object**

的引用，因此必须对从容器中取出的 **Object** 结果做类型转换。

4. 如果要进行大量的随机访问，就使用 **ArrayList**；如果要经常从 **List** 中间插入或删除元素，则应该使用 **LinkedList**。
5. 队列、双向队列以及栈的行为，由 **LinkedList** 提供支持。
6. **Map** 是一种将对象与对象相关联的设计。**HashMap** 着重于快速访问；**TreeMap** 保持“键”始终处于排序状态，所以没有 **HashMap** 快。**LinkedHashMap** 保持元素插入的顺序，也可以使用 **LRU** 算法对其重排序。
7. **Set** 不接受重复元素。**HashSet** 提供最快的查询速度，**TreeSet** 保持元素处于排序状态。**LinkedHashSet** 以插入顺序保存元素。
8. 新程序中不应该使用过时的 **Vector**、**Hashtable** 和 **Stack**。

容器是你每天都会用到的工具，它可以使你的程序更简洁、更强大、更高效。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 创建一个 **double** 数组，使用 **RandDoubleGenerator** 填充，然后打印出结果。
2. 创建一个新类 **Gerbil**，包含 **int gerbilNumber**，在构造器中初始化它（类似于本章的 **Mouse**）。添加一个方法 **hop()**，用以打印 **gerbil** 的号码。创建一个 **ArrayList**，并向其中添加一串 **Gerbil** 对象。使用 **get()** 遍历 **List**，并且对每个 **Gerbil** 调用 **hop()**。
3. 修改练习 2，使用 **Iterator** 遍历 **List**，并调用 **hop()**。
4. 使用联系 2 中的 **Gerbil** 类，将其放入 **Map** 中，将 **Gerbil** 的名字 **String**（键）与每个 **Gerbil**（值）关联起来。由 **keySet()** 获取 **Iterator**，使用它遍历 **Map**，针对每个“键”查询 **Gerbil**，然后打印出“键”，并调用 **hop()**。
5. 创建一个 **List**（**ArrayList** 和 **LinkedList** 都尝试一下），然后使用 **Collections2.countries** 填充。对 **list** 排序并打印，然后重复调用 **Collections.shuffle()** 几次，每次都打印出来，看看 **shuffle()** 每次能把 **list** 打乱到如何程度。
6. 证明除了 **Mouse** 之外，你无法将任何东西加入 **MouseListener**。
7. 修改 **MouseListener.java**，使它继承 **ArrayList** 而不是使用组合。说明此方法有何问题。
8. 改正 **CatsAndDogs.java** 程序，写一个 **Cats** 容器（使用 **ArrayList**）只接受和取出 **Cat** 对象。
9. 使用键值对填充 **HashMap**。打印结果，证明是按散列码排序的。取出键值对，按键排序，将结果置入 **LinkedHashMap**。证明后者维持元素插入的顺序。
10. 使用 **HashSet** 和 **LinkedHashSet** 重复前面的例子。
11. 创建一个新容器，用 **private ArrayList** 来保存对象。用 **Class reference** 来判断容器中的第一个对象的类型，然后只允许用户插入此类型的对象。
12. 用 **String** 数组创建一个只能存取 **String** 的容器，这样使用的时候就没有类型转换的问题了。当容器发现数组不够大的时候，应该能自动调整其内部数组的容量。在 **main()** 中比较你的容器与保存 **String** 的 **ArrayList** 的性能。

13. 重复练习 12, 做一个 `int` 的容器, 然后比较它和保存 `Integer` 对象的 `ArrayList` 的性能。性能测试应该包括“对容器中每个对象都做递增”的操作。
14. 使用 `com.bruceeckel.util` 中的使用方法, 为每种基本类型, 以及 `String` 对象各创建一个数字, 然后用生成器填充此数组, 再使用合适的 `print()` 方法打印此数组。
15. 创建一个能生成你最喜欢的电影的名字的生成器 (可以使用白雪公主或星球大战之类的), 如果名字用光了, 就绕到最前面。使用 `com.bruceeckel.util` 里面的方法来填充数组、`ArrayList`、`LinkedList` 以及两种 `Set`, 然后打印每种容器。
16. 创建一个包括两个 `String` 对象的类, 然后做一个只比较第一个字符串的 `Comparable`。用 `geography` 的生成器来生成此类对象, 填充数组和 `ArrayList`。验证一下, 排序能正常工作。再做一个只比较第二个 `String` 的 `Comparator`, 然后验证一下排序也能正常运作。然后用 `Comparator` 进行二分查询。
17. 修改练习 16, 使用字母表顺序。
18. 用 `Arrays2.RandStringGenerator` 填充 `TreeSet`, 使其按字母顺序排序。打印 `TreeSet` 看看它如何排序的。
19. 分别创建一个 `ArrayList` 和 `LinkedList`, 用 `Collections2.captials` 生成器来填充此容器。用普通的 `Iterator` 打印此 `List`, 然后用 `ListIterator`, 按隔一个位置插入一个对象的方式, 把两个列表合并起来。然后从 `List` 的末尾开始向前移动, 并且执行插入操作。
20. 写一个方法, 使用 `Iterator` 遍历 `Collection`, 并打印每个对象的 `hashCode()`。填充各种类型的 `Collection`, 然后对其使用此方法。
21. 纠正 `InfiniteRecursion.java` 中的问题。
22. 写一个类, 在其中创建一个已经初始化的对象数组。使用此数组填充 `List`。使用 `subList()` 生成此 `List` 的子集, 然后使用 `removeAll()` 将子集从 `List` 中移除。
23. 修改第七章的练习 6, 使用 `ArrayList` 保存 `Rodent`, 并使用 `Iterator` 遍历 `Rodent` 的序列。记住, `ArrayList` 只能保存 `Object`, 所以在其中使用 `Rodent` 元素时必须做类型转换。
24. 依据 `Queue.java` 的例子, 创建 `Deque` 类, 并做测试。
25. 在 `Statistics.java` 使用 `TreeMap`。添加代码, 测试 `HashMap` 和 `TreeMap` 的性能区别。
26. 生成一个 `Map` 和 `Set`, 使其包含所有以“A”开头的国家。
27. 使用 `Collections2.countries`, 用同样的数据多次填充 `Set`, 然后验证此 `Set` 中没有重复的元素。使用不同的 `Set` 做此测试。
28. 修改 `Statistics.java`, 写一个程序重复做测试, 观察是否某个数字比别的数字出现的次数多。
29. 使用 `Counter` 对象的 `HashSet` 重写 `Statistics.java` (修改 `Counter` 使其可以在 `HashSet` 中使用)。看看哪种方法更好?
30. 使用 `String` “键”和你选择的对象填充 `LinkedHashMap`。然后从中提取键值对, 以键排序, 然后重新插入此 `Map`。
31. 修改练习 16 的类, 使它能在 `HashSet` 中使用, 并可作 `HashMap` 中的键。
32. 参考 `SlowMap.java`, 写一个 `SlowSet`。
33. 创建一个 `FastTraversalLinkedList`, 令其内部使用 `LinkedList` 进行快速插入

和查询，并使用 `ArrayList` 做快速遍历，还有 `get()` 操作。修改 `ArrayPerformance.java` 对其做测试。

34. 将 `Map1.java` 中的测试应用与 `SlowMap`，验证并修改它，使其能正常工作。
35. 令 `SlowMap` 实现完整的 `Map` 接口。
36. 修改 `MapPerformance.java`，令其包含对 `SlowMap` 的测试。
37. 修改 `SlowMap`，使用 `MPair` 对象的 `ArrayList` 代替两个 `ArrayList`。验证其是否工作正常。使用 `MapPerformance.java` 测试新 `Map` 的速度。然后修改 `put()` 方法，令其插入键值对后就执行 `sort()`，修改 `get()`，令其使用 `Collections.binarySearch()` 查询键。比较新旧版本的性能。
38. 向 `CountedString` 添加一个 `char` 属性，在构造器中对其初始化，修改 `hashCode()` 和 `equals()`，令其包含对此 `char` 的计算。
39. 修改 `SimpleHashMap`，令其能够报告冲突，通过添加相同的数据来做测试，令你能够看到冲突。
40. 修改 `SimpleHashMap`，令其报告要探测多少次才能发现冲突。也就是说，插入元素时，对 `Iterator` 调用多少次 `next()` 才能在 `LinkedList` 中发现此元素已经存在。
41. 实现 `SimpleHashMap` 的 `clear()` 和 `remove()` 方法。
42. 令 `SimpleHashMap` 实现完整的 `Map` 接口。
43. 向 `SimpleHashMap` 添加一个 `private rehash()` 方法，它在负载因子到达 0.75 时会被调用。`rehash()` 将桶的数量扩大一倍，然后查找大于桶的数量的第一个质数。
44. 模仿 `SimpleHashMap.java`，写一个 `SimpleHashSet`，并作测试。
45. 修改 `SimpleHashMap` 令其使用 `ArrayList` 代替 `LinkedList`。修改 `MapPerformance.java` 令其比较两种不同实现的性能。
46. 使用 JDK 的 HTML 帮助文档(可由 java.sun.com 下载)，查询 `HashMap` 类。创建一个 `HashMap`，使用元素填充它，决定负载因子。测试查询速度，然后创建一个容量更大的 `HashMap`，尝试是否能提高速度，将旧 `Map` 中的内容复制到新 `Map` 中，然后对新的 `Map` 测试查询速度。
47. 第八章中 `GreenhouseController.java` 那个例子由四个文件组成。修改 `Controller.java` 中的 `Controller` 类，使用 `LinkedList` 代替 `ArrayList`，然后使用 `Iterator` 遍历事件集合。
48. (挑战题) 写一个自己做散列的 `Map` 类，定制键的类型：这里使用 `String`。不要继承 `Map`。令 `put()` 和 `get()` 方法只接受 `String` 对象作为键，而不是 `Object`。任何使用到键的地方都不要使用通用的类型 `Object`，只使用 `String`，以此避免类型转换的开销。你的目标是令此定制的实现尽可能地快。修改 `MapPerformance.java`，比较你的版本与 `HashMap` 的性能。
49. (挑战题) 在 Java 源码类库中找到 `List` 的源代码。复制此代码，然后做一个只保存 `int` 的 `intList`。考虑如何为所有的基本类型做一个特别的 `List`。思考如果要令 `LinkedList` 能够使用所有的基本类型，应该如何做？
50. 修改 `c08:Month.java`，使其实现 `Comparable` 接口。
51. 修改 `CountedString.java` 中的 `hashCode()`，通过 `id` 删除重复的元素，并且证明 `CountedString` 仍能正常作为键使用。这种方式有没有问题？

第十二章 Java I/O 系统

对白酿成语言的设计者来说，创建一个好的输入/输出(I/O)系统是一项更艰难的任务。

现有的大量不同方案已经说明了这一点。挑战似乎来自于要涵盖所有的可能性。不仅存在各种用于通信的 I/O 源端和接收端（文件、控制台、网络链接等），而且还需要以多种不同的方式与它们进行通信（顺序、随机存取、缓冲、二进制、按字符、按行、按字等）。

Java 类库的设计者是通过创建大量的类来解决这个难题的。一开始，可能会对 Java I/O 系统提供了如此多的类而感到不知所措（具有讽刺意味的是，Java I/O 设计的初衷是为了避免过多的类）。自从 Java 1.0 版本以来，Java 的 I/O 类库发生了明显改变，在原来面向字节的类中添加了面向字符和基于 Unicode 的类。在 JDK1.4 中，添加了 `nio` 类（对于“新 I/O”这个称呼，从现在这个名字我们仍将要若干年）用于改进性能及功能。因此，在充分理解 java I/O 系统以便正确地运用之前，我们需要学习相当数量的类。另外，很有必要理解 I/O 类库的演化过程，即使我们的第一反应是“不要用历史打扰我，只需要告诉我怎么用。”问题是，如果缺乏历史的眼光，很快我们就会对什么时候该使用某些类，什么时候不该使用它们而感到迷惑。

本章就介绍 Java 标准类库中各种各样的类以及它们的用法。

文件类

在学习那些真正用于在流中读写数据的类之前，让我们先看看一个实用工具，它提供了一个用于帮助我们处理文件目录事务的类库。

文件类这个名字具有一定的误导性；我们可能会认为它指代的是文件，实际上却并非如此。它既能代表一个特定文件的名称又能代表一个目录下的文件集合的名称。如果它指的是一个文件集，我们就可以对此集合调用 `list()` 方法，这个方法会返回一个字符数组。我们很容易就可以理解返回的是一个数组而不是某个更具灵活性的类容器，因为元素的个数是固定的，如果我们想取得另一个目录下的列表，只需要再创建一个不同的文件对象就可以了。实际上，“文件路径”对这个类来说是个更好的名字。本节所示范的这个类的用法，包括了与它相关的 `FilenameFilter` 接口。

目录列表器

假设我们想查看一个目录列表，可以用两种方法列出文件对象。如果我们调用不带参数的 `list()` 方法，便可以获得此文件对象包含的全部列表。然而，如果我们想获得一个受限列表——例如，想得到所有扩展名为 `.java` 的文件——那么我们就要用到“目录过滤器”，这个类会告诉我们怎样显示符合条件的文件对象。

下面给出一个例子的代码，注意：通过使用 `java.util.Arrays.sort()` 和第 11 章中定义的 `AlphabeticComparator`，可以很容易地对结果进行排序（按字母顺序）。

```

//: c12:DirList.java
// Displays directory listing using regular expressions.
// {Args: "D.*\\.java"}
import java.io.*;
import java.util.*;
import java.util.regex.*;
import com.bruceeckel.util.*;

public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new DirFilter(args[0]));
        Arrays.sort(list, new AlphabeticComparator());
        for(int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
}

class DirFilter implements FilenameFilter {
    private Pattern pattern;
    public DirFilter(String regex) {
        pattern = Pattern.compile(regex);
    }
    public boolean accept(File dir, String name) {
        // Strip path information, search for regex:
        return pattern.matcher(
            new File(name).getName()).matches();
    }
}

} ///: ~

```

这里，DirFilter 类“实现”了 FilenameFilter 接口。有必要先看看 FilenameFilter 接口是多么的简单：

```

public interface FilenameFilter {
    boolean accept(File dir, String name);
}

```

这表示此种类型的对象一定要提供一个 `accept()` 方法。创建这个类的目的在于把 `accept()` 方法提供给 `list()` 使用，使 `list()` 可以回调 `accept()` 进而以决定哪些文件包含在

列表中。因此，这种结构也常常称为“回调（call back）”。更具体地说，这是一个策略模式的例子，因为 `list()` 实现了基本的功能，而且我们按照 `FilenameFilter` 的形式提供了这个策略，以便完善 `list()` 在提供服务时所需的算法。因为 `list()` 接受 `FilenameFilter` 对象作为参数，这意味着我们可以传递实现了 `FilenameFilter` 接口的任何对象，用以选择（甚至在运行时）`list()` 方法的行为。回调的目的就是提供了代码行为的灵活性。

DirFilter 说明：正因为一个接口仅是一组方法集，所以我们也就没有被限定为只能编写那些方法。（然而，在一个接口中，我们必须为其所有方法提供定义。）在这种情形下，也就可以创建 `DirFilter` 的构造器了。

`accept()` 方法必须接受一个文件对象，此对象代表某个特定文件所在目录以及包含那个文件名的一个字符串。我们可以选择使用或忽视这些参数，但是至少会用到那个文件的名称。记住一点：`list()` 方法会为此目录对象下的每个文件名调用 `accept()`，来判断该文件是否包含在内；判断结果由 `accept()` 返回的布尔值表示。

为了确定所处理的元素仅有文件名并且不包含路径信息，我们必须持有一个 `String` 对象，并在其外创建一个文件对象，然后调用 `getName()`，该方法剔除掉所有的路径信息（以平台独立的方式）。接着 `accept()` 会使用一个正则表达式的 `matcher` 对象，来查看此正则表达式 `regex` 是否匹配这个文件的名称。通过使用 `accept()`，`list()` 方法会返回一个数组。

匿名内部类

这个例子很适合用一匿名内部类（第 8 章介绍过）进行改写。首先创建一个 `filter()` 方法，它会返回一个指向 `FilenameFilter` 的引用：

```
//: c12:DirList2.java
// Uses anonymous inner classes.
// {Args: "D.*\*.java"}
import java.io.*;
import java.util.*;
import java.util.regex.*;
import com.bruceeckel.util.*;

public class DirList2 {
    public static FilenameFilter filter(final String regex) {
        // Creation of anonymous inner class:
        return new FilenameFilter() {
            private Pattern pattern = Pattern.compile(regex);
            public boolean accept(File dir, String name) {
                return pattern.matcher(
                    new File(name).getName()).matches();
            }
        }; // End of anonymous inner class
    }
}
```

```

    }
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(filter(args[0]));
        Arrays.sort(list, new AlphabeticComparator());
        for(int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
} ///: ~

```

注意: 传向 `filter()` 的参数必须是 `final` 类型的。这在匿名内部类中是必须的, 这样它才能够使用在该类范围之外的对象。

这个设计有所改进, 因为现在 `FilenameFilter` 类紧密地和 `DirList2` 绑定在一起。然而, 我们可以进一步修改该方法, 定义一个作为 `list()` 参数的匿名内部类; 这样一来程序会变得更小。

```

///: c12:DirList3.java
// Building the anonymous inner class "in-place."
// {Args: "D.*\*.java"}
import java.io.*;
import java.util.*;
import java.util.regex.*;
import com.bruceeckel.util.*;

public class DirList3 {
    public static void main(final String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new FilenameFilter() {
                private Pattern pattern = Pattern.compile(args[0]);
                public boolean accept(File dir, String name) {
                    return pattern.matcher(
                        new File(name).getName()).matches();
                }
            });
        Arrays.sort(list, new AlphabeticComparator());
        for(int i = 0; i < list.length; i++)

```

```

        System.out.println(list[i]);
    }
} ///: ~

```

既然匿名内部类直接使用 `args[0]`, 那么传递给 `main()` 方法的参数现在就是 `final` 类型的。

这个例子展示了匿名内部类怎样通过创建特定的、一次性的类来解决问题的。此方法的一个优点就是将解决特定问题的代码隔离聚拢于一点。而另一方面, 这种方法却不易阅读, 因此要谨慎使用。

目录的检查及创建

文件类不仅仅只表示存在的文件或目录。我们也可以用文件对象来创建新的目录或不存在的整个目录路径。我们还可以查看文件的特性 (如: 大小, 最后修改日期, 读/写), 来检查某个文件对象代表的是一个文件还是一个目录, 并可以删除这个文件。下面这段程序展示了文件类的一些其他方法 (请参考 java.sun.com 上 HTML 文档, 获得全套说明)。

```

///: c12:MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
// {Args: MakeDirectoriesTest}
import com.bruceeckel.simpletest.*;
import java.io.*;

public class MakeDirectories {
    private static Test monitor = new Test();
    private static void usage() {
        System.err.println(
            "Usage: MakeDirectories path1 ... \n" +
            "Creates each path \n" +
            "Usage: MakeDirectories -d path1 ... \n" +
            "Deletes each path \n" +
            "Usage: MakeDirectories -r path1 path2 \n" +
            "Renames from path1 to path2");
        System.exit(1);
    }
    private static void fileData(File f) {
        System.out.println(
            "Absolute path: " + f.getAbsolutePath() +
            "\n Can read: " + f.canRead() +
            "\n Can write: " + f.canWrite() +
            "\n getName: " + f.getName() +
            "\n getParent: " + f.getParent() +
            "\n getPath: " + f.getPath() +

```



```

        "\n length: " + f.length() +
        "\n lastModified: " + f.lastModified());
    if(f.isFile())
        System.out.println("It's a file");
    else if(f.isDirectory())
        System.out.println("It's a directory");
}
public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
        if(args.length != 3) usage();
        File
            old = new File(args[1]),
            rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);
        return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
        count++;
        del = true;
    }
    count--;
    while(++count < args.length) {
        File f = new File(args[count]);
        if(f.exists()) {
            System.out.println(f + " exists");
            if(del) {
                System.out.println("deleting..." + f);
                f.delete();
            }
        }
        else { // Doesn't exist
            if(!del) {
                f.mkdirs();
                System.out.println("created " + f);
            }
        }
        fileData(f);
    }
    if(args.length == 1 &&

```

```

        args[0].equals("MakeDirectoriesTest"))
monitor.expect(new String[] {
    "%% (MakeDirectoriesTest exists"+
        "|created MakeDirectoriesTest)",
    "%% Absolute path: "
        + "\\S+MakeDirectoriesTest",
    "%% Can read: (true|false)",
    "%% Can write: (true|false)",
    " getName: MakeDirectoriesTest",
    " getParent: null",
    " getPath: MakeDirectoriesTest",
    "%% length: \\d+",
    "%% lastModified: \\d+",
    "It's a directory"
});
    }
} ///:~

```

我们可以看到在 `fileDate()` 中，用到了多种不同的文件特征查询方法来显示文件或目录路径的信息。

`main()` 方法首先调用的是 `renameTo()`，用来重命名（或移动）一个文件到由参数所指示的另一个完全不同的路径（也就是另一文件对象）下面。这同样适用于任意长度的文件目录。

实践上面的程序，我们会发现，我们可以产生任意复杂的目录路径，因为 `makedirs()` 可以为我们做好这一切。

输入和输出

I/O 类库中通常使用“流（stream）”这个抽象概念，它代表任何有能力产出数据的数据源对象或者是有能力接收数据的接收端对象。“流”屏蔽了实际的 I/O 设备中处理数据的细节。

Java 类库中的 I/O 类分成输入和输出两部分，可以在 JDK 文档里的类层次结构中查看到。通过继承，任何自 `InputStream` 或 `Reader` 衍生而来的类都含有名为 `read()` 的基本方法，用于读取单个字节或者字节数组。同样地，任何自 `OutputStream` 或 `Writer` 衍生而来的类都含有名为 `write()` 的基本方法，用于写单个字节或者字节数组。但是，我们通常不会用到这些方法，它们存在是因为别的类可以使用它们，以便提供更有用的接口。因此，我们很少使用一个单一的类来创建流对象，相反我们会通过叠合多个对象来提供所期望的功能。实际上，Java 中“流”类库让人迷惑的主要原因就在于：创建一个单一的结果流，却需要创建多个对象。

有必要按照这些类的功能对它们进行分类。在 Java1.0 中，类库的设计者首先限定与输入有关的所有类都应该从 `InputStream` 继承，而与输出有关的所有类都应该从 `OutputStream` 继承。

InputStream 类型

`InputStream` 的作用是用来表示那些从不同数据源产生输入的类。这些数据源包括：

- 1. 字节数组
- 2. `String` 对象
- 3. 文件
- 4. “管道”，工作方式与实际管道相似：从一端输入，从另一端输出。
- 5. 一个由其他种类的流组成的序列，以便我们可以将它们收集合并到某一单一的流内。
- 6. 其他数据源，如 Internet 连接等（在 *Thinking in Enterprise Java* 中讲述）。

每一种数据源都有相应的 `InputStream` 子类。另外，`FilterInputStream` 也属于一种 `InputStream`，为“decorator”类提供基类，其中，“decorator”类可以把属性或有用的接口与输入流连接在一起。我们稍后再讨论它。

表 12-1. `InputStream` 类型

类	功能	构造器参数
		如何使用
ByteArray-InputStream	允许将内存的缓冲区当作 InputStream 使用。	从中提取出字节的缓冲区 作为一种数据源：将其与 FilterInputStream 对象相连以提供有用接口
StringBuffer-InputStream	将 String 转换成 InputStream 。	字符串。底层实际实现是 StringBuffer . 作为一种数据源：将其与 FilterInputStream 对象相连以提供有用接口
File-InputStream	用于从文件中读取信息。	字符串，表示文件名、文件或 FileDescriptor 对象。 作为一种数据源：将其与 FilterInputStream 对象相连以提供有用接口
Piped-InputStream	产生用于写入相关 PipedOutput-Stream	PipedOutputStream 作为多线程中数据源：将

	的数据。实现“管道化”概念。	其与 FilterInputStream 对象相连以提供有用接口。
Sequence-InputStream	将两个或多个 InputStream 对象转换成单一 InputStream 。	两个 InputStream 对象或一个容纳 InputStream 对象的 Enumeration 。 作为一种数据源：将其与 FilterInputStream 对象相连以提供有用接口
Filter-InputStream	抽象类，作为修饰器的接口。其中，修饰器为其他的 InputStream 类提供有用功能 见表 12-3。	见表 12-3。 见表 12-3。

OutputStream 类型

这部分包含的类决定了我们要输出到什么地方：字节数组（非字符串，并假设我们可以用字节数组创建一个）、文件或管道。

另外，**FilterOutputStream** 为“修饰器（decorator）”类提供了一个基类，“修饰器”类把属性或者有用的接口与输出流连接了起来。

表 12-2. **OutputStream** 类型

类	功能	构造器参数
		如何使用
ByteArray-OutputStream	在内存中创建缓冲区。所有送往 stream 的数据都要放置在此缓冲区。	缓冲区初始化尺寸 用于指定数据的目的地：将其与 FilterOutputStream 对象相连以提供有用接口。
File-OutputStream	用于将信息写至文件。	字符串，表示文件名、文件或 FileDescriptor 对象 指定数据的目的地：Co 将其与 FilterOutputStream 对象相连以提供有用接口。
Piped-OutputStream	任何写入其中的信息都会	PipedInputStream

	自动作为相关 PipedInput-Stream 的输出。 实现“管道化”概念。	指定多线程数据的目的地： 将其与 FilterOutputStream 对象 相连以提供有用接口。
Filter-OutputStream	抽象类，作为装饰器接口。 其中，装饰器为其他 OutputStream 提供 有用功能 见表 12-4。	见表 12-4.
		见表 12-4.

添加属性和有用的接口

利用层叠的数个对象为单个对象动态地和透明地添加职责的方式，称作“修饰器”模式。（模式¹是 *Thinking in Patterns*（用java）中讨论的主题，见 www.BruceEckel.com）。修饰器模式规定所有封装于初始对象内部的对象具有相同的接口。这使得修饰器的基本应用具有透明性——我们可以向修饰过或没有修饰过的对象发送相同的消息。这正是Java I/O类库里存在“filter”类的原因所在：抽象类“filter”是所有修饰类的基类。（修饰器必须具有和它所修饰的对象相同的接口，但是修饰器也可以扩展接口，这种情况发生在几种“filter”类中）。

在直接使用扩展子类的方法时，如果导致产生了大量的、用以满足所需的各种可能性组合的子类，这时通常就会使用修饰器——处理太多的子类已不太实际。Java I/O类库需要多种不同性质的组合，这正是使用修饰器模式的理由所在²。但是，修饰器模式也有一个缺点：在我们编写程序时，它给我们提供了相当多的灵活性（因为我们可以很容易地混合和匹配属性），但是它同时也增加了我们代码的复杂性。Java IO类库操作不便的原因在于：我们必须创建许多类——“核心”IO类型加上所有的修饰器——才能得到我们所希望的单个IO对象。

FilterInputStream 和 FilterOutputStream 是提供给修饰接口用于控制特定输入流（InputStream）和输出流（OutputStream）的两个类，它们的名字并不是很直观。FilterInputStream 和 FilterOutputStream 自 I/O 类库中的基类——输入流（InputStream）和输出流（OutputStream）衍生而来，这两个类是修饰器的必要条件（以便能为所有正在被修饰的对象提供通用接口）。

通过 FilterInputStream 从 InputStream 中读入数据

FilterInputStream 类能够完成两件完全不同的事情。其中，DataInputStream 允许我们读取不同的基本类型数据以及 String 对象（所有方法都以“read”开头，例如 readByte()，

¹ 设计模式，Erich Gamma *et al*，Addison-Wesley 1995。

² 很难说这就是一个很好的设计选择，尤其是，与其他程序设计语言中的简单I/O类库相比较。但它却是如此选择的一个恰当理由。

readFloat()等等)。搭配相应的 DataOutputStream，我们就可以通过数据“流”将基本类型的数据从一个地方迁移到另一个地方。具体是哪些“地方”是由表 10.1 中的那些类决定的。

其他类则在内部修改 InputStream 的行为方式：缓冲或未缓冲、是否保留它所读过的行（允许我们查询行数或设置行数），以及是否把单一字符推回输入流等等。最后两个类看起来更像是为了创建一个编译器（也就是说，它们看起来很像是在对构建 Java 编译器提供支持），因此我们在常规编程中不会用到它们。

我们几乎每次都要对输入进行缓冲，不管我们正在连接的是什么 I/O 设备，所以，I/O 类库把无缓冲输入而不是缓冲输入作为特殊情况（或仅是方法调用）就显得更加合理了。

Table 12-3. FilterInputStream 类型

类	功能	构造器参数
		如何使用
Data-InputStream	与 DataOutputStream 与搭配使用，因此我们可以按照可移植方式从流读取基本数据类型（ int , char , long , 等）	InputStream 包含用于读取基本类型数据的全部接口。
Buffered-InputStream	使用它可以防止每次读取时都得进行实际写操作。 代表“使用缓冲区”	InputStream , 缓冲区大小 本质上不提供接口只是使用缓冲区所必需的。 与接口对象搭配。
LineNumber-InputStream	跟踪输入流中的行数; 可用 getLineNumber() 和 setLineNumber(int) .	InputStream 仅增加了行号，因此可能要与接口对象搭配使用。
Pushback-InputStream	具有“one byte push-back buffer”。因此可以将读到的最后一个字符回退。	InputStream 通常作为编译器的浏览器，之所以包含在内是因为 Java 编译器的需要，我们可能永远不会用到。

通过 FilterOutputStream 向 OutputStream 写入

与 DataInputStream 对应的是 DataOutputStream，它可以对各种基本数据类型以及 String 对象格式化到“流”中，以便在任何机器上的任何 DataInputStream 都能够读取它们。所有方法都以“write”开头，例如 writeByte(), writeFloat() 等等。

PrintStream 最初的目的便是为了以可视化格式打印所有的基本数据类型以及 String 对象。这和 DataOutputStream 不同，后者的目的是将数据元素置入“流”中，以便 DataOutputStream 能够可移植地重构它们。

PrintStream 内有两个重要的方法：print()和 println()。它们已进行了重载处理，可打印出各种数据类型。print()和 println()之间的差异是后者在操作完毕后会添加一个换行符。

PrintStream 可能会有些问题，因为它捕捉了所有的 IOExceptions(因此，我们必须使用 checkError () 自行测试错误状态，如果发生错误返回 true)。另外，PrintStream 也未完全国际化，不能以平台无关的方式处理换行动作(这些问题在 PrintWriter 中得到了解决，在后面讲述)。

BufferedOutputStream 是一个修改过的 OutputStream，它对数据流使用缓冲技术；因此当每次向流写入时，我们不必每次都进行实际上的物理写动作。所以我们可能更常用到它。

Table 12-4. FilterOutputStream 类型

类	功能	构造器参数
		如何使用
Data-OutputStream	与 DataInputStream 搭配使用，因此可以按照可移植方式向流中写入基本数据类型数据 (int, char, long, 等)	OutputStream
		包含用于写入基本类型数据的全部接口。
PrintStream	用于产生格式化输出。 其中 DataOutputStream 处理数据的存储， PrintStream 处理显示。	OutputStream, 选择 boolean 值以指示是否在每次换行时清空缓冲区。
		应该是对 OutputStream 对象的“final”封装。我们可能经常使用到它。
Buffered-OutputStream	使用它以避免每次发送数据时都要进行实际的写操作 代表“使用缓冲区。” 我们可以调用 flush() 清空缓冲区。	OutputStream, 缓冲区大小。
		本质上并不提供接口，只是使用缓冲区所必需的。 与接口对象搭配。

读和写

Java1.1 对基本的 I/O“流”类库进行了重大的修改。当我们初次看见 Reader 和 Writer 类时，可能会以为这是两个用来替代 InputStream 和 OutputStreamt 的类。但实际上并不是这样。尽管一些原始的“流”类库不再被使用（如果使用它们，则会收到编译器的警告信息），但是 InputStream 和 OutputStreamt 在以面向字节形式的 I/O 中仍可以提供极有价值的功能，Reader 和 Writer 则提供兼容 Unicode 与面向字符的 I/O 的功能。另外：

- 1. Java1.1 向 InputStream 和 OutputStreamt 继承层次结构中添加了一些新类，所以很明显在这些层次结构中的类是不会被取代的。
- 2. 有时我们必须把来自于“字节”层次结构中的类和“字符”层次结构中的类结合起来使用。为了实现这个目的，要用到“适配器（adapter）”类：InputStreamReader 可以把 InputStream 转换为 Reader，而 OutputStreamWriter 可以把 OutputStream 转换为 Writer。

设计 Reader 和 Writer 继承层次结构主要是为了国际化。老的 I/O 流继承层次结构仅支持 8 位字节流，并且不能很好地处理 16 位的 Unicode 字符。既然 Unicode 是用来国际化的（Java 本身的 char 也是 16 位的 Unicode），因此添加 Reader 和 Writer 继承层次结构就是为了在所有的 I/O 操作中都支持 Unicode。另外，新类库的设计使得它的操作比旧类库更快。

一如本书惯例，我会尽力给出所有类的概观，但是我还要假定你会自行使用 JDK 文档查看细节，例如方法的详尽列表。

数据的来源和去处

几乎所有原始的 Java I/O“流”类都有相应的 Reader 和 Writer 类来提供其本身就具备的 Unicode 操作。然而，在某些场合面向字节的 InputStreams 和 OutputStream 才是正确的解决方案；特别地，java.util.zip 类库就是面向字节的而不是面向字符的。因此，最明智的做法是尽量尝试使用 Reader 和 Writer，一旦程序代码无法成功编译，我们就会发现自己不得不使用面向字节的类库。

下面的表展示了在两个继承层次结构中，信息的来源和去处（即数据物理上来自哪里及去向哪里）之间的对应关系

Sources & Sinks: Java 1.0 class	Corresponding Java 1.1 class
InputStream	Reader adapter: InputStreamReader
OutputStream	Writer adapter:

	OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(no corresponding class)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

大体上，我们会发现这两个不同的继承层次结构中的接口相似但不完全相同。

更改“流”的行为

对于输入流和输出流来说，为了满足特殊需要，我们会使用 `FilterInputStream` 和 `FilterOutputStream` 的修饰器子类来修改“流”。`Reader` 和 `Writer` 的类继承层次结构继续沿用相同的思想——但是并不完全相同。

在下表中，相对于前一表格来说，左右之间的对应关系的近似程度更加粗略一些。造成这种差别的原因是因为类的组织形式不同；尽管 `BufferedOutputStream` 是 `FilterOutputStream` 的子类，但是 `BufferedWriter` 并不是 `FilterWriter` 的子类（尽管 `FilterWriter` 是抽象类而且没有任何子类。因此把它放在那里，也只是把它作为一个占位符或仅仅让我们不会疑惑它在哪里）。然而，这些类的接口却十分相似。

Filters: Java 1.0 class	Corresponding Java 1.1 class
FilterInputStream	<code>FilterReader</code>
FilterOutputStream	<code>FilterWriter</code> (抽象类，没有子类)
BufferedInputStream	<code>BufferedReader</code> (也有 <code>readLine()</code>)
BufferedOutputStream	<code>BufferedWriter</code>
DataInputStream	使用 <code>DataInputStream</code> (除了当我们需要使用 <code>readLine()</code> ，这时应该使用 <code>BufferedReader</code>)
PrintStream	<code>PrintWriter</code>
LineNumberInputStream (deprecated)	<code>LineNumberReader</code>
StreamTokenizer	<code>StreamTokenizer</code>

	(而是使用接受 Reader 的构造器)
PushBackInputStream	PushBackReader

有一点很清楚：无论我们何时使用 `readLine()`，都不应该使用 `DataInputStream`（这会遭到编译器的强烈反对），而应该使用 `BufferedReader`。除了这一点，`DataInputStream` 仍是 I/O 类库的首选成员。

为了更容易地过渡到使用 `PrintWriter`，它提供了一个既能接受 `Writer` 对象又能接受任何 `OutputStream` 对象的构造器。然而，`PrintWriter` 对格式化的支持程度并不比 `PrintStream` 强；二者的接口几乎是一样的。

`PrintWriter` 构造器还有一个选项，就是自动执行清空，如果构造器设置选择此项，则在每个 `Println()` 执行之后，便会自动清空。

未发生变化的类

有一些介于 Java1.0 和 Java1.1 之间的类则留置不变。

Java 1.0 classes without corresponding Java 1.1 classes
DataOutputStream
File
RandomAccessFile
SequenceInputStream

特别是 `DataOutputStream`，在使用时没有任何变化；因此如果以“可传输的”格式存储和检索数据，我们就要用到 `InputStream` 和 `OutputStream` 继承层次结构。

自我独立的类：RandomAccessFile

`RandomAccessFile` 适用于由大小已知的记录组成的文件，以便我们可以使用 `seek()` 将记录从一处转移到另一处，然后读取或者修改记录。文件中记录的大小不一定都相同，只要我们能够确定那些记录有多大以及它们在文件中的位置即可。

最初，我们可能难以相信 `RandomAccessFile` 不是 `InputStream` 或者 `OutputStream` 继承层次结构的一部分。除了实现了 `DataInput` 和 `DataOutput` 接口

（`DataInputStream` 和 `DataOutputStream` 也实现了这两个接口）之外，它和这两个继承层次结构没有任何关联。它甚至不使用 `InputStream` 和 `OutputStream` 类中已经存在的任何功能。它是一个完全独立的类，从头开始编写其所有方法的类。这么做的原因是因为

`RandomAccessFile` 拥有和别的 I/O 类型本质不同的行为，因为我们可以一个文件内向前和向后移动。在任何情况下，它都是自我独立的，直接从 `Object` 衍生而来。

从本质上来说，`RandomAccessFile` 的工作方式类似于把 `DataInputStream` 和 `DataOutputStream` 组合起来使用，其中方法 `getFilePointer()` 用于查找当前所处的文件位置，`seek()` 用于在文件内移至新的位置，`length()` 用于判断文件的最大尺寸。另外，其构造器还需要第二参数（和 C 中的 `fopen()` 相同）用来指示我们只是想随机读（“r”）还是既读又写（“rw”）。它并不支持只写文件，这表明 `RandomAccessFile` 若是从 `DataInputStream` 继承而来也可能会运行得很好。

只有 `RandomAccessFile` 支持搜寻方法，并且只适用于文件。`BufferedInputStream` 却允许标注（`mark()`）位置（其值存储于内部某个简单变量内）和重新设定位置（`reset()`），但这些功能很有限，不是非常有用。

在 JDK1.4 中，`RandomAccessFile` 的大多数功能但不是全部，由 `NIO` 存储映射文件（memory-mapped file）所取代，本章稍后会讲述。

I/O 流的典型使用方式

尽管我们可以通过不同的方式组合 I/O“流”类，但我们可能也就只会用到其中的几种组合。下面的例子可以作为基本参考；它展示了典型 I/O 配置的创建和使用。注意每个配置都以一个注释编号和标题开始，其中标题对应于随后文本内的适当说明。

```
//: c12:IOStreamDemo.java
// Typical I/O stream configurations.
// {RunByHand}
// {Clean: IODemo.out,Data.txt,rtest.dat}
import com.bruceeckel.simpletest.*;
import java.io.*;

public class IOStreamDemo {
    private static Test monitor = new Test();
    // Throw exceptions to console:
    public static void main(String[] args)
        throws IOException {
        // 1. Reading input by lines:
        BufferedReader in = new BufferedReader(
            new FileReader("IOStreamDemo.java"));
        String s, s2 = new String();
        while((s = in.readLine()) != null)
            s2 += s + "\n";
        in.close();

        // 1b. Reading standard input:
```

```

BufferedReader stdin = new BufferedReader(
    new InputStreamReader(System.in));
System.out.print("Enter a line:");
System.out.println(stdin.readLine());

// 2. Input from memory
StringReader in2 = new StringReader(s2);
int c;
while((c = in2.read()) != -1)
    System.out.print((char)c);

// 3. Formatted memory input
try {
    DataInputStream in3 = new DataInputStream(
        new ByteArrayInputStream(s2.getBytes()));
    while(true)
        System.out.print((char)in3.readByte());
} catch(EOFException e) {
    System.err.println("End of stream");
}

// 4. File output
try {
    BufferedReader in4 = new BufferedReader(
        new StringReader(s2));
    PrintWriter out1 = new PrintWriter(
        new BufferedWriter(new FileWriter("IODemo.out")));
    int lineCount = 1;
    while((s = in4.readLine()) != null )
        out1.println(lineCount++ + ": " + s);
    out1.close();
} catch(EOFException e) {
    System.err.println("End of stream");
}

// 5. Storing & recovering data
try {
    DataOutputStream out2 = new DataOutputStream(
        new BufferedOutputStream(
            new FileOutputStream("Data.txt")));
    out2.writeDouble(3.14159);
    out2.writeUTF("That was pi");
    out2.writeDouble(1.41413);
    out2.writeUTF("Square root of 2");
}

```

```

        out2.close();
        DataInputStream in5 = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("Data.txt")));
        // Must use DataInputStream for data:
        System.out.println(in5.readDouble());
        // Only readUTF() will recover the
        // Java-UTF String properly:
        System.out.println(in5.readUTF());
        // Read the following double and String:
        System.out.println(in5.readDouble());
        System.out.println(in5.readUTF());
    } catch (EOFException e) {
        throw new RuntimeException(e);
    }

    // 6. Reading/writing random access files
    RandomAccessFile rf =
        new RandomAccessFile("rtest.dat", "rw");
    for(int i = 0; i < 10; i++)
        rf.writeDouble(i*1.414);
    rf.close();
    rf = new RandomAccessFile("rtest.dat", "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();
    rf = new RandomAccessFile("rtest.dat", "r");
    for(int i = 0; i < 10; i++)
        System.out.println("Value " + i + ": " +
            rf.readDouble());
    rf.close();
    monitor.expect("IOStreamDemo.out");
}
} ///:~

```

下面是程序中编号部分的说明。

输入流

第一到第四部分演示了输入流的创建和使用，第四部分还展示了输出流的简单用法。

1. 缓冲输入文件

如果想要打开一个文件用于字符输入，我们可以使用以 `String` 或 `File` 对象作为文件名的 `FileInputStream`。为了提高速度，我们希望对那个文件进行缓冲，那么我们将作为结果的引用传给一个 `BufferedReader` 构造器。由于 `BufferedReader` 也提供 `readline()` 方法，所以这是我们的最终对象和进行读取的接口。一旦抵达文件末尾 `readLine()` 将返回 `null`，所以可以用它来终止 `while` 循环。

字符串 `s2` 用来累积文件的全部内容(包括必须添加的换行符，因为 `readline()` 已将它们删掉)。它在程序的后面部分将会用到。最后，调用 `close()` 关闭文件。从技术上讲，当运行 `finalize()` 时就会调用 `close()`，那么程序结束时也应该发生这种情况(无论垃圾回收器是否存在)。然而，这种行为的实现并不是一致的，因此唯一安全的做法就是为文件明确调用 `close()`。

1b 部分展示了怎样包装 `System.in` 来读取来自控制台的输入。`System.in` 是一个 `InputStream`，而 `BufferedReader` 需要的是 `Reader` 参数，因此引入 `InputStream` 来执行转换。

2. 从内存输入

这部分获取已包含文件全部内容的字符串 `s2`，并用它创建一个 `StringReader`。然后调用 `read()` 每次读取一个字符，并把它发送到控制台。注意 `read()` 是以 `int` 的形式返回下一字节，因此必须强制转换为 `char` 才能正确打印。

3. 格式化的内存输入

如果要读取格式化数据，我们要用到 `DataInputStream`，它是一个面向字节的 I/O 类(不是面向字符的)。因此我们必须使用 `InputStream` 类而不是 `Reader` 类。当然，我们可以用 `InputStream` 以字节的形式读取任何数据(例如一个文件)，不过，在这里使用的是字符串。为了将字符串转换为成适用于 `ByteArrayInputStream` 的字节数组，`String` 包含了一个可以实现此项工作的 `getBytes()` 方法。至此，我们就持有了一个可传递给 `DataInputStream` 的 `InputStream`。

如果我们从 `DataInputStream` 用 `readByte()` 一次一个字节地来读取字符，那么由于任何字节的值都是合法的结果，因此返回值不能用来检测输入是否结束。相反，我们可以使用 `available()` 方法查看还有多少可供存取的字符。下面这个例子演示了怎样一次一个字节地读取文件：

```
//: c12:TestEOF.java
// Testing for end of file while reading a byte at a time.
import java.io.*;
```

```

public class TestEOF {
    // Throw exceptions to console:
    public static void main(String[] args)
        throws IOException {
        DataInputStream in = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("TestEOF.java")));
        while(in.available() != 0)
            System.out.print((char)in.readByte());
        }
    } ///: ~

```

注意：`available()` 的工作方式会随着所读取的媒介类型的不同而有所不同。字面意思就是“在没有阻塞的情况下所能读取的字节数。”对于文件，意味着整个文件，但是对于不同类型的流，可能就不是这样，因此要谨慎使用。

我们也可以通过捕获异常来检测输入的末端。但是，使用异常进行流控制，被认为是对异常特性的错误使用方式。

4. 文件输出

这个例子也展示了如何向文件写入数据。首先，创建一个与指定文件连接的 `FileWriter`。实际上，我们通常会用 `BufferedWriter` 将其包装起来用以缓冲输出（尝试移除此包装来感受到对性能的影响——缓冲往往能显著地增加 I/O 操作的性能）。然后为了格式化把它转换成 `PrintWriter`。按照这种方式创建的数据文件可作为普通文本文件读取。

当读入行时，行号就会增加。注意并未用到 `LineNumberInputStream`，因为这个类没有多大帮助，所以我们没必要用它。正如在此所见，记录我们自己的行号很微不足道。

一旦读完数据流，`readLine()` 会返回 `null`。我们可以看到要为 `out1` 显式 `close()`，如果我们不为所有的输出文件调用 `close()`，就会发现缓冲区内容不会刷新清空，那么它们也就不完整。

输出流

输出流可以按写入数据的方式划分为两类：一种写是为了让人们使用写入的数据，另一种是为了 `DataInputStream` 可以再次读取它们。`RandomAccessFile` 是独立的，尽管它的数据格式兼容于 `DataInputStream` 和 `DataOutputStream`。

5. 存储和恢复数据

`PrintStream` 可以对数据进行格式化，以便人们的阅读。但是为了使输出可供另一个流恢复的数据，我们必需用 `DataOutputStream` 写入数据，并用 `DataInputStream` 恢复数据。当然，这些流可以是任何形式，但在这里我们采用的是一个文件，并且对于读和写都进行了缓冲处理。注意 `DataOutputStream` 和 `DataInputStream` 是面向字节的，因此要使用 `InputStream` 和 `OutputStreams`。

如果我们使用 `DataOutputStream` 写入数据，Java 保证我们可以使用 `DataInputStream` 准确地读取数据——无论读和写数据的平台怎么不同。这一点具有不可思议的价值，因为我们都知人们曾经花费了大量时间去处理平台相关的数据问题。只要两个平台上都有 Java，这种问题就不会再发生³。

当我们使用 `DataOutputStream` 时，写字符串并且让 `DataInputStream` 能够恢复它的唯一可靠的做法就是使用 UTF-8 编码，在例子第五部分中是用 `writeUTF()` 和 `readUTF()` 来实现的。UTF-8 是 Unicode 的变体，后者把所有字符都存储成两个字节的形势。如果我们使用的只是 ASCII 或者几乎都是 ASCII 字符（只占 7 位），这么做就显得极其浪费空间和带宽，所以 UTF-8 将 ASCII 字符编码成单一字节的形势，而非 ASCII 字符则编码成两到三个字节的形势。另外，字符串的长度存储在前两个字节中。但是，`writeUTF()` 和 `readUTF()` 使用的是适合于 Java 的 UTF-8 变体（在 JDK 文档中有这些方法的详尽描述），因此如果我们用一个非 Java 程序读取用 `writeUTF()` 所写的字符串时，必须编写一些特殊代码才能正确读取字符串。

有了 `writeUTF()` 和 `readUTF()`，我们就可以用 `DataOutputStream` 把字符串和其他数据类型相混合，我们知道字符串完全可以作为 Unicode 来存储，并且可以很容易地使用 `DataInputStream` 来恢复它。

`writeDouble()` 将 `double` 类型的数字存储到流中，并用相应的 `readDouble()` 恢复它（对于其他的数据类型，也有类似方法用于读写）。但是为了保证所有的读方法都能够正常工作，我们必须知道流中数据项所在的确切位置，因为极有可能将保存的 `double` 数据作为一个简单的字节序列、`char` 或其他格式读入。因此，我们必须要么为文件中的数据采用固定的格式，要么将额外的信息保存到文件中，以便能够对其进行解析从而确定数据的存放位置。注意：对象序列化（本章稍后会介绍）可能是更容易的存储和读取复杂数据结构的方式。

6. 读写随机访问文件

正如先前所指，`RandomAccessFile` 除了实现 `DataInput` 和 `DataOutput` 接口之外，几乎完全独立于 I/O 继承层次结构的其他部分。所以不能将其与 `InputStream` 及 `OutputStream` 子类的任何部分组合起来。尽管把一个 `ByteArrayInputStream` 当作一个随机访问元素对待也具有实际意义，但是我们只能用 `RandomAccessFile` 打开文件。我们必须假定 `RandomAccessFile` 已经被正确缓冲，因为我们不能为它添加这样的功能。

³ XML 是跨越不同的计算机平台进行数据迁移的另一种解决方法，并且所有平台都不需要 Java。JDK 1.4 中 `javax.Xml.*` 类库包提供 XML 工具。这些在网站 www.MindView.net 的 *Thinking in Enterprise Java* 有所论及。

可以自行选择的是第二个构造器参数：我们可指定以“只读”（r）方式或“读写”（rw）方式打开一个 `RandomAccessFile` 文件。

使用 `RandomAccessFile`，类似于组合使用了 `DataInputStream` 和 `DataOutputStream`（因为它实现了同等的接口）。另外，我们会看到利用 `seek()`，可以在文件中到处移动，并修改文件中的某个值。

随着 JDK 1.4 中新的 I/O 的出现，你可能会考虑使用内存映射文件来代替 `RandomAccessFile`。

管道流

`PipedInputStream`, `PipedOutputStream`, `PipedReader` 及 `PipedWriter` 在本章只是简单地提到。但这并不表明它们没有什么用处，它们的价值只有在我们开始理解多线程之后才会显现，因为管道流用于线程之间的通信。在第 13 章会用一个示例进行讲述。

文件读写的实用工具

通常，程序的任务就是读取文件到内存，修改，然后再写出。Java I/O 类库的问题之一就是：它需要我们编写相当多的代码去执行这些常用操作——没有任何帮助类功能可以为我们做这一切。更糟糕的是，修饰器会使得记住如何打开文件变得相当困难。因此，在我们的类库中添加帮助类就显得相当有意义，这样就可以很容易地为我们完成这些基本任务。下面这个程序包含的静态方法可以像简单字符串那样读写文本文件。另外，我们可以创建一个 `TextFile` 类，它用一个 `ArrayList()` 来持有文件的若干行（如此，当我们操纵文件内容时，就可以使用 `ArrayList` 的所有功能）。

```
//: com:bruceeckel:util:TextFile.java
// Static functions for reading and writing text files as
// a single string, and treating a file as an ArrayList.
// {Clean: test.txt test2.txt}
package com.bruceeckel.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList {
    // Tools to read and write files as single strings:
    public static String
    read(String fileName) throws IOException {
        StringBuffer sb = new StringBuffer();
        BufferedReader in =
            new BufferedReader(new FileReader(fileName));
        String s;
        while((s = in.readLine()) != null) {
```

```

        sb.append(s);
        sb.append("\n");
    }
    in.close();
    return sb.toString();
}

public static void
write(String fileName, String text) throws IOException {
    PrintWriter out = new PrintWriter(
        new BufferedWriter(new FileWriter(fileName)));
    out.print(text);
    out.close();
}

public TextFile(String fileName) throws IOException {
    super(Arrays.asList(read(fileName).split("\n")));
}

public void write(String fileName) throws IOException {
    PrintWriter out = new PrintWriter(
        new BufferedWriter(new FileWriter(fileName)));
    for(int i = 0; i < size(); i++)
        out.println(get(i));
    out.close();
}

// Simple test:
public static void main(String[] args) throws Exception {
    String file = read("TextFile.java");
    write("test.txt", file);
    TextFile text = new TextFile("test.txt");
    text.write("test2.txt");
}

} ///:~

```

所有的方法都直接将 `IOException` 传递给调用者。`Read()` 将每行添加到 `StringBuffer` (为了更有效) 中, 并且为每行加上换行符, 因为在读的过程中换行符会被去除掉。接着返回一个包含整个文件的字符串。`Write()` 打开文本并将其写入文件。在这两个方法完成时, 都要记着 `close()` 文件。

这个构造器利用 `read()` 方法将文件转换成字符串, 接着使用 `String.split()` 以换行符为界把结果划分成行 (若要频繁使用这个类, 我们可以重写此构造器以提高性能)。遗憾的是没有相应的“`join`”方法, 所以那个非静态的 `write()` 方法必须一行一行地输出这些行。

在 `main()` 方法中, 通过执行一个基本测试来确保这些方法运转, 尽管这个程序的代码很少, 但使用它还是会节约大量的时间并且变得很轻松, 在本章后面一些例子中就可以感受到这一点。

标准 I/O

“标准 I/O”这个术语参考的是 Unix 中“程序所使用的单一信息流”这个概念（在 windows 和其他许多操作系统中，也有相似形式的实现）。程序的所有输入都可以来自于“标准输入”，它的所有输出也都可以发送到“标准输出”，以及所有的错误信息都可以发送到“标准错误”。标准 I/O 的意义在于：我们可以很容易地把程序串联起来，一个程序的标准输出可以成为另一程序的标准输入。这真是一个强大的工具。

从标准输入读取

按照标准 I/O 模型，Java 提供了 `System.in`，`System.out` 和 `System.err`。在整本书里，我们已经看到了怎样用 `System.out` 将数据写出到标准输出，其中 `System.out` 已经事先被包装成了 `PrintStream` 对象。`System.err` 同样也是 `PrintStream`，但 `System.in` 却是一个没有被包装的未经加工的 `InputStream`。这意味尽管我们可以立即使用 `System.out` 和 `System.err`，但是在读取 `System.in` 之前必须对其进行包装。

通常我们会用 `readLine()` 一次一行地读取输入，因此我们会将 `System.in` 包装成 `BufferedReader` 来使用。为此，我们必须用 `InputStreamReader` 把 `System.in` 转换成 `Reader`。下面这个例子将直接重复你所输入的每一行。

```
//: c12:Echo.java
// How to read from standard input.
// {RunByHand}
import java.io.*;

public class Echo {
    public static void main(String[] args)
        throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = in.readLine()) != null && s.length() != 0)
            System.out.println(s);
        // An empty line or Ctrl-Z terminates the program
    }
} ///: ~
```

使用异常规范是因为 `readLine()` 会抛出 `IOException`。注意，`System.in` 和大多数流一样，通常应该对它进行缓冲。

将 `System.out` 转换成 `PrintWriter`

`System.out` 是一个 `PrintStream`，而 `PrintStream` 是一个 `OutputStream`。
`PrintWriter` 有一个可以接受 `OutputStream` 作为参数的构造器。因此，只要我们需要，就可以使用那个构造器把 `System.out` 转换成 `PrintWriter`。

```
//: c12:ChangeSystemOut.java
// Turn System.out into a PrintWriter.
import com.bruceeckel.simpletest.*;
import java.io.*;

public class ChangeSystemOut {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        PrintWriter out = new PrintWriter(System.out, true);
        out.println("Hello, world");
        monitor.expect(new String[] {
            "Hello, world"
        });
    }
} ///:~
```

标准 I/O 重定向

Java 的 `System` 类提供一些简单的静态方法调用，允许我们对标准输入、输出和错误 I/O 流进行重定向：

`setIn(InputStream)`
`setOut(PrintStream)`
`setErr(PrintStream)`

如果我们突然开始在显示器上创建大量输出，而这些输出滚动的如此之快以至于无法阅读时，重定向输出就显得极为有用⁴。对于“我们想重复测试特定用户的输入序列”的命令行程序来说，重定向输入就很有价值。下例简单演示了这些方法的使用：

```
//: c12:Redirecting.java
// Demonstrates standard I/O redirection.
// {Clean: test.out}
import java.io.*;

public class Redirecting {
    // Throw exceptions to console:
```

⁴ 第 13 章展示了一种更方便的解决方案：一个 GUI 程序，具有带滚动的文本区域。

```

public static void main(String[] args)
throws IOException {
    PrintStream console = System.out;
    BufferedInputStream in = new BufferedInputStream(
        new FileInputStream("Redirecting.java"));
    PrintStream out = new PrintStream(
        new BufferedOutputStream(
            new FileOutputStream("test.out")));
    System.setIn(in);
    System.setOut(out);
    System.setErr(out);
    BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)
        System.out.println(s);
    out.close(); // Remember this!
    System.setOut(console);
}
} ///:~

```

这个程序将标准输入附加在文件上，并将标准输出和标准错误重定向到另一个文件。

I/O 重定向操纵的是字节流，而不是字符流，因此我们使用的是 `InputStream` 和 `OutputStream` 而不是 `Reader` 和 `Writer`。

新 I/O

JDK1.4 的 `java.nio.*` 包中引入了 Java 新的 I/O 类库，其目的在于提高速度。实际上，旧的 I/O 包已经使用 `nio` 重新实现过，以便充分利用这种速度提高，因此，即使我们不显式地用 `nio` 编写代码，也能从中受益。速度的提高在文件 I/O 和网络 I/O 中都有可能发生，我们在这里只研究前者⁵，对于后者，将会在 *Thinking in Enterprise Java* 中涉及到。

速度的提高来自于所使用的结构更接近于操作系统执行 I/O 的方式：通道和缓冲器。我们可以把它想象成一个煤矿；通道是一个包含煤层（数据）的矿藏，而缓冲器则我们派送到矿藏的卡车。卡车载满煤炭而归，我们再从卡车上获得煤炭。也就是说，我们并没有直接和通道交互；我们只是和缓冲器交互，并把缓冲器派送到通道。通道要么从缓冲器获得数据，要么向缓冲器发送数据。

唯一直接与通道交互的缓冲器是 `ByteBuffer`——也就是说，可以存储未加工字节的缓冲器。当我们查询 JDK 文档中的 `java.nio.ByteBuffer` 时，会发现它是相当基础的类：通过告知分配多少存储空间来创建一个 `ByteBuffer` 对象，并且还有一个方法选择的集用于以未加工

⁵ 此部分由 Chintan Thakker 撰稿编写。

的字节形式或原始的数据类型输出和读取数据。但是，没办法输出或读取对象，即使是字符串对象也不行。这种处理虽然是低水平但却正好，因为这是大多数操作系统中更有效的映射方式。

旧 I/O 类库中有三个类被改进了，用以产生 `FileChannel`，它们是：`FileInputStream`，`FileOutputStream` 以及用于既读又写的 `RandomAccessFile`。注意这些是字节操纵流，与低层的 `nio` 特性一致。`Reader` 和 `Writer` 的字符模式类不能用于产生通道，但是 `java.nio.channels.Channels` 类能提供实用方法在通道中产生 `Reader` 和 `Writer`。

下面的简单实例演示了上面三种类型的流，用以产生可写的、可读可写的及可读的通道。

```
//: c12:GetChannel.java
// Getting channels from streams
// {Clean: data.txt}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class GetChannel {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        // Write a file:
        FileChannel fc =
            new FileOutputStream("data.txt").getChannel();
        fc.write(ByteBuffer.wrap("Some text ".getBytes()));
        fc.close();
        // Add to the end of the file:
        fc =
            new RandomAccessFile("data.txt", "rw").getChannel();
        fc.position(fc.size()); // Move to the end
        fc.write(ByteBuffer.wrap("Some more".getBytes()));
        fc.close();
        // Read the file:
        fc = new FileInputStream("data.txt").getChannel();
        ByteBuffer buff = ByteBuffer.allocate(BSIZE);
        fc.read(buff);
        buff.flip();
        while(buff.hasRemaining())
            System.out.print((char)buff.get());
    }
} ///: ~
```

对于这里所展示的任何“流”类，`getChannel()` 将会产生一个 `FileChannel`。通道是一种相当基础的东西：我们可以向它传送用于读写的 `ByteBuffer`，并且可以锁定文件的某些区域用于独占式访问（稍后讲述）。

将字节存放于 **ByteBuffer** 的方法之一是：使用一种“put”方法直接对它们进行填充，填入一或多个字节，或基本数据类型的值。不过，正如你所见，我们也可以使用 **warp()** 方法将已存在的字节数组“包装”到 **ByteBuffer** 中。一旦如此，就不再复制底层的数组，而是把它作为所产生的 **ByteBuffer** 的存储器，我们称之为数组支持的 **ByteBuffer**。

data.txt 文件被 **RandomAccessFile** 再次打开。注意我们可以在文件内随处移动 **FileChannel**；在这里，我们把它移到最后以便附加其他的写操作。

对于只读访问，我们必须显式地使用静态的 **allocate()** 方法来分配 **ByteBuffer**。**nio** 的目标就是要快速移动大量数据，因此 **ByteBuffer** 的大小就显得尤为重要——实际上，这里使用的 **1K** 可能比我们通常应该使用的要小一点（必须通过实际运行我们的应用来找到最佳尺寸）。

也可以使用 **allocateDirect()** 而不是 **allocate()** 来获取更快的速度，用于产生一个在更高层次上耦合操作系统的“直接”缓冲器。但是，这种分配的开支会更大，并且具体实现也随操作系统的不同而不同，因此必须再次实际运行我们的应用来查看直接缓冲是否可以使我们获得速度上的优势。

一旦我们调用了 **read()**，就会告知 **FileChannel** 向 **ByteBuffer** 存储字节，我们必须调用缓冲器上的 **flip()**，让它做好让别人读取字节的准备（是的，这似乎有一点拙劣，但是请记住它是基于很低水平的，适用于获取最大速度）。如果我们打算使用缓冲器执行进一步的 **read()** 操作，我们也必须得调用 **clear()** 来为每个 **read()** 做好准备。这在下面这个简单文件复制的程序中可以看到：

```
//: c12:ChannelCopy.java
// Copying a file using channels and buffers
// {Args: ChannelCopy.java test.txt}
// {Clean: test.txt}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ChannelCopy {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        if(args.length != 2) {
            System.out.println("arguments: sourcefile destfile");
            System.exit(1);
        }
        FileChannel
            in = new FileInputStream(args[0]).getChannel(),
            out = new FileOutputStream(args[1]).getChannel();
        ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
        while(in.read(buffer) != -1) {
            buffer.flip(); // Prepare for writing
```

```

        out.write(buffer);
        buffer.clear(); // Prepare for reading
    }
}
} ///: ~

```

我们看到一个 `FileChannel` 被用于读而打开，另一个被用于写而打开。`ByteBuffer` 被分配了空间，当 `FileChannel.read()` 返回 `-1` 时（一个分界符，毋庸置疑，它源于 `Unix` 和 `C`），表示我们已经到达了输入的末尾。每次 `read()` 操作之后，就会将数据输入到缓冲器中，`flip()` 则是准备缓冲器以便它的信息可以由 `write()` 提取。`Write()` 操作之后，信息仍在缓冲器中，接着 `clear()` 操作则对所有的内部指针重新安排，以便缓冲器在另一个 `read()` 操作期间，能够做好接受数据的准备。

然而，上面那个程序并不是处理此类操作的理想方式。特殊方法 `transferTo()` 和 `transferFrom()` 则允许我们将一个通道和另一个通道直接相连：

```

///: c12:TransferTo.java
// Using transferTo() between channels
// {Args: TransferTo.java TransferTo.txt}
// {Clean: TransferTo.txt}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class TransferTo {
    public static void main(String[] args) throws Exception {
        if(args.length != 2) {
            System.out.println("arguments: sourcefile destfile");
            System.exit(1);
        }
        FileChannel
            in = new FileInputStream(args[0]).getChannel(),
            out = new FileOutputStream(args[1]).getChannel();
        in.transferTo(0, in.size(), out);
        // Or:
        // out.transferFrom(in, 0, in.size());
    }
} ///: ~

```

我们并不是经常做这类事情，但是了解这一点还是有好处的。

转换数据

让我们回过头看 `GetChannel.java` 这个程序，我们会发现为了输出文件中的信息，我们必须每次只读取一个字节的数据，然后将其强制转换成 `char` 类型。这种方法似乎有点原始——如果我们查看一下 `java.nio.CharBuffer` 这个类，将会发现它有一个 `toString()` 方法是这样定义的：“返回一个包含缓冲器中所有字符的字符串”。既然 `ByteBuffer` 可以看作是具有 `asCharBuffer()` 方法的 `CharBuffer`，那么为什么不用它呢？因为正如下面的 `exception()` 语句中第一行所见，这种方法并不能解决问题：

```
//: c12:BufferToText.java
// Converting text to and from ByteBuffers
// {Clean: data2.txt}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import com.bruceeckel.simpletest.*;

public class BufferToText {
    private static Test monitor = new Test();
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        FileChannel fc =
            new FileOutputStream("data2.txt").getChannel();
        fc.write(ByteBuffer.wrap("Some text".getBytes()));
        fc.close();
        fc = new FileInputStream("data2.txt").getChannel();
        ByteBuffer buff = ByteBuffer.allocate(BSIZE);
        fc.read(buff);
        buff.flip();
        // Doesn't work:
        System.out.println(buff.asCharBuffer());
        // Decode using this system's default Charset:
        buff.rewind();
        String encoding = System.getProperty("file.encoding");
        System.out.println("Decoded using " + encoding + ": "
            + Charset.forName(encoding).decode(buff));
        // Or, we could encode with something that will print:
        fc = new FileOutputStream("data2.txt").getChannel();
        fc.write(ByteBuffer.wrap(
            "Some text".getBytes("UTF-16BE")));
        fc.close();
        // Now try reading again:
        fc = new FileInputStream("data2.txt").getChannel();
```

```

        buff.clear();
        fc.read(buff);
        buff.flip();
        System.out.println(buff.asCharBuffer());
        // Use a CharBuffer to write through:
        fc = new FileOutputStream("data2.txt").getChannel();
        buff = ByteBuffer.allocate(24); // More than needed
        buff.asCharBuffer().put("Some text");
        fc.write(buff);
        fc.close();
        // Read and display:
        fc = new FileInputStream("data2.txt").getChannel();
        buff.clear();
        fc.read(buff);
        buff.flip();
        System.out.println(buff.asCharBuffer());
        monitor.expect(new String[] {
            "????",
            "%% Decoded using [A-Za-z0-9_\\-]+: Some text",
            "Some text",
            "Some text\0\0\0"
        });
    }
} ///: ~

```

缓冲器容纳的是普通的字节，为了把它们转换成字符，我们要么在输入它们的时候对其进行编码（这样，它们输出时才具有意义），要么在将其从缓冲器输出时对它们进行解码。可以使用 `java.nio.charset.Charset` 类实现这些功能，该类提供了把数据编码成多种不同类型的字符集的工具：

```

///: c12:AvailableCharsets.java
// Displays Charsets and aliases
import java.nio.charset.*;
import java.util.*;
import com.bruceeckel.simpletest.*;

public class AvailableCharsets {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Map charSets = Charset.availableCharsets();
        Iterator it = charSets.keySet().iterator();
        while(it.hasNext()) {
            String csName = (String)it.next();
            System.out.print(csName);
            Iterator aliases = ((Charset)charSets.get(csName))

```

```

        .aliases().iterator();
    if(aliases.hasNext())
        System.out.print(": ");
    while(aliases.hasNext()) {
        System.out.print(aliases.next());
        if(aliases.hasNext())
            System.out.print(", ");
    }
    System.out.println();
}

monitor.expect(new String[] {
    "Big5: csBig5",
    "Big5-HKSCS: big5-hkscs, Big5_HKSCS, big5hkscs",
    "EUC-CN",
    "EUC-JP: eucjis, x-eucjp, csEUCPkdFmtjapanese, " +
    "eucjp, Extended_UNIX_Code_Packed_Format_for" +
    "_Japanese, x-euc-jp, euc_jp",
    "euc-jp-linux: euc_jp_linux",
    "EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, " +
    "ksc5601-1987, euc_kr, ks_c_5601-1987, " +
    "euckr, csEUCKR",
    "EUC-TW: cns11643, euc_tw, euctw",
    "GB18030: gb18030-2000",
    "GBK: GBK",
    "ISCI191: iscii, ST_SEV_358-88, iso-ir-153, " +
    "csISO153GOST1976874",
    "ISO-2022-CN-CNS: ISO2022CN_CNS",
    "ISO-2022-CN-GB: ISO2022CN_GB",
    "ISO-2022-KR: ISO2022KR, csISO2022KR",
    "ISO-8859-1: iso-ir-100, 8859_1, ISO_8859-1, " +
    "ISO8859_1, 819, csISOLatin1, IBM-819, " +
    "ISO_8859-1:1987, latin1, cp819, ISO8859-1, " +
    "IBM819, ISO_8859_1, I1",
    "ISO-8859-13",
    "ISO-8859-15: 8859_15, csISOLatin9, IBM923, cp923," +
    " 923, L9, IBM-923, ISO8859-15, LATIN9, " +
    "ISO_8859-15, LATIN0, csISOLatin0, " +
    "ISO8859_15_FDIS, ISO-8859-15",
    "ISO-8859-2", "ISO-8859-3", "ISO-8859-4",
    "ISO-8859-5", "ISO-8859-6", "ISO-8859-7",
    "ISO-8859-8", "ISO-8859-9",
    "JIS0201: X0201, JIS_X0201, csHalfWidthKatakana",
    "JIS0208: JIS_C6626-1983, csISO87JISX0208, x0208, " +
    "JIS_X0208-1983, iso-ir-87",

```

```

        "JIS0212: jis_x0212-1990, x0212, iso-ir-159, " +
        "csISO159JISC02121990",
        "Johab: ms1361, ksc5601_1992, ksc5601-1992",
        "KOI8-R",
        "Shift_JIS: shift-jis, x-sjis, ms_kanji, " +
        "shift_jis, csShiftJIS, sjis, pck",
        "TIS-620",
        "US-ASCII: IBM367, ISO646-US, ANSI_X3.4-1986, " +
        "cp367, ASCII, iso_646.irv:1983, 646, us, iso-ir-6," +
        " csASCII, ANSI_X3.4-1968, ISO_646.irv:1991",
        "UTF-16: UTF_16",
        "UTF-16BE: X-UTF-16BE, UTF_16BE, ISO-10646-UCS-2",
        "UTF-16LE: UTF_16LE, X-UTF-16LE",
        "UTF-8: UTF8", "windows-1250", "windows-1251",
        "windows-1252: cp1252",
        "windows-1253", "windows-1254", "windows-1255",
        "windows-1256", "windows-1257", "windows-1258",
        "windows-936: ms936, ms_936",
        "windows-949: ms_949, ms949", "windows-950: ms950",
    });
}
} ///:~

```

让我们返回到 `BufferToText.java`，如果我们想 `rewind()` 缓冲器（为了返回到数据开始部分），接着使用平台的缺省字符集对数据进行 `decode()`，那么作为结果的 `CharBuffer` 可以很好地输出打印到控制台。可以使用 `System.getProperty("file.encoding")` 发现缺省字符集，它会产生代表字符集名称的字符串。将该字符串传送给 `Charset.forName()` 用以产生 `Charset` 对象，可以用它对字符串进行解码。

另一选择是在读文件时，使用能够产生可打印的输出的字符集进行 `encode()`，正如我们在 `BufferToText.java` 中第三部分看到的那样。这里，`UTF-16BE` 可以把文本写到文件中，当读取时，我们只需要把它转换成 `CharBuffer`，就会产生所期望的文本。

最后，让我们来看看若是通过 `CharBuffer` 向 `ByteBuffer` 写入，会发生什么情况（后面将会深入了解）。注意我们为一个 `ByteBuffer` 分配了 24 个字节。既然一个字符需要 2 个字节，那么一个 `ByteBuffer` 足可以容纳 12 个字符，但是“Some text”只有 9 个字符。剩余的内容为零的字节仍出现在由它的 `toString()` 所产生的 `CharBuffer` 的表示中，我们可以在输出中看到。

获取原始类型

尽管 `ByteBuffer` 只能保存字节类型的数据，但是它具有可以从它所容纳的字节中产生出各种不同原始类型值的方法。下面这个例子展示了怎样使用这些方法来插入和抽取各种数值：

```

//: c12:GetData.java
// Getting different representations from a ByteBuffer
import java.nio.*;
import com.bruceeckel.simpletest.*;

public class GetData {
    private static Test monitor = new Test();
    private static final int BSIZE = 1024;
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        // Allocation automatically zeroes the ByteBuffer:
        int i = 0;
        while(i++ < bb.limit())
            if(bb.get() != 0)
                System.out.println("nonzero");
        System.out.println("i = " + i);
        bb.rewind();
        // Store and read a char array:
        bb.asCharBuffer().put("Howdy!");
        char c;
        while((c = bb.getChar()) != 0)
            System.out.print(c + " ");
        System.out.println();
        bb.rewind();
        // Store and read a short:
        bb.asShortBuffer().put((short)471142);
        System.out.println(bb.getShort());
        bb.rewind();
        // Store and read an int:
        bb.asIntBuffer().put(99471142);
        System.out.println(bb.getInt());
        bb.rewind();
        // Store and read a long:
        bb.asLongBuffer().put(99471142);
        System.out.println(bb.getLong());
        bb.rewind();
        // Store and read a float:
        bb.asFloatBuffer().put(99471142);
        System.out.println(bb.getFloat());
        bb.rewind();
        // Store and read a double:
        bb.asDoubleBuffer().put(99471142);
        System.out.println(bb.getDouble());
        bb.rewind();
    }
}

```

```

        monitor.expect(new String[] {
            "i = 1025",
            "H o w d y ! ",
            "12390", // Truncation changes the value
            "99471142",
            "99471142",
            "9.9471144E7",
            "9.9471142E7"
        });
    }
} ///:~

```

在分配一个 `ByteBuffer` 之后，可以通过检测它的值来查看缓冲器配置是否将其内容自动置零——它确实是这样做了。这里一共检测了 1024 个值（由缓冲器的 `limit()` 决定），并且所有的值都是零。

向 `ByteBuffer` 插入基本类型数据的最简单的方法是：利用 `asCharBuffer()`、`asShortBuffer()` 等获得该缓冲器上的视图，然后使用视图的 `put()` 方法。我们会发现此方法适用于所有基本数据类型。仅有一个小小的例外，即使用 `ShortBuffer` 的 `put()` 方法时，需要进行类型转换（注意类型转化会截取或改变结果）。而其他所有的视图缓冲器在使用 `put()` 方法时，不需要进行类型转换。

视图缓冲器

“视图缓冲器（view buffer）”可以让我们通过某个特定的基本数据类型的视窗查看其底层的 `ByteBuffer`。`ByteBuffer` 依然是实际存储数据的地方，“支持”着前面的视图，因此，对视图的任何修改都会映射成为对 `ByteBuffer` 中数据的修改。正如我们在上一示例看到的那样，这使我们可以很方便地向 `ByteBuffer` 插入数据。视图还允许我们从 `ByteBuffer` 一次一个地（与 `ByteBuffer` 所支持的相同）或者成批地（放入数组中）读取基本数据。在下面这个例子中，通过 `IntBuffer` 操纵 `ByteBuffer` 中的整数：

```

///: c12: IntBufferDemo.java
// Manipulating ints in a ByteBuffer with an IntBuffer
import java.nio.*;
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;

public class IntBufferDemo {
    private static Test monitor = new Test();
    private static final int BSIZE = 1024;
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        IntBuffer ib = bb.asIntBuffer();
        // Store an array of int:

```

```

ib.put(new int[] { 11, 42, 47, 99, 143, 811, 1016 });
// Absolute location read and write:
System.out.println(ib.get(3));
ib.put(3, 1811);
ib.rewind();
while(ib.hasRemaining()) {
    int i = ib.get();
    if(i == 0) break; // Else we'll get the entire buffer
    System.out.println(i);
}
monitor.expect(new String[] {
    "99",
    "11",
    "42",
    "47",
    "1811",
    "143",
    "811",
    "1016"
});
}
} ///: ~

```

先用重载过的 `put()` 方法存储一个整数数组。接着 `get()` 和 `put()` 的方法调用直接访问底层的 `ByteBuffer` 中的某个整数位置。注意，这些对绝对位置的访问通过直接与 `ByteBuffer` 对话的方式也同样可以做用于基本类型。

一旦底层的 `ByteBuffer` 通过视图缓冲器被填满整数或其他基本类型时，就可以直接被写到通道中了。正像从通道中读取那样容易，然后使用视图缓冲器可以把任何数据都转化成某一特定的基本类型。在下面的例子中，通过在同一个 `ByteBuffer` 上建立不同的视图缓冲器，将同一字节序列翻译成了 `short`, `int`, `float`, `long`, 和 `double` 类型的数据。

```

//: c12:ViewBuffers.java
import java.nio.*;
import com.bruceeckel.simpletest.*;

public class ViewBuffers {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.wrap(
            new byte[]{ 0, 0, 0, 0, 0, 0, 0, 0, 'a' });
        bb.rewind();
        System.out.println("Byte Buffer");
        while(bb.hasRemaining())
            System.out.println(bb.position() + " -> " + bb.get());
    }
}

```

```

CharBuffer cb =
    ((ByteBuffer)bb.rewind()).asCharBuffer();
System.out.println("Char Buffer");
while(cb.hasRemaining())
    System.out.println(cb.position() + " -> " + cb.get());
FloatBuffer fb =
    ((ByteBuffer)bb.rewind()).asFloatBuffer();
System.out.println("Float Buffer");
while(fb.hasRemaining())
    System.out.println(fb.position() + " -> " + fb.get());
IntBuffer ib =
    ((ByteBuffer)bb.rewind()).asIntBuffer();
System.out.println("Int Buffer");
while(ib.hasRemaining())
    System.out.println(ib.position() + " -> " + ib.get());
LongBuffer lb =
    ((ByteBuffer)bb.rewind()).asLongBuffer();
System.out.println("Long Buffer");
while(lb.hasRemaining())
    System.out.println(lb.position() + " -> " + lb.get());
ShortBuffer sb =
    ((ByteBuffer)bb.rewind()).asShortBuffer();
System.out.println("Short Buffer");
while(sb.hasRemaining())
    System.out.println(sb.position() + " -> " + sb.get());
DoubleBuffer db =
    ((ByteBuffer)bb.rewind()).asDoubleBuffer();
System.out.println("Double Buffer");
while(db.hasRemaining())
    System.out.println(db.position() + " -> " + db.get());
monitor.expect(new String[] {
    "Byte Buffer",
    "0 -> 0",
    "1 -> 0",
    "2 -> 0",
    "3 -> 0",
    "4 -> 0",
    "5 -> 0",
    "6 -> 0",
    "7 -> 97",
    "Char Buffer",
    "0 -> \0",
    "1 -> \0",
    "2 -> \0",

```



```

        "3 -> a",
        "Float Buffer",
        "0 -> 0.0",
        "1 -> 1.36E-43",
        "Int Buffer",
        "0 -> 0",
        "1 -> 97",
        "Long Buffer",
        "0 -> 97",
        "Short Buffer",
        "0 -> 0",
        "1 -> 0",
        "2 -> 0",
        "3 -> 97",
        "Double Buffer",
        "0 -> 4.8E-322"
    });
}
} ///: ~

```

ByteBuffer 是一个被“包装”过的 8 字节数组，然后通过各种不同的基本类型的视图缓冲器把它显示了出来。我们可以在下图中看到，当从不同类型的缓冲器读取时，数据显示的方式也不同。

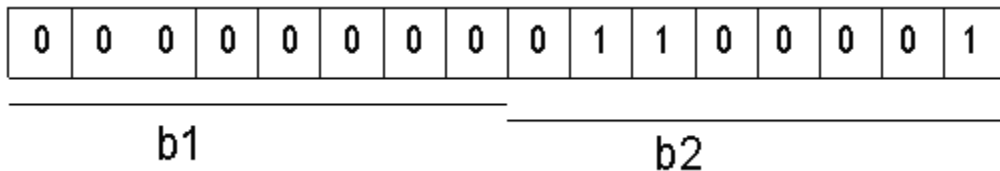
0	0	0	0	0	0	0	97	bytes
						a		chars
0		0		0		97		shorts
0				97				ints
0.0				1.36E-43				floats
97								longs
4.8E-322								doubles

这与上面程序的输出相对应。

Endians

不同的机器可能会使用不同的字节排序方法来存储数据。“Big endian（高位优先）”将最重要的字节存放在地址最低的存储器单元。而“little endian（低位优先）”则是将最重要的字节放在地址最高的存储器单元。当存储量大于一个字节时，像 int, float 等，我们就要考虑字节的顺序问题了。ByteBuffer 是以 Big endian 的形式存储数据的，并且数据在网上传送时也常常使用 Big endian 形式。我们可以使用带有参数 `ByteOrder.BIG_ENDIAN` 或 `ByteOrder.LITTLE_ENDIAN` 的 `order()` 方法改变 ByteBuffer 的 endian-ness。

考虑包含下面两个字节的 ByteBuffer：



如果我们以 short (`ByteBuffer.asShortBuffer()`) 形式读取数据，得到的数字是 97 (00000000 01100001)；但是如果将 ByteBuffer 更改成 little endian，仍以 short 形式读取数据，得到的数字却是 24832 (01100001 00000000)。

这个例子展示了怎样通过 endian 设置来改变字符中的字节次序：

```
//: c12:Endians.java
// Endian differences and data storage.
import java.nio.*;
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;

public class Endians {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
        bb.asCharBuffer().put("abcdef");
        System.out.println(Arrays2.toString(bb.array()));
        bb.rewind();
        bb.order(ByteOrder.BIG_ENDIAN);
        bb.asCharBuffer().put("abcdef");
        System.out.println(Arrays2.toString(bb.array()));
        bb.rewind();
        bb.order(ByteOrder.LITTLE_ENDIAN);
        bb.asCharBuffer().put("abcdef");
```

```

        System.out.println(Arrays2.toString(bb.array()));
        monitor.expect(new String[]{
            "[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]",
            "[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]",
            "[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]"
        });
    }
} ///: ~

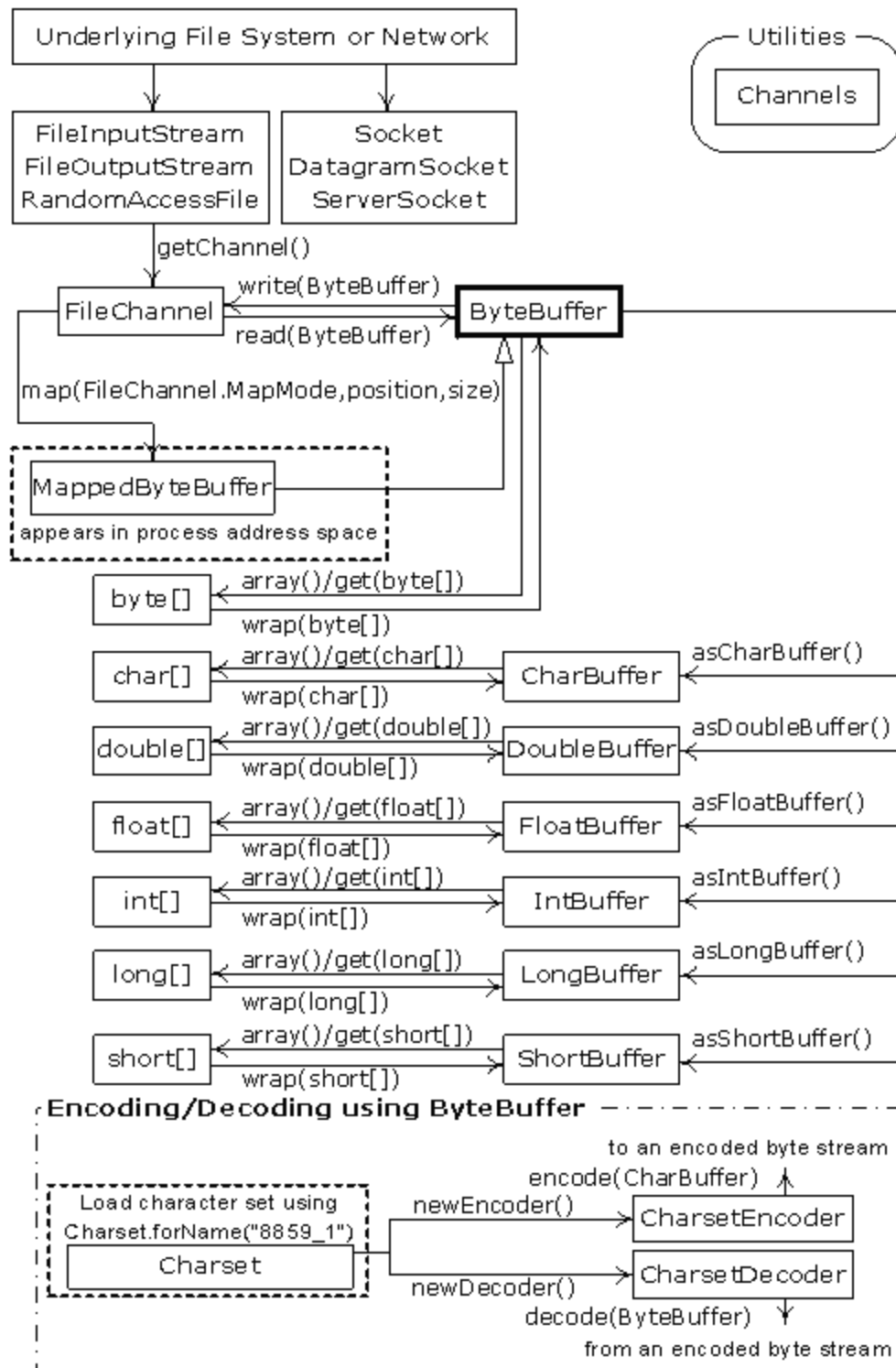
```

ByteBuffer 有足够的空间存储作为外部缓冲器的 `charArray` 中的所有字节，因此可以调用 `array()` 方法显示视图底层的字节。`array()` 方法是“可选的”，并且我们只能对由数组支持的缓冲器调用此方法；否则，将会抛出 `UnsupportedOperationException`。

通过 `CharBuffer` 视图可以将 `charArray` 插入到 `ByteBuffer` 中。在底层的字节被显示时，我们会发现缺省次序和随后的 `big endian` 次序相同；然而 `little endian` 次序则与之相反，交换了这些字节次序。

用缓冲器操纵数据

这个图解阐明了 `nio` 类之间的关系，便于我们理解怎么移动和转换数据。例如，如果我们想把一个字节数组写到文件中去，那么我们就应该使用 `ByteBuffer.wrap()` 方法把字节数组包装起来，然后用 `getChannel()` 方法在 `FileOutputStream` 上打开一个通道，接着将来自于 `ByteBuffer` 的数据写到 `FileChannel` 中。



注意：ByteBuffer 是将数据移进移出通道的唯一方式，并且我们只能创建一个独立的基本类型缓冲器，或者使用“as”方法从 ByteBuffer 中获得。也就是说，我们不能把基本类型的缓冲器转换成 ByteBuffer。然而，由于我们可以经由视图缓冲器将基本类型数据移进移出 ByteBuffer，所以这也就不是什么真正意义上的限制了。

缓冲器的细节

缓冲器由数据和可以高效地访问及操纵这些数据的四个索引组成，这四个索引是：**mark**，**position**，**limit** 和 **capacity**。下面是用于设置和复位索引以及查询数据的方法。

capacity()	返回缓冲区容量
clear()	清空缓冲区,将 <i>position</i> 设置为 0, <i>limit</i> 设置为容量。我们可以调用此方法覆写缓冲区。
flip()	将 <i>limit</i> 设置为 <i>position</i> , <i>position</i> 设置为 0。此方法用于准备从缓冲区读取已经写入的数据。
limit()	返回 <i>limit</i> 值
limit(int lim)	设置 <i>limit</i> 值。
mark()	将 <i>mark</i> 设置为 <i>position</i> 。
position()	返回 <i>position</i> 值。
position(int pos)	设置 <i>position</i> 值。
remaining()	返回 (<i>limit</i> - <i>position</i>)。
hasRemaining()	若有介于 <i>position</i> 和 <i>limit</i> 之间的元素，则返回 trueReturns true 。

在缓冲器中插入和提取数据的方法会更新这些索引，用于反映所发生的变化。

下面的示例用到一个很简单的算法（交换相邻字符）对 **CharBuffer** 中的字符进行编码和译码。

```
//: c12:UsingBuffers.java
import java.nio.*;
import com.bruceeckel.simpletest.*;

public class UsingBuffers {
    private static Test monitor = new Test();
    private static void symmetricScramble(CharBuffer buffer){
        while(buffer.hasRemaining()) {
            buffer.mark();
            char c1 = buffer.get();
            char c2 = buffer.get();
            buffer.reset();
            buffer.put(c2).put(c1);
        }
    }
    public static void main(String[] args) {
        char[] data = "UsingBuffers".toCharArray();
```

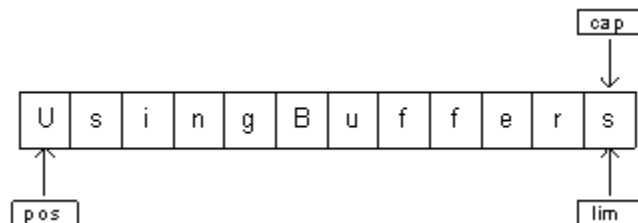
```

ByteBuffer bb = ByteBuffer.allocate(data.length * 2);
CharBuffer cb = bb.asCharBuffer();
cb.put(data);
System.out.println(cb.rewind());
symmetricScramble(cb);
System.out.println(cb.rewind());
symmetricScramble(cb);
System.out.println(cb.rewind());
monitor.expect(new String[] {
    "UsingBuffers",
    "sUniBgfuefsr",
    "UsingBuffers"
});
}
} ///:~

```

尽管你可以通过对某个 `char` 数组调用 `wrap()` 方法来直接产生一个 `CharBuffer`，但是在本例中取而代之的是分配一个底层的 `ByteBuffer`，产生的 `CharBuffer` 只是 `ByteBuffer` 上的一个视图而已。这里要强调我们总是以操纵使用 `ByteBuffer` 为目标这个事实，因为它可以和通道进行交互。

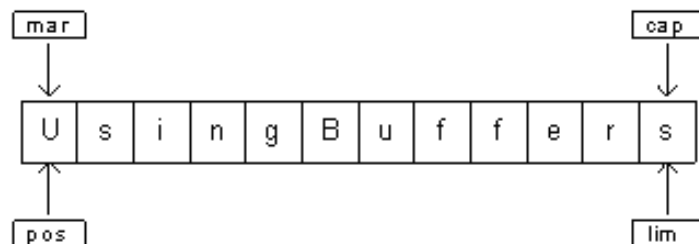
下面是 `Put()` 之后，缓冲器的样子：



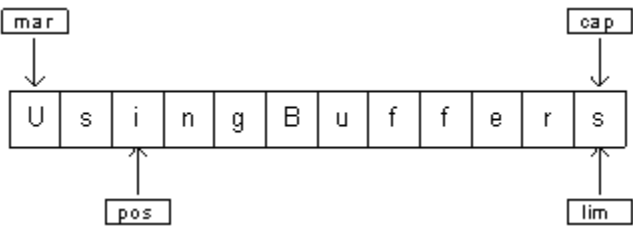
`position` 指针指向缓冲器中的第一个元素，`capacity` 和 `limit` 则指向最后一个元素。

在程序的 `symmetricScramble()` 方法中，迭代执行 `while` 循环直到 `position` 等于 `limit`。一旦调用缓冲器上的相对的 `get()` 或 `put()` 功能，`position` 指针就会随之相应改变。我们也可以调用绝对的、包含一个参数的 `get()` 和 `put()` 方法，参数指待 `get()` 或 `put()` 的发生位置。不过，这些方法不会改变缓冲器的 `position` 指针。

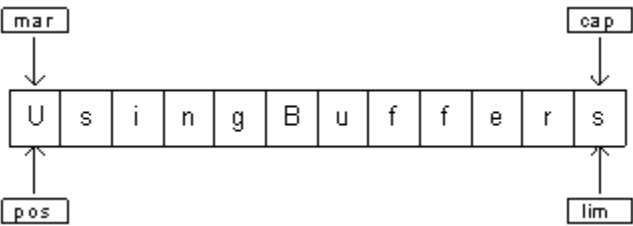
当操纵到 `while` 循环时，使用 `mark()` 调用来设置 `mark` 的值。此时，缓冲器状态如下：



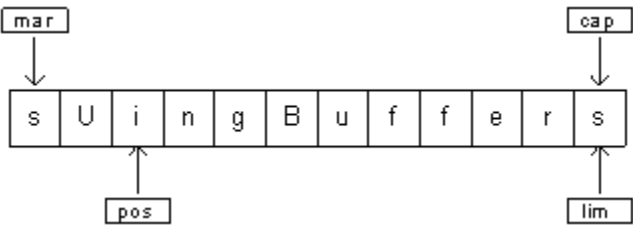
两个相对的 `get()` 调用把前两个字符保存到变量 `c1` 和 `c2` 中，调用完这两个方法后，缓冲器如下：



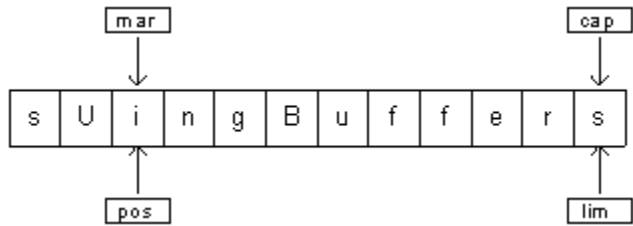
为了实现交换，我们要在 `position=0` 时写入 `c2`，`position=1` 时写入 `c1`。我们也可以使用绝对输入方法来实现，或者使用 `reset()` 把 `position` 的值设为 `mark` 的值：



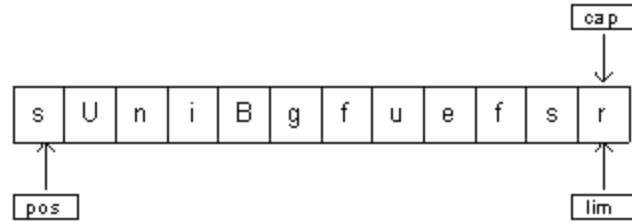
这两个 `put()` 方法先写 `c2`，接着写 `c1`：



在下一次循环迭代期间，将 `mark` 设置成 `position` 的当前值：



这个过程将会持续到遍历完整个缓冲器。在 `while` 循环的最后，`position` 指向缓冲器的末尾。所以如果想要打印缓冲器，只能打印出 `position` 和 `limit` 之间的字符。因此，如果想显示缓冲器的全部内容，我们必须使用 `rewind()` 把 `position` 设置到缓冲器的开始位置。下面是调用 `rewind()` 之后缓冲器的状态（`mark` 的值则变得不明确）：



当再次调用 `symmetricScramble()` 功能时，会对 `CharBuffer` 进行同样的处理，并将其恢复到初始状态。

存储器映射文件

存储器映射文件允许我们创建和修改那些因为太大而不能放入内存的文件。有了存储器映射文件，我们就可以假定整个文件都在内存中，而且可以完全把它当作非常大的数组来访问。这种方法极大地简化了用于修改文件的代码。下面是一个小例子：

```
//: c12:LargeMappedFiles.java
// Creating a very large file using mapping.
// {RunByHand}
// {Clean: test.dat}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class LargeMappedFiles {
    static int length = 0x8FFFFFFF; // 128 Mb
    public static void main(String[] args) throws Exception {
        MappedByteBuffer out =
            new RandomAccessFile("test.dat", "rw").getChannel()
                .map(FileChannel.MapMode.READ_WRITE, 0, length);
        for(int i = 0; i < length; i++)
            out.put((byte)'x');
        System.out.println("Finished writing");
        for(int i = length/2; i < length/2 + 6; i++)
            System.out.print((char)out.get(i));
    }
} ///:~
```

为了既能写又能读，我们先由 `RandomAccessFile` 开始，获得该文件上的通道，然后调用 `map()` 产生 `MappedByteBuffer`，这是一种特殊类型的直接缓冲器。注意我们必须指定映射文件的初始位置和映射区域的长度。这意味着我们可以映射某个大文件的一个较小的部分。

MappedByteBuffer 由 ByteBuffer 继承而来，因此它具有 ByteBuffer 的所有方法。这里，我们仅仅展示了非常简单的 put() 和 get()，但是我们同样可以使用像 asCharBuffer() 等这样的用法。

前面那个程序创建的文件为 128M，这可能比操作系统允许的空间要大。看似我们可以一次访问到整个文件，因为只有一部分文件放入了内存，文件的其他部分被交换了出去。用这种方式，很大的文件（可达 2GB）也可以很容易地被修改。注意底层操作系统的文件映射工具是被用来最大化地提高性能的。

性能

尽管“旧”的 I/O 流在用 nio 实现后性能有所提高，但是映射文件访问往往可以更加显著地加快速度。下面的程序进行了简单的性能比较。

```
//: c12:MappedIO.java
// {Clean: temp.tmp}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedIO {
    private static int numOfInts = 4000000;
    private static int numOfUbuffInts = 200000;
    private abstract static class Tester {
        private String name;
        public Tester(String name) { this.name = name; }
        public long runTest() {
            System.out.print(name + ": ");
            try {
                long startTime = System.currentTimeMillis();
                test();
                long endTime = System.currentTimeMillis();
                return (endTime - startTime);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
        public abstract void test() throws IOException;
    }
    private static Tester[] tests = {
        new Tester("Stream Write") {
            public void test() throws IOException {
                DataOutputStream dos = new DataOutputStream(
                    new BufferedOutputStream(
```

```

        new FileOutputStream(new File("temp.tmp"))));
    for(int i = 0; i < numOfInts; i++)
        dos.writeInt(i);
    dos.close();
}
},
new Tester("Mapped Write") {
    public void test() throws IOException {
        FileChannel fc =
            new RandomAccessFile("temp.tmp", "rw")
                .getChannel();
        IntBuffer ib = fc.map(
            FileChannel.MapMode.READ_WRITE, 0, fc.size())
            .asIntBuffer();
        for(int i = 0; i < numOfInts; i++)
            ib.put(i);
        fc.close();
    }
},
new Tester("Stream Read") {
    public void test() throws IOException {
        DataInputStream dis = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("temp.tmp")));
        for(int i = 0; i < numOfInts; i++)
            dis.readInt();
        dis.close();
    }
},
new Tester("Mapped Read") {
    public void test() throws IOException {
        FileChannel fc = new FileInputStream(
            new File("temp.tmp")).getChannel();
        IntBuffer ib = fc.map(
            FileChannel.MapMode.READ_ONLY, 0, fc.size())
            .asIntBuffer();
        while(ib.hasRemaining())
            ib.get();
        fc.close();
    }
},
new Tester("Stream Read/Write") {
    public void test() throws IOException {
        RandomAccessFile raf = new RandomAccessFile(

```

```

        new File("temp.tmp"), "rw");
    raf.writeInt(1);
    for(int i = 0; i < numOfUbuffInts; i++) {
        raf.seek(raf.length() - 4);
        raf.writeInt(raf.readInt());
    }
    raf.close();
}
},
new Tester("Mapped Read/Write") {
    public void test() throws IOException {
        FileChannel fc = new RandomAccessFile(
            new File("temp.tmp"), "rw").getChannel();
        IntBuffer ib = fc.map(
            FileChannel.MapMode.READ_WRITE, 0, fc.size())
            .asIntBuffer();
        ib.put(0);
        for(int i = 1; i < numOfUbuffInts; i++)
            ib.put(ib.get(i - 1));
        fc.close();
    }
}
};
public static void main(String[] args) {
    for(int i = 0; i < tests.length; i++)
        System.out.println(tests[i].runTest());
}
} ///: ~

```

正如在本书前面的例子中所看到的那样，`runTest()` 是一个模板方法 (Template Method)，为在匿名内部子类中定义的 `test()` 的各种实现提供了测试框架。每种子类都将执行一种测试，因此 `test()` 方法为我们进行各种 I/O 操作提供了原型。

尽管映射写似乎要用到 `FileOutputStream`，但是映射文件中的所有输出必须使用 `RandomAccessFile`，正如前面程序代码中的读/写一样。

下面是一次运行的输出结果：

```

Stream Write: 1719
Mapped Write: 359
Stream Read: 750
Mapped Read: 125
Stream Read/Write: 5188
Mapped Read/Write: 16

```

注意 `test()` 方法包括初始化各种 I/O 对象的时间，因此，即使建立映射文件的花费很大，但是整体受益比起 I/O 流来说还是很显著的。

文件加锁

JDK1.4 引入了文件加锁机制，允许我们同步访问一个共享文件。不过，竞争同一文件的两个线程可能在不同的 java 虚拟机上，或者一个是 Java 线程，另一个是操作系统中其他的某个本地线程。文件锁对其他的操作系统进程是可见的，因为 Java 的文件加锁直接映射到了本地操作系统的加锁机构。

下面是一个关于文件加锁的简单例子。

```
//: c12:FileLocking.java
// {Clean: file.txt}
import java.io.FileOutputStream;
import java.nio.channels.*;

public class FileLocking {
    public static void main(String[] args) throws Exception {
        FileOutputStream fos= new FileOutputStream("file.txt");
        FileLock fl = fos.getChannel().tryLock();
        if(fl != null) {
            System.out.println("Locked File");
            Thread.sleep(100);
            fl.release();
            System.out.println("Released Lock");
        }
        fos.close();
    }
} ///: ~
```

通过对 `FileChannel` 调用 `tryLock()` 或 `lock()`，就可以获得整个文件的 `FileLock`。

(`SocketChannel`、`DatagramChannel` 和 `ServerSocketChannel` 不需要加锁，因为它们是从单进程实体继承而来；我们通常不在两个进程之间的共享网络 socket) `tryLock()` 是非阻塞式的。它设法获取锁，但是如果不能获得（当其他一些进程已经持有相同的锁，并且不共享时），它将直接从方法调用返回。`lock()` 则要阻塞直至锁可以获得，或调用 `lock()` 的线程中断，或调用 `lock()` 的通道关闭。使用 `FileLock.release()` 可以释放锁。

也可以使用如下方法对文件的一部分上锁：

```
tryLock(long position, long size, boolean shared)
```

或者

`lock(long position, long size, boolean shared)`

其中，加锁的区域由 `size—position` 决定。第三个参数指定是否是共享锁。

尽管无参数的加锁方法将根据文件尺寸的变化而变化，但是具有固定尺寸的锁不随文件尺寸的变化而变化。如果你获得了某一区域上（从 `position` 到 `position+size`）的锁，当文件增大超出 `position+size` 时，那么在 `position+size` 之外的部分不会被锁定。无参数的加锁方法会对整个文件进行加锁，甚至文件变大后也是如此。

对独占锁或者共享锁的支持必须由底层的操作系统提供。如果操作系统不支持共享锁并为每一个请求都创建一个锁，那么它就会使用独占锁。锁的类型（共享或独占）可以通过 `FileLock.isShared()` 进行查询。

对映射文件的部分加锁

如前面提到的，文件映射通常应用于极大的文件。因此我们可能需要对如此巨大的文件进行部分加锁，以便其他进程可以修改文件中未被加锁的部分。例如，数据库就是这样，因此多个用户可以同时访问到它。

下面例子中有两个线程，分别加锁文件的不同部分。

```
//: c12:LockingMappedFiles.java
// Locking portions of a mapped file.
// {RunByHand}
// {Clean: test.dat}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class LockingMappedFiles {
    static final int LENGTH = 0x8FFFFFFF; // 128 Mb
    static FileChannel fc;
    public static void main(String[] args) throws Exception {
        fc =
            new RandomAccessFile("test.dat", "rw").getChannel();
        MappedByteBuffer out =
            fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);
        for(int i = 0; i < LENGTH; i++)
            out.put((byte)'x');
        new LockAndModify(out, 0, 0 + LENGTH/3);
        new LockAndModify(out, LENGTH/2, LENGTH/2 + LENGTH/4);
    }
    private static class LockAndModify extends Thread {
        private ByteBuffer buff;
```

```

private int start, end;
LockAndModify(ByteBuffer mbb, int start, int end) {
    this.start = start;
    this.end = end;
    mbb.limit(end);
    mbb.position(start);
    buff = mbb.slice();
    start();
}
public void run() {
    try {
        // Exclusive lock with no overlap:
        FileLock fl = fc.lock(start, end, false);
        System.out.println("Locked: " + start + " to " + end);
        // Perform modification:
        while(buff.position() < buff.limit() - 1)
            buff.put((byte)(buff.get() + 1));
        fl.release();
        System.out.println("Released: " + start + " to " + end);
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}
}
} ///:~

```

线程类 `LockAndModify` 创建了缓冲区和用于修改的 `slice()`，然后在 `run()` 中，获得文件通道上的锁（我们不能获得缓冲器上的锁——只能是通道上的）。`lock()` 调用类似于在一个工程上获得线程锁——我们现在处在“临界区”，即对该部分的文件具有独占访问权。

如果有 Java 虚拟机，它会自动释放锁，或者关闭加锁的通道。不过我们也可以像程序中那样，显式地为 `FileLock` 调用 `release()` 释放锁。

压缩

Java I/O 类库中的类支持对压缩格式的数据流的读写。它们对现有的 I/O 类进行封装，以提供压缩功能。

这些类不是从 `Reader` 和 `Writer` 类衍生出来的，而是属于 `InputStream` 和 `OutputStream` 继承层次结构的一部分。这样做是因为压缩类库是按字节方式处理的，而不是字符。不过有时我们可能会被迫要混合使用两种类型的数据流（注意我们可以使用 `InputStreamReader` 和 `OutputStreamWriter` 在两种类型间方便地进行转换）。

压缩类	功能
CheckedInputStream	GetChecksum() 为任何 InputStream 产生校验和 (不仅是解压缩).
CheckedOutputStream	GetChecksum() 为任何 OutputStream 产生校验和 (不仅是解压缩).
DeflaterOutputStream	用于压缩的基类
ZipOutputStream	DeflaterOutputStream that 将数据压缩成 Zip 文件格式.
GZIPOutputStream	DeflaterOutputStream 将数据压缩成 GZIP 文件格式.
InflaterInputStream	用于解压缩的基类
ZipInputStream	InflaterInputStream 解压缩 Zip 文件格式的数据.
GZIPInputStream	InflaterInputStream 解压缩 GZIP 文件格式的数据.

尽管存在许多种压缩算法，但是 Zip 和 GZIP 可能是最常用的。因此我们可以很容易地使用多种可读写这些格式的工具来操纵我们的压缩数据。

用 GZIP 进行简单压缩

GZIP 接口非常简单，因此如果我们只想对单个数据流进行压缩（而不是一系列互异数据），那么它就可能是比较适合的选择。下面是对单个文件进行压缩的例子：

```
//: c12:GZIPcompress.java
// {Args: GZIPcompress.java}
// {Clean: test.gz}
import com.bruceeckel.simpletest.*;
import java.io.*;
import java.util.zip.*;

public class GZIPcompress {
    private static Test monitor = new Test();
    // Throw exceptions to console:
    public static void main(String[] args)
        throws IOException {
        if(args.length == 0) {
            System.out.println(
                "Usage: \nGZIPcompress file\n" +
                "\tUses GZIP compression to compress " +
```

```

        "the file to test.gz");
        System.exit(1);
    }
    BufferedReader in = new BufferedReader(
        new FileReader(args[0]));
    BufferedOutputStream out = new BufferedOutputStream(
        new GZIPOutputStream(
            new FileOutputStream("test.gz")));
    System.out.println("Writing file");
    int c;
    while((c = in.read()) != -1)
        out.write(c);
    in.close();
    out.close();
    System.out.println("Reading file");
    BufferedReader in2 = new BufferedReader(
        new InputStreamReader(new GZIPInputStream(
            new FileInputStream("test.gz"))));
    String s;
    while((s = in2.readLine()) != null)
        System.out.println(s);
    monitor.expect(new String[] {
        "Writing file",
        "Reading file"
    }, args[0]);
}
} ///: ~

```

压缩类的使用非常直观——我们直接将输出流封装成 `GZIPOutputStream` 或 `ZipOutputStream`，并将输入流封装成 `GZIPInputStream` 或 `ZipInputStream` 即可。其他全部操作就是通常的 IO 读写。这个例子把面向字符的流和面向字节的流混合了起来：输入用 `Reader` 类。而 `GZIPOutputStream` 的构造器只能接受 `OutputStream` 对象，不能接受 `Writer` 对象。在打开文件时，`GZIPInputStream` 就会被转换成 `Reader`。

用 **Zip** 进行多文件保存

支持 `Zip` 格式的 `Java` 库更加全面。利用它可以方便地保存多个文件，甚至有一个独立的类使得读取 `Zip` 文件更加方便。这个类库使用的是标准 `Zip` 格式，所以能与当前那些可通过因特网下载到的压缩工具很好地协作。下面这个例子具有与前例相同的形式，但它能根据需要来处理任意多个命令行参数。另外，它示例了用 `Checksum` 类来计算和校验文件的“校验和”（`Checksum`）的方法。一共有两种 `Checksum` 类型：`Adler32`（速度要快一些）和 `CRC32`（慢一些，但更准确）。

```

//: c12: ZipCompress.java

```



```

// Uses Zip compression to compress any
// number of files given on the command line.
// {Args: ZipCompress.java}
// {Clean: test.zip}
import com.bruceeckel.simpletest.*;
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ZipCompress {
    private static Test monitor = new Test();
    // Throw exceptions to console:
    public static void main(String[] args)
        throws IOException {
        FileOutputStream f = new FileOutputStream("test.zip");
        CheckedOutputStream csum =
            new CheckedOutputStream(f, new Adler32());
        ZipOutputStream zos = new ZipOutputStream(csum);
        BufferedOutputStream out =
            new BufferedOutputStream(zos);
        zos.setComment("A test of Java Zipping");
        // No corresponding getComment(), though.
        for(int i = 0; i < args.length; i++) {
            System.out.println("Writing file " + args[i]);
            BufferedReader in =
                new BufferedReader(new FileReader(args[i]));
            zos.putNextEntry(new ZipEntry(args[i]));
            int c;
            while((c = in.read()) != -1)
                out.write(c);
            in.close();
        }
        out.close();
        // Checksum valid only after the file has been closed!
        System.out.println("Checksum: " +
            csum.getChecksum().getValue());
        // Now extract the files:
        System.out.println("Reading file");
        FileInputStream fi = new FileInputStream("test.zip");
        CheckedInputStream csumi =
            new CheckedInputStream(fi, new Adler32());
        ZipInputStream in2 = new ZipInputStream(csumi);
        BufferedInputStream bis = new BufferedInputStream(in2);
        ZipEntry ze;

```

```

while((ze = in2.getNextEntry()) != null) {
    System.out.println("Reading file " + ze);
    int x;
    while((x = bis.read()) != -1)
        System.out.write(x);
}
if(args.length == 1)
    monitor.expect(new String[] {
        "Writing file " + args[0],
        "%% Checksum: \\d+",
        "Reading file",
        "Reading file " + args[0]}, args[0]);
System.out.println("Checksum: " +
    csumi.getChecksum().getValue());
bis.close();
// Alternative way to open and read zip files:
ZipFile zf = new ZipFile("test.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
    ZipEntry ze2 = (ZipEntry)e.nextElement();
    System.out.println("File: " + ze2);
    // ... and extract the data as before
}
if(args.length == 1)
    monitor.expect(new String[] {
        "%% Checksum: \\d+",
        "File: " + args[0]
    });
}
} ///: ~

```

对于每一个要加入压缩文档的文件，都必须调用 `putNextEntry()`，并将其传递给一个 `ZipEntry` 对象。`ZipEntry` 对象包含了一个功能广泛的接口，允许你获取和设置 `Zip` 文件内该特定项上所有可利用的数据：名字、压缩的和未压缩的文件大小、日期、CRC 校验和、额外域的数据、注释、压缩方法以及它是否是一个目录入口等等。然而，即使 `Zip` 格式提供了设置密码的方法，但 Java 的 `Zip` 类库并不提供这方面的支持。虽然 `CheckedInputStream` 和 `CheckedOutputStream` 同时支持 `Adler32` 和 `CRC32` 两种类型的校验和，但是 `ZipEntry` 类只有一个支持 `CRC` 的接口。虽然这是一个底层 `Zip` 格式的限制，但却限制了你不能使用速度更快的 `Adler32`。

为了能够解压缩文件，`ZipInputStream` 提供了一个 `getNextEntry()` 方法返回下一个 `ZipEntry`，如果该 `ZipEntry` 存在的话。有一个更简便的方法来解压缩文件——利用 `ZipFile` 对象读取文件。该对象有一个 `entries()` 方法用来向 `ZipEntry` 返回一个 `Enumeration`（枚举）。

如果要读取校验和，必须拥有对与之相关联的 `Checksum` 对象的访问权限。在这里保留了指向 `CheckedOutputStream` 和 `CheckedInputStream` 对象的一个引用。但是，也可以只保留一个指向 `Checksum` 对象的引用。

`Zip` 流中有一个令人困惑的方法 `setComment()`。正如在前面 `ZipCompress.java` 中所示，我们可以在写一个文件时写注释，但却没有任何方法恢复 `ZipInputStream` 内的注释。似乎只能通过 `ZipEntry`，注释才能完全在逐条获取的基础上被支持。

当然，`GZIP` 或 `Zip` 库的使用并不仅仅局限于文件——它可以压缩任何东西，包括需要通过网络发送的数据。

Java 档案文件（JAR）

`Zip` 格式也被应用于 `JAR`（Java ARchive）文件格式中。这种文件格式就象 `Zip` 一样可以将一组文件压缩到单个压缩文件中。同 `Java` 中其他任何东西一样，`JAR` 文件是跨平台的，所以不必担心跨平台的问题。声音和图像文件可以像类文件一样被包含在其中。

`JAR` 文件非常有用，尤其是在涉及因特网应用的时候。如果不采用 `JAR` 文件，`Web` 浏览器为了下载构成一个应用的所有文件时必须重复多次请求 `Web` 服务器。而且所有这些文件都是未经压缩的。如果将所有这些文件合并到一个 `JAR` 文件中，只需向远程服务器发出一次请求即可。同时，由于采用了压缩技术，可以使传输时间更短。另外，出于安全的考虑，`JAR` 文件中的每个条目都可以加上数字化签名（可参考第 14 章签名的例子）。

一个 `JAR` 文件由一组压缩文件构成，同时还有一张描述了所有这些文件的“文件清单”。（可自行创建文件清单，也可以由 `jar` 程序自动生成）。在 `JDK` 文档中，可以找到与 `JAR` 文件清单相关的更多资料。

`Sun` 的 `JDK` 自带的 `jar` 程序可根据我们的选择自动压缩文件。可以用命令行的形式调用它：

`jar [options] destination [manifest] inputfile(s)`

`options` 只是一个字母集合（不必输入任何“-”或其他任何标识符）。以下这些选项字符在 `Unix` 和 `Linux` 系统中的 `tar` 文件中也具有相同的意义。具体意义如下所示：

c	创建一个新的或空的压缩文档。
t	列出目录表。
x	解压所有文件。
x file	解压该文件。
f	意指：“我打算指定一个文件名。” 如果我们没有用到这个选项， jar 假设所有的输入都来自于标准输入，在创建一个文件时，输出对象也假设为标准输出。.
m	表示第一个参数将是用户自建的清单文件的名字。
v	产生详细输出，描述 jar 所做的工作。
O	只储存文件，不压缩文件。（用来创建一个可放在类路径中的

	JAR 文件)。
M	不自动创建文件清单。

如果想要将包含子目录的文件压缩到 JAR 文件中，那么该子目录会被自动添加到 JAR 文件中，且包括该子目录的所有子目录。路径信息也会被保留。

下面是一些调用 `jar` 的典型方法：

```
jar cf myJarFile.jar *.class
```

这条语句创建了一个名为 `myJarFile.jar` 的 JAR 文件，该文件包含了当前目录中的所有类文件，以及自动产生的清单文件。

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

与前例类似，但添加了一个名为 `myManifestFile.mf` 的用户自建清单文件。

```
jar tf myJarFile.jar
```

产生 `myJarFile.jar` 内所有文件的一个目录表。

```
jar tvf myJarFile.jar
```

添加“verbose”（详尽）标志，提供有关 `myJarFile.jar` 中的文件的更详细的信息。

```
jar cvf myApp.jar audio classes image
```

假定 `audio`，`classes` 和 `image` 是子目录，这条语句将所有子目录合并到文件 `myApp.jar` 中，其中也包括了“verbose”标志。当 `jar` 程序运行时，该标志可以提供更详细的信息。

如果用 `0`（零）选项创建一个 JAR 文件，那么该文件就可放入类路径（CLASSPATH）中：

```
CLASSPATH="lib1.jar;lib2.jar;"
```

然后 Java 就可以在 `lib1.jar` 和 `lib2.jar` 中搜索目标类文件了。

`jar` 工具的功能没有 `zip` 工具那么强大。例如，不能够对一个已经存在的 JAR 文件作添加或更新文件的操作，只能从头创建一个 JAR 文件。同时，也不能将文件移动至一个 JAR 文件，并在移动后将它们删除。然而，在一种平台上创建的 JAR 文件可以被在其他任何平台上的 `jar` 工具透明地阅读（这个问题有时会困扰 `zip` 工具）。

你将会在第 14 章看到，JAR 文件也被用来为 Java Beans 打包。

对象序列化

Java 的对象序列化（Object Serialization）将那些实现了 `Serializable` 接口的对象转换成一个字节序列，并可以在以后将这个字节序列完全恢复为原来的对象。这一过程甚至可通过网络进行。这意味着序列化机制能自动弥补不同操作系统之间的差异。也就是说，可以在运行 Windows 系统的计算机上创建一个对象，将其序列化，通过网络将它发送给一台运行

Unix 系统的计算机，然后在那里准确地重新组装，而你却不必担心数据在不同机器上的表示会不同，也不必关心字节的顺序或者其他任何细节。

就其本身来说，对象的序列化是非常有趣的，因为利用它可以实现“轻量级持久化（lightweight persistence）”。“持久化”意味着一个对象的生存周期并不取决于程序是否正在执行；它可以生存于程序的调用之间。通过将一个序列化对象写入磁盘，然后在重新调用程序时恢复该对象，就能够实现持久化的效果。之所以称其为“轻量级”，是因为不能用某种“persistent”（持久）关键字来简单地定义一个对象，并让系统自动维护其他细节问题（尽管将来有可能实现）。相反，对象必须在程序中显式地序列化和重组。如果需要一个更严格的持久化机制，可以考虑使用 Java 数据对象(JDO)或者像 Hibernate 之类的工具 (<http://hibernate.sourceforge.net>)。更多的细节可参考《Thinking in Enterprise Java》，可从 www.BruceEckel.com 下载。

对象序列化的概念加入到语言中是为了提供对两种主要特性的支持。Java 的“远程方法调用”（RMI, Remote Method Invocation）使存活于其他计算机上的对象使用起来就像是存活于本机上一样。当向远程对象发送消息时，需要通过对象序列化来传输参数和返回值。在《Thinking in Enterprise Java》中有对 RMI 的具体讨论。

对 Java Beans 来说对象的序列化也是必需的，可参看第 14 章。使用一个 Bean 时，一般情况下是在设计阶段对它的状态信息进行配置。这种状态信息必须保存下来，并在程序启动以后，进行恢复；具体工作由对象序列化完成。

只要对象实现了 Serializable 接口（该接口仅是一个标记接口，不包括任何方法），对象的序列化处理就会非常简单。当序列化的概念被加入到语言中时，许多标准库类都发生了改变，以便能够使之序列化——其中包括所有原始数据类型的封装器、所有容器类以及许多其他的東西。甚至 Class 对象也可以被序列化。

为了序列化一个对象，首先要创建某些 OutputStream 对象，然后将其封装在一个 ObjectOutputStream 对象内。这时，只需调用 writeObject() 即可将对象序列化，并将其发送给 OutputStream。要将一个序列重组为一个对象，需要将一个 InputStream 封装在 ObjectInputStream 内，然后调用 readObject()。和往常一样，我们最后获得的是指向一个向上转型为 Object 的句柄，所以必须向下转型，以便能够直接对其进行设置。

对象序列化特别“聪明”的一个地方是它不仅保存了对象的“全景图”，而且能追踪对象内包含的所有引用并保存那些对象；接着又能对每个这样的对象内包含的引用进行追踪；以此类推。这种情况有时被称为“对象网”，单个对象可与之建立连接，而且它还包含了对对象的引用数组和成员对象。如果必须保持一套自己的对象序列化机制，那么维护那些可追踪到所有链接的代码可能会显得非常麻烦。然而，由于 Java 的对象序列化似乎找不出什么缺点，所以请尽量不要自己动手，让它用优化的算法自动维护整个对象网。下面这个例子通过对链接的对象生成一个“Worm”（蠕虫）对序列化机制进行了测试。每个对象都与 Worm 中的下一段链接，同时又与属于不同类（Data）的对象引用数组链接：

```
//: c12:Worm.java
// Demonstrates object serialization.
// {Clean: worm.out}
import java.io.*;
import java.util.*;
```

```

class Data implements Serializable {
    private int n;
    public Data(int n) { this.n = n; }
    public String toString() { return Integer.toString(n); }
}

```

```

public class Worm implements Serializable {
    private static Random rand = new Random();
    private Data[] d = {
        new Data(rand.nextInt(10)),
        new Data(rand.nextInt(10)),
        new Data(rand.nextInt(10))
    };
    private Worm next;
    private char c;
    // Value of i == number of segments
    public Worm(int i, char x) {
        System.out.println("Worm constructor: " + i);
        c = x;
        if(--i > 0)
            next = new Worm(i, (char)(x + 1));
    }
    public Worm() {
        System.out.println("Default constructor");
    }
    public String toString() {
        String s = ":" + c + "(";
        for(int i = 0; i < d.length; i++)
            s += d[i];
        s += ")";
        if(next != null)
            s += next;
        return s;
    }
    // Throw exceptions to console:
    public static void main(String[] args)
        throws ClassNotFoundException, IOException {
        Worm w = new Worm(6, 'a');
        System.out.println("w = " + w);
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("worm.out"));
        out.writeObject("Worm storage\n");
        out.writeObject(w);
    }
}

```

```

        out.close(); // Also flushes output
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("worm.out"));
        String s = (String)in.readObject();
        Worm w2 = (Worm)in.readObject();
        System.out.println(s + "w2 = " + w2);
        ByteArrayOutputStream bout =
            new ByteArrayOutputStream();
        ObjectOutputStream out2 = new ObjectOutputStream(bout);
        out2.writeObject("Worm storage\n");
        out2.writeObject(w);
        out2.flush();
        ObjectInputStream in2 = new ObjectInputStream(
            new ByteArrayInputStream(bout.toByteArray()));
        s = (String)in2.readObject();
        Worm w3 = (Worm)in2.readObject();
        System.out.println(s + "w3 = " + w3);
    }
} ///: ~

```

更有趣的是，Worm 内的 Data 对象数组是用随机数初始化的（这样就不用怀疑编译器保留了某种原始信息）。每个 Worm 段都用一个 Char 标记。该 Char 是在递归生成链接的 Worm 列表时自动产生的。要创建一个 Worm，必须告诉构造器你所希望的它的长度。在产生下一个引用时，要调用 Worm 构造器，并将长度减 1，以此类推。最后一个 next 句柄则为 null（空），表示已到达 Worm 的尾部。

以上这些操作都使得事情变得更加复杂，从而加大了对对象序列化的难度。然而，真正的序列化过程却是非常简单的。一旦从另外某个流创建了 ObjectOutputStream，writeObject() 就会将对象序列化。注意也可以为一个 String 调用 writeObject()。也可以用与 DataOutputStream 相同的方法写入所有原始数据类型（它们具有同样的接口）。

有两个独立的代码段看起来是相似的。一个读写的是文件，而另一个读写的是一个字节数组（ByteArray）。可利用序列化将对象读写到任何 DataInputStream 或者 DataOutputStream，甚至包括网络；正如在《Thinking in Enterprise Java》中所述。某一次运行后的输出结果如下：

```

Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(414):b(276):c(773):d(870):e(210):f(279)
Worm storage
w2 = :a(414):b(276):c(773):d(870):e(210):f(279)
Worm storage

```

```
w3 = :a(414):b(276):c(773):d(870):e(210):f(279)
```

可以看出，重组的对象确实包含了原对象中的所有链接。

注意在对一个可序列化（**Serializable**）对象进行重组的过程中，没有调用任何构造器，包括缺省的构造器。整个对象都是通过从 **InputStream** 中取得数据恢复而来的。

对象序列化是面向字节的，因此采用 **InputStream** 和 **OutputStream** 层次结构。

寻找类

读者或许会奇怪将一个对象从它的序列化状态中恢复出来有哪些工作是必须的呢？举个例子来说，假如我们将一个对象序列化，并通过网络将其作为文件传送给另一台计算机。那么，另一台计算机上的程序可以只利用该文件内容来重组这个对象吗？

回答这个问题的最好方法就是做一个实验。下面这个文件位于本章的子目录下：

```
//: c12:Alien.java
// A serializable class.
import java.io.*;
public class Alien implements Serializable { } ///: ~
```

用于创建和序列化一个 **Alien** 对象的文件位于相同的目录下：

```
//: c12:FreezeAlien.java
// Create a serialized output file.
// {Clean: X.file}
import java.io.*;

public class FreezeAlien {
    // Throw exceptions to console:
    public static void main(String[] args) throws Exception {
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("X.file"));
        Alien zorcon = new Alien();
        out.writeObject(zorcon);
    }
} ///: ~
```

这个程序不但能捕获和处理异常，而且将异常抛出到 **main()** 方法之外，以便通过控制台产生报告。

一旦该程序被编译和运行，它就会在 **c12** 目录下产生一个名为 **X.file** 的文件。以下代码位于一个名为 **xfiles** 的子目录下

```
//: c12:xfiles:ThawAlien.java
// Try to recover a serialized file without the
```



```
// class of object that's stored in that file.
// {ThrowsException}
import java.io.*;

public class ThawAlien {
    public static void main(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(new File("../X.file")));
        Object mystery = in.readObject();
        System.out.println(mystery.getClass());
    }
} ///:~
```

打开文件和读取 `mystery` 对象中的内容都需要 `Alien` 的 `Class` 对象；而 Java 虚拟机(JVM)找不到 `Alien.class`（除非它正好在类路径内，而本例却不在类路径之内）。这样就会得到一个名叫 `ClassNotFoundException` 的异常（同样地，除非能够验证 `Alien` 存在，否则它等于消失）。必须保证 Java 虚拟机（JVM）能找到相关的 `.class` 文件。

序列化的控制

正如大家看到的那样，缺省的序列化机制并不难操纵。然而，如果有特殊的需要那又该怎么办呢？例如，也许你有考虑特殊的安全问题，而且你不希望对象的某一部分被序列化；或者一个对象被重组以后，某子对象需要重新创建，从而不必将该子对象序列化。

在这些特殊情况下，可通过实现 `Externalizable` 接口代替实现 `Serializable` 接口来对序列化过程进行控制。这个 `Externalizable` 接口继承了 `Serializable` 接口，同时增添了两个方法：`writeExternal()`和 `readExternal()`。这两个方法会在序列化和重组的过程中被自动调用，以便执行一些特殊操作。

下面这个例子展示了 `Externalizable` 接口方法的简单实现。注意 `Blip1` 和 `Blip2` 除了细微的差别之外，几乎完全一致（研究一下代码，看看你是否能发现）：

```
///: c12:Blips.java

// Simple use of Externalizable & a pitfall.
// {Clean: Blips.out}
import com.bruceeckel.simpletest.*;
import java.io.*;
import java.util.*;

class Blip1 implements Externalizable {
    public Blip1() {
        System.out.println("Blip1 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
```

```

        System.out.println("Blip1.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        System.out.println("Blip1.readExternal");
    }
}

class Blip2 implements Externalizable {
    Blip2() {
        System.out.println("Blip2 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        System.out.println("Blip2.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        System.out.println("Blip2.readExternal");
    }
}

public class Blips {
    private static Test monitor = new Test();
    // Throw exceptions to console:
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        System.out.println("Constructing objects:");
        Blip1 b1 = new Blip1();
        Blip2 b2 = new Blip2();
        ObjectOutputStream o = new ObjectOutputStream(
            new FileOutputStream("Blips.out"));
        System.out.println("Saving objects:");
        o.writeObject(b1);
        o.writeObject(b2);
        o.close();
        // Now get them back:
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("Blips.out"));
        System.out.println("Recovering b1:");
        b1 = (Blip1)in.readObject();
        // OOPS! Throws an exception:
        //! System.out.println("Recovering b2:");
        //! b2 = (Blip2)in.readObject();
    }
}

```

```

        monitor.expect(new String[] {
            "Constructing objects:",
            "Blip1 Constructor",
            "Blip2 Constructor",
            "Saving objects:",
            "Blip1.writeExternal",
            "Blip2.writeExternal",
            "Recovering b1:",
            "Blip1 Constructor",
            "Blip1.readExternal"
        });
    }
} ///: ~

```

上例中没有恢复 **Blip2** 对象，因为那样做会导致一个异常。你找出 **Blip1** 和 **Blip2** 之间的区别了吗？**Blip1** 的构造方法是“公共的”（**public**），**Blip2** 的构造方法却不是，这样就会在恢复时造成异常。试试将 **Blip2** 的构造方法变成“**public**”，然后删除`///`注释标记，看看是否能得到正确的结果。

恢复 **b1** 后，会调用 **Blip1** 缺省构造器。这与恢复一个可序列化（**Serializable**）对象不同。对于后者，对象完全以它存储的二进制位为基础重组，而不调用构造器。而对一个 **Externalizable** 对象，所有普通的缺省构造器都会被调用（包括在域定义时的初始化），然后调用 `readExternal()`。必须注意这一点——所有缺省的构造器都会被调用——以此来使 **Externalizable** 对象产生正确的行为。

下面这个例子示范了如何完整保存和恢复一个 **Externalizable** 对象：

```

///: c12:Blip3.java

// Reconstructing an externalizable object.
import com.bruceeckel.simpletest.*;
import java.io.*;
import java.util.*;

public class Blip3 implements Externalizable {
    private static Test monitor = new Test();
    private int i;
    private String s; // No initialization
    public Blip3() {
        System.out.println("Blip3 Constructor");
        // s, i not initialized
    }
    public Blip3(String x, int a) {
        System.out.println("Blip3(String x, int a)");
        s = x;
        i = a;
        // s & i initialized only in nondefault constructor.
    }
}

```

```

    }
    public String toString() { return s + i; }
    public void writeExternal(ObjectOutput out)
    throws IOException {
        System.out.println("Blip3.writeExternal");
        // You must do this:
        out.writeObject(s);
        out.writeInt(i);
    }
    public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException {
        System.out.println("Blip3.readExternal");
        // You must do this:
        s = (String)in.readObject();
        i = in.readInt();
    }
    public static void main(String[] args)
    throws IOException, ClassNotFoundException {
        System.out.println("Constructing objects:");
        Blip3 b3 = new Blip3("A String ", 47);
        System.out.println(b3);
        ObjectOutputStream o = new ObjectOutputStream(
            new FileOutputStream("Blip3.out"));
        System.out.println("Saving object:");
        o.writeObject(b3);
        o.close();
        // Now get it back:
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("Blip3.out"));
        System.out.println("Recovering b3:");
        b3 = (Blip3)in.readObject();
        System.out.println(b3);
        monitor.expect(new String[] {
            "Constructing objects:",
            "Blip3(String x, int a)",
            "A String 47",
            "Saving object:",
            "Blip3.writeExternal",
            "Recovering b3:",
            "Blip3 Constructor",
            "Blip3.readExternal",
            "A String 47"
        });
    }
}

```

```
} ///: ~
```

其中，域 `s` 和 `i` 只在第二个构造器中被初始化，而不在缺省的构造器中被初始化的。这意味着假如不在 `readExternal()` 中初始化 `s` 和 `i`，`s` 就会为 `null`，而 `i` 就会为零（因为在创建对象的第一步中将对象的存储空间清除为 0）。如果注释掉跟随于“You must do this”后面的两行代码，然后运行程序，就会发现当对象重组后，`s` 是 `null`，而 `i` 是零。

我们如果从一个 `Externalizable` 对象继承，通常需要调用基类版本的 `writeExternal()` 和 `readExternal()` 来为基类组件提供恰当的存储和恢复功能。

因此，为了正常运行，我们不仅需要在 `writeExternal()` 方法（没有任何缺省行为来为 `Externalizable` 对象写入任何成员对象）中将来自对象的重要信息写入，还必须在 `readExternal()` 方法中恢复数据。起先，可能会有一点迷惑，因为 `Externalizable` 对象的缺省构造行为使其看起来似乎像某种自动发生的存储与恢复操作。但实际上并非如此。

transient（瞬时）关键字

当我们序列化进行控制时，可能存在某个特定子对象不想让 Java 的序列化机制自动保存与恢复。如果子对象表示的是我们不想序列化的敏感信息（如密码），通常就会面临这种情况。即使对象中的这些信息是“`private`”（私有）属性，一经序列化处理，人们就可以通过读取文件，或者拦截网络传输的方式来访问到它。

有一种防止对象的敏感部分被序列化的办法，就是将我们自己的类实现为 `Externalizable`，如前面所示。这样一来，没有任何东西可以自动序列化，并且我们可以在 `writeExternal()` 内部只对所需部分进行显式的序列化。

然而，如果我们正在操作的是一个 `Serializable` 对象，那么所有序列化操作都会自动进行。为了能够予以控制，可以用 `transient`（瞬时）关键字逐个域地关闭序列化，它意旨“不用麻烦你保存或恢复数据——我自己会处理的”。

例如，假设某个 `Login` 对象保存某个特定的登录会话信息。登录的合法性通过校验之后，我们想把数据保存下来，但不包括密码。为做到这一点，最简单的办法是实现 `Serializable`，并将 `password` 域标志为 `transient`。下面是具体的代码：

```
///: c12:Logon.java

// Demonstrates the "transient" keyword.
// {Clean: Logon.out}
import java.io.*;
import java.util.*;

public class Logon implements Serializable {
    private Date date = new Date();
    private String username;
    private transient String password;
    public Logon(String name, String pwd) {
```

```

        username = name;
        password = pwd;
    }
    public String toString() {
        String pwd = (password == null) ? "(n/a)" : password;
        return "logon info: \n    username: " + username +
            "\n    date: " + date + "\n    password: " + pwd;
    }
    public static void main(String[] args) throws Exception {
        Logon a = new Logon("Hulk", "myLittlePony");
        System.out.println( "logon a = " + a);
        ObjectOutputStream o = new ObjectOutputStream(
            new FileOutputStream("Logon.out"));
        o.writeObject(a);
        o.close();
        Thread.sleep(1000); // Delay for 1 second
        // Now get them back:
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("Logon.out"));
        System.out.println("Recovering object at "+new Date());
        a = (Logon)in.readObject();
        System.out.println("logon a = " + a);
    }
} ///: ~

```

我们可以看到，其中的 `date` 和 `username` 域是一般数据（不是 `transient`），所以它们会被自动序列化。而 `password` 是 `transient`，所以不会被自动保存到磁盘；另外，自动序列化机制也不会尝试去恢复它。输出如下：

```

logon a = logon info:
    username: Hulk
    date: Mon Oct 21 12:10:13 MDT 2002
    password: myLittlePony
Recovering object at Mon Oct 21 12:10:14 MDT 2002
logon a = logon info:
    username: Hulk
    date: Mon Oct 21 12:10:13 MDT 2002
    password: (n/a)

```

当对象被恢复时，`password` 域就会变成 `null`。注意必须用 `toString()` 检查 `password` 是否为 `null`，因为如果用重载的“+”运算符来连接 `String` 对象，并且该运算符如果遇到一个 `null` 引用，我们就会得到 `NullPointerException` 异常（新版 Java 可能会包含避免这个问题的代码）。

我们还可以发现：`date` 域被存储到了磁盘并从磁盘上被恢复了出来，而且没有再重新生成。

由于 `Externalizable` 对象在缺省情况下不保存它们的任何域，所以 `transient` 关键字只能和 `Serializable` 对象一起使用。

Externalizable 的替代方法

如果我们不是特别想要实现 `Externalizable` 接口，那么就还有另一种方法。我们可以实现 `Serializable` 接口，并添加（注意我说的是“添加”，而非“重载”或者“实现”）名为 `writeObject()` 和 `readObject()` 的方法。这样一旦对象被序列化或者被反序列化，就会自动地分别调用这两个方法。也就是说，只要我们提供了这两个方法，就会使用它们而不是缺省的序列化机制。

这些方法必须具有准确的方法签名：

```
private void writeObject(ObjectOutputStream stream)
    throws IOException;
```

```
private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException
```

从设计的观点来看，现在事情变得真是不可思议。首先，我们可能会认为由于这些方法不是基类或者 `Serializable` 接口的一部分，所以应该在它们自己的接口中进行定义。但是注意它们被定义成了 `private`，这意味着它们仅能被这个类的其他成员调用。然而，实际上我们并没有从这个类的其他方法中调用它们，而是 `ObjectOutputStream` 和 `ObjectInputStream` 对象的 `writeObject()` 和 `readObject()` 方法调用我们对象的 `writeObject()` 和 `readObject()` 方法（注意关于这里用到的相同方法名，我尽量抑制住不去谩骂。一句话：混乱）。你可能想知道 `ObjectOutputStream` 和 `ObjectInputStream` 对象是怎样访问你的类中的 `private` 方法的。我们只能假设这正是序列化神奇的一部分。

在任何情况下，在接口中定义的所有东西都自动是 `public` 的，因此如果 `writeObject()` 和 `readObject()` 必须是 `private` 的，那么它们不会是接口的一部分。因为我们必须要完全遵循其方法签名，所以其效果就和实现了接口一样。

在你调用 `ObjectOutputStream.writeObject()` 时，会检查你所传递的 `Serializable` 对象，看看是否实现了它自己的 `writeObject()`。如果是这样，就跳过正常的序列化过程并调用它的 `writeObject()`。`readObject()` 的情形与此相同。

还有另外一个技巧。在我们的 `writeObject()` 内部，可以调用 `defaultWriteObject()` 来选择执行缺省的 `writeObject()`。类似地，在 `readObject()` 内部，我们可以调用 `defaultReadObject()`。下面这个简单的例子演示了如何对一个 `Serializable` 对象的存储与恢复进行控制：

```
//: c12: SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods.
import com.bruceeckel.simpletest.*;
import java.io.*;
```

```

public class SerialCtl implements Serializable {
    private static Test monitor = new Test();
    private String a;
    private transient String b;
    public SerialCtl(String aa, String bb) {
        a = "Not Transient: " + aa;
        b = "Transient: " + bb;
    }
    public String toString() { return a + "\n" + b; }
    private void writeObject(ObjectOutputStream stream)
    throws IOException {
        stream.defaultWriteObject();
        stream.writeObject(b);
    }
    private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException {
        stream.defaultReadObject();
        b = (String)stream.readObject();
    }
    public static void main(String[] args)
    throws IOException, ClassNotFoundException {
        SerialCtl sc = new SerialCtl("Test1", "Test2");
        System.out.println("Before:\n" + sc);
        ByteArrayOutputStream buf= new ByteArrayOutputStream();
        ObjectOutputStream o = new ObjectOutputStream(buf);
        o.writeObject(sc);
        // Now get it back:
        ObjectInputStream in = new ObjectInputStream(
            new ByteArrayInputStream(buf.toByteArray()));
        SerialCtl sc2 = (SerialCtl)in.readObject();
        System.out.println("After:\n" + sc2);
        monitor.expect(new String[] {
            "Before:",
            "Not Transient: Test1",
            "Transient: Test2",
            "After:",
            "Not Transient: Test1",
            "Transient: Test2"
        });
    }
} ///:~

```


在这个例子中，有一个 `String` 域是普通域，而另一个是 `transient`（瞬时）域，用来证明非 `transient` 域由 `defaultWriteObject()` 方法保存，而 `transient` 域必须在程序中明确保存和恢复。域是在构造器内部而不是在定义处进行初始化的，以此可以证实它们在反序列化期间没有被一些自动机制初始化。

如果我们打算使用缺省机制写入对象的非 `transient` 部分，那么我们必须调用 `defaultWriteObject()` 作为 `writeObject()` 中的第一个操作，并让 `defaultReadObject()` 作为 `readObject()` 中的第一个操作。这些都是奇怪的方法调用。例如，如果我们正在为 `ObjectOutputStream` 调用 `defaultWriteObject()` 并且没有传递任何参数，然而不知何故它却可以运行，并且知道对象的引用以及如何写入非 `transient` 部分。奇怪之极。

对 `Transient` 对象的存储和恢复使用到了我们比较熟悉的代码。请再考虑一下在这里发生了什么事情。在 `main()` 中，创建 `SerialCtl` 对象，然后将其序列化到 `ObjectOutputStream` 里（注意在这种情况下，使用的是缓冲区而不是文件——这对于 `ObjectOutputStream` 来说则是完全一样的）。序列化发生在下面这行代码当中：

```
o.writeObject(sc);
```

`writeObject()` 方法必须检查 `sc`，判断它是否拥有自己的 `writeObject()` 方法（不是检查接口——这里根本就没有接口，也不是检查类的类型，而是利用反射来真正地搜索方法）。如果有，那么就会使用它。对 `readObject()` 也采用了类似的方法。或许这是解决这个问题唯一切实可行的方法，但它确实有点古怪。

版本

有时你可能想要改变可序列化类的版本（比如源类的对象可能保存在数据库中）。虽然 Java 支持这种作法，但是你可能只在特殊的情况下才这样做，此外，还需要对它进行相当深程度的了解，在这里我们就不再试图达到这一点。从 java.sun.com 处下载的 JDK 文档，对这一主题进行了非常彻底的论述。

我们会发现在 JDK 文档中有许多注解是从下面的文字开始的：

警告： 该类的序列化对象和未来的 *Swing* 版本不兼容。当前对序列化的支持只适用于短期存储或应用之间的 RMI。

这是因为 Java 的版本机制过于简单而不能在任何场合都可靠运转，尤其是对 `JavaBean` 更是如此。他们正在设法修正这一设计，也就是警告中的相关部分。

使用“持久性”

一个比较诱人的使用序列化技术的想法是：存储程序的一些状态，以便我们随后可以很容易地将程序恢复到当前状态。但是在我们能够这样做之前，必须回答几个问题。如果我们将两个都具有指向第三个对象的引用的对象进行序列化，会发生什么情况？当我们将它们的序列

化状态恢复这两个对象时，第三个对象会只出现一次吗？如果将这两个对象序列化成独立的文件，然后在代码的不同部分对它们进行反序列化，又会怎样呢？

下面这个例子展示说明上述问题：

```
//: c12:MyWorld.java

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

class House implements Serializable {}

class Animal implements Serializable {
    private String name;
    private House preferredHouse;
    Animal(String nm, House h) {
        name = nm;
        preferredHouse = h;
    }
    public String toString() {
        return name + "[" + super.toString() +
            "], " + preferredHouse + "\n";
    }
}

public class MyWorld {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        House house = new House();
        List animals = new ArrayList();
        animals.add(new Animal("Bosco the dog", house));
        animals.add(new Animal("Ralph the hamster", house));
        animals.add(new Animal("Fronk the cat", house));
        System.out.println("animals: " + animals);
        ByteArrayOutputStream buf1 =
            new ByteArrayOutputStream();
        ObjectOutputStream o1 = new ObjectOutputStream(buf1);
        o1.writeObject(animals);
        o1.writeObject(animals); // Write a 2nd set
        // Write to a different stream:
        ByteArrayOutputStream buf2 =
            new ByteArrayOutputStream();
        ObjectOutputStream o2 = new ObjectOutputStream(buf2);
        o2.writeObject(animals);
        // Now get them back:
```

```

        ObjectInputStream in1 = new ObjectInputStream(
            new ByteArrayInputStream(buf1.toByteArray()));
        ObjectInputStream in2 = new ObjectInputStream(
            new ByteArrayInputStream(buf2.toByteArray()));
        List
            animals1 = (List)in1.readObject(),
            animals2 = (List)in1.readObject(),
            animals3 = (List)in2.readObject();
        System.out.println("animals1: " + animals1);
        System.out.println("animals2: " + animals2);
        System.out.println("animals3: " + animals3);
    }
} ///: ~

```

这里有一件有趣的事：我们可以通过一个字节数组来使用对象序列化，从而实现对任何可 `Serializable` 对象的“深度复制（deep copy）”（深度复制意味着我们复制的是整个对象网，而不仅仅是基本对象及其引用）。复制对象将在附录 A 中进行深入地探讨。

在这个例子中，`Animal` 对象包含有类型为 `House` 的域。在 `main()` 方法中，创建了一个 `Animal` 列表并将其两次数序列化，分别送至不同的流。当其被反序列化并被打印时，我们可以看到执行某次运行后的结果如下（每次运行时，对象将会处在不同的内存地址）：

```

animals: [Bosco the dog[Animal@1cde100], House@16f0472
, Ralph the hamster[Animal@18d107f], House@16f0472
, Fronk the cat[Animal@360be0], House@16f0472
]
animals1: [Bosco the dog[Animal@e86da0], House@1754ad2
, Ralph the hamster[Animal@1833955], House@1754ad2
, Fronk the cat[Animal@291aff], House@1754ad2
]
animals2: [Bosco the dog[Animal@e86da0], House@1754ad2
, Ralph the hamster[Animal@1833955], House@1754ad2
, Fronk the cat[Animal@291aff], House@1754ad2
]
animals3: [Bosco the dog[Animal@ab95e6], House@fe64b9
, Ralph the hamster[Animal@186db54], House@fe64b9
, Fronk the cat[Animal@a97b0b], House@fe64b9
]

```

当然我们期望这些被反序列化的对象地址与原来的地址不同。但请注意，在 `animals1` 和 `animals2` 中却出现了相同的地址，包括二者共享的那个指向 `house` 的引用。另一方面，当恢复 `animals3` 时，系统无法知道另一个流内的对象是第一个流内对象的别名，因此它会产生出完全不同的对象网。

只要我们将任何对象序列化到一个单一流中，我们就可以恢复出与我们写出时一样的对象网，并且没有任何意外重复复制出的对象。当然，我们可以在写出第一个对象和写出最后一个对象期间改变这些对象的状态，但是这是我们自己的责任；无论在对象被序列化时处于什么状态（无论它们和其他对象有什么样的连接关系），它们都可以被写出。

如果我们想保存系统状态，最安全的作法是将其作为“原子”操作进行序列化。如果我们序列化了某些东西，再去做其他一些工作，再来序列化更多的东西，如此等等，那么我们将无法安全地保存系统状态。取而代之的是，将构成系统状态的所有对象都置入某个单一容器内，并在一个操作中将该容器直接写出。然后同样只需一次方法调用，即可以将其恢复。

下面这个例子是一个想象的计算机辅助设计（CAD）系统，用来演示这一方法。此外，它还引入了 `static` 域的问题；如果我们查看 JDK 文档，就会发现 `Class` 是“`Serializable`”的，因此只需直接对 `Class` 对象序列化，就可以很容易地保存 `static` 域。在任何情况下，这都是一种明智的做法。

```
//: c12:CADState.java
// Saving and restoring the state of a pretend CAD system.
// {Clean: CADState.out}
//package c12;
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
    public static final int RED = 1, BLUE = 2, GREEN = 3;
    private int xPos, yPos, dimension;
    private static Random r = new Random();
    private static int counter = 0;
    public abstract void setColor(int newColor);
    public abstract int getColor();
    public Shape(int xVal, int yVal, int dim) {
        xPos = xVal;
        yPos = yVal;
        dimension = dim;
    }
    public String toString() {
        return getClass() +
            "color[" + getColor() + "] xPos[" + xPos +
            "]" yPos[" + yPos + "]" dim[" + dimension + "]\n";
    }
    public static Shape randomFactory() {
        int xVal = r.nextInt(100);
        int yVal = r.nextInt(100);
        int dim = r.nextInt(100);
        switch(counter++ % 3) {
            default:
```

```

        case 0: return new Circle(xVal, yVal, dim);
        case 1: return new Square(xVal, yVal, dim);
        case 2: return new Line(xVal, yVal, dim);
    }
}
}

```

```

class Circle extends Shape {
    private static int color = RED;
    public Circle(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

```

```

class Square extends Shape {
    private static int color;
    public Square(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
        color = RED;
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

```

```

class Line extends Shape {
    private static int color = RED;
    public static void
    serializeStaticState(ObjectOutputStream os)
    throws IOException { os.writeInt(color); }
    public static void
    deserializeStaticState(ObjectInputStream os)
    throws IOException { color = os.readInt(); }
    public Line(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

```

```

public class CADState {
    public static void main(String[] args) throws Exception {
        List shapeTypes, shapes;
    }
}

```

```

if(args.length == 0) {
    shapeTypes = new ArrayList();
    shapes = new ArrayList();
    // Add references to the class objects:
    shapeTypes.add(Circle.class);
    shapeTypes.add(Square.class);
    shapeTypes.add(Line.class);
    // Make some shapes:
    for(int i = 0; i < 10; i++)
        shapes.add(Shape.randomFactory());
    // Set all the static colors to GREEN:
    for(int i = 0; i < 10; i++)
        ((Shape)shapes.get(i)).setColor(Shape.GREEN);
    // Save the state vector:
    ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("CADState.out"));
    out.writeObject(shapeTypes);
    Line.serializeStaticState(out);
    out.writeObject(shapes);
} else { // There's a command-line argument
    ObjectInputStream in = new ObjectInputStream(
        new FileInputStream(args[0]));
    // Read in the same order they were written:
    shapeTypes = (List)in.readObject();
    Line.deserializeStaticState(in);
    shapes = (List)in.readObject();
}
// Display the shapes:
System.out.println(shapes);
}
} ///:~

```

Shape 类实现了 `Serializable`，所以任何自 Shape 继承的类也都会自动地是 `Serializable`。每个 Shape 都含有数据，而且每个衍生自 Shape 的类都包含一个 `static` 域，用来确定所有那些 Shape 类型的颜色（如果将 `static` 域置入基类，只会产生一个域，因为 `static` 域不能在衍生类中复制）。可对基类中的方法进行重载，以便为不同的类型设置颜色（`static` 方法不会动态绑定，所以这些都是普通的方法）。每次调用 `randomFactory()` 方法时，它都会使用不同的随机数作为 Shape 的数据，从而创建不同的 Shape。

Circle 和 Square 是 Shape 的直接扩展；唯一的差别是 Circle 是在定义时初始化 `color` 的，而 Square 是在构造器中初始化 `color` 的。我们将 Line 留到稍后再讨论。

在 `main()` 中，一个 `ArrayList` 用于容纳 Class 对象，而另一个用于容纳几何形状。如果我们不提供命令行参数，就会创建 `shapeTypes ArrayList`，并添加 Class 对象。然后创建 `shapes ArrayList`，并添加 Shape 对象。接下来，所有的 `static color` 值都会被设置成 GREEN，而且所有东西都会序列化到文件 `CADState.out` 中。

如果我们供了一个命令行参数（假设为 CADState.out），便会打开那个文件，并用它恢复程序的状态。无论是哪种情况，作为结果产生的 Shape 的 ArrayList 都会被打印出来。某次运行的结果如下：

```
$ java CADState

[class Circlecolor[3] xPos[71] yPos[82] dim[44]
, class Squarecolor[3] xPos[98] yPos[21] dim[49]
, class Linecolor[3] xPos[16] yPos[80] dim[37]
, class Circlecolor[3] xPos[51] yPos[74] dim[7]
, class Squarecolor[3] xPos[7] yPos[78] dim[98]
, class Linecolor[3] xPos[38] yPos[79] dim[93]
, class Circlecolor[3] xPos[84] yPos[12] dim[62]
, class Squarecolor[3] xPos[16] yPos[51] dim[94]
, class Linecolor[3] xPos[51] yPos[0] dim[73]
, class Circlecolor[3] xPos[47] yPos[6] dim[49]
]
```

我们可以看到，xPos，yPos 以及 dim 的值都被成功地保存和恢复了，但是对 static 信息的读取却出现了问题。所有读回的颜色都是“3”，但是真实情况却并非如此。Circle 的值为 1（定义为 RED），而 Square 的值为 0（记住，它们是在构造器中被初始化的）。看上去似乎 static 的域根本没有被序列化！确实如此——尽管 Class 类是 Serializable 的，但它却不能按我们所期望的去运行。所以假如想序列化 static 值，你必须自己动手去实现。

这正是 Line 中的 serializeStaticState() 和 deserializeStaticState() 两个 static 方法的用途。我们可以看到，它们是作为存储和读取过程的一部分被显式地调用的。（注意必须维护写入序列化文件和从该文件中读回的顺序）。因此，为了使 CADState.java 正确运转起来，我们必须这样：

1. 为几何形状添加 serializeStaticState() 和 deserializeStaticState()。
2. 移除 ArrayList shapeTypes 以及与之有关的所有代码。
3. 在几何形状内添加对新的序列化和反序列化静态方法的调用。

另一个要注意的问题是安全，因为序列化也会将 private 数据保存下来。如果我们有安全问题，那么应将其标记成 transient。但是这之后，我们还必须要设计一种安全的保存信息的方法，以便在执行恢复时，我们可以复位那些 private 变量。

Preferences

JDK1.4 引入了 Preferences API，它比对象序列化更接近于持久化，因为它可以自动存储和读取信息。不过，它只能用于小的受限的数据集合——我们只能存储原始类型和字符串，并且每个字符串的存储长度不能超过 8K（不是很小，但我们也并不想用它来创建任何重要的东西）。正如其名，Preferences API 用于存储和读取用户的 Preferences 以及程序配置项的设置。

Preferences 是一个键-值集合（类似映射），存储在一个节点层次结构中。尽管节点层次结构可用来创建更为复杂的结构，但它通常是创建以你的类的名字命名的单一节点，然后将信息存储于其中。下面是一个简单的例子：

```
//: c12:PreferencesDemo.java
import java.util.prefs.*;
import java.util.*;

public class PreferencesDemo {
    public static void main(String[] args) throws Exception {
        Preferences prefs = Preferences
            .userNodeForPackage(PreferencesDemo.class);
        prefs.put("Location", "Oz");
        prefs.put("Footwear", "Ruby Slippers");
        prefs.putInt("Companions", 4);
        prefs.putBoolean("Are there witches?", true);
        int usageCount = prefs.getInt("UsageCount", 0);
        usageCount++;
        prefs.putInt("UsageCount", usageCount);
        Iterator it = Arrays.asList(prefs.keys()).iterator();
        while(it.hasNext()) {
            String key = it.next().toString();
            System.out.println(key + ": " + prefs.get(key, null));
        }
        // You must always provide a default value:
        System.out.println(
            "How many companions does Dorothy have? " +
            prefs.getInt("Companions", 0));
    }
} ///: ~
```

这里用的是 `userNodeForPackage()`，但我们也可以选择用 `systemNodeForPackage()`；虽然可以任意选择，但最好将“user”用于单个用户的 preferences，将“system”用于通用的安装配置。既然 `main()` 是静态的，因此 `PreferencesDemo.class` 可以用来标识节点，但是在非静态方法内部，我们通常使用 `getClass()`。我们不一定非要把当前的类作为节点标志符，但这仍不失为一种很有用的方法。

一旦我们创建了节点，就可以用它来加载或者读取数据了。在这个例子中，向节点载入了各种不同类型的数据项，然后获取其 `keys()`。它们是以 `String[]` 的形式返回的，如果你习惯于 `keys()` 在集合类库里面，那么这个返回结果可能并不是你所期望的。现在，把它们转化成一个列表，用来产生一个迭代器，从而打印出关键字和值。注意 `get()` 的第二个参数，如果某个关键字下没有任何条目，那么这个参数就是所产生的缺省值。当在一个关键字集合内迭代时，我们总要确信条目是存在的，因此用 `null` 作为缺省值是安全的，但是通常我们会获得一个具名的关键字，就像下面这条语句：


```
prefs.getInt("Companions", 0));
```

在通常情况下，我们希望提供一个合理的缺省值。实际上，典型的习惯用法可见下面几行：

```
int usageCount = prefs.getInt("UsageCount", 0);
usageCount++;
prefs.putInt("UsageCount", usageCount);
```

这样，在我们第一次运行程序时，UsageCount 的值是 0，但在随后引用中，它将会是非零值。

在我们运行 PreferencesDemo.java 时，会发现每次运行程序时，UsageCount 的值都会增加 1，但是数据存储到哪里了呢？在程序第一次运行之后，并没有出现任何本地文件。Preferences API 利用合适的系统资源完成了这个任务，并且这些资源会随操作系统而不同。例如在 windows 里，就使用注册表（因为它已经有“key-value”这样的节点对层次结构了）。但是最重要的一点是，它已经神奇般地为我们存储了信息，所以我们不必担心不同的操作系统是怎么运作的。

还有更多的 Preferences API，请参阅 JDK 文档，很容易理解更深的细节。

正则表达式

在本章最后，让我们看看正则表达式，它是在JDK1.4版本中新引入的内容，而且与标准的Unix实用工具，像sed和awk，以及程序语言像Python 和 Perl（一些人认为这是Perl成功的关键因素）完整一致。在技术上，它们是字符串操纵工具（先前这些任务代理给了Java中的String、StringBuffer和StringTokenizer这些类），但它们通常和I/O联合使用，所以这部分内容放在本章也就不是很牵强的了⁶。

正则表达式是强大而灵活的文本处理工具。它们可以让我们以编程方式指定那些可以在输入字符串中发现的复杂的文本模式。一旦我们发现了这些模式，那么我们就可以按照任何我们所希望的方式进行了。尽管正则表达式的语法一开始令人感到发怵，但是它们提供了一种紧凑的、动态的语言，能够以一种完全通用的方式来解决各种字符串处理，例如：匹配、选择、编辑及验证问题。

创建正则表达式

我们先在可以用正则表达式来构造的集合中选取一个很有用的子集，以此开始学习正则表达式。用于创建正则表达式的完全构造列表可以在 javadocs 的模式类的包 java.util.regex 中找到。

Characters	
B	字符 B
\xhh	具有 16 进制值 0xhh 的字符

⁶ 在第 4 版本中，会添加一章专门讲述字符串。由Mike Shea撰稿编写。

\uhhhh	用 16 进制 0xhhhh 表示的 Unicode 字符
\t	制表符
\n	换行符
\r	回车
\f	换页
\e	Escape

正则表达式的威力在定义字符类的时候开始显现。下面是一些创建字符类的典型方式以及一些预定义的类。

Character Classes	
.	表示任何字符
[abc]	包含 a、b、c 的任何字符(和 a b c 相同)
[^abc]	除了 a, b, c 之外的任何字符(否定)
[a-zA-Z]	任何从 a 到 z 或从 A 到 Z 的任何字符 (范围)
[abc(hij)]	任意 a,b,c,h,i,j 字符 (与 a b c h i j 相同) (合并)
[a-z&&(hij)]	任意 h, i, 或 j (交)
\s	whitespace 符 (空格, tab, 换行, 换页, 回车)
\S	非 whitespace 符([^\s])
\d	数字 [0-9]
\D	非数字 [^0-9]
\w	word character [a-zA-Z_0-9]
\W	非 word character [^\w]

如果你有处理其他语言中正则表达式的经验,那么你就会很快注意到二者处理反斜杠的方式有点不同。在其他语言中,“\\”意味着“我想在一个正则表达式中插入一个无格式的字面意义上的反斜杠,不要给它赋予任何特殊意义。”在 Java 中,“\\”意味着“我正在插入一个正则表达式反斜杠,那么随后的字符具有特殊意义。”例如,我们想指示一个或多个字符,我们的正则表达式串仍旧是“\\w+”这种形式。如果我们想插入一个字面意义上的反斜杠,我们得这样表示“\\\\”。然而,像换行符和制表符这样的字符只需要用到一个反斜杠:“\n\t”。

这里显示的仅是一个样板;你可以对 `java.util.regex.PatternJDK` 文档的页面设置书签或者是添加到“开始”菜单中,这样你就可以很容易地访问到所有可能的正则表达式模式了。

Logical Operators	
XY	XY
X Y	X 或 Y

(X)	<i>capturing group</i> . 随后，我们可以用表达式\i 引用第 i 个 captured group
-----	---

Boundary Matchers	
^	一行的开始
\$	一行的结束
\b	词界
\B	非词界
\G	上一级结尾

作为一个例子，下面的每一个表达式都是合法的正则表达式，并且所有表达式的都会成功匹配字符序列“Rudolph”。

```
Rudolph
[rR]udolph
[rR][aeiou][a-z]ol.*
R.*
```

量词

量词描述了一个模式吸收输入文本的方式：

- 贪婪的：量词总是贪婪的，除非有其他的选项被设置。贪婪表达式会为所有可能的模式发现尽可能多的匹配。导致此问题的一个典型理由就是假定我们的模式仅能匹配第一个可能的字符组，如果它确实是贪婪的，那么它就会继续往下匹配。
- 勉强的：用问号来指定，这个量词匹配满足模式所需的最少字符数。因此也称作懒惰的、最少匹配、非贪婪的或不贪婪的（*lazy, minimal matching, non-greedy, or ungreedy*）。
- 占有的：量词当前只有在 java 语言中才可用（在其他语言中不可用），并且它也更高级，因此我们大概不会立刻用到它。当正则表达式被应用于字符串时，它会产生相当多的状态以便在匹配失败时可以回溯。而占有量词并不保存这些中间状态，因此它们可以防止回溯。它们常常用于防止正则表达式失控，因此可以使正则表达式执行起来显得更有效。

Greedy	Reluctant	Possessive	Matches
X?	X??	X?+	X, 一或无
X*	X*?	X*+	X, 0 或多个
X+	X+?	X++	X, 1 或多个

X{n}	X{n}?	X{n}+	X, 恰好 n 次
X{n,}	X{n,}?	X{n,}+	X, 至少 n 次
X{n,m}	X{n,m}?	X{n,m}+	X, 至少 n 次不多于 m 次

我们应该非常清楚地意识到表达式‘X’通常必须要用圆括号括起来，以便它能够按照我们期望的效果去执行。例如：

```
abc+
```

看起来它似乎应该匹配一个或多个‘abc’序列，如果我们把它应用于输入字符串‘abcabcabc’，我们实际上会获得 3 个匹配。然而，这个表达式实际上表示的是：匹配‘ab’，后面跟随一个或多个‘c’。为了匹配一个或多个完整的‘abc’字符串，我们必须这样表示：

```
(abc)+
```

你会发现在使用正则表达式时，很容易混淆。因为它是一种在 Java 之上的新语言。

字符序列

JDK 1.4 定义了一个新接口 CharSequence，它建立了从 String 或 StringBuffer 类中抽象出一个字符序列的定义：

```
interface CharSequence {
    charAt(int i);
    length();
    subSequence(int start, int end);
    toString();
}
```

String、StringBuffer 和 CharBuffer 类都已经修改过以实现新的 CharSequence 接口。许多正则表达式操作都接受 CharSequence 类型的参数。

模式和匹配器

作为第一个示例，下述类可以用来测试一个输入字符串是否违反了正则表达式。第一个参数是输入的用于匹配的字符串，随后的一个或多个正则表达式将会被应用于输入字符串。在 unix/linux 环境下，正则表达式必须以命令行的形式进行引用。

当我们创建正则表达式并查看它们是否产生我们所预期的行为时，下面的程序有助于我们对这些正则表达式进行测试。

```
//: c12:TestRegularExpression.java
// Allows you to easily try out regular expressions.
```

```
// {Args: abcabcabcddefabc "abc+" "(abc)+" "(abc){2,}" }
import java.util.regex.*;

public class TestRegularExpression {
    public static void main(String[] args) {
        if(args.length < 2) {
            System.out.println("Usage: \n" +
                "java TestRegularExpression " +
                "characterSequence regularExpression+");
            System.exit(0);
        }
        System.out.println("Input: \"" + args[0] + "\"");
        for(int i = 1; i < args.length; i++) {
            System.out.println(
                "Regular expression: \"" + args[i] + "\"");
            Pattern p = Pattern.compile(args[i]);
            Matcher m = p.matcher(args[0]);
            while(m.find()) {
                System.out.println("Match \"" + m.group() +
                    "\" at positions " +
                    m.start() + "-" + (m.end() - 1));
            }
        }
    }
} ///:~
```

在 java 中，正则表达式是通过 java.util.regex 包里面的 Pattern 和 Matcher 类来实现的。一个 Pattern 对象表示一个正则表达式的编译版本。静态的 compile（）方法将一个正则表达式字符串编译成 Pattern 对象。正如在前面例子中所看到的那样，我们可以使用 matcher（）方法和输入字符串从编译过的 Pattern 对象中产生 Matcher 对象。Pattern 还有一个

```
static boolean ( regex, input)
```

用于快速识别，如果可以在 input 中发现 regex 的话；以及一个 split（）方法，它可以产生一个按照 regex 的匹配而断开的字符串数组。

调用以输入字符串作为参数的 Pattern.matcher（）可以产生 Matcher 对象。接着可以使用 Matcher 对象访问结果，并利用一些方法来判断各种类型的匹配是成功或是失败：

```
boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)
```

如果模式匹配整个输入字符串，则 `matches()` 方法成功，而且，如果输入字符串从一开始就是模式的一个匹配，那么 `lookingAt()` 方法也是成功的。

find()

`Matcher.find()` 用于发现应用于 `CharSequence` 的多重模式匹配。例如：

```
//: c12:FindDemo.java
import java.util.regex.*;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class FindDemo {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Matcher m = Pattern.compile("\\w+")
            .matcher("Evening is full of the linnet's wings");
        while(m.find())
            System.out.println(m.group());
        int i = 0;
        while(m.find(i)) {
            System.out.print(m.group() + " ");
            i++;
        }
        monitor.expect(new String[] {
            "Evening",
            "is",
            "full",
            "of",
            "the",
            "linnet",
            "s",
            "wings",
            "Evening vening ening ning ing ng g is is s full " +
            "full ull ll l of of f the the he e linnet linnet " +
            "innet nnet net et t s s wings wings ings ngs gs s "
        });
    }
} ///: ~
```

模式“`\\w+`”表示“一个到多个单词字符”，因此，它直接将输入断开成单词。`find()`就像是一种迭代器，可以在输入字符串中向前移动迭代。不过在 `find()` 的第二版本中，可以给出一个整型参数，告知字符的所在位置以便开始进行搜索——这个版本可以将搜索位置重新设置为参数值，正如我们在输出中所看到的一样。

组

组是由圆括号分开的正则表达式，随后可以根据它们的组号进行调用。第 0 组表示整个匹配表达式，第 1 组表示第 1 个用圆括号括起来的组，等等。因此，在表达式

`A(B(C))D`

中，有 3 个组：第 0 组 `ABCD`，第 1 组是 `BC` 以及第 2 组 `C`。

`Matcher` 对象有一些方法可以向我们提供有关组的信息：

`public int groupCount()` 返回本匹配器的模式中分组的数目。第 0 组不包括在内。

`public String group()` 返回前一次匹配操作（例如：`find()`）的第 0 组（整个匹配）。

`public String group(int i)` 返回在前一次匹配操作期间指定的组。如果匹配成功，但是指定的组没有匹配输入字符串的任何部分，将返回 `null`。

`public int start(int group)` 返回在前一次匹配操作中找到的组的起始下标。

`public int end(int group)` 返回在前一次匹配操作中找到的组的最后一个字符下标加一的值。

下面是正则表达式组的例子：

```
//: c12:Groups.java
import java.util.regex.*;
import com.bruceeckel.simpletest.*;

public class Groups {
    private static Test monitor = new Test();
    static public final String poem =
        "Twas brillig, and the slithy toves\n" +
        "Did gyre and gimble in the wabe.\n" +
        "All mimsy were the borogoves,\n" +
        "And the mome raths outgrabe.\n\n" +
        "Beware the Jabberwock, my son,\n" +
        "The jaws that bite, the claws that catch.\n" +
        "Beware the Jubjub bird, and shun\n" +
        "The frumious Bandersnatch.";
    public static void main(String[] args) {
        Matcher m =
            Pattern.compile("(?m)(\\S+)\\s+(\\S+)\\s+(\\S+))$")
                .matcher(poem);
```

```

while(m.find()) {
    for(int j = 0; j <= m.groupCount(); j++)
        System.out.print "[" + m.group(j) + ""];
    System.out.println();
}
monitor.expect(new String[]{
    "[the slithy toves]" +
    "[the][slithy toves][slithy][toves]",
    "[in the wabe.][in][the wabe.][the][wabe.]",
    "[were the borogoves,]" +
    "[were][the borogoves,][the][borogoves,]",
    "[mome raths outgrabe.]" +
    "[mome][raths outgrabe.][raths][outgrabe.]",
    "[Jabberwock, my son,]" +
    "[Jabberwock,][my son,][my][son,]",
    "[claws that catch.]" +
    "[claws][that catch.][that][catch.]",
    "[bird, and shun][bird,][and shun][and][shun]",
    "[The frumious Bandersnatch.][The]" +
    "[frumious Bandersnatch.][frumious][Bandersnatch.]"
});
}
} ///: ~

```

这首诗来自于 Lewis Carroll 的 *Through the Looking Glass* 中的“Jabberwocky”。我们可以看到这个正则表达式模式有许多圆括号分组，由任意数目的非空格字符（‘\S+’）伴随任意数目的空格字符（‘\s+’）所组成。目的是捕获每行的最后 3 个词；每行最后以‘\$’结束。不过，在正常情况下，将‘\$’与整个输入序列的末端相匹配。所以我们一定要显式地告知正则表达式注意输入序列中的换行符。这可以由序列开头的模式标记‘(?m)’来完成（模式标记马上就会介绍）。

start() 和 end()

在匹配操作成功之后，start() 返回先前匹配的起始位置的下标，而 end() 返回所匹配的最后字符的下标加一的值。在匹配操作失败之后，（或先于一个正在进行的匹配操作去尝试）调用 start() 或 end()，将会产生 IllegalStateException。下面的示例还同时展示了 matches() 和 lookingAt() 的用法：

```

//: c12:StartEnd.java
import java.util.regex.*;
import com.bruceeckel.simpletest.*;

public class StartEnd {
    private static Test monitor = new Test();

```



```

public static void main(String[] args) {
    String[] input = new String[] {
        "Java has regular expressions in 1.4",
        "regular expressions now expressing in Java",
        "Java represses oracular expressions"
    };
    Pattern
        p1 = Pattern.compile("re\\w*"),
        p2 = Pattern.compile("Java.*");
    for(int i = 0; i < input.length; i++) {
        System.out.println("input " + i + ": " + input[i]);
        Matcher
            m1 = p1.matcher(input[i]),
            m2 = p2.matcher(input[i]);
        while(m1.find())
            System.out.println("m1.find() '" + m1.group() +
                "' start = " + m1.start() + " end = " + m1.end());
        while(m2.find())
            System.out.println("m2.find() '" + m2.group() +
                "' start = " + m2.start() + " end = " + m2.end());
        if(m1.lookingAt()) // No reset() necessary
            System.out.println("m1.lookingAt() start = "
                + m1.start() + " end = " + m1.end());
        if(m2.lookingAt())
            System.out.println("m2.lookingAt() start = "
                + m2.start() + " end = " + m2.end());
        if(m1.matches()) // No reset() necessary
            System.out.println("m1.matches() start = "
                + m1.start() + " end = " + m1.end());
        if(m2.matches())
            System.out.println("m2.matches() start = "
                + m2.start() + " end = " + m2.end());
    }
    monitor.expect(new String[] {
        "input 0: Java has regular expressions in 1.4",
        "m1.find() 'regular' start = 9 end = 16",
        "m1.find() 'ressions' start = 20 end = 28",
        "m2.find() 'Java has regular expressions in 1.4'" +
            " start = 0 end = 35",
        "m2.lookingAt() start = 0 end = 35",
        "m2.matches() start = 0 end = 35",
        "input 1: regular expressions now " +
            "expressing in Java",
        "m1.find() 'regular' start = 0 end = 7",
    });
}

```

```

        "m1.find() 'ressions' start = 11 end = 19",
        "m1.find() 'ressing' start = 27 end = 34",
        "m2.find() 'Java' start = 38 end = 42",
        "m1-lookingAt() start = 0 end = 7",
        "input 2: Java represses oracular expressions",
        "m1.find() 'represses' start = 5 end = 14",
        "m1.find() 'ressions' start = 27 end = 35",
        "m2.find() 'Java represses oracular expressions' " +
        "start = 0 end = 35",
        "m2-lookingAt() start = 0 end = 35",
        "m2.matches() start = 0 end = 35"
    });
}
} ///:~

```

注意 `find()` 可以输入的任意位置去定位正则表达式，而 `lookingAt()` 和 `matches()` 只有在正则表达式与输入的最开始处就开始匹配时，才会成功。`matches()` 只有在整个输入都匹配正则表达式时才会成功，而 `lookingAt()` 只要输入的第一部分匹配就会成功⁷。

模式标记

另一可供选择的 `compile()` 方法接受影响正则表达式的匹配行为的标记作为其参数：

```
Pattern Pattern.compile(String regex, int flag)
```

其中，标记来自于下面 `Pattern` 类中的常量：

Compile Flag	Effect
Pattern.CANON_EQ	两个字符当且仅当它们的完全规范分解相匹配时，就认为它们是匹配的。例如，如果我们指定这个标记，表达式 <code>"a\u030A"</code> 就会匹配字符串 <code>"?"</code> 。缺省情况下，匹配不考虑规范的等价性。
Pattern.CASE_INSENSITIVE (?i)	缺省情况下，大小写不敏感的匹配假定只有在 <code>US-ASCII</code> 字符集中的字符才能进行。这个标记允许我们的模式匹配不必考虑大小写（大写或小写）。通过指定 <code>UNICODE_CASE</code> 标记并与此标记结合起来，基于 <code>Unicode</code> 大小写不敏感的匹配就可以使能。

⁷ 我真不清楚他们是怎么提出这些方法名的，或者说他们参考了什么。但是可以确定一点，那些提出了如此不直观的方法名的人仍在 `Sun` 公司任职。并且他们那种很显然没有考虑代码设计的方针仍占有一席之地。很抱歉说了这些风凉话，但是几年后这种事就会变得非常令人厌倦。

Pattern.COMMENTS (?x)	在这种模式下，空格符将被忽略掉，并且以#开始直到行末的注释也会被忽略掉。通过嵌入的标记表达式也可以使能 Unix 的行模式
Pattern.DOTALL (?s)	在 <code>dotall</code> 模式下，表达式 <code>.</code> 匹配所有字符，包括行终结符。缺省情况下， <code>.</code> 表达式不匹配行终结符。
Pattern.MULTILINE (?m)	在 多行模式下，表达式 <code>^</code> 和 <code>\$</code> 分别匹配一行的开始和结束。 <code>^</code> 还匹配输入字符串的开始，而 <code>\$</code> 也还匹配输入字符串的结尾。缺省情况下，这些表达式仅匹配输入的完整字符串的开始和结束。
Pattern.UNICODE_CASE (?u)	当指定这个标记，并且使能 <code>CASE_INSENSITIVE</code> 时，大小写不敏感的匹配将按照与 Unicode 标准相一致的方式进行。缺省情况下，大小写不敏感的匹配假定只能在 US-ASCII 字符集中的字符才能匹配。
Pattern.UNIX_LINES (?d)	在这种模式下，在 <code>.</code> 、 <code>^</code> 和 <code>\$</code> 行为中，只识别行终结符 <code>\n</code> 。

在这些标记中，特别有用的是 `Pattern.CASE_INSENSITIVE`、`Pattern.MULTILINE` 和 `Pattern.COMMENTS`（有助于理清程序和文档管理）。注意大多数标记的行为也可以通过向我们的正则表达式中（在我们所期望模式产生效用的位置之前）插入加括号的字符（见表中标记下面的内容）来获得。

我们可以通过“OR”（`|`）操作，将这些标记和其他标记相结合：

```
//: c12:ReFlags.java
import java.util.regex.*;
import com.bruceeckel.simpletest.*;

public class ReFlags {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Pattern p = Pattern.compile("^java",
            Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
        Matcher m = p.matcher(
            "java has regex\nJava has regex\n" +
            "JAVA has pretty good regular expressions\n" +
            "Regular expressions are in Java");
        while(m.find())
            System.out.println(m.group());
    }
}
```

```

        monitor.expect(new String[] {
            "java",
            "Java",
            "JAVA"
        });
    }
} ///:~

```

这个程序创建了一个模式，可以匹配以 “java”、“Java”、“JAVA”等开始的行，它尝试着为一个多行集里面的每一行进行匹配（匹配起始于字符序列的首部，结束于字符序列中每一行的终结符）。注意 `group()` 方法仅产生匹配部分。

split()

Splitting 将输入字符串断开成字符串对象数组，断开边界由正则表达式确定：

```

String[] split(CharSequence charseq)
String[] split(CharSequence charseq, int limit)

```

这是一个快速而方便的方法，可以按照通用边界断开输入文本。

```

//: c12:SplitDemo.java
import java.util.regex.*;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class SplitDemo {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        String input =
            "This!!unusual use!!of exclamation!!points";
        System.out.println(Arrays.asList(
            Pattern.compile("!!").split(input)));
        // Only do the first three:
        System.out.println(Arrays.asList(
            Pattern.compile("!!").split(input, 3)));
        System.out.println(Arrays.asList(
            "Aha! String has a split() built in!".split(" ")));
        monitor.expect(new String[] {
            "[This, unusual use, of exclamation, points]",
            "[This, unusual use, of exclamation!!points]",
            "[Aha!, String, has, a, split(), built, in!]"
        });
    }
}

```

```
} ///: ~
```

第二种形式的 `split()` 限制了发生分裂 (`split`) 的数目。

注意正则表达式如此有价值，以至于某些操作也可以添加到 `String` 类中，包括 `split()` (就像这里展示的)、`matches()`、`replaceFirst()` 和 `replaceAll()`。这些操作运行起来就像是对它们的 `Pattern` 和 `Matcher` 的补充一样。

替换操作

在我们开始替换文本的时候，正则表达式就变得尤为有用了。下面是一些可用的方法：

`replaceFirst(String replacement)` 用 `replacement` 替换输入字符串中最先匹配的那部分。

`replaceAll(String replacement)` 用 `replacement` 替换输入字符串中所有的匹配部分。

`appendReplacement(StringBuffer sbuf, String replacement)` 逐步地在 `sbuf` 中执行替换，而不是像 `replaceFirst()` 那样仅替换第一个匹配或者像 `replaceAll()` 是替换所有的匹配。这是个非常重要的方法，因为它允许我们通过调用某些方法并执行一些其他处理来产生 `replacement` (而 **`replaceFirst()`** 和 **`replaceAll()`** 只能输入固定字符串)。有了这个方法，我们就可以通过编程来实现将目标拆分成组以及创建功能强大的替换。

`appendTail(StringBuffer sbuf, String replacement)` 在一个或多个 `appendReplacement()` 调用之后被调用，以便复制输入字符串的剩余部分。

下面这个例子展示了所有的替换操作。另外，开始部分的注释文本块被提取了出来，并用正则表达式对其进行处理，以便将其用作该例子其他部分的输入。

```
///: c12:TheReplacements.java
import java.util.regex.*;
import java.io.*;
import com.bruceeckel.util.*;
import com.bruceeckel.simpletest.*;

/*! Here's a block of text to use as input to
   the regular expression matcher. Note that we'll
   first extract the block of text by looking for
   the special delimiters, then process the
   extracted block. !*/

public class TheReplacements {
    private static Test monitor = new Test();
    public static void main(String[] args) throws Exception {
```

```

String s = TextFile.read("TheReplacements.java");
// Match the specially-commented block of text above:
Matcher mInput =
    Pattern.compile("/\\*!(.*)!\\*/", Pattern.DOTALL)
        .matcher(s);
if(mInput.find())
    s = mInput.group(1); // Captured by parentheses
// Replace two or more spaces with a single space:
s = s.replaceAll(" {2,}", " ");
// Replace one or more spaces at the beginning of each
// line with no spaces. Must enable MULTILINE mode:
s = s.replaceAll("(?m) ^ +", "");
System.out.println(s);
s = s.replaceFirst("[aeiou]", "(VOWEL1)");
StringBuffer sbuf = new StringBuffer();
Pattern p = Pattern.compile("[aeiou]");
Matcher m = p.matcher(s);
// Process the find information as you
// perform the replacements:
while(m.find())
    m.appendReplacement(sbuf, m.group().toUpperCase());
// Put in the remainder of the text:
m.appendTail(sbuf);
System.out.println(sbuf);
monitor.expect(new String[]{
    "Here's a block of text to use as input to",
    "the regular expression matcher. Note that we'll",
    "first extract the block of text by looking for",
    "the special delimiters, then process the",
    "extracted block. ",
    "H(VOWEL1)rE's A bLOck Of tExt tO UsE As InpUt tO",
    "thE rEgUIAr ExprEssIO n mAtChEr. NOtE thAt wE'll",
    "flrst ExtrAct thE bLOck Of tExt by lOOkIng fOr",
    "thE spEcIAl dElImItErS, thEn prOcEss thE",
    "ExtrActEd bLOck. "
});
}
} ///:~

```

上面的例子打开了文件并使用本章前面介绍的 `TextFile.read()` 方法对其进行读取。创建了 `mInput` 用以匹配在 `/*!` 和 `!*/` 之间的所有文本（注意分组的圆括号）。然后，多于两个的空格被减缩成一个，并且每行开始的所有空格都被删掉（为了对所有的行而不仅是输入的开始部分都能这样处理，必须使用多行模式）。这两个替换都是用等价的 `replaceAll()` 来执

行的（在这种情形下，显得更方便）。注意既然每个替换在程序中只能被使用一次，因此这样做与把它作为一个模式进行预先编译相比，不会有任何额外的花费。

`replaceFirst()` 仅对找到的第一个匹配进行替换。另外，`replaceFirst()` 和 `replaceAll()` 中的替换字符串仅是字面意义上的，因此如果我们想在每个替换上执行一些操作，它们就不会有太大帮助。如果确实要那样的话，我们需要使用 `appendReplacement()`，它可以让我们将任意数量的代码编写进执行替换的过程。在前一个例子中，选择 `group()` 然后进行处理——在这种情形下，将正则表达式发现的元音字母设置为大写——作为正在创建的 `sbuf` 的结果。通常地，我们会逐步地执行所有替换，然后调用 `appendTail()`，但是如果我们想模仿 `replaceFirst()`（或者“替换第 *n* 个”），我们仅需要执行一次替换，然后调用 `appendTail()` 把剩余部分输入到 `sbuf` 即可。

`appendReplacement()` 也可以让我们通过用“`$g`”（其中‘*g*’是组号）的形式，来在替代字符串中直接引用被捕获的组。然而，这只适合于较为简单的处理，并且它不会在前面的程序中产生你所期望的结果。

reset ()

使用 `reset ()` 方法，可以将现有的 `Matcher` 对象应用于一个新的字符序列：

```
//: c12:Resetting.java
import java.util.regex.*;
import java.io.*;
import com.bruceeckel.simpletest.*;

public class Resetting {
    private static Test monitor = new Test();
    public static void main(String[] args) throws Exception {
        Matcher m = Pattern.compile("[frb][aiu][gx]")
            .matcher("fix the rug with bags");
        while(m.find())
            System.out.println(m.group());
        m.reset("fix the rig with rags");
        while(m.find())
            System.out.println(m.group());
        monitor.expect(new String[]{
            "fix",
            "rug",
            "bag",
            "fix",
            "rig",
            "rag"
        });
    }
}
```

```
} ///: ~
```

没有任何参数的 `reset()` 将 `Matcher` 设置到当前序列的开始部分。

正则表达式和 Java I/O

到目前为止，展示的大多数例子中，正则表达式都是应用于静态字符串的。下面的例子展示了使用正则表达式在文件中搜寻匹配的一种方法。受 Unix 的 `grep` 的启发，`JGrep.java` 接受两个参数：一个文件名及我们想要匹配的正则表达式。输出结果显示发生匹配的所有行以及在行内的匹配位置。

```
//: c12:JGrep.java
// A very simple version of the "grep" program.
// {Args: JGrep.java "\\b[Ssct]\\w+"}
import java.io.*;
import java.util.regex.*;
import java.util.*;
import com.bruceeckel.util.*;

public class JGrep {
    public static void main(String[] args) throws Exception {
        if(args.length < 2) {
            System.out.println("Usage: java JGrep file regex");
            System.exit(0);
        }
        Pattern p = Pattern.compile(args[1]);
        // Iterate through the lines of the input file:
        ListIterator it = new TextFile(args[0]).listIterator();
        while(it.hasNext()) {
            Matcher m = p.matcher((String)it.next());
            while(m.find())
                System.out.println(it.nextIndex() + ": " +
                                    m.group() + ": " + m.start());
        }
    }
} ///: ~
```

在上例中，文件被作为一个 `TextFile` 对象（在本章前面已经介绍过）打开。由于一个 `TextFile` 对象内部的 `ArrayList` 包含了文件中的行，由这个数组可以产生一个 `ListIterator`。因此，其结果是一个可以让我们在文件的行之间进行移动（向前向后）的迭代器。

每个输入行都被用来产生一个 `Matcher`，并可以用 `find()` 浏览结果。注意 `ListIterator.nextIndex()` 用于跟踪行号。

测试参数打开 JGrep.java 文件，将其读入作为输入，然后搜索以[Ssct]开始的单词。

需要 StringTokenizer 吗？

正则表达式提供的新功能可能会使我们想知道最初的 StringTokenizer 类是否仍旧重要。在 JDK1.4 之前，将字符串分离成几部分的方法是：利用 StringTokenizer 将该字符串“用标记断开”。但是现在利用正则表达式，同样的事情做起来变得更加容易和更加简洁了。

```
//: c12:ReplacingStringTokenizer.java
import java.util.regex.*;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class ReplacingStringTokenizer {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        String input = "But I'm not dead yet! I feel happy!";
        StringTokenizer stoke = new StringTokenizer(input);
        while(stoke.hasMoreElements())
            System.out.println(stoke.nextToken());
        System.out.println(Arrays.asList(input.split(" ")));
        monitor.expect(new String[] {
            "But",
            "I'm",
            "not",
            "dead",
            "yet!",
            "I",
            "feel",
            "happy!",
            "[But, I'm, not, dead, yet!, I, feel, happy!]"
        });
    }
} ///: ~
```

通过使用正则表达式，我们还可以使用更复杂的模式来断开字符串——而这用 StringTokenizer 来实现要困难得多。这样说似乎比较稳妥：正则表达式取代了先前版本中所有与“用标记断开”相关的类。

想了解更多正则表达式，请参阅 *Mastering Regular Expressions* (第二版本)，Jeffrey E. F. Friedl 著 (O'Reilly, 2002)

总结

Java I/O 流类库的确能满足我们的基本需求：我们可以通过控制台、文件、内存块甚至因特网进行读写。通过继承，我们可以创建新类型的 `input` 和 `output` 对象。并且通过重新定义 `toString()` 方法，我们甚至可以为流接受的对象类型进行简单扩充。当我们向期望收到一个字符串的方法传送一个对象时，会自动调用 `toString()` 方法。（这是 Java 有限的自动类型转换功能。）

在 I/O 流类库的文档和设计中，仍留有一些没有解决的问题。例如，当我们打开一个文件以便输出时，我们可以指定一旦试图覆盖该文件就抛出一个异常——有的编程系统允许我们自行指定想要打开的输出文件，只要它尚不存在。在 Java 中，我们似乎应该使用一个 `File` 对象来判断某个文件是否存在，因为如果我们以 `FileOutputStream` 或者 `FileWriter` 打开，那么它肯定会被覆盖。

I/O 流类库使我们喜忧参半。它确实能做许多事情，而且具有可移植性。但是如果我们没有理解修饰器模式，那么这种设计就不是很直觉，因此，在学习和传授它的过程中，需要额外的开销。而且它并不完善；例如我应该可以不必去写像 `TextFile` 这样的应用，并且它没有任何对输出格式化的种类的支持，而事实上其他所有语言的 I/O 包都提供这种支持。

然而，一旦我们理解了修饰器模式，并开始在某些情况下使用该类库以利用其能够提供的灵活性，那么你就开始从这个设计中受益了。到那个时候，为此额外多写几行代码的开销应该不至于使人觉得太麻烦。

如果你在本章没有发现想要找寻的东西（这里仅仅是一个初步介绍，其涵盖面并不全面），可以在《Java I/O》中找到更深入的论述，Elliotte Rusty Harold 著（O'Reilly, 1999）。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 打开一个文本文件，每次读取一行内容。将每行作为一个 `String` 读入，并将那个 `String` 对象置入一个 `LinkedList` 中。按相反的顺序打印出 `LinkedList` 中的所有行。
2. 修改练习 1，使要读取的文件的名字作为一个命令行参数来提供。
3. 修改练习 2，同样也打开一个文本文件，以便将文字写入其中。将 `LinkedList` 中的行随同行号一起写入文件（不要试图使用“`LineNumber`”类）。
4. 修改练习 2，强制 `LinkedList` 中的所有行都变成大写形式，将结果发给 `System.out`。
5. 修改练习 2，接受附加的命令行参数，用来表示要在文件中查找的单词。打印出包含了欲查找单词的所有文本行。
6. 修改 `DirList.java`，以便 `FilenameFilter` 能够真正打开每个文件，并检查命令行参数所表示的文件是否存在，以此检查结果来决定是否接受这个文件。
7. 修改 `DirList.java`，用以产生当前目录下和满足给定的正则表达式子目录下的所有文件名。提示：使用递归遍历子目录。

8. 创建一个叫做 `SortedDirList` 的类，具有一个可以接受文件路径信息，并能构建该路径下所有文件的排序目录列表的构造器。创建两个重载的 `list()`，根据参数产生整个列表或者列表子集。添加 `size()` 方法，能够接受文件名并产生该文件的大小。
9. 利用第 11 章的工具，修改 `WordCount.java`，使其能够按照字母序排序。
10. 修改 `WordCount.java`，使其可以使用一个包含了一个字符串和一个计数值的类来存储每个不同的单词，并且使用一个对象集合来维护这个单词列表。
11. 修改 `IOStreamDemo.java`，以便可以使用 `LineNumberReader` 来记录行数。注意继续使用编程方式地实现跟踪会更简单。
12. 从 `IOStreamDemo.java` 的第四部分开始，编写一个程序，用来比较有缓冲的和无缓冲的 I/O 方式在向文件写入时的性能差别。
13. 修改 `IOStreamDemo.java` 的第五部分，消除由首次调用 `in5.readUTF()` 时所产生的行内的空格。
14. 按照书中描述的，修改 `CADState.java`。
15. 复制 `Blips.java` 并重命名为 `BlipCheck.java`，然后将 `Blip2` 重命名为 `BlipCheck`（使其成为 `Public`，并删除类 `Blips` 中的公共作用域）。删除文件中的 `///标记，然后执行含有这几个错误行的程序。接下来，注解掉 BlipCheck 的缺省构造器。执行之并解释它可以运行的原因。注意编译后我们必须使用 java Blips 执行程序，因为 main() 方法仍在类 Blips 中。`
16. 注解掉 `Blip3.java` 中自“You must do this:”开始的两行，运行之。解释说明结果并说出该结果与这两行在程序中运行时所产生的结果不同的原因。
17. （中等难度）找到第 8 章中由 4 个文件组成的 `GreenhouseController.java` 示例。`GreenhouseController` 包含了硬编码的事件集。改变程序以便可以从文本文件中读取事件及事件的相对时间。（挑战在于：使用设计模式中的 *factory* 方法来创建事件——请参阅网站 www.BruceEckel.com 上 *Thinking in Patterns* (Java 版)）。
18. 创建并测试一个实用方法，可以打印出 `CharBuffer` 中的内容直到字符不能再打印为止。
19. 试着将本章例子中的 `ByteBuffer.allocate()` 语句改为 `ByteBuffer.allocateDirect()`。用来证实性能之间的差异，但是请注意程序的启动时间是否发生了明显的改变。
20. 为语句“Java now has regular expressions”进行评测，看看下面的表达式是否能够在其中找到匹配：

```
^Java
\breg.*
n.w\s+h(a|i)s
s?
s*
s+
s{4}
s{1.}
s{0,3}
```

1. 将正则表达式

```
(?i)((^[aeiou])|(\s+[aeiou]))\w+?[aeiou]\b
```

应用于

"Arline ate eight apples and one orange while Anita hadn't any"

1. 修改 JGrep.java 使其能够接收一个标记参数(例如 Pattern.CASE_INSENSITIVE、Pattern.MULTILINE)
2. 修改 JGrep.java, 让其使用 Java 的 nio 内存映射文件。
3. 修改 JGrep.java 以接受目录名称或文件名作为参数(如果提供的是目录名, 应该搜寻目录下的所有文件)。提示: 可以用:

`String[] filenames = new File(".").list()`

来创建一个文件列表。

第十三章 并发

对象技术使你得以把程序划分成若干独立的部分。通常，你还需要把程序转换成彼此分离的，能独立运行的子任务。

每一个这些独立的子任务都被称为一个“线程”（thread）。你要这样去编写程序：每个线程都好像是在独自运行并且占有自己的处理器。处理器时间确实是通过某些底层机制进行分配的，不过一般来说，你不必考虑这些，这使得编写多线程程序的任务变得容易得多了。

所谓“进程”（process），是一个独立运行着的程序，它有自己的地址空间。“多任务”（multitasking）操作系统通过周期性地处理器切换到不同的任务，使其能够同时运行不止一个进程（程序），而每个进程都象是连续运行、一气呵成的。线程是进程内部的一控制序列流。因此一个进程内可以具有多个并发执行的线程。

多线程有多种用途，不过通常用法是，你的程序中的某个部分与一个特定的事件或资源联系在了一起，而你又不想让这种联系阻碍程序其余部分的运行。这时，可以创建一个与这个事件或资源相关联的线程，并且让此线程独立于主程序运行。

学习并发编程就像进入了一个全新的领域，有点类似于学习一门新的编程语言，或者至少要学习一整套新的语言概念。随着对线程的支持在大多数微机操作系统中的出现，在编程语言和程序库中也出现了对线程的扩展。总的来说，线程编程：

1. 不仅看起来神秘，而且需要你改变编程时的思维方式。
2. 各种语言中对线程的支持都很相似，所以只要理解了线程概念，那么在别的语言中要用到线程的话就有了共同语言。

尽管对线程的支持使Java看起来更复杂，不过这并不全是Java的错，线程本身就很讲究技巧。

要理解并发编程，其难度与理解多态机制差不多。如果你花了工夫，就能明白其基本机制，但要想真正地掌握它的实质，就需要深入的学习和理解。本章的目标就是要让你对并发的基本知识打下坚实的基础，使你能够理解其概念并编写出合理的多线程程序。注意，你可能会很容易就变得过分自信，所以在你编写任何复杂程序之前，应该学习一下专门讨论这个主题的书籍。

动机

使用并发最强制性的原因之一就是产生能够作出响应的用户界面。考虑一个程序，它要执行某项CPU深度占用的计算，这样就会导致用户的输入被忽略，也就无法作出响应。问题的实质是：程序需要一边连续进行计算，同时还要把控制权交给用户界面，这样程序才能响应用户的操作。如果你有一个“退出”按钮，你一定不希望程序的每段代码里都检测按钮状态，但你还是希望对这个按钮能够作出响应，就好像你定期对其进行检测一样。

传统的方法不可能一边连续执行其操作，同时又把控制权交给程序的其余部分。事实上，这听起来就像是不可能完成的任务，就好象让一个处理器同时出现在两个地方，但这恰恰是并发编程所能提供的错觉效果。

并发还可以用来优化程序的吞吐量。比如，在你等待数据到达输入/输出端口的时候，你可以进行其他重要的工作。要是不用线程的话，唯一可行的办法就是不断查询输入/输出端口，这种方法不仅笨拙，而且很困难。

如果你有一台有多处理器的机器，多个线程就可以分布在多个处理器上，这可以极大地提高吞吐量。这种情况通常出现在有多个强劲处理器的web服务器上，在这种环境下，程序对于每个用户请求都将分配一个线程，这样就可以把大量的请求分配给多个处理器来处理。

需要牢记的是，具有多个线程的程序，必须也能够在只有单处理器的机器上运行。因此，不使用任何线程而写出具有同样功能的程序也是可能的。然而，使用多线程的重要好处是可以使程序的组织更有条理，因而能大大简化程序设计。对于某些类型的问题，比如模拟视频游戏，如果没有对并发的支持将会非常难以解决。

线程模型为编程带来了便利，它简化了在单一程序中交织在一起同时运行的多个操作。在使用线程时，处理器将轮流给每个线程分配其占用时间。每个线程都觉得自己一直在占用处理器，但事实上处理器时间是划分成片段分配给了所有的线程。例外情况是程序运行在具有多个处理器的机器上，但线程的一大好处是可以使你从这个层次抽身出来，即代码不必知道它是运行在具有一个还是多个处理器的机器上。所以，线程是一种建立透明的、可扩展的程序的方法，如果程序运行得太慢，为机器增添一个处理器就能很容易地加快程序的运行速度。多任务和多线程往往是使用多处理机系统的最合理方式。

在单处理器机器上，线程会降低一些运行效率，但是，从程序设计、资源平衡、用户使用方便等方面来看，还是非常值得的。一般来说，线程使你能得到更加松散耦合的设计；否则的话，你将不得不在部分代码中直接关注那些通常由线程处理的工作。

基本线程

写一个线程最简单的做法是从`java.lang.Thread`继承，这个类已经具有了创建和运行线程所必要的架构。`Thread`最重要的方法是`run()`，你得重载这个方法，以实现你要的功能。这样，`run()`里的代码就能够与程序里的其它线程“同时”执行。

下面的例子创建了五个线程，每个线程由一个唯一的数字来标识，这个数字由一个静态成员变量产生。`Thread`类的`run()`方法被重载，它的动作是在每次循环里计数值减一，当计数值为零的时候返回（在`run()`方法返回的地点，将由线程机制终止此线程）。

```
//: c13:SimpleThread.java
// Very simple Threading example.
import com.bruceeckel.simpletest.*;

public class SimpleThread extends Thread {
    private static Test monitor = new Test();
    private int countDown = 5;
    private static int threadCount = 0;
```

```

public SimpleThread() {
    super("" + ++threadCount); // Store the thread
name
    start();
}
public String toString() {
    return "#" + getName() + ": " + countDown;
}
public void run() {
    while(true) {
        System.out.println(this);
        if(--countDown == 0) return;
    }
}
public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
        new SimpleThread();
    monitor.expect(new String[] {
        "#1: 5",
        "#2: 5",
        "#3: 5",
        "#5: 5",
        "#1: 4",
        "#4: 5",
        "#2: 4",
        "#3: 4",
        "#5: 4",
        "#1: 3",
        "#4: 4",
        "#2: 3",
        "#3: 3",
        "#5: 3",
        "#1: 2",
        "#4: 3",
        "#2: 2",
        "#3: 2",
        "#5: 2",
        "#1: 1",
        "#4: 2",
        "#2: 1",
        "#3: 1",
        "#5: 1",
        "#4: 1"
    }, Test.IGNORE_ORDER + Test.WAIT);
}

```

```
    }
} //:~
```

通过调用 `Thread` 类相应的构造器，可以给线程对象指定一个名字。这个名字可以在 `toString()` 里用 `getName()` 得到。

`Thread` 对象的 `run()` 方法一般总会有某种形式的循环，使得线程一直运行下去直到不再需要，所以你要设定跳出循环的条件（或者，就像前面的程序那样，直接从 `run()` 返回）。通常，`run()` 被写成无限循环的形式，这就意味着，除非有某个条件使得 `run()` 终止，否则它将永远运行下去（在本章后面你将看到如何安全地通知线程终止）。

你可以看到在 `main()` 里创建并运行了一些线程。`Thread` 类的 `start()` 方法将为线程执行特殊的初始化动作，然后调用 `run()` 方法。所以整个步骤是：首先调用构造器来构造对象，在构造器中调用了 `start()` 方法来配置线程，然后由线程执行机制调用 `run()`。如果你不调用 `start()`（在后面的例子你将看到，你不必在构造器里调用 `start()`），线程永远不会启动。

因为线程调度机制的行为不是确定性的，所以每次运行该程序都会产生不同的输出结果。实际上，你要是在不同的 JDK 版本下运行这个简单的程序，就会发现程序输出的差异非常大。比如，以前版本的 JDK 经常都是不切片时间的，所以线程 1 可能首先循环执行完毕，然后是线程 2 完成其所有循环，如此下去。这样的做法除了启动这些线程开销更加昂贵以外，在实质上，与调用一个子程序然后马上完成该子程序所有循环的做法类似。用 JDK 1.4 你能得到与 `SimpleThread.java` 类似的输出，这表明了调度器执行了更合适的时间切片行为，每个线程看起来都得到了有秩序的服务。总的说来，JDK 这种行为上的变化并没有被 Sun 所提到，所以你不能对线程的行为作任何假设。应付这类问题最好的办法就是在编写线程代码时尽可能保守些。

当在 `main()` 中创建若干个 `Thread` 对象的时候，并没有获得它们中任何一个的引用。对于普通的对象，这会使它成为垃圾回收器要回收的目标，但对于 `Thread` 对象就不会了。每个 `Thread` 对象需要“注册”自己，所以实际上在某个地方存在着对它的引用，垃圾收集器只有在线程离开了 `run()` 并且死亡之后才能把它清理掉。

让步

如果你知道 `run()` 方法中已经完成了所需的工作，你可以给线程调度机制一个暗示：你的工作已经做得差不多了，可以让别的线程使用处理器了。这个暗示将通过调用 `yield()` 方法的形式来作出。（不过这只是一个暗示，没有任何机制保证它将会被采纳。）

我们可以修改前面的例子，在每次循环之后调用 `yield()`。

```
//: c13:YieldingThread.java
// Suggesting when to switch threads with yield().
import com.bruceeckel.simpletest.*;

public class YieldingThread extends Thread {
    private static Test monitor = new Test();
    private int countDown = 5;
    private static int threadCount = 0;
```



```

public YieldingThread() {
    super("" + ++threadCount);
    start();
}
public String toString() {
    return "#" + getName() + ": " + countDown;
}
public void run() {
    while(true) {
        System.out.println(this);
        if(--countDown == 0) return;
        yield();
    }
}
public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
        new YieldingThread();
    monitor.expect(new String[] {
        "#1: 5",
        "#2: 5",
        "#4: 5",
        "#5: 5",
        "#3: 5",
        "#1: 4",
        "#2: 4",
        "#4: 4",
        "#5: 4",
        "#3: 4",
        "#1: 3",
        "#2: 3",
        "#4: 3",
        "#5: 3",
        "#3: 3",
        "#1: 2",
        "#2: 2",
        "#4: 2",
        "#5: 2",
        "#3: 2",
        "#1: 1",
        "#2: 1",
        "#4: 1",
        "#5: 1",
        "#3: 1"
    }, Test.IGNORE_ORDER + Test.WAIT);
}

```

```

    }
} ///:~

```

使用`yield()`以后，程序的输出显得比较均衡。但要注意的是，如果输出的字符串再长一点的话，你就会得到与`SimpleThread.java`大致相同的输出。你可以试一试，逐步改变`toString()`方法，每次输出更长的字符串，以观察效果。因为调度机制是抢占式的，它能决定在需要的时候中断一个线程并切换到别的线程，所以如果输入/输出（在`main()`所在的线程内执行）占用了太多的时间，它将在`run()`有机会调用`yield()`之前被中断。一般来说，`yield()`使用的机会并不多，你要是想对程序做认真的调整的话，就不能依赖于它。

休眠

另一种能控制线程行为的方法是调用`sleep()`，这将使线程停止执行一段时间，该时间由你给定的毫秒数决定。在前面的例子里，要是把对`yield()`的调用换成`sleep()`，将得到如下结果：

```

//: c13:SleepingThread.java
// Calling sleep() to wait for awhile.
import com.bruceeckel.simpletest.*;

public class SleepingThread extends Thread {
    private static Test monitor = new Test();
    private int countDown = 5;
    private static int threadCount = 0;
    public SleepingThread() {
        super("" + ++threadCount);
        start();
    }
    public String toString() {
        return "#" + getName() + ": " + countDown;
    }
    public void run() {
        while(true) {
            System.out.println(this);
            if(--countDown == 0) return;
            try {
                sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
    public static void
    main(String[] args) throws InterruptedException {

```

```

        for(int i = 0; i < 5; i++)
            new SleepingThread().join();
monitor.expect(new String[] {
    "#1: 5",
    "#1: 4",
    "#1: 3",
    "#1: 2",
    "#1: 1",
    "#2: 5",
    "#2: 4",
    "#2: 3",
    "#2: 2",
    "#2: 1",
    "#3: 5",
    "#3: 4",
    "#3: 3",
    "#3: 2",
    "#3: 1",
    "#4: 5",
    "#4: 4",
    "#4: 3",
    "#4: 2",
    "#4: 1",
    "#5: 5",
    "#5: 4",
    "#5: 3",
    "#5: 2",
    "#5: 1"
});
    }
} ///:~

```

在你调用`sleep()`方法的时候，必须把它放在`try`块中，这是因为`sleep()`方法在休眠时间到期之前有可能被中断。如果某人持有对此线程的引用，并且在此线程上调用了`interrupt()`方法，就会发生这种情况。（如果对线程调用了`wait()`或`join()`方法，`interrupt()`也会对线程有影响，所以对这些方法的调用也必须放在类似的`try`块中，我们将在后面学习这些方法）。通常，如果你想使用`interrupt()`来中断一个挂起的线程，那么挂起的时候最好使用`wait()`而不是`sleep()`，这样就不可能在`catch`子句里的结束了。这里，我们把异常作为一个`RuntimeException`重新抛出，这遵循了“除非知道如何处理，否则不要捕获异常”的原则。

你将会发现程序的输出是确定的，每个线程都在下一个线程开始之前递减计数。这是因为在每个线程之上都使用了`join()`（很快你就会学到），所以`main()`在继续执行之前会等待线程结束。要是你不使用`join()`，你可能会发现线程可以以任意顺序执行，这说明`sleep()`也不是控制线程执行顺序的方法，它仅仅使线程停止执行一段时间。你得到的唯

一保证是线程将休眠至少 100 毫秒，这个时间段也可能更长，因为在休眠时间到期之后，线程调度机制也需要时间来恢复线程。

如果你必须要控制线程的执行顺序，你最好是根本不用线程，而是自己编写以特定顺序彼此控制的协作子程序。

优先权

线程的“优先权”（priority）能告诉调度程序其重要性如何。尽管处理器处理现有线程集的顺序是不确定的，但是如果有许多线程被阻塞并在等待运行，那么调度程序将倾向于让优先权最高的线程先执行。然而，这并不是意味着优先权较低的线程将得不到执行（也就是说，优先权不会导致死锁）。优先级较低的线程仅仅是执行的频率较低。

下面的程序修改了 SimpleThread.java，用以演示优先权。线程的优先权是通过使用 setPriority() 方法进行调整的。

```
//: c13:SimplePriorities.java
// Shows the use of thread priorities.
import com.bruceeckel.simpletest.*;

public class SimplePriorities extends Thread {
    private static Test monitor = new Test();
    private int countDown = 5;
    private volatile double d = 0; // No optimization
    public SimplePriorities(int priority) {
        setPriority(priority);
        start();
    }
    public String toString() {
        return super.toString() + ": " + countDown;
    }
    public void run() {
        while(true) {
            // An expensive, interruptable operation:
            for(int i = 1; i < 100000; i++)
                d = d + (Math.PI + Math.E) / (double)i;
            System.out.println(this);
            if(--countDown == 0) return;
        }
    }
    public static void main(String[] args) {
        new SimplePriorities(Thread.MAX_PRIORITY);
        for(int i = 0; i < 5; i++)
            new SimplePriorities(Thread.MIN_PRIORITY);
        monitor.expect(new String[] {
```

```

        "Thread[Thread-1,10,main]: 5",
        "Thread[Thread-1,10,main]: 4",
        "Thread[Thread-1,10,main]: 3",
        "Thread[Thread-1,10,main]: 2",
        "Thread[Thread-1,10,main]: 1",
        "Thread[Thread-2,1,main]: 5",
        "Thread[Thread-2,1,main]: 4",
        "Thread[Thread-2,1,main]: 3",
        "Thread[Thread-2,1,main]: 2",
        "Thread[Thread-2,1,main]: 1",
        "Thread[Thread-3,1,main]: 5",
        "Thread[Thread-4,1,main]: 5",
        "Thread[Thread-5,1,main]: 5",
        "Thread[Thread-6,1,main]: 5",
        "Thread[Thread-3,1,main]: 4",
        "Thread[Thread-4,1,main]: 4",
        "Thread[Thread-5,1,main]: 4",
        "Thread[Thread-6,1,main]: 4",
        "Thread[Thread-3,1,main]: 3",
        "Thread[Thread-4,1,main]: 3",
        "Thread[Thread-5,1,main]: 3",
        "Thread[Thread-6,1,main]: 3",
        "Thread[Thread-3,1,main]: 2",
        "Thread[Thread-4,1,main]: 2",
        "Thread[Thread-5,1,main]: 2",
        "Thread[Thread-6,1,main]: 2",
        "Thread[Thread-4,1,main]: 1",
        "Thread[Thread-3,1,main]: 1",
        "Thread[Thread-6,1,main]: 1",
        "Thread[Thread-5,1,main]: 1"
    }, Test.IGNORE_ORDER + Test.WAIT);
}
} ///:~

```

在这个版本中，`toString()` 方法被重载，并在里面使用了 `Thread.toString()` 方法来打印线程的名称（你可以通过构造器来自己设置这个名称；这里是自动生成的名称，如 `Thread-1`, `Thread-2` 等），线程的优先权，以及线程所属的“线程组”。因为线程是自标识的，所以本例中没有使用 `threadNumber`。重载的 `toString()` 方法还打印了线程的倒计数值。

你可以看到 `thread 1` 的优先权最高，其余线程的优先权被设为最低。

在 `run()` 里，加入了 100,000 次开销相当大的浮点运算，包括 `double` 类型的加法与除法。变量 `d` 用 `volatile` 修饰，以确保不进行优化。如果没有加入这些运算的话，你就看不到设置优先级的效果（试一试：把包含 `double` 运算的 `for` 循环注释掉）。有了这些运算，你就能观察到 `thread 1` 被线程调度机制优先选择（至少在我的 windows 2000 机器上是这样）。

尽管向控制台打印也是开销大的操作，但在那种情况下看不出优先权的效果，因为向控制台打印不能被中断（否则的话，在多线程情况下控制台显示就乱套了），而数学运算是可以中断的。运算时间要足够长，这样线程调度机制才来得及进行改变，并注意到thread1的优先权，使其被优先选择。

对于已存在的线程，你可以用 `getPriority()` 方法得到其优先权，也可以在任何时候使用 `setPriority()` 方法更改其优先权（这并不局限于像 `SimplePriorities.java` 里那样在构造器中修改）。

尽管JDK有 10 个优先级别，但它与多数操作系统都不能映射得很好。比如，Windows 2000 有 7 个优先级且不是固定的，所以这种映射关系也是不确定的（尽管Sun的Solaris有 2^{31} 个优先级）。唯一可移植的策略是当你调整优先级的时候，只使用 `MAX_PRIORITY`，`NORM_PRIORITY`，和 `MIN_PRIORITY` 三种级别。

后台线程

所谓“后台” (daemon) 线程，是指程序运行的时候，在后台提供一种通用服务的线程，并且这种服务并不属于程序中不可或缺的部分。因此，当所有的非后台线程结束，程序也就终止了。反过来说，只要有任何非后台线程还在运行，程序就不会终止。比如，执行 `main()` 的就是一个非后台线程。

```
//: c13:SimpleDaemons.java
// Daemon threads don't prevent the program from
ending.

public class SimpleDaemons extends Thread {
    public SimpleDaemons() {
        setDaemon(true); // Must be called before start()
        start();
    }
    public void run() {
        while(true) {
            try {
                sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println(this);
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new SimpleDaemons();
    }
} ///:~
```

你必须在线程启动之前调用 `setDaemon()` 方法，才能把它设置为后台线程。在 `run()` 里面，线程被设定为休眠一段时间。一旦所有的线程都启动了，程序马上会在所有的线程能打印信息之前立刻终止，这是因为没有非后台线程（除了 `main()`）使得程序保持运行。因此，程序未打印任何信息就终止了。

你可以通过调用 `isDaemon()` 方法来确定线程是否是一个后台线程。如果是一个后台线程，那么它创建的任何线程将被自动设置成后台线程，如下例所示：

```
//: c13:Daemons.java
// Daemon threads spawn other daemon threads.
import java.io.*;
import com.bruceeckel.simpletest.*;

class Daemon extends Thread {
    private Thread[] t = new Thread[10];
    public Daemon() {
        setDaemon(true);
        start();
    }
    public void run() {
        for(int i = 0; i < t.length; i++)
            t[i] = new DaemonSpawn(i);
        for(int i = 0; i < t.length; i++)
            System.out.println("t[" + i + "].isDaemon() =
"
                + t[i].isDaemon());
        while(true)
            yield();
    }
}

class DaemonSpawn extends Thread {
    public DaemonSpawn(int i) {
        start();
        System.out.println("DaemonSpawn " + i + "
started");
    }
    public void run() {
        while(true)
            yield();
    }
}

public class Daemons {
    private static Test monitor = new Test();
```

```

        public static void main(String[] args) throws
Exception {
            Thread d = new Daemon();
            System.out.println("d.isDaemon() = " +
d.isDaemon());
            // Allow the daemon threads to
            // finish their startup processes:
            Thread.sleep(1000);
            monitor.expect(new String[] {
                "d.isDaemon() = true",
                "DaemonSpawn 0 started",
                "DaemonSpawn 1 started",
                "DaemonSpawn 2 started",
                "DaemonSpawn 3 started",
                "DaemonSpawn 4 started",
                "DaemonSpawn 5 started",
                "DaemonSpawn 6 started",
                "DaemonSpawn 7 started",
                "DaemonSpawn 8 started",
                "DaemonSpawn 9 started",
                "t[0].isDaemon() = true",
                "t[1].isDaemon() = true",
                "t[2].isDaemon() = true",
                "t[3].isDaemon() = true",
                "t[4].isDaemon() = true",
                "t[5].isDaemon() = true",
                "t[6].isDaemon() = true",
                "t[7].isDaemon() = true",
                "t[8].isDaemon() = true",
                "t[9].isDaemon() = true"
            }, Test.IGNORE_ORDER + Test.WAIT);
        }
    } ///:~

```

在 Daemon 线程中把后台标志设置为“真”，然后派生出许多子线程，这些线程并没有被明确设置是否为后台线程，不过它们的确是后台线程。接着，线程进入了无限循环，并在循环里调用 `yield()` 方法来把控制权交给其它进程。

一旦 `main()` 完成其工作，就没什么能阻止程序终止了，因为除了后台线程之外，已经没有任何线程在运行了。`main()` 线程被设定为睡眠一段时间，所以你可以观察到所有后台线程启动后的结果。不这样的话，你就只能看见一些后台线程创建时得到的结果。（试试调整 `sleep()` 休眠的时间，以观察这个行为。）

加入到某个线程

一个线程可以在其它线程之上调用 `join()` 方法，其效果是等待一段时间直到第二个线程结束才继续执行。如果某个线程在另一个线程 `t` 上调用 `t.join()`，此线程将被挂起，直到目标线程 `t` 结束才恢复（即 `t.isAlive()` 返回为假）。

你也可以在调用 `join()` 时带上一个超时参数（单位可以是毫秒或者毫秒加纳秒），这样如果目标线程在这段时间到期时还没有结束的话，`join()` 方法总能返回。

对 `join()` 方法的调用可以被中断，做法是在调用线程上调用 `interrupt()` 方法，这时需要用到 `try-catch` 子句。

下面这个例子演示了所有这些操作：

```
//: c13:Joining.java
// Understanding join().
import com.bruceeckel.simpletest.*;

class Sleeper extends Thread {
    private int duration;
    public Sleeper(String name, int sleepTime) {
        super(name);
        duration = sleepTime;
        start();
    }
    public void run() {
        try {
            sleep(duration);
        } catch (InterruptedException e) {
            System.out.println(getName() + " was
interrupted. " +
                "isInterrupted(): " + isInterrupted());
            return;
        }
        System.out.println(getName() + " has awakened");
    }
}

class Joiner extends Thread {
    private Sleeper sleeper;
    public Joiner(String name, Sleeper sleeper) {
        super(name);
        this.sleeper = sleeper;
        start();
    }
    public void run() {
```

```

        try {
            sleeper.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(getName() + " join
completed");
    }
}

public class Joining {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Sleeper
            sleepy = new Sleeper("Sleepy", 1500),
            grumpy = new Sleeper("Grumpy", 1500);
        Joiner
            dopey = new Joiner("Dopey", sleepy),
            doc = new Joiner("Doc", grumpy);
        grumpy.interrupt();
        monitor.expect(new String[] {
            "Grumpy was interrupted. isInterrupted():
false",
            "Doc join completed",
            "Sleepy has awakened",
            "Dopey join completed"
        }, Test.AT_LEAST + Test.WAIT);
    }
} //::~~

```

Sleeper 是一个 Thread 类型，它要休眠一段时间，这段时间是通过构造器传进来的参数所指定的。在 run() 中，sleep() 方法有可能在指定的时间期满时返回，但也可能被中断。在 catch 子句中，将根据 isInterrupted() 的返回值报告这个中断。当另一个线程在该线程上调用 interrupt() 时，将给该线程设定一个标志，表明该线程已经被中断。然而，异常被捕获时将清除这个标志，所以在异常被捕获的时候这个标志总是为假。除异常之外，这个标志还可用于其它情况，比如线程可能会检查其中断状态。

Joiner 线程将通过在 Sleeper 对象上调用 join() 方法来等待 Sleeper 醒来。在 main() 里面，每个 Sleeper 都有一个 Joiner，你可以在输出中发现，如果 Sleeper 被中断或者是正常结束，Joiner 将和 Sleeper 一同结束。

编码的变体

到目前为止你所看到的所有简单例子中，线程对象都继承自Thread。这么做很合理，因为很显然，这些对象仅仅是作为线程而创建的，并不具有其它任何行为。然而，你的类也许已经继承了其它的类，在这种情况下，就不可能同时继承Thread（Java并不支持多重继承）。这时，你可以使用“实现Runnable接口”的方法作为替代。要实现Runnable接口，只需实现run（）方法，Thread也是从Runnable接口实现而来的。

以下例子演示了这种方法的要点：

```
//: c13:RunnableThread.java
// SimpleThread using the Runnable interface.

public class RunnableThread implements Runnable {
    private int countDown = 5;
    public String toString() {
        return "#" + Thread.currentThread().getName() +
            ": " + countDown;
    }
    public void run() {
        while(true) {
            System.out.println(this);
            if(--countDown == 0) return;
        }
    }
    public static void main(String[] args) {
        for(int i = 1; i <= 5; i++)
            new Thread(new RunnableThread(), "" +
i).start();
        // Output is like SimpleThread.java
    }
} ///:~
```

Runnable 类型的类只需一个 run（）方法，但是如果你想要对这个 Thread 对象做点别的事情（比如在 toString（）里调用 getName（）），那么你就必须通过调用 Thread.currentThread（）方法明确得到对此线程的引用。这里采用的 Thread 构造器接受一个 Runnable 对象和一个线程名称作为其参数。

当某个对象具有Runnable接口，即表明它有run（）方法，但也就仅此而已，不像从Thread继承的那些类，它本身并不带有任何和线程有关的特性。所以要从Runnable对象产生一个线程，你必须像例子中那样建立一个单独的Thread对象，并把Runnable对象传给专门的Thread构造器。然后你可以对这个线程对象调用start（），去执行一些通常的初始化动作，然后调用run（）。

Runnable接口的方便之处在于所有事物都属于同一个类；也就是说，Runnable允许把基类和其它接口混在一起。如果你要访问某些资源，只需直接去做，而不用通过别的对象。不过，内部类也能同样方便地访问外部类的所有部分，所以，成员访问并不是使用Runnable接口形成混和类，而不是“一个Thread子类类型的内部类”的强制因素。

当你使用了Runnable，你通常的意思就是，要用run()方法中所实现的这段代码创建一个进程(process)，而不是创建一个对象表示该进程。这一点是有争议的，取决于你认为把线程作为一个对象来表示，或是作为完全不同的一个实体，即进程来表示，这两种方式哪一种更具实际意义¹。如果你觉得应该是一个进程，你就不必拘泥于面向对象的原则，即“所有事物都是对象”。这也意味着，如果仅仅是想开启一个进程以驱动程序的某个部分，就没有理由把整个类写成是Runnable类型的。因此，使用内部类把和线程有关的代码隐藏在类的内部，似乎更合理，如下所示：

```
//: c13:ThreadVariations.java
// Creating threads with inner classes.
import com.bruceeckel.simpletest.*;

// Using a named inner class:
class InnerThread1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner extends Thread {
        Inner(String name) {
            super(name);
            start();
        }
        public void run() {
            while(true) {
                System.out.println(this);
                if(--countDown == 0) return;
                try {
                    sleep(10);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
        public String toString() {
            return getName() + ": " + countDown;
        }
    }
    public InnerThread1(String name) {
```

¹ Java 1.0 已经支持Runnable，而Java 1.1 才引入内部类，这也部分说明了Runnable存在的原因。此外，传统的多线程模式着眼于要运行的功能，而不是表示成对象。我的习惯是尽可能从Thread继承；这样看起来更清楚和灵活。

```

        inner = new Inner(name);
    }
}

```

// Using an anonymous inner class:

```

class InnerThread2 {
    private int countDown = 5;
    private Thread t;
    public InnerThread2(String name) {
        t = new Thread(name) {
            public void run() {
                while(true) {
                    System.out.println(this);
                    if(--countDown == 0) return;
                    try {
                        sleep(10);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        };
        t.start();
    }
}

```

// Using a named Runnable implementation:

```

class InnerRunnable1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner implements Runnable {
        Thread t;
        Inner(String name) {
            t = new Thread(this, name);
            t.start();
        }
        public void run() {
            while(true) {
                System.out.println(this);
                if(--countDown == 0) return;
                try {

```

```

        Thread.sleep(10);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}

public String toString() {
    return t.getName() + ": " + countDown;
}

public InnerRunnable1(String name) {
    inner = new Inner(name);
}
}

// Using an anonymous Runnable implementation:
class InnerRunnable2 {
    private int countDown = 5;
    private Thread t;
    public InnerRunnable2(String name) {
        t = new Thread(new Runnable() {
            public void run() {
                while(true) {
                    System.out.println(this);
                    if(--countDown == 0) return;
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        }, name);
        t.start();
    }
}

// A separate method to run some code as a thread:
class ThreadMethod {
    private int countDown = 5;

```

```

private Thread t;
private String name;
public ThreadMethod(String name) { this.name =
name; }
public void runThread() {
    if(t == null) {
        t = new Thread(name) {
            public void run() {
                while(true) {
                    System.out.println(this);
                    if(--countDown == 0) return;
                    try {
                        sleep(10);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        };
        t.start();
    }
}
}

```

```

public class ThreadVariations {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        new InnerThread1("InnerThread1");
        new InnerThread2("InnerThread2");
        new InnerRunnable1("InnerRunnable1");
        new InnerRunnable2("InnerRunnable2");
        new ThreadMethod("ThreadMethod").runThread();
        monitor.expect(new String[] {
            "InnerThread1: 5",
            "InnerThread2: 5",
            "InnerThread2: 4",
            "InnerRunnable1: 5",
            "InnerThread1: 4",
            "InnerRunnable2: 5",
            "ThreadMethod: 5",
            "InnerRunnable1: 4",

```

```

        "InnerThread2: 3",
        "InnerRunnable2: 4",
        "ThreadMethod: 4",
        "InnerThread1: 3",
        "InnerRunnable1: 3",
        "ThreadMethod: 3",
        "InnerThread1: 2",
        "InnerThread2: 2",
        "InnerRunnable2: 3",
        "InnerThread2: 1",
        "InnerRunnable2: 2",
        "InnerRunnable1: 2",
        "ThreadMethod: 2",
        "InnerThread1: 1",
        "InnerRunnable1: 1",
        "InnerRunnable2: 1",
        "ThreadMethod: 1"
    }, Test.IGNORE_ORDER + Test.WAIT);
}
} ///:~

```

InnerThread1 创建了一个命名的内部类，它继承自 Thread，并且在构造器中创建了一个此内部类的实例。如果你需要在别的方法中访问此内部类（比如新的方法），这么做就显得很合理。不过，绝大多数时候创建一个线程的原因仅仅是为了使用 Thread 的功能，所以建立一个命名的内部类往往没有必要。InnerThread2 演示了另一种选择：在构造器内部建立了一个匿名内部类，它也继承自 Thread，同时它被向上转型为对 Thread 的引用 t。如果该类的别的方法需要访问 t，它们可以通过 Thread 的接口访问，并且不需要知道这个对象的确切类型。

例子中的第三、四个类和前两个大致相同，但是它们实现了 Runnable 接口而不是从 Thread 继承。这只不过表明了 in 实现 Runnable 接口的情况下，并没有带来更多好处，而且实际上代码要稍微复杂一些（读起来也是）。所以，除非被迫使用 Runnable，否则我更倾向于使用 Thread。

ThreadMethod 类演示了如何在方法内部创建一个线程。当你调用该方法准备运行这个线程时，在线程启动后该方法返回。当线程只是做一些辅助工作，而不是作为类的基本功能的时候，这种方案就比在类的构造器中启动一个线程显得更合理。

建立有响应的用户界面

如前所述，使用线程的动机之一就是建立有响应的用户界面。尽管我们要到第 14 章才接触到图形用户界面，你在本章还是可以看到一个基于控制台用户界面的简单例子。下面的例子有两个版本：一个在全神贯注于运算，所以不能读取控制台输入；另一个把运算放在线程里单独运行，此时就可以在进行运算的同时监听控制台输入。

```

///: c13:ResponsiveUI.java

```



```

// User interface responsiveness.
import com.bruceeckel.simpletest.*;

class UnresponsiveUI {
    private volatile double d = 1;
    public UnresponsiveUI() throws Exception {
        while(d > 0)
            d = d + (Math.PI + Math.E) / d;
        System.in.read(); // Never gets here
    }
}

public class ResponsiveUI extends Thread {
    private static Test monitor = new Test();
    private static volatile double d = 1;
    public ResponsiveUI() {
        setDaemon(true);
        start();
    }
    public void run() {
        while(true) {
            d = d + (Math.PI + Math.E) / d;
        }
    }
    public static void main(String[] args) throws
Exception {
        //! new UnresponsiveUI(); // Must kill this
process
        new ResponsiveUI();
        Thread.sleep(300);
        System.in.read(); // 'monitor' provides input
        System.out.println(d); // Shows progress
    }
} ///:~

```

UnresponsiveUI 在一个无限的 while 循环里执行运算，显然程序不可能到达读取控制台输入的那一行（编译器被欺骗了，相信 while 的条件使得程序能到达读取控制台输入的那一行）。如果你把建立 UnresponsiveUI 的那一行解除注释再运行程序，那么要终止它的话，就只能杀死（kill）这个进程。

要想让程序有响应，就得把计算程序放在 run() 方法中，这样它就能让出处理器给别的程序。当你按下回车键的时候，你可以看到计算确实在作为后台程序运行，同时还在等待用户输入（基于测试的原因，控制台输入这一行使用 com.bruceeckel.simpletest.Test 对象自动提交给 System.in.read()，这将在第 15 章中解释）。

共享受限资源

你可以把单线程程序当作在问题域求解的单一实体，每次只能做一件事情。因为只有一个实体，所以你永远不用担心诸如“两个实体试图同时使用同一个资源”这样的问题，比如：两个人在同一个地方停车，两个人同时走过一扇门，甚至是两个人同时说话。

在多线程的环境中，可以同时做多件事情。但是，“两个或多个线程同时使用同一个受限资源”的问题也出现了。必须防止这种资源访问的冲突，否则，就可能发生两个线程同时试图访问同一个银行帐户，向同一个打印机打印，改变同一个值等诸如此类的问题。

不正确地访问资源

考虑下面的例子，这里的类“保证”当你调用`getValue()`方法时，总能返回一个偶数。同时，另外一个名为“Watcher”的线程不断调用`getValue()`方法并检查返回值是否真的为偶数。这看起来好像没什么意义，因为从代码上看，返回值显然是偶数。但这恰恰是令人惊奇之处。下面是该程序的第一个版本：

```
//: c13:AlwaysEven.java
// Demonstrating thread collision over resources by
// reading an object in an unstable intermediate
// state.

public class AlwaysEven {
    private int i;
    public void next() { i++; i++; }
    public int getValue() { return i; }
    public static void main(String[] args) {
        final AlwaysEven ae = new AlwaysEven();
        new Thread("Watcher") {
            public void run() {
                while(true) {
                    int val = ae.getValue();
                    if(val % 2 != 0) {
                        System.out.println(val);
                        System.exit(0);
                    }
                }
            }
        }.start();
        while(true)
            ae.next();
    }
}
```

```

    }
} ///:~

```

在main()中，建立了一个AlwaysEven对象，它必须是final的，因为它要被一个继承自Thread的匿名内部类所访问。如果线程读出的值不是偶数，它将把这个值打印出来（以证明它捕获了对象的不稳定状态）并退出程序。

这个例子表明了使用线程会遇到的基本问题。你永远不会知道线程是何时运行的。想象一下，你坐在座位旁边，手里有一把叉子，准备叉起盘子里最后一块食物，当叉子碰到食物的时候，它忽然消失了（因为你的线程被挂起，另一个线程跑进来偷走了食物）。这就是你在写并发程序时要面临的问题。

你试图使用一个资源的同时，有时并不关心它是否正在被访问（比如别的盘子里的食物）。但为了让多线程能工作，你就需要某种方法来防止两个线程访问同一个资源，至少是在某个关键时间段内避免此问题。

要防止这类冲突，只要在线程使用资源的时候给它加一把锁就行了。访问资源的第一个线程给资源加锁，接着其它线程就只能等到锁被解除以后才能访问资源，这时某个线程就可以对资源加锁以进行访问。如果把汽车的前排座椅看成是受限资源的话，那么你的小孩大喊一声“我要坐”，就相当于在声明上锁。

一个资源测试框架

在我们继续之前，先来建立一个小型框架，试试能否简化对这种类型的线程例子的测试工作。我们可以把在多个例子中出现的常用代码分离出来做到这一点。首先，注意到“观察者”线程实际上在观察特定对象内部是否违例了约束条件。也就是说，假设对象具有保持其内部状态的条件，但如果你能从外部观察到对象的非法中间状态，那么从客户的观点来看，约束条件确实遭到了破坏（这并不是说对象在非法的中间状态下就不能存在，而是这种状态不该被客户观察到）。所以，我们不仅要检查约束条件是否被违反，还要知道这个违例的值是多少。要想用一次方法调用就得到这两个结果，我们得把它们捆绑在一个标记接口里，这个接口仅仅用来在代码中提供有意义的名字：

```

//: c13:InvariantState.java
// Messenger carrying invariant data
public interface InvariantState {} ///:~

```

在这种模式下，有关成功或是失败的信息被编码到类的名称和类型中，以使结果更可读。表示成功的类是：

```

//: c13:InvariantOK.java
// Indicates that the invariant test succeeded
public class InvariantOK implements InvariantState
{} ///:~

```

要表示失败，InvariantFailure 对象将包括一个对象，此对象表示了有关失败原因的信息，这样就可以显示出来：

```
//: c13:InvariantFailure.java
// Indicates that the invariant test failed

public class InvariantFailure implements
InvariantState {
    public Object value;
    public InvariantFailure(Object value) {
        this.value = value;
    }
} ///:~
```

现在我们可以定义一个接口，任何需要对约束条件进行测试的类必须实现这个接口：

```
//: c13:Invariant.java
public interface Invariant {
    InvariantState invariant();
} ///:~
```

在创建通用的“观察者”线程之前，要注意到本章中的某些例子也许不能在所有的平台上都按预期效果运行。这里的许多例子是故意演示在多线程环境下对单线程行为的影响，而这种情况并非总是发生²。相反，试图演示这种违例的例子也许不会（或者没有成功）造成任何影响。这时，我们需要某种方法在一段时间后能终止程序。下面的类通过继承标准类库中的Timer类做到了这一点：

```
//: c13:Timeout.java
// Set a time limit on the execution of a program
import java.util.*;

public class Timeout extends Timer {
    public Timeout(int delay, final String msg) {
        super(true); // Daemon thread
        schedule(new TimerTask() {
            public void run() {
                System.out.println(msg);
                System.exit(0);
            }
        }, delay);
    }
}
```

²一些例子是在一台有双处理器的win2k机器上编写的，这样冲突更容易发生。然而，在单处理器机器上运行同样的程序时可能会在相当长时间内没有冲突，正是这种不确定性使得多线程编程如此困难。你可以想象一下，你在单处理器机器上编写程序，并且认为代码是线程安全的，一旦把程序拿到一台多处理器机器上运行的时候，马上出了问题。

```
} ///:~
```

延迟的单位是毫秒，时间到期的话，将打印消息。注意通过调用`super(true)`，此线程将作为一个后台线程而创建，所以要是你的程序采用别的方式结束，此线程将不会阻止程序终止。`Timer.schedule()`方法传入了一个`TimerTask`的子类（此处作为匿名内部类创建）对象，这个对象的`run()`方法将在`schedule()`的第二个参数`delay`指定的时间到期之后执行。使用`Timer`一般来说比直接写代码调用`sleep()`要简单和清晰。此外，设计`Timer`的目的就是承担大量并发调度任务，所以它能成为很有用的工具。

现在我们可以用 `InvariantWatcher` 线程里使用 `Invariant` 接口和 `Timeout` 类了：

```
//: c13:InvariantWatcher.java
// Repeatedly checks to ensure invariant is not
violated

public class InvariantWatcher extends Thread {
    private Invariant invariant;
    public InvariantWatcher(Invariant invariant) {
        this.invariant = invariant;
        setDaemon(true);
        start();
    }
    // Stop everything after awhile:
    public
    InvariantWatcher(Invariant invariant, final int
timeOut){
        this(invariant);
        new Timeout(timeOut,
            "Timed out without violating invariant");
    }
    public void run() {
        while(true) {
            InvariantState state = invariant.invariant();
            if(state instanceof InvariantFailure) {
                System.out.println("Invariant violated: "
                    + ((InvariantFailure)state).value);
                System.exit(0);
            }
        }
    }
} ///:~
```

构造器接受一个`Invariant`对象的引用作为参数，它是要测试的对象，然后启动线程。第二个构造器调用第一个构造器，然后创建了一个`Timeout`，用来在一定的时间延迟之后终止所有的线程，如果程序中没有违反约束条件，那么线程就不可能因违反约束条件而终止，

此时就要用到它Timeout了。在run()中，当前的InvariantState被获取和测试，如果有违例的话，违例的值将被打印出来。注意，我们不能在此线程里抛出异常，因为这只会终止线程，而不是终止程序。

现在 AlwaysEven.java 可以用这个框架重写：

```
//: c13:EvenGenerator.java
// AlwaysEven.java using the invariance tester

public class EvenGenerator implements Invariant {
    private int i;
    public void next() { i++; i++; }
    public int getValue() { return i; }
    public InvariantState invariant() {
        int val = i; // Capture it in case it changes
        if(val % 2 == 0)
            return new InvariantOK();
        else
            return new InvariantFailure(new Integer(val));
    }
    public static void main(String[] args) {
        EvenGenerator gen = new EvenGenerator();
        new InvariantWatcher(gen);
        while(true)
            gen.next();
    }
} ///:~
```

在定义invariant()方法的时候，你必须把所有相关的值都存放到局部变量中。这样，你才能返回你真正测试的那个值，否则在返回的时候这个值可能已经（被别的线程）改变。

在这种情况下，问题已经不在于对象是否处于违反约束条件的状态，而是当对象处于这种中间的不稳定状态时，别的线程可能会调用它的方法。

资源冲突

对于EvenGenerator，最糟糕的莫过于客户线程可能会发现它处于不稳定的中间状态。尽管对象最终被观察到处于合法的状态，而且其内部一致性能够得到维护。但如果两个线程确实是在修改同一个对象，共享资源的冲突将变得更糟糕，因为这有可能会把对象设置成不正确的状态。

考虑简单的“信号量”（semaphore）概念，它可以看成是在两个线程之间进行通讯的标志对象。如果信号量的值是零，则它监控的资源是可用的，但如果这个值是非零的，则被监

控的资源不可用，所以线程必须等待。当资源可用的时候，线程增加信号量的值，然后继续执行并使用这个被监控的资源。因为把增加和减少当作是原子操作（也就是不能被中断），信号量能够保证两个线程同时访问同一资源的时候不至于冲突。

如果信号量能正确守护它所监控的资源，那么它一定不会处于不稳定的状态。下面是一个信号量概念的简化版本：

```
//: c13:Semaphore.java
// A simple threading flag

public class Semaphore implements Invariant {
    private volatile int semaphore = 0;
    public boolean available() { return semaphore ==
0; }
    public void acquire() { ++semaphore; }
    public void release() { --semaphore; }
    public InvariantState invariant() {
        int val = semaphore;
        if(val == 0 || val == 1)
            return new InvariantOK();
        else
            return new InvariantFailure(new Integer(val));
    }
} ///:~
```

类的核心部分很直接，包括了`available()`，`acquire()`，和`release()`方法。既然线程在获取资源的时候要检查其可用性，所以信号量的值一定不能是 0 或 1 以外的值，这将由`invariant()`来测试。

但是请看，当测试 Semaphore 的线程一致性时发生的情况：

```
//: c13:SemaphoreTester.java
// Colliding over shared resources

public class SemaphoreTester extends Thread {
    private volatile Semaphore semaphore;
    public SemaphoreTester(Semaphore semaphore) {
        this.semaphore = semaphore;
        setDaemon(true);
        start();
    }
    public void run() {
        while(true)
            if(semaphore.available()) {
                yield(); // Makes it fail faster
            }
    }
}
```

```

        semaphore.acquire();
        yield();
        semaphore.release();
        yield();
    }
}

public static void main(String[] args) throws
Exception {
    Semaphore sem = new Semaphore();
    new SemaphoreTester(sem);
    new SemaphoreTester(sem);
    new InvariantWatcher(sem).join();
}
} ///:~

```

SemaphoreTester创建了一个线程，此线程不断进行测试，以检查Semaphore对象是否可用，可用的话就获取它，然后释放。注意，semaphore的字段被标记为volatile，以确保编译器不会对任何读取此值的操作进行优化。

在main()里，建立了两个SemaphoreTester线程，你会发现很快就发生了违反约束条件的事件。其原因是，一个线程可能从对available()的调用返回为真，但在此线程调用acquire()的时候，另一个线程可能已经调用了acquire()并增加了semaphore字段的值。此时InvariantWatcher可能会发现字段的值太大；如果两个线程都调用了release()以减少字段值后就会出现负数，这时就会出现值太小的情况。注意，主线程在InvariantWatcher上调用了join()，这将使程序一直运行，直到发生失败。

在我的机器上，我发现如果加上对yield()的调用，会导致约束条件的违例情况出现得更快，不过这会因操作系统和JVM实现的不同而有所不同。你应该去掉对yield()的调用，自己试验一下；失败也许要很久才会发生，这也说明了当你编写多线程代码的时候，发现程序的瑕疵是多么困难。

这个类强调了并发编程的风险：如果写这么简单的类也会出问题的话，你永远不能信任对并发编程作出的任何假设。

解决共享资源竞争

基本上所有的多线程模式，在解决线程冲突问题的时候，都是采用“序列化”(serialize)访问共享资源的方案。这意味着在给定时刻只允许一个线程访问共享资源。通常这是通过在代码前面加上一条锁语句来实现的，这就保证了在一段时间内只有一个线程运行这段代码。因为锁语句产生了一种互相排斥的效果，所以常常称为“互斥量”(mutex)。

考虑一下屋子里的浴室：多个人（即多个线程）都希望能单独使用浴室（即共享资源）。为了使用浴室，一个人先敲门，看看是否能使用。如果没人的话，他就进入浴室并且锁上

门。这时其它人要使用浴室的话，就会被“阻挡”，所以他们要在浴室门口等待，直到浴室可以使用。

当浴室使用完毕，就该把浴室给其他人使用了，这个比喻就有点不太准确了。事实上，人们并没有排队，我们也不能确定谁将是下一个使用浴室的人，因为线程调度机制并不是确定性的。实际情况是：等待使用浴室的人们簇拥在浴室门口，当锁住浴室门的那个人打开锁准备离开的时候，离门最近的那个人可能进入浴室。如前所述，可以通过`yield()`和`setPriority()`来给线程调度机制一些建议，但这些建议未必会有多大效果，这取决于你的具体平台和JVM实现。

Java以提供关键字`synchronized`的形式，为防止资源冲突提供了内置支持。它的行为很像Semaphore类：当线程要执行被`synchronized`关键字守护的代码片断的时候，它将检查信号量是否存在，然后获取信号量，执行代码，释放信号量。不同的是，`synchronized`内置于语言，所以这种防护始终存在，不像Semaphore那样要明确使用才能工作。

典型的共享资源是以对象形式存在的内存片断，但也可以是文件，输入/输出端口，或者是打印机。要控制对共享资源的访问，你得先把它包装进一个对象。然后把所有要访问这个方法的方法标记为`synchronized`。也就是说，一旦某个线程处于一个标记为`synchronized`的方法中，那么在这个线程从该方法返回之前，其它要调用类中任何标记为`synchronized`方法的线程都会被阻塞。

一般来说类的数据成员都被声明为私有的，只能通过方法来访问这些数据。所以你可以把方法标记为`synchronized`来防止资源冲突。下面是如何声明`synchronized`方法：

```
synchronized void f() { /* ... */ }
synchronized void g(){ /* ... */ }
```

每个对象都含有一个单一的锁（也称为监视器），这个锁本身就是对象的一部分（你不用写任何特殊代码）。当你在对象上调用其任意`synchronized`方法的时候，此对象都被加锁，这时对象上的其它`synchronized`方法只有等到前一个方法调用完毕并释放了锁之后才能被调用。在上个例子里，如果对对象调用了`f()`，对于这个对象就只能等到`f()`调用结束并释放了锁之后，才能调用`g()`。所以，对于某个对象，其所有`synchronized`方法共享同一个锁，这能防止多个线程同时访问对象所在的内存。

一个线程可以多次获得对象的锁。如果一个方法在同一个对象上调用了第二个方法，后者又调用了同一对象上的另一个方法，就会发生这种情况。JVM负责跟踪对象被加锁的次数。如果一个对象被解锁，其计数为0。在线程第一次给对象加锁的时候，计数变为1。每次线程在这个对象上获得了锁，计数都会增加。显然，只有首先获得了锁的线程才能允许继续获取多个锁。每当线程离开一个`synchronized`方法，计数减少，当计数为零的时候，锁被完全释放，此时别的线程就可以使用此资源。

针对每个类，也有一个锁（作为类的Class对象的一部分），所以`synchronized static`方法可以在类的范围内防止对静态数据的并发访问。

同步控制 EvenGenerator

通过在 EvenGenerator.java 中加入 synchronized 关键字，我们就可以防止不希望的线程访问：

```
//: c13:SynchronizedEvenGenerator.java
// Using "synchronized" to prevent thread collisions

public
class SynchronizedEvenGenerator implements
Invariant {
    private int i;
    public synchronized void next() { i++; i++; }
    public synchronized int getValue() { return i; }
    // Not synchronized so it can run at
    // any time and thus be a genuine test:
    public InvariantState invariant() {
        int val = getValue();
        if(val % 2 == 0)
            return new InvariantOK();
        else
            return new InvariantFailure(new Integer(val));
    }
    public static void main(String[] args) {
        SynchronizedEvenGenerator gen =
            new SynchronizedEvenGenerator();
        new InvariantWatcher(gen, 4000); // 4-second
timeout
        while(true)
            gen.next();
    }
} ///:~
```

你可以注意到next()和getValue()都使用了synchronized进行修饰。要是你只同步控制其中一个方法的话，那么另一个就可以随意地忽略对象锁，从而出现不负责任的调用。关键是：每个访问关键共享资源的方法必须全部是synchronized的，否则就会出错。另一方面，InvariantState没有同步控制，因为它只是进行测试，我们希望它在任意时刻都能被调用，这样它才能真正检查对象的状态。

原子操作

在有关Java线程的讨论中，一个常被提到的认识是“原子操作不需要进行同步控制”。“原子操作”（atomic operation）即不能被线程调度机制中断的操作；一旦操作开始，那

么它一定可以在可能发生的“上下文切换”（context switch）之前（切换到其它线程执行）执行完毕。

还有一个常被提到的知识是，如果问题中的变量类型是除long或double以外的基本类型，对这种变量进行简单的赋值或者返回值操作的时候，才算是原子操作。不包括long和double的原因是因为它们比其它基本类型要大，所以JVM不能把对它的读取或赋值当成是单一原子操作（也许JVM能够这么做，但这并不能保证）。然而，你只要给long或double加上volatile，操作就是原子的了。

如果你把原子操作的概念尝试着应用到 SynchronizedEvenGenerator.java，你将注意到

```
public synchronized int getValue() { return i; }
```

是符合这个定义的。但如果试着去掉synchronized，测试将会失败。因为尽管return i确实是原子操作，去掉synchronized的话，将会出现当对象还处于不稳定的中间状态的时候就被别的线程读取了。所以在试图做这样的优化之前，你必须真正知道自己在做什么。而这并没有简单可行的规则。

作为第二个例子，考虑一下更简单的情况：一个产生序列号的类³。每次调用nextSerialNumber()，它必须向调用者返回一个唯一值：

```
//: c13:SerialNumberGenerator.java

public class SerialNumberGenerator {
    private static volatile int serialNumber = 0;
    public static int nextSerialNumber() {
        return serialNumber++;
    }
} ///:~
```

SerialNumberGenerator基本上与你能想到的一样简单，如果你有C++或其它低级语言的背景，你可能认为自增加操作是一个原子操作，因为它通常可以用一条微处理器指令实现。然而，在JVM中的自增加操作并不是原子的，它牵涉到一次读和一次写，所以即使在这样简单的操作中，也为线程出问题提供了空间。

serialNumber字段标记成volatile，其原因是每个线程都可能拥有一个本地栈以维护一些变量的复本。如果把一个变量定义成volatile，就等于告诉编译器不要做任何优化，这些优化可能会移除那些使字段与线程里的本地数据复本保持同步的读写操作。

要证明这一点，我们需要一个不会用完内存的集合，这时，检测到问题要花很长时间。这里的CircularSet重用了用来存储整型数组的内存，并假设在你访问数组时，覆写值时发生冲突的可能性最小。add()和contains()方法标记为synchronized以防止线程冲突。

³受到Joshua Bloch的《Effective Java》的启发，Addison-Wesley 2001，第190页。

```

//: c13:SerialNumberChecker.java
// Operations that may seem safe are not,
// when threads are present.

// Reuses storage so we don't run out of memory:
class CircularSet {
    private int[] array;
    private int len;
    private int index = 0;
    public CircularSet(int size) {
        array = new int[size];
        len = size;
        // Initialize to a value not produced
        // by the SerialNumberGenerator:
        for(int i = 0; i < size; i++)
            array[i] = -1;
    }
    public synchronized void add(int i) {
        array[index] = i;
        // Wrap index and write over old elements:
        index = ++index % len;
    }
    public synchronized boolean contains(int val) {
        for(int i = 0; i < len; i++)
            if(array[i] == val) return true;
        return false;
    }
}

public class SerialNumberChecker {
    private static CircularSet serials =
        new CircularSet(1000);
    static class SerialChecker extends Thread {
        SerialChecker() { start(); }
        public void run() {
            while(true) {
                int serial =
                    SerialNumberGenerator.nextSerialNumber();
                if(serials.contains(serial)) {
                    System.out.println("Duplicate: " + serial);
                    System.exit(0);
                }
                serials.add(serial);
            }
        }
    }
}

```

```

    }
}
public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
        new SerialChecker();
    // Stop after 4 seconds:
    new Timeout(4000, "No duplicates detected");
}
} ///:~

```

SerialNumberChecker含有一个静态的CircularSet，后者包含了所有已经生成的序列号，以及一个获取序列号并能确保其唯一的嵌套线程。通过建立多个线程来争夺序列号，你会发现线程很快就会得到重复的序列号，（注意这个程序在你的机器上可能并不冲突，但在一台多处理器的机器上确实检查到了冲突）。要解决这个问题，就得给nextSerialNumber()方法加上synchronized关键字。

原子操作只有在对基本类型进行读取或赋值的时候才被认为是安全的。不过，正如在EvenGenerator.java中所见，原子操作也很容易访问到对象尚处于不稳定状态时的值，所以你不能做任何假设。不仅如此，原子操作也不保证对long和double类型能工作（尽管有些JVM实现确实能保证对long和double类型操作的原子性，但如果你依赖于这一点，你的代码就失去了可移植性）。

最安全的就是使用以下方针：

1. 如果你要对类中的某个方法进行同步控制，最好同步所有方法。如果你忽略了其中一个，通常很难确定这么做是否会有负面影响。
2. 当去除方法的同步控制时，要非常小心。通常这么做是基于性能方面的考虑，但在JDK1.3 和JDK1.4 中，同步控制所需的负担已经大大减少。此外，你只应该在使用了性能评价工具证实了同步控制确实是性能瓶颈的时候，才能这么做。

修正信号量

现在考虑 Semaphore.java。看起来我们能通过给三个方法加上 synchronized 标记，来修正这个问题，就像这样：

```

///: c13:SynchronizedSemaphore.java
// Colliding over shared resources

public class SynchronizedSemaphore extends Semaphore
{
    private volatile int semaphore = 0;
    public synchronized boolean available() {
        return semaphore == 0;
    }
}

```

```

    }
    public synchronized void acquire() { ++semaphore; }
    public synchronized void release() { --semaphore; }
    public InvariantState invariant() {
        int val = semaphore;
        if(val == 0 || val == 1)
            return new InvariantOK();
        else
            return new InvariantFailure(new Integer(val));
    }
    public static void main(String[] args) throws
Exception {
        SynchronizedSemaphore sem =new
SynchronizedSemaphore();
        new SemaphoreTester(sem);
        new SemaphoreTester(sem);
        new InvariantWatcher(sem).join();
    }
} ///:~

```

首先，SynchronizedSemaphore 类就显得很奇怪：它是从 Semaphore 类继承而来，且所有被重载的方法都标记为 synchronized，但这些方法的基类版本却不是。Java 并不允许你在重载的时候改变方法的签名，但这却没有产生出错信息。这是因为 synchronized 关键字不属于方法签名的一部分，所以你能把它加进来，而它也并不局限于重载。

从Semaphore类继承的原因是为了重用SemaphoreTester类。当你运行程序的时候你会发现程序还是会产生InvariantFailure错误。

为什么会失败呢？当线程检测到Semaphore可用时，即调用available()并且返回为真的时候，对象上的锁已经被释放。这时，另一个线程可能会冲进来，并在前一个线程增加semaphore值的时候抢先增加。同时，前一个线程还在假设Semaphore对象是可用的，所以会继续向前并盲目的进入acquire()方法，这就使对象处于不稳定的状态。这只不过是并发编程首要规则的又一次教训而已：永远不要做任何假设。

这个问题的唯一解决方案是，把测试可用性操作和获取操作作为一个单一的原子操作，这也就是synchronized关键字与对象的锁协作所共同提供的功能。也就是说，Java的锁和synchronized关键字属于内置的信号量机制，所以你不必自己再去发明一个。

临界区

有时，你只是希望防止多个线程同时访问方法内部的部分代码而不是整个方法。通过这种方式分离出来的代码段被称为“临界区”(critical section)，它也使用synchronized关键字建立。这里，synchronized被用来指定某个对象，此对象的锁被用来对花括号内的代码进行同步控制：

```

synchronized(syncObject) {
    // This code can be accessed
    // by only one thread at a time
}

```

这也被称为“同步控制块”（synchronized block），在进入此段代码前，必须得到 syncObject 对象的锁。如果其它线程已经得到这个锁，那么就得等到锁被释放以后，才能进入临界区。

通过使用同步控制块而不是对整个方法进行同步控制，可以使多个线程访问对象的时间性能得到显著提高，下面的例子比较了这两种同步控制方法。此外，它也演示了如何把一个非保护类型的类，在其它类的保护和控制之下，应用于多线程的环境：

```

//: c13:CriticalSection.java
// Synchronizing blocks instead of entire methods.
Also
// demonstrates protection of a non-thread-safe class
// with a thread-safe one.
import java.util.*;

class Pair { // Not thread-safe
    private int x, y;
    public Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Pair() { this(0, 0); }
    public int getX() { return x; }
    public int getY() { return y; }
    public void incrementX() { x++; }
    public void incrementY() { y++; }
    public String toString() {
        return "x: " + x + ", y: " + y;
    }
}

public class PairValuesNotEqualException
    extends RuntimeException {
    public PairValuesNotEqualException() {
        super("Pair values not equal: " + Pair.this);
    }
}

// Arbitrary invariant -- both variables must be
// equal:
public void checkState() {
    if(x != y)
        throw new PairValuesNotEqualException();
}

```

```

    }
}

// Protect a Pair inside a thread-safe class:
abstract class PairManager {
    protected Pair p = new Pair();
    private List storage = new ArrayList();
    public synchronized Pair getPair() {
        // Make a copy to keep the original safe:
        return new Pair(p.getX(), p.getY());
    }
    protected void store() { storage.add(getPair()); }
    // A "template method":
    public abstract void doTask();
}

// Synchronize the entire method:
class PairManager1 extends PairManager {
    public synchronized void doTask() {
        p.incrementX();
        p.incrementY();
        store();
    }
}

// Use a critical section:
class PairManager2 extends PairManager {
    public void doTask() {
        synchronized(this) {
            p.incrementX();
            p.incrementY();
        }
        store();
    }
}

class PairManipulator extends Thread {
    private PairManager pm;
    private int checkCounter = 0;
    private class PairChecker extends Thread {
        PairChecker() { start(); }
        public void run() {
            while(true) {
                checkCounter++;
            }
        }
    }
}

```



```

        pm.getPair().checkState();
    }
}

public PairManipulator(PairManager pm) {
    this.pm = pm;
    start();
    new PairChecker();
}

public void run() {
    while(true) {
        pm.doTask();
    }
}

public String toString() {
    return "Pair: " + pm.getPair() +
        " checkCounter = " + checkCounter;
}
}

public class CriticalSection {
    public static void main(String[] args) {
        // Test the two different approaches:
        final PairManipulator
            pm1 = new PairManipulator(new PairManager1()),
            pm2 = new PairManipulator(new PairManager2());
        new Timer(true).schedule(new TimerTask() {
            public void run() {
                System.out.println("pm1: " + pm1);
                System.out.println("pm2: " + pm2);
                System.exit(0);
            }
        }, 500); // run() after 500 milliseconds
    }
} ///:~

```

正如注释中注明的，Pair 不是线程安全的，因为它的约束条件（虽然是任意的）需要两个变量维护成相同的值。此外，如本章前面所述，自增操作不是线程安全的，并且因为没有方法被标记为 synchronized，所以你不能保证一个 Pair 对象在多线程程序中不会被破坏。

PairManager 类持有一个 Pair 对象并控制对它的访问。注意唯一的 public 方法是 getPair()，它是被同步控制的。对于抽象方法 doTask()，它的同步控制将在实现的时候进行处理。

至于PairManager类的结构，它的一些功能已经在基类中实现，并且其一个或多个抽象方法将在派生类中定义，这种结构在“设计模式”（Design Patterns）中称为“模板方法”（Template Method）⁴。设计模式使你得以把变化封装在代码里；在此，发生变化的部分是模板方法doTask()。在PairManager1中整个doTask()方法是被同步控制的，但在PairManager2中的doTask()方法使用同步控制块进行同步。注意到synchronized关键字不属于方法签名的一部分，所以可以在重载方法的时候加上去。

PairManager2 值得注意，store()是一个protected方法，它不能被一般客户使用，只能被其子类使用。所以，对这个方法的调用没有必要进行同步控制，而是被放在同步控制块外面。

同步控制块必须指定一个对象才能进行同步，通常，最合理的对象就是在其上调用方法的当前对象：synchronized(this)，在PairManager2中采用了这种方法。这样，当为同步控制块请求锁的时候，对象的其它同步控制方法就不能被调用了。所以其效果不过是缩小了同步控制的范围。

有时这并不符合你的要求，这时你可以创建一个单独的对象，并对其进行同步控制。下面的例子演示了当对象中的方法在不同的锁上同步的时候，两个线程可以访问同一个对象：

```
//: c13:SyncObject.java
// Synchronizing on another object
import com.bruceeckel.simpletest.*;

class DualSynch {
    private Object syncObject = new Object();
    public synchronized void f() {
        System.out.println("Inside f()");
        // Doesn't release lock:
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println("Leaving f()");
    }
    public void g() {
        synchronized(syncObject) {
            System.out.println("Inside g()");
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

⁴参见《Design Patterns》，Gamma等著，Addison-Wesley 1995。

```

        System.out.println("Leaving g()");
    }
}

}

public class SyncObject {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        final DualSynch ds = new DualSynch();
        new Thread() {
            public void run() {
                ds.f();
            }
        }.start();
        ds.g();
        monitor.expect(new String[] {
            "Inside g()",
            "Inside f()",
            "Leaving g()",
            "Leaving f()"
        }, Test.WAIT + Test.IGNORE_ORDER);
    }
} ///:~

```

DualSynch 对象的f()方法在this上同步（通过在整个方法上同步），g()的同步控制块在syncObject对象上同步。因此，两个同步控制相互独立。在main()中通过创建调用f()的线程演示了这一点。main()线程用来调用g()。从输出中你能观察到两个方法同时运行，所以它们没有在对方的同步控制上阻塞。

我们回到CriticalSection.java，通过在一个线程中运行doTask()，在另一个线程中运行内部类PairChecker的实例，创建了PairManipulator来测试两种不同类型的PairManager。要跟踪进行测试的频率，PairChecker在每次成功的时候增加checkCounter计数。在main()中，建立了2个PairManipulator对象，并让它们运行一段时间。当Timer时间到期时，将执行run()方法，它将显示每个PairManipulator的结果，然后退出。运行一下程序，你能看到类似下面的结果：

```

pm1: Pair: x: 58892, y: 58892 checkCounter = 44974
pm2: Pair: x: 73153, y: 73153 checkCounter = 100535

```

尽管你每次运行的结果可能会有很大不同，但一般来说，对于PairChecker的检查频率，PairManager1.doTask()不允许有PairManager2.doTask()那样多。后者采用同步控制块进行同步，所以对象不加锁的时间更长。这也是宁愿使用同步控制块而不是对整个方法进行同步控制的典型原因：使得其它线程能更多地访问（在安全的情况下尽可能多）。

当然，所有的同步控制都要靠程序员的勤奋工作：必须把访问共享资源的代码段包装进一个合适的同步控制块。

线程状态

一个线程可以处于以下四种状态之一：

1. 新建 (new)：线程对象已经建立，但还没有启动，所以它还不能运行。
2. 就绪 (Runnable)：在这种状态下，只要调度程序把时间片分配给线程，线程就可以运行。也就是说，在任意时刻，线程可以运行也可以不运行。只要调度程序能分配时间片给线程，它就可以运行；这不同于死亡和阻塞状态。
3. 死亡 (Dead)：线程死亡的通常方式是从 `run()` 方法返回。在 Java 2 废弃 `stop()` 以前，你也可以调用它，但这很容易让你的程序进入不稳定状态。还有一个 `destroy()` 方法（这个方法从来没被实现过，也许以后也不会被实现，它也属于被废止的）。在本章的后面你将学习一种与调用 `stop()` 功能等价的方式。
4. 阻塞 (Blocked)：线程能够运行，但有某个条件阻止它的运行。当线程处于阻塞状态时，调度机制将忽略线程，不会分配给线程任何处理器时间。直到线程重新进入了就绪状态，它才有可能执行操作。

进入阻塞状态

当一个线程被阻塞时，必然存在某种原因使其不能继续运行。一个线程进入阻塞状态，可能有如下原因：

1. 你通过调用 `sleep(milliseconds)` 使线程进入休眠状态，在这种情况下，线程在指定的时间内不会运行。
2. 你通过调用 `wait()` 使线程挂起。直到线程得到了 `notify()` 或 `notifyAll()` 消息，线程才会进入就绪状态。我们将在下一节验证这一点。
3. 线程在等待某个输入/输出完成。
4. 线程试图在某个对象上调用其同步控制方法，但是对象锁不可用。

在较早的代码中，你也可能看到 `suspend()` 和 `resume()` 用来阻塞和唤醒线程，但是在 Java 2 中这些方法被废止了（因为可能导致死锁），所以本书不讨论这些内容。

线程之间协作

在理解了线程之间可能存在相互冲突，以及怎样避免冲突之后，下一步就是学习怎样使线程之间相互协作。这种协作关键是通过线程之间的握手来进行的，这种握手可以通过Object的方法wait()和notify()来安全的实现。

等待与通知

调用sleep()的时候锁并没有被释放，理解这一点很重要。另一方面，wait()方法的确释放了锁，这就意味着在调用wait()期间，可以调用线程中对象的其他同步控制方法。当一个线程在方法里遇到了对wait()的调用的时候，线程的执行被挂起，对象上的锁被释放。

有两种形式的wait()。第一种接受毫秒作为参数，意思与sleep()方法里参数的意思相同，都是指“在此期间暂停”。不同之处在于，对于wait()：

1. 在wait()期间锁是释放的。
2. 你可以通过notify()、notifyAll()，或者时间到期，从wait()中恢复执行。

第二种形式的wait()不要参数；这种用法更常见。wait()将无限等待直到线程接收到notify()或者notifyAll()消息。

wait()，notify()，以及notifyAll()的一个比较特殊的方面是这些方法是基类Object的一部分，而不是像Sleep()那样属于Thread的一部分。尽管开始看起来有点奇怪，仅仅针对线程的功能却作为通用基类的一部分而实现，不过这是有道理的，因为这些功能要用到的锁也是所有对象的一部分。所以，你可以把wait()放进任何同步控制方法里，而不用考虑这个类是继承自Thread还是实现了Runnable接口。实际上，你只能在同步控制方法或同步控制块里调用wait()，notify()和notifyAll()（因为不用操作锁，所以sleep()可以在非同步控制方法里调用）。如果你在非同步控制方法里调用这些方法，程序能通过编译，但运行的时候，你将得到IllegalMonitorStateException异常，伴随着一些含糊的消息，比如“当前线程不是拥有者”。消息的意思是，调用wait()，notify()和notifyAll()的线程在调用这些方法前必须“拥有”（获取）对象的锁。

你能够让另一个对象执行这种操作以维护其自己的锁。要这么做的话，你必须首先得到对象的锁。比如，如果你要在对象x上调用notify()，那么你就必须在能够取得x的锁的同步控制块中这么做：

```
synchronized(x) {  
    x.notify();  
}
```

特别地，当你在等待某个条件，这个条件必须由当前方法以外的因素才能改变的时候（典型地，这个条件被另一个线程所改变），就应该使用wait()。你也不希望在线程里测试条件的时候空等；这也称为“忙等”，它会极大占用CPU时间。所以wait()允许你在等待外部条件的时候，让线程休眠，只有在收到notify()或notifyAll()的时候线程才唤醒并对变化进行检查。所以，wait()为在线程之间进行同步控制提供了一种方法。

例如，考虑一个餐馆，有一个厨师和一个服务员。服务员必须等待厨师准备好食物。当厨师准备好食物的时候，他通知服务员，后者将得到食物然后继续等待。这是一个线程协作的极好的例子：厨师代表了生产者，服务员代表了消费者。下面是模拟这个场景的代码：

```
//: c13:Restaurant.java
// The producer-consumer approach to thread
cooperation.
import com.bruceeckel.simpletest.*;

class Order {
    private static int i = 0;
    private int count = i++;
    public Order() {
        if(count == 10) {
            System.out.println("Out of food, closing");
            System.exit(0);
        }
    }
    public String toString() { return "Order " + count; }
}

class WaitPerson extends Thread {
    private Restaurant restaurant;
    public WaitPerson(Restaurant r) {
        restaurant = r;
        start();
    }
    public void run() {
        while(true) {
            while(restaurant.order == null)
                synchronized(this) {
                    try {
                        wait();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            System.out.println(
                "Waitperson got " + restaurant.order);
        }
    }
}
```

```

        restaurant.order = null;
    }
}

class Chef extends Thread {
    private Restaurant restaurant;
    private WaitPerson waitPerson;
    public Chef(Restaurant r, WaitPerson w) {
        restaurant = r;
        waitPerson = w;
        start();
    }
    public void run() {
        while(true) {
            if(restaurant.order == null) {
                restaurant.order = new Order();
                System.out.print("Order up! ");
                synchronized(waitPerson) {
                    waitPerson.notify();
                }
            }
            try {
                sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

public class Restaurant {
    private static Test monitor = new Test();
    Order order; // Package access
    public static void main(String[] args) {
        Restaurant restaurant = new Restaurant();
        WaitPerson waitPerson = new
WaitPerson(restaurant);
        Chef chef = new Chef(restaurant, waitPerson);
        monitor.expect(new String[] {
            "Order up! Waitperson got Order 0",
            "Order up! Waitperson got Order 1",
            "Order up! Waitperson got Order 2",
            "Order up! Waitperson got Order 3",

```

```

        "Order up! Waitperson got Order 4",
        "Order up! Waitperson got Order 5",
        "Order up! Waitperson got Order 6",
        "Order up! Waitperson got Order 7",
        "Order up! Waitperson got Order 8",
        "Order up! Waitperson got Order 9",
        "Out of food, closing"
    }, Test.WAIT);
}
} ///:~

```

Order是一个简单的能自己计数的类，但要注意它也包含了终止程序的方法；当订单数累计到10时，将调用System.exit()。

WaitPerson(服务员)必须知道自己所工作的Restaurant(餐馆)，因为他们必须从餐馆的“订单窗口”取出订单restaurant.order。在run()中，WaitPerson调用wait()进入等待模式，停止线程的执行直到被Chef(厨师)的notify()方法所唤醒。因为是很简单的程序，我们知道只有一个线程在等待WaitPerson对象的锁：即WaitPerson线程自己。正因为这个原因，使得调用notify()是安全的。在更复杂的情况下，多个线程可能在等待同一个特定的锁，所以你不知道哪个线程被唤醒。解决方法是调用notifyAll()，它将唤醒所有等待这个锁的线程。每个线程必须自己决定是否对这个通知作出反应。

注意对wait()的调用被包装在一个while()语句里，它在测试的正是等待的条件。开始看起来可能很奇怪——如果你在等一个订单，那么一旦你被唤醒，订单必须是可用的，对不对？问题是在多线程程序里，一些别的线程可能在WaitPerson苏醒的同时冲进来抢走订单。唯一安全的方法就是对于wait()总是使用如下方式：

```

while(conditionIsNotMet)
    wait( );

```

这可以保证在你跳出等待循环之前条件将被满足，如果你被不相干的条件所通知（比如notifyAll()），或者在你完全退出循环之前条件已经被改变，你被确保可以回来继续等待。

一个Chef对象必须知道他/她工作的餐馆（这样可以通过restaurant.order下订单）和取走食物的WaitPerson，这样才能在订单准备好的时候通知WaitPerson。在这个简化过的例子中，由Chef产生订单对象，然后通知WaitPerson订单已经准备好了。

请注意对notify()的调用必须首先获取WaitPerson对象的锁。WaitPerson.run()里对wait()的调用将自动释放这个锁，所以这是可能的。因为要调用notify()必须获取锁，这就能保证如果两个线程试图在同一个对象上调用notify()时不会互相冲突。

上面的例子中，一个线程只有一个单一的地点来存储某个对象，这样另一个线程就可以在以后使用这个对象。然而，在一个典型的生产者—消费者实现中，你要使用先进先出的队列来存放被生产和消费的对象。要对这个问题做更多的了解，请参阅本章后面的练习。

线程间使用管道进行输入/输出

通过输入/输出在线程间进行通信通常很有用。线程库以“管道”(pipes)的形式对线程间的输入/输出提供了支持。它们在 Java 输入/输出库中的对应物就是 `PipedWriter` 类(允许线程向管道写)和 `PipedReader` 类(允许不同线程从同一个管道中读取)。这个模型可以看成是生产者-消费者问题的变体，这里的管道就是一个封装好的解决方案。

下面是一个简单例子，两个线程使用一个管道进行通信：

```
//: c13:PipedIO.java
// Using pipes for inter-thread I/O
import java.io.*;
import java.util.*;

class Sender extends Thread {
    private Random rand = new Random();
    private PipedWriter out = new PipedWriter();
    public PipedWriter getPipedWriter() { return out; }
    public void run() {
        while(true) {
            for(char c = 'A'; c <= 'z'; c++) {
                try {
                    out.write(c);
                    sleep(rand.nextInt(500));
                } catch(Exception e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
}

class Receiver extends Thread {
    private PipedReader in;
    public Receiver(Sender sender) throws IOException
    {
        in = new PipedReader(sender.getPipedWriter());
    }
    public void run() {
        try {
            while(true) {
                // Blocks until characters are there:
                System.out.println("Read: " +
(char)in.read());
            }
        }
    }
}
```

```

    }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

public class PipedIO {
    public static void main(String[] args) throws
Exception {
        Sender sender = new Sender();
        Receiver receiver = new Receiver(sender);
        sender.start();
        receiver.start();
        new Timeout(4000, "Terminated");
    }
} ///:~

```

Sender和Receiver代表了两个线程，它们执行某些任务并且需要互相通信。Sender创建了一个PipedWriter，它是一个单独的对象，但是对于Receiver，PipedReader的建立必须在构造器中与一个PipedWriter相关联。Sender把数据放进Writer然后休眠一段随机的时间。然而，Receiver没有调用sleep()和wait()。但当它调用read()时，如果没有数据，它将自动阻塞。这样你不用使用wait()循环，就得到了生产者-消费者的效果。

注意到sender和receiver是在main()中启动的，即对象构造完毕以后。如果你启动了一个没有构造完毕的对象，在不同的平台上管道可能会产生不一致的行为。

更高级的协作

本章只介绍了最基本的协作方式（比如生产者-消费者，通常用wait()，notify()和notifyAll()实现）。这些能解决大多数线程协作的问题，不过还有很多更高级的方法，它们在一些更深入的书籍中有介绍（尤其是在本章末尾介绍的Lea的著作）。

死锁

因为线程可以阻塞，并且对象可以具有同步控制方法，用以防止别的线程在锁还没有释放的时候就访问这个对象。所以就可能出现这种情况：某个线程在等待另一个线程，而后者又等待别的线程，这样一直下去，直到这个链条上的线程又在等待第一个线程释放锁。你将得到一个线程之间相互等待的连续循环，没有哪个线程能继续。这被称之为“死锁”（deadlock）。

如果你运行一个程序，而它马上就死锁了，你当时就能知道出了问题，并且可以跟踪下去。真正的问题在于，你的程序可能看起来工作良好，但是具有潜在的死锁危险。这时，死锁可能发生，而事先却没有任何征兆，所以它会潜伏在你的程序里，直到客户发现它出乎意料地发生（并且你可能很难重现这个问题）。因此，在编写并发程序的时候，进行仔细的程序设计以防止死锁是一个关键部分。

让我们看一下经典的死锁现象，它是由Dijkstra提出的：哲学家就餐问题。其内容是指定五个哲学家（不过这里的例子中将允许任意数目）。这些哲学家将用部分的时间思考，部分的时间就餐。当他们思考的时候，不需要任何共享资源，但当他们就餐的时候，他们坐在桌子旁并且只有有限数量的餐具。在问题的原始描述中，餐具是叉子，要吃到桌子中央盘子里的意大利面条需要用两把叉子，不过把餐具看成是筷子更合理；很明显，哲学家要就餐就需要两根筷子。

问题中引入的难点是：作为哲学家，他们很穷，所以他们只能买的起五根筷子。他们围坐在桌子周围，每人之间放一根筷子。当一个哲学家要就餐的时候，他/她必须同时得到左边和右边的筷子。如果一个哲学家左边或右边已经有人在使用筷子了，那么这个哲学家就必须等待。

注意，这个问题之所以有趣，其原因是它显示了一个程序可能看起来运行正确，但确实存在死锁的危险。要演示这一点，命令行参数允许你调整哲学家的数量以及一个影响哲学家思考时间的因子。如果你有许多哲学家，并且/或者他们花很多时间去思考，那么尽管存在死锁可能，你可能永远也看不到死锁。默认的命令行参数是倾向于使死锁尽快发生：

```
//: c13:DiningPhilosophers.java
// Demonstrates how deadlock can be hidden in a
program.
// {Args: 5 0 deadlock 4}
import java.util.*;

class Chopstick {
    private static int counter = 0;
    private int number = counter++;
    public String toString() {
        return "Chopstick " + number;
    }
}

class Philosopher extends Thread {
    private static Random rand = new Random();
    private static int counter = 0;
    private int number = counter++;
    private Chopstick leftChopstick;
    private Chopstick rightChopstick;
    static int ponder = 0; // Package access
```

```

    public Philosopher(Chopstick left, Chopstick right)
    {
        leftChopstick = left;
        rightChopstick = right;
        start();
    }

    public void think() {
        System.out.println(this + " thinking");
        if(ponder > 0)
            try {
                sleep(rand.nextInt(ponder));
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
    }

    public void eat() {
        synchronized(leftChopstick) {
            System.out.println(this + " has "
                + this.leftChopstick + " Waiting for "
                + this.rightChopstick);
            synchronized(rightChopstick) {
                System.out.println(this + " eating");
            }
        }
    }

    public String toString() {
        return "Philosopher " + number;
    }

    public void run() {
        while(true) {
            think();
            eat();
        }
    }
}

public class DiningPhilosophers {
    public static void main(String[] args) {
        if(args.length < 3) {
            System.err.println("usage:\n" +
                "java DiningPhilosophers
numberOfPhilosophers " +
                "ponderFactor deadlock timeout\n" +

```

```

        "A nonzero ponderFactor will generate a random
" +
        "sleep time during think().\n" +
        "If deadlock is not the string " +
        "'deadlock', the program will not deadlock.\n"
+
        "A nonzero timeout will stop the program after
" +
        "that number of seconds.");
    System.exit(1);
}
Philosopher[] philosopher =
    new Philosopher(Integer.parseInt(args[0]));
Philosopher.ponder = Integer.parseInt(args[1]);
Chopstick
    left = new Chopstick(),
    right = new Chopstick(),
    first = left;
int i = 0;
while(i < philosopher.length - 1) {
    philosopher[i++] =
        new Philosopher(left, right);
    left = right;
    right = new Chopstick();
}
if(args[2].equals("deadlock"))
    philosopher[i] = new Philosopher(left, first);
else // Swapping values prevents deadlock:
    philosopher[i] = new Philosopher(first, left);
// Optionally break out of program:
if(args.length >= 4) {
    int delay = Integer.parseInt(args[3]);
    if(delay != 0)
        new Timeout(delay * 1000, "Timed out");
}
}
} ///:~

```

Chopstick和Philosopher都使用一个自动增加的静态计数器，来给每个元素一个标识数字。每个Philosopher对象都有一个对左边和右边Chopstick对象的引用；这些是哲学家进餐之前必须得到的餐具。

静态字段ponder表示哲学家是否花费时间进行思考。如果值是非零的，那么在think()方法里线程将休眠一段由这个值随机产生的时间。通过这种方法，就可以展示，线程（哲

学家)在其它任务(思考)上花费的时越多,那么它们在请求共享资源(筷子)的时候发生冲突的可能性就越低,这样你就能确信程序没有死锁,尽管事实上可能是有的。

在`eat()`里,哲学家通过同步控制获取左边的筷子。如果筷子不可用,那么哲学家将等待(阻塞)。在获取了左边的筷子之后,再用同样的方法获取右边的筷子。在就餐完毕之后,先释放右边的筷子,然后释放左边的筷子。

在`run()`中,每个哲学家不断重复思考和就餐的动作。

`main()`方法至少需要三个参数,如果不满足的话,就打印一条使用信息。第三个参数可以是字符串`"deadlock"`,在这种情况下,使用会发生死锁的程序。如果是其它字符串就使用不产生死锁的程序。最后一个参数(可选的)是一个到期因子,表示一段时间之后将退出程序(无论是否发生了死锁)。作为本书所附代码的测试程序的一部分,并且是自动运行的程序,设置一个过期时间是很有必要的。

在创建了`Philosopher`数组并且设置了`ponder`值之后,建立了两个`Chopstick`对象,并且将书组的第一个元素存储在`first`变量里供以后使用。除最后一个元素之外,数组里的每个引用都是通过创建一个新的`Philosopher`对象,并传递给它左筷子和右筷子对象来进行初始化的。在每次初始化之后,左边的筷子放到右边,右边的给一个新的`Chopstick`对象,供下一个`Philosopher`使用。

在死锁版本的程序里,最后一个`Philosopher`被给予了左边的筷子和前面存储的第一个筷子。这是因为最后一个哲学家就坐在第一个哲学家的旁边,他们共享第一根筷子。在这种安排下,在某个时间点上就可能出现,所有的哲学家都准备就餐,并且在等待旁边的哲学家放下他们手中的筷子,此时程序死锁。

试着用不同的命令行参数运行程序,以观察程序的行为,尤其要注意那些使程序看起来可以正常运行,没有死锁的方式。

要修正死锁问题,你必须明白,当以下四个条件同时满足时,就会发生死锁:

1. 互斥条件:线程使用的资源中至少有一个是不能共享的。这里,一根筷子一次就只能被一个哲学家使用。
2. 至少有一个进程持有一个资源,并且它在等待获取一个当前被别的进程持有的资源。也就是说,要发生死锁,哲学家必须拿着一根筷子并且等待另一根。
3. 资源不能被进程抢占。所有的进程必须把资源释放作为普通事件。哲学家很有礼貌,他们不会从其他哲学家那里抢筷子。
4. 必须有循环等待,这时,一个进程等待其它进程持有的资源,后者又在等待另一个进程持有的资源,这样一直下去,直到有一个进程在等待第一个进程持有的资源,使得大家都被锁住。在这里,因为每个哲学家都试图先得到左边的筷子,然后得到右边的筷子,所以发生了循环等待。在上面的例子中,针对最后一个哲学家,通过交换构造器中的初始化顺序,打破了死锁条件,使得最后一个哲学家

先拿右边的筷子，后拿左边的筷子。

因为要发生死锁的话，所有这些条件必须全部满足，所以你要防止死锁的话，只需破坏其中一个即可。在程序中，防止死锁最容易的方法是破坏条件 4。有这个条件的原因是每个哲学家都试图用特定的顺序拿筷子：先左后右。正因为如此，就可能会发生“每个人都拿着左边的筷子，并等待右边的筷子”的情况，这就是循环等待条件。然而，如果最后一个哲学家被初始化成先拿右边的筷子，后拿左边的筷子，那么这个哲学家将永远不会阻止其左边的哲学家拿起他/她右边的筷子，这就打破了循环等待。这只是问题的解决方法之一，你也可以通过破坏其它条件来防止死锁（具体细节请参考更高级的线程书籍）。

Java对死锁并没有提供语言层面的支持；能否通过小心的程序设计以避免死锁，取决于你自己。对于正在试图调试一个有死锁的程序的程序员来说，没有什么更好消息。

正确的停止方法

Java 2 所引入的能够减小死锁可能性的变化就是废弃了Thread类的stop(),suspend()和resume()方法。

废弃stop()方法的原因是它不释放线程获得的锁，如果对象处于不一致的状态(受损状态)，其它线程就可能读取和修改这个状态。导致产生的后果可能非常微妙，并难以检测。如果不使用stop()，你就要使用一个标志告诉线程什么时候通过从run()方法返回以终止自己。下面是个简单的例子：

```
//: c13:Stopping.java
// The safe way to stop a thread.
import java.util.*;

class CanStop extends Thread {
    // Must be volatile:
    private volatile boolean stop = false;
    private int counter = 0;
    public void run() {
        while(!stop && counter < 10000) {
            System.out.println(counter++);
        }
        if(stop)
            System.out.println("Detected stop");
    }
    public void requestStop() { stop = true; }
}

public class Stopping {
    public static void main(String[] args) {
        final CanStop stoppable = new CanStop();
```

```

        stoppable.start();
        new Timer(true).schedule(new TimerTask() {
            public void run() {
                System.out.println("Requesting stop");
                stoppable.requestStop();
            }
        }, 500); // run() after 500 milliseconds
    }
} ///:~

```

Stop标志必须是volatile的，所以run()方法肯定能看到它（否则的话，这个值可能在本地有缓存）。这个线程的任务是打印 10000 个数字，所以当counter >= 10000 或某人请求停止的时候，线程终止。注意requestStop()没有进行同步控制，因为stop是布尔型（把它设为“真”是一个原子操作）并且有volatile标志。

在main()中，启动了CanStop线程，然后设置了一个Timer，用来在半秒后调用requestStop()。Timer的构造器传入了一个参数true，这使其成为一个后台线程，使得它不会阻碍程序的终止。

中断阻塞线程

有时线程会阻塞，比如在等待输入的时候，并且它也不能像前面的例子那样轮询标志。这时，你可以使用Thread.interrupt()方法来跳出阻塞代码：

```

///: c13:Interrupt.java
// Using interrupt() to break out of a blocked thread.
import java.util.*;

class Blocked extends Thread {
    public Blocked() {
        System.out.println("Starting Blocked");
        start();
    }
    public void run() {
        try {
            synchronized(this) {
                wait(); // Blocks
            }
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("Exiting run()");
    }
}

```



```

public class Interrupt {
    static Blocked blocked = new Blocked();
    public static void main(String[] args) {
        new Timer(true).schedule(new TimerTask() {
            public void run() {
                System.out.println("Preparing to
interrupt");
                blocked.interrupt();
                blocked = null; // to release it
            }
        }, 2000); // run() after 2000 milliseconds
    }
} ///:~

```

Blocked.run() 中的 wait() 产生了阻塞线程。当 Timer 到期时，调用了对象的 Interrupt() 方法。然后 blocked 的引用被设为 null，这样垃圾回收器就可以清理这个对象了（虽然在这里微不足道，但对于长期运行的程序就很重要了）。

线程组

线程组持有一个线程集合。线程组的价值可以引用 Joshua Bloch⁵（他是 Sun 公司的软件架构师，曾经修正并大大改良了 JDK1.2 中的集合类库）的话来概括：

“最好把线程组看成是一次不成功的尝试，你只要忽略它就好了。”

如果你已经花费了时间和精力并试图领会线程组的价值（我已经这么做过），你可能会奇怪为什么在这个主题上 Sun 公司没有更多的官方声明，在此之前，多年来发生在 Java 身上的其它任何改变也存在着同样的问题。诺贝尔经济学奖获得者 Joseph Stiglitz 的生活哲学看起来可以应用到这里⁶。它被称为“承诺升级理论”（The Theory of Escalating Commitment）：

“继续错误的后果由别人承担，承认错误的后果要由自己承担。”

线程组还是存在着些许用处的。如果线程组里的线程抛出了一个未捕获的异常，ThreadGroup.uncaughtException() 将被调用，它将向标准错误流打印栈轨迹。如果要修改这个行为，你得重载这个方法。

⁵ 《Effective Java》，Joshua Bloch 著，Addison-Wesley 2001 第 211 页。

⁶ 在贯穿于 Java 的使用经验中，许多地方都存在类似问题。嗯，为什么不那么做呢？我已经咨询过许多采用了这个理念的项目。

总结

明白什么时候应该使用并发，什么时候应该避免使用并发是非常关键的。使用它的原因主要是：要处理很多任务，它们交织在一起，能够更有效地使用计算机（包括在多个处理器上透明地分配任务的能力），能够更好地组织代码，或者更便于用户使用。资源均衡的经典案例是在等待输入/输出时使用处理器。使用户方便的经典案例是在长时间的下载过程中监视“停止”按钮是否被按下。

线程的一个额外好处是它们提供了轻量级的执行上下文切换（大约 100 条指令），而不是重量级的进程上下文切换（要上千条指令）。因为一个给定进程内的所有线程共享相同的内存空间，轻量级的上下文切换只是改变了程序的执行序列和局部变量。进程切换（重量级的上下文切换）必须改变所有内存空间。

多线程的主要缺陷有：

1. 等待共享资源的时候性能降低。
2. 需要处理线程的额外 CPU 耗费。
3. 糟糕的程序设计导致不必要的复杂度。
4. 有可能产生一些病态行为，如饿死、竞争、死锁和活锁。
5. 不同平台导致的不一致性。比如，当我在编写书中的一些例子时，发现竞争条件在某些机器上很快出现，但在别的机器上根本不出现。如果你在后一种机器上做开发，那么当你发布程序的时候就要大吃一惊了。

因为多个线程可能共享一个资源，比如内存中的对象，而且你必须确定多个线程不会同时读取和改变这个资源，这就是线程产生的最大难题。这需要明智的使用 `synchronized` 关键字，它仅仅是个工具，同时它会引入潜在的死锁条件，所以要对它有透彻的理解。

此外，线程应用上也有一些特定技巧。Java 被设计用来让你建立足够多的对象来解决你的问题，至少理论上是如此。（比如，为工程上的有限元分析而创建几百万个对象，在 Java 中并不可行。）然而，你要创建的线程数目看起来还是有个上界，因为达到了一定数量之后，线程性能会很差。这个临界点很难检测，通常依赖于操作系统和 JVM；它可以是少于一百，也可能是几千。不过通常我们只是创建少数线程来解决问题，所以这个限制并不严重；尽管对于更一般的设计来说，这可能会是一个约束。

线程导致的一个明显违反直觉的结果是：线程是可调度的，在 `run()` 的主循环中加入对 `yield()` 甚至是 `sleep()` 的调用，可以使程序运行得更“快”。这绝对使问题看起来更具有技巧性，特别是更长的延迟似乎可以导致性能提升。其原因是，更短的延迟导致休眠结束，调度程序在当前线程准备好休眠之前发生中断，这就迫使调度程序暂停线程，过一段时间后重新启动线程，此时线程才能完成工作然后休眠。多余的上下文切换将导致性能

降低，而使用 `yield()` 或 `sleep()` 能防止这种多余的切换。要搞清楚这个问题会变得多么复杂，就要花额外的时间进行思考。

有关线程更高级的讨论，请参考《*Concurrent Programming in Java*》第二版，Doug Lea 著，Addison-Wesley，2000。

练习

所选习题的答案都可以在《*The Thinking in Java Annotated Solution Guide*》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 从 `Thread` 继承一个类，并重载 `run()` 方法。在 `run()` 中，打印一条消息，然后调用 `sleep()`。重复三遍以后从 `run()` 返回。在构造器中打印一条启动消息，重载 `finalize()` 并打印一条结束消息。再写一个线程类，在其 `run()` 中调用 `System.gc()` 和 `System.runFinalization()`，在调用的时候分别打印消息。为这些线程类创建几个实例，运行它们并观察结果。

2. 在 `Daemons.java` 中使用不同的休眠时间，并观察结果。

3. 找到第八章 `GreenhouseController.java` 这个例子，它包含了四个文件。`Event.java` 中的 `Event` 类是基于观察时间来实现的。把 `Event` 改成一个线程，并修改其它的设计使得它们能与这个新的基于线程的 `Event` 一起工作。

4. 修改上一个练习，使用 `java.util.Timer` 控制系统运行。

5. 把 `SimpleThread.java` 中的所有线程修改成后台线程，并验证一旦 `main()` 退出，程序立刻终止。

6. 创建一个程序，产生大量 `Timer` 对象，在时间到期后让它们执行一些简单的工作，来演示 `java.util.Timer` 可以伸缩到很大数目（如果你要比较好的效果，可以看看前面“窗体和小应用程序”这一章，那里使用 `Timer` 对象在屏幕上画点，但在此向控制台打印消息已经足够了）。

7. 演示类中的同步控制方法可以调用同一个类上的另一个同步控制方法。后者也能调用同一个类上的第三个同步方法。创建一个单独的 `Thread` 对象调用并第一个同步控制方法。

8. 建立两个 `Thread` 的子类，一个在 `run()` 中启动，然后调用 `wait()`，另一个在 `run()` 中捕获第一个线程对象的引用。它应该在过几秒种后对第一个线程调用 `notifyAll()`，使得第一个线程能打印一条消息。

9. 写一个“忙等待”的例子。一个线程休眠一段时间然后把一个标志设为真。第二个线程在一个 `while` 循环中观察这个标志（这就是“忙等待”），当标志变为真的时候，把这个标志设置为假，并向控制台报告这个变化。注意程序在“忙等待”里花费的时间，然后重写这个程序，这时使用 `wait()` 而不是“忙等待”。

10. 修改 `Restaurant.java`, 使用 `notifyAll()`, 观察程序的行为有何不同。
11. 修改 `Restaurant.java`, 加入多个服务员, 并能够显示谁得到了各自的订单。
12. 修改 `Restaurant.java`, 使得多个服务员向多个厨师产生订单请求, 厨师要准备食品, 并通知那个给他订单的服务员, 你可能需要队列来保存进来的和出去的订单。
13. 修改上个练习, 增加 `Customer` 对象, 它们作为线程实现。客户将向服务员下订单, 服务员把订单给厨师, 厨师将准备好食品并通知相应的服务员, 服务员再把食物交给相应的客户。
14. 修改 `PipedIO.java`, 使得 `Sender` 向一个文本文件里读取和发送文本行。
15. 改变 `DiningPhilosophers.java`, 使得哲学家拿下一根可用的筷子(当哲学家用完筷子之后, 他们把筷子放在一个筷笼里。当哲学家要就餐的时候, 他们就从筷笼里取出两根可用的筷子)。这消除了死锁的可能吗? 你能仅仅通过减少可用的筷子数目就重新引入死锁吗?
16. 从 `java.util.Timer` 继承一个类, 像 `Stopping.java` 里那样实现 `requestStop()` 方法。
17. 修改 `SimpleThread.java`, 使得所有线程在完毕之前接收到 `interrupt()`。
18. 使用 `wait()` 和 `notify()` 解决一个生产者, 一个消费者的问题。生产者不能覆盖消费者的缓冲区, 这在生产的速度大于消费的时候有可能发生。如果消费者的速度大于生产者, 那么它一定不能把同样的数据读取两遍。不要对生产者和消费者的速度做任何假设。

第十四章 创建 Windows 与 Applet 程序

设计中要遵循的一条基本原则是：“让简单的事情变得容易、让困难的事情变得可行”¹。

Java 1.0 版本中的图形用户接口（GUI, graphical user interface），其最初的设计目标是帮助程序员编写在所有平台上都能良好表现的 GUI 程序。遗憾的是，这个目标没有达到。Java 1.0 提供的“抽象窗口程序包”（AWT, Abstract Window Toolkit）在所有的系统上表现得都不太好，而且限制颇多；你只能使用四种字体，也不能访问存在于本地操作系统上的任何成熟的 GUI 组件。Java 1.0 的 AWT 编程模型非常笨拙，并且不是面向对象的。在我课上的一个学生（在 Java 的创建期间他曾经在 Sun 工作）解释了其原因：最初版本的 AWT 是在一个月内分析、设计和实现的。从生产率上看，这确实很惊人，不过这也是能够说明为什么精心设计如此重要的反面教材。

Java 1.1 的 AWT 中引入了事件模型后，情况有所好转，这是一种更清晰的且面向对象的编程模型。随之而来的是 JavaBean 标准的加入，它最初是为了使可视化编程环境的创建变得更容易而引入的构件编程模型。Java 2 (JDK 1.2) 最终完成了从旧式的 Java 1.0 AWT 到新标准的转换：“Java 基础类库”（JFC）几乎替换了所有内容，其中有关 GUI 的部分被称为“Swing”。Swing 是一组易于使用、易于理解的 JavaBean，它能够通过拖放操作（也可以通过手工编写）来创建令人满意的 GUI 程序。软件工业界里的“三次修订”规则（产品在修订三次之后才会成熟）看起来对编程语言也同样适用。

本章只介绍流行的 Java 2 Swing 库，并且合理地假定 Swing 就是 Java 最终的 GUI 库²。如果出于某些原因，你需要使用以前那个“老式”的 AWT（比如你在为以前的代码做支持，或者由于浏览器的限制），那么你可以在本书第一版中（可以从 www.BruceEckel.com 下载，本书配套光盘中也有）找到相关介绍。注意，Java 中仍然存在某些 AWT 构件，有时你必须使用它们。

在本章的前面部分，你将学习到使用 Swing 构件编写 applet 与编写常规程序有何不同，如何编写既是 applet，又是应用程序的代码，这样它就既能通过浏览器，也能通过命令行运行。本书中提供的所有 GUI 例子几乎都能既作为 applet，又作为应用程序运行。

请注意，本章没有完整地介绍 Swing 提供的构件，对于提到的类，也不会讨论其所有方法。这里的讨论尽可能简单。Swing 库非常庞大，本章的目的仅仅是让你打下坚实的基础，熟悉其中的基本概念。如果你需要更复杂的功能，只要深入研究，Swing 几乎可以实现任何你想要的功能。

在这里，我假定你已经从 java.sun.com 下载并安装了 HTML 格式的 JDK 文档，你可以浏览 `javax.swing` 包里面的类，里面有 Swing 库包括方法在内的完整细节。Swing 库在设计上简单明了，所以这些信息一般就足以解决你的问题。已经出版了很多（也很厚）专

¹ 另一种说法称为“最小吃惊原则”，也就是说，“别让用户感到惊讶。”

² 注意，IBM 公司为 Eclipse 项目 (www.Eclipse.org) 开发了一套全新的开源 GUI 库，你可以把它作为 Swing 之外的选择。

门研究 **Swing** 的书籍，如果你需要更深入的内容，比如希望修改 **Swing** 的默认行为，可以参考这些书。

在学习**Swing**的时候，你会发现：

1. **Swing**比你在其它语言或开发环境中见过的编程模型要好得多。**JavaBean**（将在本章快结束的时介绍）是这个库的框架。
2. “**GUI 构造工具**”（可视化编程环境）对于完整的 **Java** 开发环境而言，是必不可少的方面。**JavaBean** 和 **Swing** 使得 **GUI 构造工具**能够在你用图形工具向窗体上放置组件的同时帮助你编写代码。这不仅在编写 **GUI** 程序期间加快了开发速度，而且它使得你可以进行更多的试验，从而具备能够通过试验产生更多设计的能力，以得到更好的设计。
3. **Swing**库设计上的简单性和合理性使得即使你使用**GUI**构造工具而不是手工编写代码，得到的代码仍然是可读的；这解决了以前**GUI**构造工具的大问题，它们很容易产生不可读的代码。

Swing 包含了所有你希望在流行的用户接口中看到的组件：从带图片的按钮，到树形和表格组件。这个库虽然庞大，但它的设计理念是：使用组件的复杂程度与任务的难度相匹配；如果任务很简单，你不用写很多代码，但对于复杂的工作，就要写复杂的代码才行。这意味着它的门槛很低，而且在需要的时候你能得到更强的功能。

Swing 中有一个非常令人称道的原则，称为“正交使用（orthogonality of use）”。意思是，一旦你理解了库中的某个通用概念，你就可以把这个概念应用到其它地方。比如标准的命名约定，我在编写例子的时候，常常在没有翻阅任何资料的情况下，仅仅通过方法的名称就能正确猜测出其功能。从库的设计上来说，这是个相当好的特性。再比如，在通常情况下你可以把一个组件“插”到另一个组件里面，而且能正常工作。

所有组件都是“轻量级的”，所以速度不成问题。为了可移植性考虑，**Swing** 完全用 **Java** 编写。

Swing 自动支持键盘导航；你可以不用鼠标运行 **Swing** 程序，而且这也不用额外编写代码。要支持滚动也不用费工夫；只要在把组件加入窗体之前，先把它包装进一个 **JScrollPane** 组件即可。要使用类似工具提示这样的功能，通常一行代码即可。

Swing还支持一种非常先进的功能，称为“可插式外观（pluggable look and feel）”，意思是UI的外观可以动态改变，以适应不同平台和操作系统下用户的习惯。你甚至可以（不过很难）自己发明一种外观。

applet 基础

Java具有创建applet的能力，它是在Web浏览器中运行的小程序。这种程序必须是安全的，所以其功能就有所限制。不过，对于Web编程的主要问题之一，即客户端编程来说，applet 仍然是一个有力的工具。

Applet 的限制

编写 applet 很受限制，你的行为总是受到 Java 运行时刻安全系统的监视，所以这也常常被称为“在沙箱内（sandbox）”编程。

不过，你也可以摆脱沙箱的限制，编写 applet 之外的常规应用程序，这时你就能使用本地操作系统提供的其它功能了。本书到目前为止，我们一直在写这样的常规应用程序，不过它们属于没有任何图形界面的控制台应用程序。可以使用 Swing 为这些常规应用程序编写图形用户界面。

applet能作什么呢？看看设计人员对它的期望就知道了：增强浏览器中网页的功能。不过，在互联网上冲浪的时候，你很难知道网页是否来自恶意站点，所以浏览器上运行的任何代码必须是安全的。因此，你可能也注意到了，applet最大的限制是：

1. Applet不能访问本地盘。你不会同意在未经允许的情况下，applet读取你的私人信息并在互联网上传播，所以不能让applet读取硬（磁）盘信息。当然也不能向硬（磁）盘写，否则就为病毒打开了方便之门。Java为applet提供了数字签名。当你允许一些经过签名的applet（签名的来源是可以信任的）访问本地机器的时候，就可以放宽一些限制。本章后面你将会看到这种例子，以及有关Java Web Start的例子，它是一种通过互联网把程序安全地传送到客户端的技术。
2. 因为每次都要下载所有内容，其中的每一个类都要单独向服务器发送请求下载，这就使Applet显示起来更慢了。浏览器可能会缓存applet，但这一点并不能得到保证。所以，你应该总是把applet打包到一个JAR（Java ARchive）文件中，它能把applet的所有组件（.class文件以及图形和声音文件）压缩到一个单一文件中，这样就能在一次服务器请求中下载了。JAR文件中的每个单独实体都能使用“数字签名”。

Applet 的优势

要是你能容忍applet的限制，那么你就能享受它所带来的很多好处，尤其是在建造客户/服务器或其它网络应用程序的时候：

1. 不存在安装问题。Applet具有真正的平台无关性（包括播放声音文件的能力，等等。），所以你不必针对不同平台对代码作任何修改，也不需要作任何安装调试的工作。实际上，安装会在用户载入含有applet的网页时自动发生，所以程序的更新也是在后台自动进行的。在传统的客户/服务器系统中，创建和安装新版本的客户端软件总是非常困难。
2. 你也不必担心不好的代码会损坏别人的系统，因为安全机制已经内置于Java语言的核心以及applet的架构之中了。以上两点使得Java可以应用于所谓的“企业内部互联网客户/服务器应用（intranet client/server application）”。这种应用只存在于公司内部或者操作受限的区域内。后者是指用户环境（Web浏览器及其附属程序）可以被指定和/或被控制的区域。

Applet能自动与HTML集成，所以你可以用HTML这个内置的平台无关的文档系统来支持applet。这一点非常有趣，因为把文档当作程序的一部分已经司空见惯，反过来让程序成为文档的一部分确并不常见。

应用框架

类库常常根据功能进行分组。例如，Java标准库中的String和ArrayList就属于这种拿来就能用的库。还有一些专门设计的库，可以作为“积木”（building block）来创建其它类。其中有一种就是应用框架（application framework），其功能是通过提供单个类或一组类来帮助你构建应用。对于某个特定的应用场合，这些类能够产生在每个特定类型的应用中都需要的基本行为。然后，你只要通过从类中继承并重载感兴趣的方法，就能定制自己需要的行为。应用框架的默认控制机制将在恰当的时候调用被你重载的方法。因为应用框架能够把程序中所有与众不同的部分限制在可被重载的方法中，所以它是对“将从不不变的事物与可变的事物相分离”这个原则的典型示例。³

Applet 就是基于应用框架编写的。你需要从 JApplet 类继承，并且重载相应的方法。对于网页上的 applet，下列方法可以对其创建和执行进行控制：

方法	功能
init()	被自动调用，用来对 applet 进行首次初始化，其中包括组件布局。你必须重载此方法。
start()	每当 applet 进入 Web 浏览器视野的时候，此方法被调用，使得 applet 能开启它的常规功能（尤其是被 stop() 关闭的功能）。它还会在 init() 之后被调用。
stop()	每当 applet 离开 Web 浏览器视野的时候，此方法被调用，使得 applet 能关闭它的昂贵操作。它还会在 destroy() 之前被调用。
destroy()	当 applet 不再被使用，要从网页中卸载 applet 以最终释放资源的时候，此方法被调用。

有了这些信息，你就可以编写一个简单的 applet 了：

```
//: c14:Applet1.java
// Very simple applet.
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
}
```

³ 应用框架是“模板方法”（一种设计模式）的应用。


```
} ///:~
```

注意，`applet` 不必具有 `main()` 方法。它已经被包装到应用框架内部了；你只要把初始化代码放进 `init()` 即可。

在这个程序中，唯一的动作就是使用 `JLabel` 类（同其它组件类似，它在 `AWT` 中对应的是 `Label`，所以你在使用 `Swing` 组件的时候，常常会看到开头为“J”的组件名称），在 `applet` 上放置了一个文本标签。这个类的构造器接受一个 `String` 作为参数，可以用它来创建一个标签。在上面的例子中，这个标签被放在窗体上。

在 `init()` 方法中，使用 `add()` 把所有组件加入窗体。你可能会觉得应该能直接调用 `add()`，实际上这也是以前在 `AWT` 中的做法。不过，`Swing` 要求你把所有组件加入到窗体的“内容面板”（`content pane`）中，所以在调用 `add()` 之前你得先调用 `getContentPane()`。

在 Web 浏览器中运行 applet

要运行这个程序，你必须先把它嵌入某个网页，然后使用支持 `Java` 的 `Web` 浏览器来观看这个网页。要把 `applet` 嵌入网页，你得在网页的 `HTML` 源文件⁴中加上特殊标签，这就能告诉网页如何装载并运行这个 `applet`。

以前，这个过程非常简单，那时候 `Java` 本身也很简单，大家都使用相同的技术，浏览器内部对 `Java` 的支持也相同。那时，只需在网页里加入非常简单的 `HTML` 代码即可，像这样：

```
<applet code=Applet1 width=100 height=50>
</applet>
```

后来就是浏览器和语言的激烈争斗，最终我们（程序员和类似的最终用户）失败了。不久，`Sun` 认识到我们不能再指望浏览器来支持正确风格的 `Java`，唯一的解决之道就是提供某种符合浏览器扩展机制的插件。通过使用扩展机制（为了获得竞争优势，浏览器的提供商不会在不破坏所有第三方扩展的情况下禁止这个功能），`Sun` 得以保证 `Java` 技术不会被竞争对手提供的浏览器拒之门外。

对于 `Internet Explorer`，扩展机制就是 `ActiveX` 控件；对于 `Netscape`，扩展机制是 `plug-in`。在你的 `HTML` 代码中，你必须提供二者都支持的标签。好在你能使用 `JDK` 附带的 `HTMLconverter` 工具来自动生成必要的标签。在使用 `HTMLconverter` 工具对前面的 `applet` 标签进行处理之后，以下就是对于 `Applet1` 所得到的最简单的 `HTML` 页面代码：

```
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<OBJECT
    classid = "clsid:CAFEEFAC-0014-0001-0000-ABCDEFEDCBA"
```

⁴ 这里假设读者已经熟悉了 `HTML` 的基本内容，它并不难理解，也有很多参考书和资料。

```

        codebase =
"http://java.sun.com/products/plugin/autodl/jinstall-1_4_1-
windows-i586.cab#Version=1,4,1,0"
        WIDTH = 100 HEIGHT = 50 >
        <PARAM NAME = CODE VALUE = Applet1 >
        <PARAM NAME = "type" VALUE =
"application/x-java-applet;jpi-version=1.4.1">
        <PARAM NAME = "scriptable" VALUE = "false">
        <COMMENT>
        <EMBED
            type =
"application/x-java-applet;jpi-version=1.4.1"
            CODE = Applet1
            WIDTH = 100
            HEIGHT = 50
            scriptable = false
            pluginspage =
"http://java.sun.com/products/plugin/index.html#download">
        <NOEMBED>
        </NOEMBED>
        </EMBED>
        </COMMENT>
    </OBJECT>
    <!--
    <APPLET CODE = Applet1 WIDTH = 100 HEIGHT = 50>
    </APPLET>
    -->
    <!--"END_CONVERTED_APPLET"-->

```

其中有些行太长了，要换行处理才能完整显示。本书源程序（可以从www.BruceEckel.com下载）中的代码可以正常工作，不用担心其换行是否正确。

`code` 值给出了 `applet` 所在的.class 文件的名称。`width` 和 `height` 指定了 `applet` 的初始大小（和以前一样，以像素为单位）。`applet` 标签中还可以加入其它属性：在互联网上查找其它.class 文件的地址（`codebase`），对齐方式（`align`），使 `applet` 之间能够相互通讯的特殊标识符（`name`），以及 `applet` 能够读取的参数信息。参数具有如下形式：

```
<param name="identifier" value = "information">
```

而参数的数量并没有限制。

本书的源代码包（可以从www.BruceEckel.com免费下载）中为书中的每一个 `applet` 都提供了HTML页面，因此，有很多 `applet` 标签的例子，可以从本章源代码对应的 `index.html` 文件中找到。至于把 `applet` 嵌入网页的细节部分，你可以在java.sun.com上查到完整且最新的说明。

使用 Appletviewer 工具

Sun 公司的 JDK 中包括一个名为 Appletviewer 的工具，它能从 HTML 文件中抽取出 “<applet>” 标签，然后只运行这个 applet 而不显示其它的 HTML 内容。这是因为 Appletviewer 会忽略除<applet>标签以外的所有信息，所以你可以把这个标签作为注释放在 Java 源程序里面：

```
// <applet code=MyApplet width=200 height=100></applet>
```

这样，你就可以通过 “appletviewer MyApplet.java” 这个命令来运行 applet，而且你也不用为了测试而专门写一个小的 HTML 文件。例如，你可以在 Applet1.java 里加上作为注释出现的 HTML 标记：

```
//: c14:Applet1b.java
// Embedding the applet tag for Appletviewer.
// <applet code=Applet1b width=100 height=50></applet>
import javax.swing.*;
import java.awt.*;

public class Applet1b extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
} ///:~
```

现在你可以使用下面的命令来运行 applet 了：

```
appletviewer Applet1b.java
```

在本书中，为了简化对applet的测试，使用了这种方式。不久，你将学习另一种编程方式，它可以让你不使用Appletviewer工具，而是从命令行来运行applet。

测试 applet

要想在没有任何网络连接的情况下对applet进行简单的测试，你可以通过启动Web浏览器，然后打开包含applet标签的HTML文件的方式来实现。随着HTML文件被加载，浏览器将发现其中的applet标签，然后根据code指定的值来查找相应的.class文件。当然，它将根据CLASSPATH中指定的路径进行查找，如果.class文件不在这个指定的路径中，浏览器将在状态栏给出错误信息，表明它没有找到.class文件。

如果你要在网站上做这样的测试，情况会稍微复杂一些。首先，你必须有一个网站，对大多数人来说是指远程的互联网服务提供商（ISP）。既然applet只不过是单个或几个文件，所有不需要ISP对Java做任何特殊支持。你还要能够用某种方式把你的HTML文件和.class

文件从本地机器移动到ISP机器的正确目录下。这通常使用支持文件传输协议（FTP）的软件来完成的，有很多以免费软件或自由软件发布的程序可以使用。所以，你所要做的就是通过FTP把文件移动到ISP的机器上，然后使用浏览器连接到网站的HTML文件；如果applet出现并且能够工作，那么一切正常，对吗？

这里有个容易犯错的地方。如果客户端的浏览器不能在服务器上找到.class文件，它会在本地机器上CLASSPATH指定的路径内查找。所以，applet也许没有从服务器上正确载入，但在测试过程中一切正常，因为浏览器在本地机器上找到了.class文件。不过，当其它人连接的时候，他/她的浏览器就不能发现这些文件了。所以当你测试的时候，要确保删除本地机器上有关的.class文件（或.jar文件），以验证它们确实存在于服务器上的正确位置。

我曾经遇到过的最隐秘的错误之一，发生在我简单地把applet放进包（package）中的时候。在上传了HTML文件和applet之后，因为包名称的原因，造成了applet在服务器路径上的冲突。不过，我的浏览器在本地机器上的CLASSPATH路径中找到了这个文件。所以只有我能正常载入这个applet。在applet标签的CODE参数中指定包含包名称在内的完整类名非常重要。在许多已发布的applet例子中，applet并没有放进包。但对于产品级代码，最好还是使用包。

从命令行运行 applet

有时，你希望写一个视窗程序，它要有些别的功能，而不仅仅是嵌入网页。也许你希望它能做一些“常规”程序能做的事情，但仍然具有Java提供的便于移植的能力。在本书前面的章节中，我们编写了许多命令程序，但在某些操作环境（比如Macintosh）中，并没有命令行。正是基于这样的许多原因，我们希望能用Java编写一个非applet的视窗程序。这当然是个合理的要求。

Swing库使你创建的应用能够保持底层操作环境的外观。对于创建视窗应用来说，只有当你使用最新版本的Java及相关工具，使你交付的应用不会令用户迷惑的时候，才具有实际意义⁵。如果因为某种原因，你不得不使用较旧版本的Java，那么在编写重要的视窗程序之前一定要深思熟虑。

你常常会需要编写一个能既能作为视窗，又能作为applet被调用的类。这在测试applet的时候尤其方便，因为与使用浏览器或者Appletviewer相比，通常在命令行下运行得到的applet程序要更快且更容易。

要编写一个能够从控制台命令运行的applet，你只要给applet添加一个main()方法，然后在JFrame⁶中创建一个applet的实例。作为一个简单的例子，我们看一下修改后的Applet1b.java，它可以同时作为应用程序和applet运行：

```
//: c14:Applet1c.java
// An application and an applet.
// <applet code=Applet1c width=100 height=50></applet>
```

⁵ 依我看，在你学习了Swing之后，你就不会在Swing之前的东西上浪费时间了。

⁶ 如前所述，“Frame”这个词已经被AWT用了，所以Swing只好用“JFrame”。

```

import javax.swing.*;
import java.awt.*;

public class Applet1c extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
    // A main() for the application:
    public static void main(String[] args) {
        JApplet applet = new Applet1c();
        JFrame frame = new JFrame("Applet1c");
        // To close the application:
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(applet);
        frame.setSize(100, 50);
        applet.init();
        applet.start();
        frame.setVisible(true);
    }
} ///:~

```

在Applet中只加入了main()方法，其余部分都没有改变。创建applet以后，把它加入JFrame，这样applet就能显示了。

因为在这种情况下，没有浏览器来帮你初始化applet，所以在main()中，applet被明确初始化然后启动。当然，这还不能提供浏览器的所有行为，浏览器还会帮你调用stop()和destory()，但大多数情况下，这样就够了。如果这确实是个问题的话，你可以自己强制调用⁷。

注意最后一行：

```
frame.setVisible(true);
```

如果没有这一行，你在屏幕上就什么也看不到。

一个显示框架

尽管编写既能作为 applet，又能作为应用程序运行的程序很有价值，但要是到处使用的话，不仅容易分散注意力，而且有浪费纸张之嫌。所以，本书余下部分的 Swing 例子将会使用以下的显示框架来代替。

⁷ 在进一步阅读本章之后，这句话才有意义。首先，使用类的静态成员来引用JApplet对象（而不是main()方法中的局部变量），然后在你调用System.exit()之前，在WindowAdapter.windowClosing()方法中调用applet.stop()和applet.destory()。

```

//: com:bruceeckel:swing:Console.java
// Tool for running Swing demos from the
// console, both applets and JFrames.
package com.bruceeckel.swing;
import javax.swing.*;
import java.awt.event.*;

public class Console {
    // Create a title string from the class name:
    public static String title(Object o) {
        String t = o.getClass().toString();
        // Remove the word "class":
        if(t.indexOf("class") != -1)
            t = t.substring(6);
        return t;
    }

    public static void
    run(JFrame frame, int width, int height) {
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(width, height);
        frame.setVisible(true);
    }

    public static void
    run(JApplet applet, int width, int height) {
        JFrame frame = new JFrame(title(applet));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(applet);
        frame.setSize(width, height);
        applet.init();
        applet.start();
        frame.setVisible(true);
    }

    public static void
    run(JPanel panel, int width, int height) {
        JFrame frame = new JFrame(title(panel));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(panel);
        frame.setSize(width, height);
        frame.setVisible(true);
    }
} ///:~

```

这是一个能独立使用的工具类，所以把它放在了`com.bruceeckel.swing`包里面。`Console`类完全由静态方法组成。第一个方法用来从任意对象中抽取出类名（使用RTTI），并且移

除由getClass()方法预先产生的“class”字。它先使用String的indexOf()方法来判断是否含有“class”字，然后用substring()方法来产生没有“class”或空格的新字符串。这个名字将用作视窗的标签，在run()方法里显示。

当JFrame被关闭的时候，setDefaultCloseOperation()方法使得程序退出。如果你不为JFrame调用setDefaultCloseOperation()方法或者编写等价的代码，那么缺省行为将是什么也不做，程序也就不能关闭。

run()方法被重载，以便适用于JApplet, JPanel,和JFrame。注意，它只对于JApplet,才调用init()和start()方法。

现在只要编写main()方法，并加入如下代码，就能从控制台运行applet了：

```
Console.run(new MyClass(), 500, 300);
```

这里后两个参数是显示的宽度与高度。下面是使用Console，修改后的Applet1c.java：

```
//: c14:Applet1d.java
// Console runs applets from the command line.
// <applet code=Applet1d width=100 height=50></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1d extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
    public static void main(String[] args) {
        Console.run(new Applet1d(), 100, 50);
    }
} ///:~
```

这使得在运行例子的时候，既提供了最大程度的灵活性，又消除了重复代码。

创建按钮

建立一个按钮(button)非常简单：你只要把希望在按钮上显示的标签传递给JButton的构造器即可。在后面你会看到一些更有趣的功能，比如在按钮上显示图形。

一般来说，你要在类中为按钮创建一个域，以便以后可以引用这个按钮。

JButton 是一个组件，它有自己的小窗体，这使得它能作为整个更新过程的一部分而自动被重绘。也就是说，你不必明确绘制一个按钮或者别的类似控件；你只要把它们放在窗体上，它们自己可以自动绘制。所以你只要在 `init()` 中，把按钮加入窗体：

```
//: c14:Button1.java
// Putting buttons on an applet.
// <applet code=Button1 width=200 height=50></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button1 extends JApplet {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
    }
    public static void main(String[] args) {
        Console.run(new Button1(), 200, 50);
    }
} ///:~
```

这里引入了一些新内容：在向内容面板里加入任何组件之前，先设置了一个“布局管理器”，这里设置的是 **FlowLayout**。面板能够隐式地通过布局管理器来决定组件在窗体上的位置。**Applet** 通常使用 **BorderLayout** 管理布局，但这里不能使用（在本章后面部分你将会深入学习如何控制窗体的布局），因为它的缺省行为是每加入一个组件，将完全覆盖其它组件。**FlowLayout** 使得组件可以在窗体上从左到右，从上到下连续均匀分布。

捕获事件

你可能注意到了：如果编译并运行了前面的 **applet**，当按下按钮的时候，什么也没发生。现在就是应该深入进来，编写一些代码以决定发生什么动作的时候了。事件驱动编程（包含了许多关于 **GUI** 的内容）的主要内容，就是把事件同处理事件的代码相关联起来。

在 **Swing** 中，这种关联的方式就是通过把接口（图形组件）和实现（当和组件相关的事件发生时，你要执行的代码）清楚地分离而实现的。每个 **Swing** 组件都能够报告其上所有可能发生的事件，并且它能单独报告每种事件。所以，你要是对诸如“鼠标移动到按钮上”这样的事件不感兴趣的话，那么你不注册这样的事件就可以了。这种处理事件驱动编程的方式非常直接和优雅，一旦你理解了其基本概念，就能够很容易将其应用到甚至从未见过 **Swing** 组件之上。实际上，只要是 **JavaBean**（本章后面讨论），这个模式都适用。

首先，我们针对所使用的组件，看看对什么事件感兴趣。对于JButton，这个“感兴趣的事件”就是按钮被按下。为了表明你对按钮按下事件感兴趣，可以调用JButton的addActionListener()方法。这个方法接受一个实现了ActionListener接口的对象作为参数，ActionListener接口只包含了actionPerformed()方法。所以要想把事件处理代码和JButton关联，就先要用一个类实现ActionListener接口，然后把这个类的对象通过addActionListener()方法注册给JButton。这样按钮按下的时候就会调用actionPerformed()方法了（通常这也称为回调）。

但是按钮按下的时候应该有什么结果呢？我们希望看到屏幕有所改变，所以在这里介绍一个新的Swing组件：JTextField。这个组件能够输入文本，在本例里，将由程序插入文本。尽管有很多方法可以创建JTextField，但是最简单的就是告诉其构造器你所希望的文本域宽度。一旦JTextField被放置到了窗体上，你就可以使用setText()方法来修改它的内容了（JTextField还有很多方法，不过你应该先到java.sun.com看一下HTML格式的JDK文档）。下面就是其具体程序：

```
//: c14:Button2.java
// Responding to button presses.
// <applet code=Button2 width=200 height=75></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button2 extends JApplet {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    }
    private ButtonListener bl = new ButtonListener();
    public void init() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(txt);
    }
    public static void main(String[] args) {
```

```

        Console.run(new Button2(), 200, 75);
    }
} ///:~

```

创建JTextField并把它放置在窗体上的步骤，同JButton或者其它Swing组件所采用的步骤相同。这里与前面例子的不同之处在于创建了一个ButtonListener对象，它实现了前面提到过的ActionListener接口。actionPerformed()方法的参数是ActionEvent类型，它包含了事件源和事件的所有信息。本例中，我希望表明是哪个按钮被按下；getSource()方法产生的对象表明了事件的来源，我假设(使用了类型转换)这个对象是JButton。getText()方法将返回按钮上的文本，这个文本被写进JTextField，以证明当按钮按下时代码确实被调用了。

在init()中，使用了addActionListener()方法来将ButtonListener对象注册给按钮。

通常，把ActionListener实现成匿名内部类会显得更方便，尤其是对每个监听器类只使用一个实例的时候更是如此。可以向下面这样修改Button2.java，这里使用了匿名内部类：

```

///: c14:Button2b.java
// Using anonymous inner classes.
// <applet code=Button2b width=200 height=75></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button2b extends JApplet {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    private ActionListener bl = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    };
    public void init() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(txt);
    }
}

```

```

    }
    public static void main(String[] args) {
        Console.run(new Button2b(), 200, 75);
    }
} ///:~

```

本书中的例子将倾向于（可能的话）使用匿名内部类的方式。

文本区域

除了可以有多行文本，以及更多的功能之外，`JTextArea` 与 `JTextField` 很相似。它有一个比较常用的方法 `append()`；使用它你能很容易地在 `JTextArea` 里累积大量文本，比起长期以来在命令程序中把文本打印到标准输出的做法，这使得 `Swing` 编程成为了一种进步（因为可以来回滚动）。作为一个例子，下面的程序使用第 11 章“地名生成器”的输出来填充 `JTextArea`。

```

///: c14:TextArea.java
// Using the JTextArea control.
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class TextArea extends JFrame {
    private JButton
        b = new JButton("Add Data"),
        c = new JButton("Clear Data");
    private JTextArea t = new JTextArea(20, 40);
    private Map m = new HashMap();
    public TextArea() {
        // Use up all the data:
        Collections2.fill(m, Collections2.geography,
            CountryCapitals.pairs.length);
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Iterator it = m.entrySet().iterator();
                while(it.hasNext()) {
                    Map.Entry me = (Map.Entry)(it.next());
                    t.append(me.getKey() + ": " + me.getValue()+"\n");
                }
            }
        });
    }
}

```

```

        c.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText("");
            }
        });
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(new JScrollPane(t));
        cp.add(b);
        cp.add(c);
    }

    public static void main(String[] args) {
        Console.run(new TextArea(), 475, 425);
    }
} ///:~

```

这里使用了JFrame而不是JApplet，因为它要从本地磁盘中读取数据，因此不能作为applet在HTML网页里运行。

在init()方法中，用国家及其首都名称来填充Map。注意，对于其中的两个按钮，因为在程序中你不再需要引用监听器，所以直接创建ActionListener对象并向按钮注册，而没有定义中间变量。“Add Data”按钮格式化并且添加所有数据，“Clear Data”按钮使用setText()方法来清除JTextArea中的所有文本。

在JTextArea被加入到applet的之前，先被包装进了JScrollPane，它被用来在屏幕上的文本太多的时候进行滚动控制。这么做就足以得到完整的滚动功能。由于我曾试图在其它GUI编程环境中得到类似功能，所以我对像JScrollPane这样设计良好、使用简单的组件印象非常深刻。

控制布局

在Java中，组件放置在窗体上的方式可能与你使用过的任何GUI系统都不相同。首先，它完全基于代码；没有用来控制组件布置的“资源”。第二，组件放置在窗体上的方式不是通过绝对坐标控制，而是由“布局管理器”（layout manager）根据组件加入的顺序决定其位置。使用不同的布局管理器，组件的大小、形状和位置将大不相同。此外，布局管理器还可以适应applet或应用程序视窗的大小，所以如果视窗的尺寸改变了，组件的大小、形状和位置也能够做相应的改变。

JApplet, JFrame, JWindow和JDialog都可以通过getContentPane()得到一个容器（Container），用来包含和显示组件。它有一个被称为setLayout()的方法，你可以通过这个方法来选择不同的布局管理器。其它的类，比如JPanel，可以直接包含和显示组件，所以你不用通过内容面板了，就可以直接设置布局管理器。

在本节中，我们将通过在窗体上放置一些按钮的方式，来研究不同的布局管理器（这样最简单）。这里不会捕获任何按钮事件，因为这些例子仅仅是为了演示按钮是如何布局的。

BorderLayout

Applet在缺省情况下的布局方式是BorderLayout（前面有些例子已经把布局管理器改成了FlowLayout）。如果不加入其它指令，它将接受你调用add()方法而加入的组件，把它放置在中央，然后把组件向各个方向拉伸，直到与边框对齐。

不过，BorderLayout 的功能并不仅仅于此。它还具有四个边框区域和一个中央区域的概念。当你向 BorderLayout 管理的面板加入组件的时候，你可以使用重载的 add()方法，它的第一个参数接受一个常量值。这个值可以为以下任何一个：

BorderLayout. NORTH	顶端
BorderLayout. SOUTH	底端
BorderLayout. EAST	右端
BorderLayout. WEST	左端
BorderLayout.CENTER	从中央开始填充，直到与其它组件或边框相遇

如果你没有为组件指定放置的位置，缺省情况下它将被放置到中央。

下面是一个简单的例子。由于 Japplet 在缺省情况下使用的就是 BorderLayout，所以下面的程序中使用了缺省方式：

```
//: c14:BorderLayout1.java
// Demonstrates BorderLayout.
//<applet code=BorderLayout1 width=300 height=250></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BorderLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.add(BorderLayout.NORTH, new JButton("North"));
        cp.add(BorderLayout.SOUTH, new JButton("South"));
        cp.add(BorderLayout.EAST, new JButton("East"));
        cp.add(BorderLayout.WEST, new JButton("West"));
        cp.add(BorderLayout.CENTER, new JButton("Center"));
    }
    public static void main(String[] args) {
```

```

        Console.run(new BorderLayout1(), 300, 250);
    }
} ///:~

```

对于除CENTER以外的所有位置，加入的组件将被沿着一个方向压缩到最小尺寸，同时在另一个方向上拉伸到最大尺寸。不过对于CENTER，组件将在两个方向上同时拉伸，以覆盖中央区域。

FlowLayout

它直接将组件从左到右“流动”到窗体上，直到占满上方的空间，然后向下移动一行，继续流动。

在下面的例子中，先把布局管理器设置为FlowLayout，然后在窗体上放置按钮。你将注意到，在使用FlowLayout的情况下，组件将呈现出“合适”的大小。比如，一个JButton的大小就是其标签的大小。

```

///: c14:FlowLayout1.java
// Demonstrates FlowLayout.
// <applet code=FlowLayout1 width=300 height=250></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class FlowLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            cp.add(new JButton("Button " + i));
    }

    public static void main(String[] args) {
        Console.run(new FlowLayout1(), 300, 250);
    }
} ///:~

```

使用FlowLayout，所有的组件将被压缩到它们的最小尺寸，所以可能会得到令你惊讶的效果。比如，在使用FlowLayout的时候，因为JLabel的尺寸就是其字符串的尺寸，这就使得文本右对齐不会产生任何视觉上的效果。

GridLayout

GridLayout允许你构建一个放置组件的表格，在向表格里面添加组件的时候，它们将按照从左到右，从上到下的顺序加入。在构造器中你要指定需要的行数和列数，它们将均匀分布在窗体上。

```
//: c14:GridLayout1.java
// Demonstrates GridLayout.
// <applet code=GridLayout1 width=300 height=250></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class GridLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(7, 3));
        for(int i = 0; i < 20; i++)
            cp.add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        Console.run(new GridLayout1(), 300, 250);
    }
} ///:~
```

在这个例子中有 21 个空位，但是只加入了 20 个按钮。因为GridLayout并不进行“均衡”处理，所以最后一个空位将被闲置。

GridBagLayout

GridBagLayout提供了强大的控制功能，包括精确判断窗体区域的布局，以及视窗大小变化的时候如何重新放置组件。不过，它也是最复杂的布局管理器，很难以理解。它的目的主要是辅助GUI构造工具（它可能使用GridBagLayout而不是绝对位置来控制布局）自动生成代码。如果你发现自己的设计非常复杂，以至于需要使用GridBagLayout，那么你应该使用GUI构造工具来生成这个设计。如果你觉得自己必须要掌握它的复杂细节，我推荐你参考《Core Java 2 第一卷》，Horstmann & Cornell著（Prentice Hall, 2001），或者是专门研究Swing的书作为起点。

绝对定位

也可以用下面的方式来设置图形组件的绝对位置：

1. 使用`setLayout(null)`方法把容器的布局管理器设置为空。
2. 为每个组件调用`setBounds()`或者`reshape()`方法（取决于语言的版本），为方法传递一个以像素坐标为单位的边界矩形。根据要获得的效果，你可以在构造器或者`paint()`方法中调用这些方法。

某些GUI构造工具大量使用这种方法，不过这通常不是生成代码的最佳方式。

BoxLayout

由于人们在理解和使用 `GridBagLayout` 的时候遇到了很多问题，所以 `Swing` 还提供了 `BoxLayout`，它具有 `GridBagLayout` 的许多好处，却不像 `GridBagLayout` 那么复杂。所以当你需要手工编写布局代码的时候，可以考虑使用它（再次提醒，如果你的设计过于复杂，那么就应该使用 GUI 构造工具来生成布局代码）。`BoxLayout` 使你可以在水平方向或者垂直方向控制组件的位置，并且通过被称为“支架和胶水”（struts and glue）的机制来控制组件的间隔。首先，我们使用与演示其它布局管理器相同的方式，看一看如何直接使用 `BoxLayout`：

```
///  
// Vertical and horizontal BoxLayouts.  
// <applet code=BoxLayout1 width=450 height=200></applet>  
import javax.swing.*;  
import java.awt.*;  
import com.bruceeckel.swing.*;  
  
public class BoxLayout1 extends JApplet {  
    public void init() {  
        JPanel jpv = new JPanel();  
        jpv.setLayout(new BoxLayout(jpv, BoxLayout.Y_AXIS));  
        for(int i = 0; i < 5; i++)  
            jpv.add(new JButton("jpv " + i));  
        JPanel jph = new JPanel();  
        jph.setLayout(new BoxLayout(jph, BoxLayout.X_AXIS));  
        for(int i = 0; i < 5; i++)  
            jph.add(new JButton("jph " + i));  
        Container cp = getContentPane();  
        cp.add(BorderLayout.EAST, jpv);  
        cp.add(BorderLayout.SOUTH, jph);  
    }  
    public static void main(String[] args) {  
        Console.run(new BoxLayout1(), 450, 200);  
    }  
} ///  
~
```


BoxLayout的构造器与其它布局管理器有所不同，你要把“将由**BoxLayout**控制的容器”作为第一个参数，布局的方向作为第二个参数提供给构造器。

为了简化使用方式，有一个被称为 **Box** 的特殊容器，它缺省地使用 **BoxLayout** 作为布局管理器。**Box** 具有两个静态方法来创建水平或者垂直对齐的箱子，下面的例子就使用了 **Box** 来水平和垂直放置组件。

```
//: c14:Box1.java
// Vertical and horizontal BoxLayouts.
// <applet code=Box1 width=450 height=200></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box1 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();
        for(int i = 0; i < 5; i++)
            bv.add(new JButton("bv " + i));
        Box bh = Box.createHorizontalBox();
        for(int i = 0; i < 5; i++)
            bh.add(new JButton("bh " + i));
        Container cp = getContentPane();
        cp.add(BorderLayout.EAST, bv);
        cp.add(BorderLayout.SOUTH, bh);
    }
    public static void main(String[] args) {
        Console.run(new Box1(), 450, 200);
    }
} ///:~
```

一旦创建了**Box**对象，你就可以在把组件添加到内容面板上的时候，把它作为第二个参数去传递了。

支架（**strut**）可以以像素为单位来增加组件之间的空隙。如果要使用支架，你只需在添加组件的时候，把它加入到要格开的组件之间即可：

```
//: c14:Box2.java
// Adding struts.
// <applet code=Box2 width=450 height=300></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box2 extends JApplet {
```

```

public void init() {
    Box bv = Box.createVerticalBox();
    for(int i = 0; i < 5; i++) {
        bv.add(new JButton("bv " + i));
        bv.add(Box.createVerticalStrut(i * 10));
    }
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++) {
        bh.add(new JButton("bh " + i));
        bh.add(Box.createHorizontalStrut(i * 10));
    }
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, bv);
    cp.add(BorderLayout.SOUTH, bh);
}

public static void main(String[] args) {
    Console.run(new Box2(), 450, 300);
}
} ///:~

```

支架能够把组件格开固定的距离，胶水（glue）正好相反；它尽可能地将组件分离开。所以与其说它是“胶水”，不如说它是“弹簧”（它们的设计基于所谓的“弹簧和支架”算法，所以选用的术语有点神秘）。

```

///: c14:Box3.java
// Using Glue.
// <applet code=Box3 width=450 height=300></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box3 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();
        bv.add(new JLabel("Hello"));
        bv.add(Box.createVerticalGlue());
        bv.add(new JLabel("Applet"));
        bv.add(Box.createVerticalGlue());
        bv.add(new JLabel("World"));
        Box bh = Box.createHorizontalBox();
        bh.add(new JLabel("Hello"));
        bh.add(Box.createHorizontalGlue());
        bh.add(new JLabel("Applet"));
        bh.add(Box.createHorizontalGlue());
        bh.add(new JLabel("World"));
    }
}

```

```

        bv.add(Box.createVerticalGlue());
        bv.add(bh);
        bv.add(Box.createVerticalGlue());
        getContentPane().add(bv);
    }
    public static void main(String[] args) {
        Console.run(new Box3(), 450, 300);
    }
} ///:~

```

支架只在一个方向上起效果，但“刚性区域”（**rigid area**）可以在两个方向上固定组件之间的空隙：

```

///: c14:Box4.java
// Rigid areas are like pairs of struts.
// <applet code=Box4 width=450 height=300></applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box4 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();
        bv.add(new JButton("Top"));
        bv.add(Box.createRigidArea(new Dimension(120, 90)));
        bv.add(new JButton("Bottom"));
        Box bh = Box.createHorizontalBox();
        bh.add(new JButton("Left"));
        bh.add(Box.createRigidArea(new Dimension(160, 80)));
        bh.add(new JButton("Right"));
        bv.add(bh);
        getContentPane().add(bv);
    }
    public static void main(String[] args) {
        Console.run(new Box4(), 450, 300);
    }
} ///:~

```

你要注意：“刚性区域”是有些争议的。因为它使用了绝对值，所以有人认为这么做带来的问题比带来的好处要更多。

最好的方式？

Swing功能强大；少量的代码就可以做很多事情。基于学习的目的，本书中的例子相当简单，所以手工编写它们很有意义。你能通过组合简单布局，得到非常多的效果。不过，在某些情况下，手工编写GUI窗体就不太适合了；这样做太复杂，也不是分配编程时间的合理策略。Java和Swing设计者的最初目的就是要使语言和库能对GUI构造工具提供支持，这些工具的明确目的也是为了使你更容易地获取编程经验。只要你理解了布局的方式以及如何处理事件（下面将学习到），那么如何手工放置组件的细节就显得不那么重要了；应该让合适的工具帮你去做这些事情（毕竟，Java是被设计用来提供程序员的生产率的）。

Swing 事件模型

在Swing的事件模型中，组件可以发起（触发）一个事件。每种事件的类型由单独的类表示。当事件被触发时，它将被一个或多个“监听器”接收，监听器负责处理事件。所以，事件发生的地方可以与事件处理的地方分离开。既然是以这种方式使用Swing组件，那么就只需编写组件收到事件时将被调用的代码，所以这是一个分离接口与实现的极佳例子。

所谓事件监听器，就是一个“实现了某种类型的监听器接口的”类的对象。所以程序员要做的就是，先创建一个监听器对象，然后把它注册给触发事件的组件。这个注册动作是通过调用触发事件的组件的addXXXListener()方法来完成的，这里用“XXX”表示监听器所监听的事件类型。通过观察“addListener”方法的名称，就可以很容易地知道其能够处理的事件类型，要是你把所监听事件的类型搞错了，在编译期间就会发现错误。在本章的后面将会学习到，JavaBean也是使用“addListener”方法的名称来判断某个Bean所能处理的事件类型的。

然后，所有的事件处理逻辑都将被置于监听器类的内部。要编写一个监听器类，唯一的要求就是必须实现相应的接口。你可以创建一个全局的监听器类，不过有时写成内部类会更有用。这不仅是因为将监听器类放在它们所服务的用户接口类或者业务逻辑类的内部时，可以在逻辑上对其进行分组；而且还因为（你将在后面看到）内部类对象含有一个对其外部类对象的引用，这就为跨越类和子系统边界的调用提供了一种优雅的方式。

在本章到目前为止的例子中，已经使用了Swing事件模型，本节余下部分将研究这个模型的细节。

事件与监听器的类型

所有Swing组件都具有addXXXListener()和removeXXXListener()方法。这样就可以为每个组件添加或移除相应类型的监听器。注意，每个方法的“XXX”还表示方法能接收的参数，比如addMouseListener(MouseListener m)。下表包含了相互关联的基本事件、监听器、以及通过提供addXXXListener()和removeXXXListener()方法来支持这些事件的基本组件。记住，事件模型是可以扩展的，所以将来你也许会遇到表格里没有列出的事件和监听器。

事件、监听器接口以及“添加”和“移除”方法	支持此事件的组件
ActionEvent ActionListener addActionListener() removeActionListener()	JButton, JList, JPasswordField, JMenuItem 及其派生类, 包括 JCheckBoxMenuItem, JMenu, 和 JPopupMenu
AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener()	JScrollbar 以及你编写的任何实现了 Adjustable 接口的类
ComponentEvent ComponentListener addComponentListener() removeComponentListener()	*Component 及其派生类, 包括 JButton, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, 和 JTextField
ContainerEvent ContainerListener addContainerListener() removeContainerListener()	Container 及其派生类, 包括 JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, 和 JFrame
FocusEvent FocusListener addFocusListener() removeFocusListener()	Component 及其派生类*
KeyEvent KeyListener addKeyListener() removeKeyListener()	Component 及其派生类*
MouseEvent (包括单击和移动) MouseListener addMouseListener() removeMouseListener()	Component 及其派生类*
MouseEvent⁸ (包括单击和移动) MouseMotionListener addMouseMotionListener() removeMouseMotionListener()	Component 及其派生类*
WindowEvent WindowListener	Window 及其派生类, 包括 JDialog, JFileDialog, 和

⁸ 尽管表示鼠标移动的事件似乎很有必要, 但Swing并没有提供MouseMotionEvent这样的事件。MouseEvent包含了鼠标单击和移动的事件, 所以MouseEvent在这个表第二次出现并不是一个错误。

addWindowListener() removeWindowListener()	JFrame
ItemEvent ItemListener addItemListener() removeItemListener()	JCheckBox , JCheckBoxMenuItem , JComboBox , JList , 以及任何实现了 ItemSelectable 接口的类
TextEvent TextListener addTextListener() removeTextListener()	任何从 JtextComponent 继承的类, 包括 JTextArea 和 JTextField

你可以观察到，每种组件所支持的事件类型都是某些固定的类型。为每个组件列出其支持的所有事件是相当困难的。一个比较简单的方法是修改第 10 章的 `ShowMethods.java` 程序，这样它就可以显示出你所输入的任意 Swing 组件所支持的所有事件监听器。

在第 10 章介绍了反射机制，并且使用反射，对指定的类查找其方法，既可以查找所有方法的列表，也可以查找“方法名称符合你所提供的关键字的”部分方法。反射的神奇之处在于它能自动得到一个类的所有方法，而不用遍历类的整个继承层次并在每个层次检查基类。所以，它为编程提供了极具价值并可以节省时间的工具；因为大多数 Java 方法的名称非常详细且具有描述性，所以你能查找出包含了你所感兴趣的关键字的方法名称。当你找到了所要找的方法，就可以在 JDK 文档里查看其细节了。

不过，在第 10 章的时候还没有学习 **Swing**，所以那一章的工具是以命令程序的形式编写的。下面是更好用的 GUI 版本，专门用来查找 Swing 组件里的“addListener”方法：

```

//: c14:ShowAddListeners.java
// Display the "addXXXListener" methods of any Swing class.
// <applet code=ShowAddListeners
// width=500 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.regex.*;
import com.bruceeckel.swing.*;

public class ShowAddListeners extends JApplet {
    private JTextField name = new JTextField(25);
    private JTextArea results = new JTextArea(40, 65);
    private static Pattern addListener =
        Pattern.compile("(add\\w+?Listener\\(\\. *?\\))");
    private static Pattern qualifier =
        Pattern.compile("\\w+\\.");

```

```

class NameL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String nm = name.getText().trim();
        if(nm.length() == 0) {
            results.setText("No match");
            return;
        }
        Class klass;
        try {
            klass = Class.forName("javax.swing." + nm);
        } catch(ClassNotFoundException ex) {
            results.setText("No match");
            return;
        }
        Method[] methods = klass.getMethods();
        results.setText("");
        for(int i = 0; i < methods.length; i++) {
            Matcher matcher =
                addListener.matcher(methods[i].toString());
            if(matcher.find())
                results.append(qualifier.matcher(
                    matcher.group(1)).replaceAll("") + "\n");
        }
    }
}

public void init() {
    NameL nameListener = new NameL();
    name.addActionListener(nameListener);
    JPanel top = new JPanel();
    top.add(new JLabel("Swing class name (press ENTER):"));
    top.add(name);
    Container cp = getContentPane();
    cp.add(BorderLayout.NORTH, top);
    cp.add(new JScrollPane(results));
    // Initial data and test:
    name.setText("JTextArea");
    nameListener.actionPerformed(
        new ActionEvent("", 0, ""));
}

public static void main(String[] args) {
    Console.run(new ShowAddListeners(), 500, 400);
}
} ///:~

```

你要在JtextField中输入要查找的Swing组件类的名称。查找的结果将使用正则表达式进行匹配，最终结果在JTextArea显示。

注意，这里没有使用按钮或者别的组件来让你启动一次查找。这是由于JTextField已经被ActionListener所监听。当你准备好输入并按下回车键后，列表马上就得到了更新。如果文本域的内容非空，将把此内容作为Class.forName()的参数以查找这个类。如果名称不正确，Class.forName()方法将失败，即抛出异常。这个异常被捕获，并把JTextArea内容设置为“No match”。如果输入了正确的名称（注意大小写），Class.forName()将成功返回，然后getMethods()方法将返回一个Method对象数组。

这里使用了两个正则表达式。第一个是 addListener，它查找的模式为：以“add”开头，后面跟任意字母，然后接“Listener”，最后是括号包围的参数列表。注意，整个正则表达式用“非转义”的括号包围，意思是，当发生匹配的时候，它可以被一个正则表达式“组”来访问。在 NameL.ActionPerformed()中，通过把每个 Method 对象都以字符串形式传递给 Pattern.matcher()方法，创建了一个 Matcher 对象。当在此对象上调用 find()的时候，只有发生了匹配，才会返回真，这时，你可以通过调用 group(1)来选择第一个匹配的“表达式组”。这样得到的字符串仍然包含了限定词，为了把限定词剔除掉，需要使用 qualifier Pattern 对象，这与第 9 章 ShowMethods.java 中的作法很相似。

在 init()的末尾，name 被设置了一个初始值，然后触发事件，对初始数据进行一次测试。

这个程序为查询 Swing 组件所支持的事件类型提供了一种便利方式。一旦你知道了组件支持哪些事件，你不用参考任何资料就可以处理这个事件了。你只要：

- 1. 获取事件类的名称，并移除“Event”字，然后将剩下的部分加上单词“Listener”，得到的就是内部类必须实现的监听器接口。
- 2. 实现上面的接口，为你要捕获的事件编写出方法。比如，你可能要监听鼠标移动，所以你可以为MouseEvent接口的mouseMoved()方法编写代码（你必须同时实现接口的其它方法，不过很快你会学到一种简单的方式。）。
- 3. 为第二步编写的监听器类创建一个对象。然后通过调用方法名为“add”前缀加上监听器名称的方法，向组件注册这个对象。比如addMouseListener()。

下面是一些监听器接口：

监听器接口及其适配器	接口支持的方法
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)

ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

这并不是个完整的列表，部分原因是由于事件模型允许你编写自己的事件类型和相应的监听器。所以，你常常会遇到含有自定义事件的库，本章学习到的知识可以帮助你理解如何使用这些事件。

使用监听器适配器来进行简化

你可以发现在上面的表中，某些监听器接口只有一个方法。方法多少本身无关紧要，因为你只有在编写特定方法的时候才会去实现相应接口。不过，具有多个方法的监听器接口使用起来却不太方便。比如，如果你想捕获一个鼠标单击事件（例如，某个按钮还没有替你捕获该事件），那么你就需要为mouseClicked()方法编写代码。但是因为MouseListener是一个接口，所以尽管接口里的其它方法对你来说没有任何用处，但是你还是必须要实现所有这些方法。这非常烦人。

要解决这个问题，某些（不是所有的）含有多于一个方法的监听器接口提供了相应的适配器（adapter），你可以在上面的表中看到。适配器为接口里的每个方法都提供了缺省的空实现。现在你要做的就是从适配器继承，然后重载你需要修改的方法。比如，对于典型的MouseListener，你可以这样：

```
class MyMouseListener extends MouseAdapter {
```

```

    public void mouseClicked(MouseEvent e) {
        // Respond to mouse click...
    }
}

```

适配器的出发点就是为了使编写监听器类变得更容易。

不过，适配器也有某种形式的缺陷。假设你写了一个与前面类似的 `MouseAdapter`：

```

class MyMouseListener extends MouseAdapter {
    public void MouseClicked(MouseEvent e) {
        // Respond to mouse click...
    }
}

```

那么这个适配器将不起作用，而且要想找出原因非常困难，足以让你发疯。因为除了鼠标单击的时候，方法没有被调用以外，程序的编译和运行都十分良好。你能发现这个问题吗？它出在方法的名称上：这里是 `MouseClicked()` 而没有写成 `mouseClicked()`。这个简单的大小写错误导致加入了一个新方法。它不是关闭视窗的时候应该被调用的方法，所以你不能得到希望的结果。所以尽管使用接口有些不方便，但可以保证方法被正确实现。

跟踪多个事件

作为一个有趣的试验，也为了向你证明这些事件确实可以被触发，编写一个能跟踪 `JButton` 额外行为（除了“是否被按下”事件）的 `applet` 显得很有价值。这个例子还向你演示了如何通过继承来编写自己的按钮，因为只有你自己的按钮才对这些事件感兴趣。要得到这个效果，你只要从 `JButton` 继承⁹。

`MyButton` 是 `TrackEvent` 类的内部类，所以 `MyButton` 能访问父窗体，并操作其文本区域，这正是能够把状态信息写到父窗体的文本区域内所必需的。当然，这是一个受限的解决方案，因为这样做 `MyButton` 就被局限为要与 `TrackEvent` 一起使用。这种情况有时称为“高耦合”代码：

```

//: c14:TrackEvent.java
// Show events as they happen.
// <applet code=TrackEvent width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class TrackEvent extends JApplet {

```

⁹ 在Java 1.0/1.1 中，你不能有效地通过继承得到自己的按钮对象。这只是其基础设计中的众多缺陷之一。

```

private HashMap h = new HashMap();
private String[] event = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited", "mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
};

private MyButton
    b1 = new MyButton(Color.BLUE, "test1"),
    b2 = new MyButton(Color.RED, "test2");
class MyButton extends JButton {
    void report(String field, String msg) {
        ((JTextField)h.get(field)).setText(msg);
    }
    FocusListener fl = new FocusListener() {
        public void focusGained(FocusEvent e) {
            report("focusGained", e paramString());
        }
        public void focusLost(FocusEvent e) {
            report("focusLost", e paramString());
        }
    };
    KeyListener kl = new KeyListener() {
        public void keyPressed(KeyEvent e) {
            report("keyPressed", e paramString());
        }
        public void keyReleased(KeyEvent e) {
            report("keyReleased", e paramString());
        }
        public void keyTyped(KeyEvent e) {
            report("keyTyped", e paramString());
        }
    };
    MouseListener ml = new MouseListener() {
        public void mouseClicked(MouseEvent e) {
            report("mouseClicked", e paramString());
        }
        public void mouseEntered(MouseEvent e) {
            report("mouseEntered", e paramString());
        }
        public void mouseExited(MouseEvent e) {
            report("mouseExited", e paramString());
        }
        public void mousePressed(MouseEvent e) {

```

```

        report("mousePressed", e paramString());
    }
    public void mouseReleased(MouseEvent e) {
        report("mouseReleased", e paramString());
    }
};
MouseMotionListener mml = new MouseMotionListener() {
    public void mouseDragged(MouseEvent e) {
        report("mouseDragged", e paramString());
    }
    public void mouseMoved(MouseEvent e) {
        report("mouseMoved", e paramString());
    }
};
public MyButton(Color color, String label) {
    super(label);
    setBackground(color);
    addFocusListener(fl);
    addKeyListener(kl);
    addMouseListener(ml);
    addMouseMotionListener(mml);
}
}
public void init() {
    Container c = getContentPane();
    c.setLayout(new GridLayout(event.length + 1, 2));
    for(int i = 0; i < event.length; i++) {
        JTextField t = new JTextField();
        t.setEditable(false);
        c.add(new JLabel(event[i], JLabel.RIGHT));
        c.add(t);
        h.put(event[i], t);
    }
    c.add(b1);
    c.add(b2);
}
public static void main(String[] args) {
    Console.run(new TrackEvent(), 700, 500);
}
} ///:~

```

在MyButton的构造器中，调用setBackground()方法设置了按钮的颜色。所有的监听器都是通过简单的方法调用进行注册的。

TrackEvent类包含了一个HashMap，它用来存放表示事件类型的字符串，以及对JTextField的引用，每个JTextField用来显示和相应事件有关的信息。当然，这种对应关系可以静态生成而不用放进HashMap，不过我认为你会同意这样做，因为如此一来使用和修改起来就会很容易了。尤其是，如果你要在TrackEvent中加入或移除新的事件类型，那么你只要event数组中加入或移除字符串即可，其它工作将自动完成。

调用report()的时候，将传给它事件的名称以及从事件中得到的参数字符串。它使用了外部类的HaspMap对象h来查找与事件名称相关联的JTextField，然后把第二个参数放进该文本区域。

运行这个例子很有趣，你可以观察到程序中事件发生时的实际情况。

Swing 组件一览

你已经理解了布局管理器和事件模型，现在可以学习如何使用Swing组件了。本节将带你将对Swing组件进行一次并不是很彻底的浏览，因为我们将着重于最常使用的功能。每个例子都保持尽可能的小，这样你就可以很容易地抽出所需代码，将其应用到自己的程序中。

请记住：

1. 在本章的源代码（从www.BruceEckel.com下载）中，你可以很容易的通过浏览HTML页面的方式运行每个例子，并观察结果。
2. 来自java.sun.com的JDK文档内包含了Swing所有的类和方法（这里只演示了其中的一部分）。
3. Swing中的事件使用了很好的命名习惯，所以对于某种类型的事件，很容易猜测出如何编写和安装事件的处理程序。你可以使用本章前面的查找程序ShowAddListeners.java，来帮助查询特定的组件。
4. 当程序变得复杂的时候，你应该过渡到使用GUI构造工具。

按钮（Button）

Swing提供了许多类型的按钮。所有的按钮，包括检查框，单选按钮，甚至菜单项都是从AbstractButton（因为包含了菜单项，所以将其命名为“AbstractSelector” 或者其它概括的名字，似乎更恰当一些）继承而来。很快你就会看到菜单项的使用，下面的例子演示了几种按钮：

```
//: c14:Buttons.java
// Various Swing buttons.
// <applet code=Buttons width=350 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.plaf.basic.*;
```

```

import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Buttons extends JApplet {
    private JButton jb = new JButton("JButton");
    private BasicArrowButton
        up = new BasicArrowButton(BasicArrowButton.NORTH),
        down = new BasicArrowButton(BasicArrowButton.SOUTH),
        right = new BasicArrowButton(BasicArrowButton.EAST),
        left = new BasicArrowButton(BasicArrowButton.WEST);
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(jb);
        cp.add(new JToggleButton("JToggleButton"));
        cp.add(new JCheckBox("JCheckBox"));
        cp.add(new JRadioButton("JRadioButton"));
        JPanel jp = new JPanel();
        jp.setBorder(new TitledBorder("Directions"));
        jp.add(up);
        jp.add(down);
        jp.add(left);
        jp.add(right);
        cp.add(jp);
    }
    public static void main(String[] args) {
        Console.run(new Buttons(), 350, 100);
    }
} ///:~

```

程序开始加入了来自`javax.swing.plaf.basic`的`BasicArrowButton`，然后又加入了几种不同类型的按钮。运行例子，你会发现触发器按钮（`JToggleButton`）能保持自身最新的状态：按下或者弹出。不过检查框和单选按钮看起来就差不多，也都是在开和关之间切换（它们都是从`JToggleButton`继承而来）。

按钮组（Button group）

要想让单选按钮表现出某种“排它”行为，你必须把它们加入到一个“按钮组”（`ButtonGroup`）中。不过，正如下面的例子所演示的，任何`AbstractButton`对象都可以加入到按钮组中。

为了避免重复编写大量的代码，下面这个例子使用了反射功能来产生几组不同类型的按钮。注意 `makeBPanel()` 方法，它用来创建一个按钮组和一个 `JPanel`，此方法的第二个参数

是一个字符串数组。针对其中每个字符串，将创建一个由第一参数所代表的按钮实例，然后将此按钮加入到 `Jpanel` 中：

```
//: c14:ButtonGroups.java
// Uses reflection to create groups
// of different types of AbstractButton.
// <applet code=ButtonGroups width=500 height=300></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.lang.reflect.*;
import com.bruceeckel.swing.*;

public class ButtonGroups extends JApplet {
    private static String[] ids = {
        "June", "Ward", "Beaver",
        "Wally", "Eddie", "Lumpy",
    };

    static JPanel makeBPanel(Class klass, String[] ids) {
        ButtonGroup bg = new ButtonGroup();
        JPanel jp = new JPanel();
        String title = klass.getName();
        title = title.substring(title.lastIndexOf('.') + 1);
        jp.setBorder(new TitledBorder(title));
        for(int i = 0; i < ids.length; i++) {
            AbstractButton ab = new JButton("failed");
            try {
                // Get the dynamic constructor method
                // that takes a String argument:
                Constructor ctor =
                    klass.getConstructor(new Class[] {String.class});
                // Create a new object:
                ab = (AbstractButton)
                    ctor.newInstance(new Object[] { ids[i] });
            } catch(Exception ex) {
                System.err.println("can't create " + klass);
            }
            bg.add(ab);
            jp.add(ab);
        }
        return jp;
    }

    public void init() {
```

```

        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(makeBPanel(JButton.class, ids));
        cp.add(makeBPanel(JToggleButton.class, ids));
        cp.add(makeBPanel(JCheckBox.class, ids));
        cp.add(makeBPanel(JRadioButton.class, ids));
    }

    public static void main(String[] args) {
        Console.run(new ButtonGroups(), 500, 300);
    }
} ///:~

```

边框的标题是从类的名称中得到的，并且去掉了其中的路径信息。**AbstractButton**被初始化为一个标签为“Failed”的 **JButton**对象，所以即使你忽略了异常，仍旧能够在屏幕上观察到失败。**getConstructor()**方法产生一个**Constructor**对象，这个构造器对象接受“传递给**getConstructor()**的Class数组里面指定的类型”所组成的数组作为参数。然后你要做的就是调用**newInstance()**，并且把包含真正参数的**Object**数组传递给它，在本例中就是**ids**数组中的每个字符串。

这就使这个简单的过程变得稍显复杂了。要想通过按钮得到“排它”行为，就得先创建一个按钮组，然后把你希望具有“排它”行为的按钮加入到这个按钮组中。运行程序，你将发现除了 **JButton**以外，其它按钮都具有了这种“排它”行为。

图标 (icon)

你可以在 **JLabel**或者任何从**AbstractButton**（包括 **JButton**, **JCheckBox**, **JRadioButton**, 以及几种 **JMenuItem**）继承的组件中使用 **Icon**。和 **JLabel**一起使用 **Icon**的做法非常直接（后面有例子）。下面的例子还研究了与按钮（或者从按钮继承的组件）搭配使用图标的所有方式。

你可以使用任何想用的gif文件，本例中使用的文件来自于本书的源代码包（可以从www.BruceEckel.com下载）。要打开一个文件并且得到图形，只需创建一个 **ImageIcon**对象并且把文件名传递给它即可。然后，你就能在程序中使用得到的图标了。

```

///: c14:Faces.java
// Icon behavior in Jbuttons.
// <applet code=Faces width=400 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Faces extends JApplet {

```



```

private static Icon[] faces;
private JButton jb, jb2 = new JButton("Disable");
private boolean mad = false;
public void init() {
    faces = new Icon[] {
        new ImageIcon(getClass().getResource("Face0.gif")),
        new ImageIcon(getClass().getResource("Face1.gif")),
        new ImageIcon(getClass().getResource("Face2.gif")),
        new ImageIcon(getClass().getResource("Face3.gif")),
        new ImageIcon(getClass().getResource("Face4.gif")),
    };
    jb = new JButton("JButton", faces[3]);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    jb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if(mad) {
                jb.setIcon(faces[3]);
                mad = false;
            } else {
                jb.setIcon(faces[0]);
                mad = true;
            }
            jb.setVerticalAlignment(JButton.TOP);
            jb.setHorizontalAlignment(JButton.LEFT);
        }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(faces[1]);
    jb.setPressedIcon(faces[2]);
    jb.setDisabledIcon(faces[4]);
    jb.setToolTipText("Yow!");
    cp.add(jb);
    jb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if(jb.isEnabled()) {
                jb.setEnabled(false);
                jb2.setText("Enable");
            } else {
                jb.setEnabled(true);
                jb2.setText("Disable");
            }
        }
    });
}

```

```

        cp.add(jb2);
    }
    public static void main(String[] args) {
        Console.run(new Faces(), 400, 200);
    }
} ///:~

```

许多Swing组件的构造器都接受Icon类型的参数，你也可以使用setIcon()来加入或者改变图标。本例还演示了如何让JButton（或者任何AbstractButton类型）在各种情况下显示不同的图标：按下、禁止，或者“浮动”（鼠标移动到按钮上没有点击的时候）。这使得按钮具有了相当不错的动画效果。

工具提示（Tool tip）

前面的例子给按钮添加了一个“工具提示”。你用来创建用户接口的类，绝大多数都是从JComponent继承而来的，它们包含了一个setToolTipText(String)方法。所以，对于你要放置在窗体上的组件，基本上所要做的就只是写（对于任何JComponent子类对象jc）：

```
jc.setToolTipText("My tip");
```

当鼠标停留在这个JComponent上经过一段预先指定的时间之后，在鼠标旁边弹出的小方框里就会出现你所设定的文字。

文本域（Text Field）

下面的例子演示了JTextField组件具有的其它功能：

```

//: c14:TextFields.java
// Text fields and Java events.
// <applet code=TextFields width=375 height=125></applet>
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class TextFields extends JApplet {
    private JButton
        b1 = new JButton("Get Text"),
        b2 = new JButton("Set Text");
    private JTextField
        t1 = new JTextField(30),

```

```

        t2 = new JTextField(30),
        t3 = new JTextField(30);
private String s = new String();
private UpperCaseDocument ucd = new UpperCaseDocument();
public void init() {
    t1.setDocument(ucd);
    ucd.addDocumentListener(new T1());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    DocumentListener dl = new T1();
    t1.addActionListener(new T1A());
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(t1);
    cp.add(t2);
    cp.add(t3);
}
class T1 implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
        t2.setText(t1.getText());
        t3.setText("Text: " + t1.getText());
    }
    public void removeUpdate(DocumentEvent e) {
        t2.setText(t1.getText());
    }
}
class T1A implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
        t3.setText("t1 Action Event " + count++);
    }
}
class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(t1.getSelectedText() == null)
            s = t1.getText();
        else
            s = t1.getSelectedText();
        t1.setEditable(true);
    }
}
}

```

```

class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        ucd.setUpperCase(false);
        t1.setText("Inserted by Button 2: " + s);
        ucd.setUpperCase(true);
        t1.setEditable(false);
    }
}

public static void main(String[] args) {
    Console.run(new TextFields(), 375, 125);
}
}

class UpperCaseDocument extends PlainDocument {
    private boolean upperCase = true;
    public void setUpperCase(boolean flag) {
        upperCase = flag;
    }
    public void
    insertString(int offset, String str, AttributeSet attSet)
    throws BadLocationException {
        if(upperCase) str = str.toUpperCase();
        super.insertString(offset, str, attSet);
    }
} ///:~

```

当JTextField对象t1 的动作监听器被触发时，JTextField对象t3 是要被报知该事件的对象之一。你可以观察到，只有当你按下“回车”键的时候，JTextField的动作监听器才会被触发。

JTextField对象t1 关联了多个监听器。T1 是一个DocumentListener，用来对“文档”（本例中指JTextField的内容）中的变化作出反应。它将自动把t1 的文本复制到t2。此外，t1 的“文档”被设置成PlainDocument的子类对象，就是代码中的UpperCaseDocument，它把所有字符强制变成大写。此外，它还能自动检测退格键并作出对其执行删除，调整插字符，以及处理你所期望的所有行为。

边框（Border）

JComponent 有一个 setBorder()方法，它允许你为任何可视组件设置各种边框。下面的例子使用 showBorder()方法演示了一些可用的边框。此方法先创建一个 JPanel，然后设置相应的边框。此外，它还使用 RTTI(运行时类型识别)来得到正在使用的边框名称（去掉了路径信息），然后把这个名称放进面板中间的一个 JLabel 中：

```
///: c14:Borders.java
```

```

// Different Swing borders.
// <applet code=Borders width=500 height=300></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Borders extends JApplet {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
        jp.add(new JLabel(nm, JLabel.CENTER),
            BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(2, 4));
        cp.add(showBorder(new TitledBorder("Title")));
        cp.add(showBorder(new EtchedBorder()));
        cp.add(showBorder(new LineBorder(Color.BLUE)));
        cp.add(showBorder(
            new MatteBorder(5, 5, 30, 30, Color.GREEN)));
        cp.add(showBorder(
            new BevelBorder(BevelBorder.RAISED)));
        cp.add(showBorder(
            new SoftBevelBorder(BevelBorder.LOWERED)));
        cp.add(showBorder(new CompoundBorder(
            new EtchedBorder(),
            new LineBorder(Color.RED))));
    }
    public static void main(String[] args) {
        Console.run(new Borders(), 500, 300);
    }
} ///:~

```

你还可以自己编写边框代码，然后把它们加入到按钮、标签等任何从JComponent继承的组件中去。

滚动面板（JScrollPane）

大多数时候你只想使用一个 JScrollPane 去执行滚动控制，不过你还可以去控制有哪些滚动条是可用的：水平的、垂直的、两者都用还是两者都不用：

```
///  
//: c14:JScrollPane.java  
// Controlling the scrollbars in a JScrollPane.  
// <applet code=JScrollPane width=300 height=725></applet>  
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.border.*;  
import com.bruceeckel.swing.*;  
  
public class JScrollPane extends JApplet {  
    private JButton  
        b1 = new JButton("Text Area 1"),  
        b2 = new JButton("Text Area 2"),  
        b3 = new JButton("Replace Text"),  
        b4 = new JButton("Insert Text");  
    private JTextArea  
        t1 = new JTextArea("t1", 1, 20),  
        t2 = new JTextArea("t2", 4, 20),  
        t3 = new JTextArea("t3", 1, 20),  
        t4 = new JTextArea("t4", 10, 10),  
        t5 = new JTextArea("t5", 4, 20),  
        t6 = new JTextArea("t6", 10, 10);  
    private JScrollPane  
        sp3 = new JScrollPane(t3,  
            JScrollPane.VERTICAL_SCROLLBAR_NEVER,  
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),  
        sp4 = new JScrollPane(t4,  
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,  
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),  
        sp5 = new JScrollPane(t5,  
            JScrollPane.VERTICAL_SCROLLBAR_NEVER,  
            JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS),  
        sp6 = new JScrollPane(t6,  
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,  
            JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);  
    class BIL implements ActionListener {  
        public void actionPerformed(ActionEvent e) {  
            t5.append(t1.getText() + "\n");  
        }  
    }  
}
```

```

    }
    class B2L implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t2.setText("Inserted by Button 2");
            t2.append(": " + t1.getText());
            t5.append(t2.getText() + "\n");
        }
    }
    class B3L implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String s = " Replacement ";
            t2.replaceRange(s, 3, 3 + s.length());
        }
    }
    class B4L implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t2.insert(" Inserted ", 10);
        }
    }
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        // Create Borders for components:
        Border brd = BorderFactory.createMatteBorder(
            1, 1, 1, 1, Color.BLACK);
        t1.setBorder(brd);
        t2.setBorder(brd);
        sp3.setBorder(brd);
        sp4.setBorder(brd);
        sp5.setBorder(brd);
        sp6.setBorder(brd);
        // Initialize listeners and add components:
        b1.addActionListener(new B1L());
        cp.add(b1);
        cp.add(t1);
        b2.addActionListener(new B2L());
        cp.add(b2);
        cp.add(t2);
        b3.addActionListener(new B3L());
        cp.add(b3);
        b4.addActionListener(new B4L());
        cp.add(b4);
        cp.add(sp3);
        cp.add(sp4);
    }

```

```

        cp.add(sp5);
        cp.add(sp6);
    }

    public static void main(String[] args) {
        Console.run(new JScrollPanes(), 300, 725);
    }
} ///:~

```

通过向JScrollPane的构造器中传入不同的参数，可以控制哪些滚动条是可用的。本例还使用边框进行了一些修饰。

一个迷你编辑器

通过使用 JTextPane 组件，不用费多少工夫，就能支持许多编辑操作。下面的例子是对这个组件的简单应用，其中忽略了该组件所能提供的其他的大量功能：

```

///: c14:TextPane.java
// The JTextPane control is a little editor.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class TextPane extends JFrame {
    private JButton b = new JButton("Add Text");
    private JTextPane tp = new JTextPane();
    private static Generator sg =
        new Arrays2.RandStringGenerator(7);
    public TextPane() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for(int i = 1; i < 10; i++)
                    tp.setText(tp.getText() + sg.next() + "\n");
            }
        });
        Container cp = getContentPane();
        cp.add(new JScrollPane(tp));
        cp.add(BorderLayout.SOUTH, b);
    }

    public static void main(String[] args) {
        Console.run(new TextPane(), 475, 425);
    }
} ///:~

```


按钮的功能只是添加一些随机生成的文本。JTextPane的目的是提供即时编辑文本的功能，所以这里没有append()方法。在本例中（坦白地说，这不是一个可以发挥JTextPane功能的好例子），文本必须被捕获并修改，然后使用setText()将其放回到文本面板中。

如前所述，applet的缺省布局方式是使用BorderLayout。如果你向面板中加入组件的时候没有指定任何细节，那么它将从中间开始填充面板直到与边框对齐。不过，要是你像本例中这样指定了一种环绕区域（NORTH, SOUTH, EAST或者WEST），组件将被调整到这个区域内；这里，按钮将处于屏幕的底部。

注意，JTextPane还有诸如自动换行的内置功能。其它的功能你可以参考JDK文档。

检查框（Check box）

检查框提供了一种用以进行开/关选择的方式。它包含了一个小方框和一个标签。这个方框中通常是有一个“x”标记（或者其它能表明被选中的标记）或者为空，这取决于检查框是否被选中。

通常你会使用接受标签作为参数的构造器来创建JCheckBox。你可以获取和设置状态，也可以获取和设置其标签，甚至可以在JCheckBox对象已经建立之后改变标签。

当JCheckBox被选中或清除时，将发生一个事件，你可以用与对付按钮相同的方法来捕获这个事件：使用ActionListener。在下面的例子中，将枚举所有被选中的检查框，然后在JTextArea里显示：

```
//: c14:CheckBoxes.java
// Using JCheckBoxes.
// <applet code=CheckBoxes width=200 height=200></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CheckBoxes extends JApplet {
    private JTextArea t = new JTextArea(6, 15);
    private JCheckBox
        cb1 = new JCheckBox("Check Box 1"),
        cb2 = new JCheckBox("Check Box 2"),
        cb3 = new JCheckBox("Check Box 3");
    public void init() {
        cb1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("1", cb1);
            }
        });
    }
}
```

```

        cb2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("2", cb2);
            }
        });
        cb3.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("3", cb3);
            }
        });
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(new JScrollPane(t));
        cp.add(cb1);
        cp.add(cb2);
        cp.add(cb3);
    }

    private void trace(String b, JCheckBox cb) {
        if(cb.isSelected())
            t.append("Box " + b + " Set\n");
        else
            t.append("Box " + b + " Cleared\n");
    }

    public static void main(String[] args) {
        Console.run(new CheckBoxes(), 200, 200);
    }
} ///:~

```

`trace()`方法中使用了`append()`，用来把检查框的名称及其状态显示到`JTextArea`中，所以你看看到一个积累的检查框列表，包括了检查框名称及其状态。

单选按钮（Radio button）

GUI编程中单选按钮的概念来源于电子按钮发明之前汽车无线电上的机械按钮；当你按下其中的一个，其它被按下的按钮将被弹出。所以，单选按钮用来强制你在多个选项中只能选择一个。

要设置一组关联的**JRadioButton**，你要做的就是将它们加入到一个**ButtonGroup**中（窗体上可以有任意数目的按钮组）。可以设置其中的一个按钮状态为选中(**true**)（在构造器的第二个参数中设置）。如果你把多个单选按钮的状态都设置为选中，那么只有最后设置的那个有效。

下面的简单例子使用了单选按钮。注意，你可以像其它按钮那样捕获单选按钮的事件：

```

/// c14:RadioButtons.java
// Using JRadioButtons.
// <applet code=RadioButtons width=200 height=100></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class RadioButtons extends JApplet {
    private JTextField t = new JTextField(15);
    private ButtonGroup g = new ButtonGroup();
    private JRadioButton
        rb1 = new JRadioButton("one", false),
        rb2 = new JRadioButton("two", false),
        rb3 = new JRadioButton("three", false);
    private ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText("Radio button " +
                ((JRadioButton)e.getSource()).getText());
        }
    };
    public void init() {
        rb1.addActionListener(al);
        rb2.addActionListener(al);
        rb3.addActionListener(al);
        g.add(rb1); g.add(rb2); g.add(rb3);
        t.setEditable(false);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        cp.add(rb1);
        cp.add(rb2);
        cp.add(rb3);
    }
    public static void main(String[] args) {
        Console.run(new RadioButtons(), 200, 100);
    }
} ///~

```

这里使用了文本域来显示状态。因为它仅仅用来显示而不是收集数据，所以被设置成为不可编辑（non-editable）。因此，这是可以用来代替JLabel的一种选择。

组合框（下拉列表）（Combo box（drop-down list））

与一组单选按钮的功能类似，下拉列表也用来强制用户从一组可能的元素中只选择一个。不过，这种方法更加紧凑，而且在不会使用户感到迷惑的前提下，改变下拉列表中的内容更容易（你也可以动态改变单选按钮，不过这显然不合适）。

缺省状态下，JComboBox组件与Windows操作系统下的组合框并不完全相同，后者允许你从列表中选择或者自己输入。要想得到这样的行为，你必须调用setEditable()方法。使用JComboBox，你能且只能从列表中选择一个元素。在下面的例子里，JComboBox开始已经具有一些元素，然后当一个按钮按下时，将向组合框中加入新的元素。

```
//: c14:ComboBoxes.java
// Using drop-down lists.
// <applet code=ComboBoxes width=200 height=125></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class ComboBoxes extends JApplet {
    private String[] description = {
        "Ebullient", "Obtuse", "Recalcitrant", "Brilliant",
        "Somnolent", "Timorous", "Florid", "Putrescent"
    };
    private JTextField t = new JTextField(15);
    private JComboBox c = new JComboBox();
    private JButton b = new JButton("Add items");
    private int count = 0;
    public void init() {
        for(int i = 0; i < 4; i++)
            c.addItem(description[count++]);
        t.setEditable(false);
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(count < description.length)
                    c.addItem(description[count++]);
            }
        });
        c.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText("index: " + c.getSelectedIndex() + " " +
                    ((JComboBox)e.getSource()).getSelectedItem());
            }
        });
    }
}
```

```

        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        cp.add(c);
        cp.add(b);
    }

    public static void main(String[] args) {
        Console.run(new ComboBoxes(), 200, 125);
    }
} ///:~

```

上例中的JtextField被用来显示“被选中的索引”（当前被选中元素的序号）和组合框中被选中元素的文本。

列表框（list box）

列表框和JComboBox组合框明显不同，这不仅仅体现在外观上。当你激活JComboBox组合框时，会出现下拉列表，而JList总是在屏幕上占据固定行数的空间，大小也不会改变。如果你要得到列表框中被选中的项目，你只需调用getSelectedValues()，它可以产生一个字符串数组，里面是被选中的项目名称。

Jlist组件允许多重选择：要是你按住“Ctrl”键，连续在多个项目上单击，那么原先被选中的项目仍旧保持选中状态，所以你可以选中任意多的项目。如果你选中了某个项目，任何按住“Shift”键并单击另一个项目，那么这两个项目之间的所有项目都将被选中。要从选中的项目组中去掉一个，可以按住“Ctrl”键在此项目上单击。

```

///: c14:List.java
// <applet code=List width=250 height=375></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class List extends JApplet {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };

    private DefaultListModel lItems=new DefaultListModel();
    private JList lst = new JList(lItems);
    private JTextArea t =

```

```

        new JTextArea(flavors.length, 20);
private JButton b = new JButton("Add Item");
private ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(count < flavors.length) {
            lItems.add(0, flavors[count++]);
        } else {
            // Disable, since there are no more
            // flavors left to be added to the List
            b.setEnabled(false);
        }
    }
};
private ListSelectionListener ll =
    new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e) {
            if(e.getValueIsAdjusting()) return;
            t.setText("");
            Object[] items=lst.getSelectedValues();
            for(int i = 0; i < items.length; i++)
                t.append(items[i] + "\n");
        }
    };
private int count = 0;
public void init() {
    Container cp = getContentPane();
    t.setEditable(false);
    cp.setLayout(new FlowLayout());
    // Create Borders for components:
    Border brd = BorderFactory.createMatteBorder(
        1, 1, 2, 2, Color.BLACK);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Add the first four items to the List
    for(int i = 0; i < 4; i++)
        lItems.addElement(flavors[count++]);
    // Add items to the Content Pane for Display
    cp.add(t);
    cp.add(lst);
    cp.add(b);
    // Register event listeners
    lst.addListSelectionListener(ll);
    b.addActionListener(bl);
}

```

```

        public static void main(String[] args) {
            Console.run(new List(), 250, 375);
        }
    } ///:~

```

你可以观察到列表框周围添加了边框。

如果你只是要把一个字符串数组加入JList，那么有一个更简单的办法：只要把数组传递给JList的构造器，就能自动构造列表框。上例中使用“列表模型”（list model）的唯一原因是，这样可以在程序执行的过程中操纵列表框。

JList本身没有对滚动提供直接的支持。当然，你要做的只是把JList包装进JScrollPane，它将自动帮你处理其中的细节。

页签面板

JtabbedPane 允许你创建“页签式的对话框”，它能沿着窗体的一条边放置类似文件夹上的页签，你所要做的只是在页签上单击，用以向前进入到另一个不同的对话框中。

```

///: c14:TabbedPane.java
// Demonstrates the Tabbed Pane.
// <applet code=TabbedPane width=350 height=200></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class TabbedPane extends JApplet {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private JTabbedPane tabs = new JTabbedPane();
    private JTextField txt = new JTextField(20);
    public void init() {
        for(int i = 0; i < flavors.length; i++)
            tabs.addTab(flavors[i],
                new JButton("Tabbed pane " + i));
        tabs.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                txt.setText("Tab selected: " +
                    tabs.getSelectedIndex());
            }
        });
    }
}

```

```

    });
    Container cp = getContentPane();
    cp.add(BorderLayout.SOUTH, txt);
    cp.add(tabs);
}
public static void main(String[] args) {
    Console.run(new TabbedPane(), 350, 200);
}
} ///:~

```

在Java中，使用某种形式的“标签面板”机制是非常重要的。因为在applet编程中，使用弹出式对话框是会被劝阻的，这种劝阻体现为对在applet中弹出的任何对话框都会添加一条小小的警告。

在运行程序的时候你可以观察到，如果页签太多，以至于在一行中放不下它们的时候，JTabbedPane能够自动把页签叠起来。如果是在命令行方式下运行该程序，你可以通过调整窗口大小来进行观察。

消息框（Message box）

视窗环境下通常包含了一组标准的消息框，使你能够快速地把消息通知给用户，或者是从用户那里得到信息。在Swing中，这些消息框包含在JOptionPane组件里。你有许多选择（有些非常高级），但最常用的可能就是消息对话框和确认对话框，它们分别可以通过调用静态的JOptionPane.showMessageDialog()和JOptionPane.showConfirmDialog()方法得到。下面的例子演示了JOptionPane中一些可用的消息框。

```

///: c14:MessageBoxes.java
// Demonstrates JOptionPane.
// <applet code=MessageBoxes width=200 height=150></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class MessageBoxes extends JApplet {
    private JButton[] b = {
        new JButton("Alert"), new JButton("Yes/No"),
        new JButton("Color"), new JButton("Input"),
        new JButton("3 Vals")
    };
};
private JTextField txt = new JTextField(15);
private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {

```



```

String id = ((JButton)e.getSource()).getText();
if(id.equals("Alert"))
    JOptionPane.showMessageDialog(null,
        "There's a bug on you!", "Hey!",
        JOptionPane.ERROR_MESSAGE);
else if(id.equals("Yes/No"))
    JOptionPane.showConfirmDialog(null,
        "or no", "choose yes",
        JOptionPane.YES_NO_OPTION);
else if(id.equals("Color")) {
    Object[] options = { "Red", "Green" };
    int sel = JOptionPane.showOptionDialog(
        null, "Choose a Color!", "Warning",
        JOptionPane.DEFAULT_OPTION,
        JOptionPane.WARNING_MESSAGE, null,
        options, options[0]);
    if(sel != JOptionPane.CLOSED_OPTION)
        txt.setText("Color Selected: " + options[sel]);
} else if(id.equals("Input")) {
    String val = JOptionPane.showInputDialog(
        "How many fingers do you see?");
    txt.setText(val);
} else if(id.equals("3 Vals")) {
    Object[] selections = {"First", "Second", "Third"};
    Object val = JOptionPane.showInputDialog(
        null, "Choose one", "Input",
        JOptionPane.INFORMATION_MESSAGE,
        null, selections, selections[0]);
    if(val != null)
        txt.setText(val.toString());
}
}
};

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < b.length; i++) {
        b[i].addActionListener(al);
        cp.add(b[i]);
    }
    cp.add(txt);
}

public static void main(String[] args) {
    Console.run(new MessageBoxes(), 200, 200);
}

```

```
    }  
} ///:~
```

为了能够只编写一个单一的ActionListener，我使用了检查按钮上字符串标签的方法来判断事件的来源，这有点冒险。其问题在于标签可能会有拼写错误，尤其是大小写，这种Bug很难发现。

注意，showOptionDialog()和showInputDialog()方法提供了返回对象，此对象包含了用户输入的信息。

菜单（Menu）

每个组件都能够持有一个菜单，包括 JApplet, JFrame, JDialog以及它们的子类。它们的 setJMenuBar()方法接受一个JMenuBar对象（每个组件只能持有一个JMenuBar对象）作为参数。你先把JMenu对象添加到JmenuBar中，然后把JmenuItem添加到Jmenu中。每个JmenuItem都能有一个相关联的ActionListener，用来捕获菜单项被选中时触发的事件。

与使用“资源文件”的系统不同，在 Java 和 Swing 中你必须在源代码中构造所有的菜单。下面是个非常简单的菜单例子：

```
///: c14:SimpleMenus.java  
// <applet code=SimpleMenus width=200 height=75></applet>  
import javax.swing.*;  
import java.awt.event.*;  
import java.awt.*;  
import com.bruceeckel.swing.*;  
  
public class SimpleMenus extends JApplet {  
    private JTextField t = new JTextField(15);  
    private ActionListener al = new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            t.setText(((JMenuItem)e.getSource()).getText());  
        }  
    };  
    private JMenu[] menus = {  
        new JMenu("Winken"), new JMenu("Blinken"),  
        new JMenu("Nod")  
    };  
    private JMenuItem[] items = {  
        new JMenuItem("Fee"), new JMenuItem("Fi"),  
        new JMenuItem("Fo"), new JMenuItem("Zip"),  
        new JMenuItem("Zap"), new JMenuItem("Zot"),  
        new JMenuItem("Olly"), new JMenuItem("Oxen"),
```

```

        new JMenuItem("Free")
    };
    public void init() {
        for(int i = 0; i < items.length; i++) {
            items[i].addActionListener(al);
            menus[i % 3].add(items[i]);
        }
        JMenuBar mb = new JMenuBar();
        for(int i = 0; i < menus.length; i++)
            mb.add(menus[i]);
        setJMenuBar(mb);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
    }
    public static void main(String[] args) {
        Console.run(new SimpleMenus(), 200, 75);
    }
} ///:~

```

程序中通过取模运算“`i%3`”把菜单项分配给三个Jmenu。每个JMenuItem必须有一个相关联的ActionListener；这里使用了同一个ActionListener，不过通常你要为每个JmenuItem都单独准备一个ActionListener。

JMenuItem 从 AbstractButton 继承而来，所以它具有类似按钮的行为。它提供了一个可以单独放置在下拉菜单上的条目。还有三种类型继承自 JMenuItem：JMenu 用来持有其它的 JMenuItem（这样才能实现层叠式菜单）；JCheckBoxMenuItem 提供了一个检查标记，用来表明菜单项是否被选中；JRadioButtonMenuItem 包含了一个单选按钮。

下面是一个更复杂的例子，这里仍然是冰激凌口味菜单。这个例子还演示了层叠式菜单、菜单快捷键、JCheckBoxMenuItem，以及动态改变菜单的方法：

```

///: c14:Menus.java
// Submenus, checkbox menu items, swapping menus,
// mnemonics (shortcuts) and action commands.
// <applet code=Menus width=300 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Menus extends JApplet {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",

```

```

        "Praline Cream", "Mud Pie"
    };
    private JTextField t = new JTextField("No flavor", 30);
    private JMenuBar mb1 = new JMenuBar();
    private JMenu
        f = new JMenu("File"),
        m = new JMenu("Flavors"),
        s = new JMenu("Safety");
    // Alternative approach:
    private JCheckBoxMenuItem[] safety = {
        new JCheckBoxMenuItem("Guard"),
        new JCheckBoxMenuItem("Hide")
    };
    private JMenuItem[] file = { new JMenuItem("Open") };
    // A second menu bar to swap to:
    private JMenuBar mb2 = new JMenuBar();
    private JMenu fooBar = new JMenu("fooBar");
    private JMenuItem[] other = {
        // Adding a menu shortcut (mnemonic) is very
        // simple, but only JMenuItem's can have them
        // in their constructors:
        new JMenuItem("Foo", KeyEvent.VK_F),
        new JMenuItem("Bar", KeyEvent.VK_A),
        // No shortcut:
        new JMenuItem("Baz"),
    };
    private JButton b = new JButton("Swap Menus");
    class BL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JMenuBar m = getJMenuBar();
            setJMenuBar(m == mb1 ? mb2 : mb1);
            validate(); // Refresh the frame
        }
    }
    class ML implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JMenuItem target = (JMenuItem)e.getSource();
            String actionCommand = target.getActionCommand();
            if(actionCommand.equals("Open")) {
                String s = t.getText();
                boolean chosen = false;
                for(int i = 0; i < flavors.length; i++)
                    if(s.equals(flavors[i])) chosen = true;
                if(!chosen)

```

```

        t.setText("Choose a flavor first!");
    else
        t.setText("Opening " + s + ". Mmm, mm!");
    }
}

class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        t.setText(target.getText());
    }
}

// Alternatively, you can create a different
// class for each different MenuItem. Then you
// Don't have to figure out which one it is:
class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Foo selected");
    }
}

class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Bar selected");
    }
}

class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Baz selected");
    }
}

class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        JCheckBoxMenuItem target =
            (JCheckBoxMenuItem)e.getSource();
        String actionCommand = target.getActionCommand();
        if(actionCommand.equals("Guard"))
            t.setText("Guard the Ice Cream! " +
                "Guarding is " + target.getState());
        else if(actionCommand.equals("Hide"))
            t.setText("Hide the Ice Cream! " +
                "Is it hidden? " + target.getState());
    }
}

public void init() {

```

```

ML ml = new ML();
CMIL cmil = new CMIL();
safety[0].setActionCommand("Guard");
safety[0].setMnemonic(KeyEvent.VK_G);
safety[0].addItemListener(cmil);
safety[1].setActionCommand("Hide");
safety[1].setMnemonic(KeyEvent.VK_H);
safety[1].addItemListener(cmil);
other[0].addActionListener(new FooL());
other[1].addActionListener(new BarL());
other[2].addActionListener(new BazL());
FL fl = new FL();
for(int i = 0; i < flavors.length; i++) {
    JMenuItem mi = new JMenuItem(flavors[i]);
    mi.addActionListener(fl);
    m.add(mi);
    // Add separators at intervals:
    if((i + 1) % 3 == 0)
        m.addSeparator();
}
for(int i = 0; i < safety.length; i++)
    s.add(safety[i]);
s.setMnemonic(KeyEvent.VK_A);
f.add(s);
f.setMnemonic(KeyEvent.VK_F);
for(int i = 0; i < file.length; i++) {
    file[i].addActionListener(fl);
    f.add(file[i]);
}
mb1.add(f);
mb1.add(m);
setJMenuBar(mb1);
t.setEditable(false);
Container cp = getContentPane();
cp.add(t, BorderLayout.CENTER);
// Set up the system for swapping menus:
b.addActionListener(new BL());
b.setMnemonic(KeyEvent.VK_S);
cp.add(b, BorderLayout.NORTH);
for(int i = 0; i < other.length; i++)
    fooBar.add(other[i]);
fooBar.setMnemonic(KeyEvent.VK_B);
mb2.add(fooBar);
}

```

```

        public static void main(String[] args) {
            Console.run(new Menu(), 300, 100);
        }
    } ///:~

```

在这个程序中，我把菜单项放到了几个数组中，然后通过遍历这些数组，并为每个 **JMenuItem** 调用 `add()` 方法的方式，将它们添加到菜单中。这就使添加或减少菜单项时不至于太乏味。

程序中不是创建了一个而是创建了两个 **JMenuBar**，用以演示程序运行期间可以动态替换菜单条。你可以看到如何用 **JMenu** 构造 **JMenuBar**，以及用 **JMenuItem**，**JCheckBoxMenuItem** 甚至 **JMenu**（产生子菜单）来装配 **JMenu**。当构造完一个 **JMenuBar** 后，可以使用 `setJMenuBar()` 方法把它安装到当前程序上。注意，当按钮按下时，它将通过调用 `getJMenuBar()` 来判断当前安装的是哪一个菜单条，然后替换成另一个菜单条。

在测试“Open”菜单项的时候，要注意到拼写和大小写是很关键的，如果没有任何匹配的“Open”，Java也不会报告任何错误。这种类型的字符串比较是造成程序错误的根源之一。

菜单项的选中和清除能够被自动地处理。处理 **JCheckBoxMenuItem** 的代码演示了判断菜单项是否选中的两种方式：字符串比较（如上所述，尽管能使用这种方法，但很不安全），和比较事件的目标对象。可以使用 `getState()` 方法得到是否选中的状态。你还可以用 `setState()` 方法来改变 **JCheckBoxMenuItem** 的状态。

菜单对应的事件有些不一致，这可能会引起困惑：**JMenuItem** 使用的是 **ActionListener**，而 **JCheckBoxMenuItem** 使用的是 **ItemListener**。**JMenu** 虽然对象也支持 **ActionListeners**，不过其用处并不大。一般来说，你要把监听器关联到每一个 **JMenuItem**，**JCheckBoxMenuItem**，或者 **JRadioButtonMenuItem** 上，但是在上例中 **ItemListener** 和 **ActionListener** 关联到了不同的菜单组件上。

Swing 支持助记键，或者称为“键盘快捷键”，所以你可以使用键盘而不是鼠标来选择任何从 **AbstractButton**（按钮，菜单项等等）继承而来的组件。做到这一点很简单；只要使用重载的构造器，使它的第二个参数接受快捷键的标识符即可。不过，大多数 **AbstractButton** 没有这样的构造器，所以更通用作法的是使用 `setMnemonic()` 方法。上例中为按钮和部分菜单项添加了快捷键；快捷指示符会自动地出现在组件上。

你还能看到 `setActionCommand()` 的用法。它看起来有些奇怪，因为在每种情况下的“动作命令”（action command）与菜单上的标签都完全相同。为什么不直接使用标签而是这种额外的字符串呢？问题在于对国际化的支持。如果你要把程序以另一种语言发布，最好是希望只改变菜单上的标签，而不用修改代码（毫无疑问，修改代码会引入新的错误）。为了使代码能更容易地判断与菜单关联的字符串，可以把“动作命令”作为不变量，而把菜单上的标签作为可变量。所有的代码在运行时都使用“动作命令”，这样改变菜单标签的时候就不会影响代码。注意，在本例中，并非所有菜单都是基于“动作命令”进行判断的，这是因为没有专门为它们设定“动作命令”。

大量工作都是在监听器中完成的。BL执行的是JMenuBar的交换。在ML中，采用了“找出按铃者”方式，它的作法是先得到ActionEvent的事件源，然后把它类型转换成JmenuItem，接着得到其“动作命令”的字符串，并且把它传递给级联的if语句进行处理。

尽管FL监听器处理的是风味菜单中所有不同风味的菜单项，但它的确很简单。如果你的事件处理逻辑够简单的话，这种方式值得参考。不过一般情况下，你会采用在FooL，BarL和BazL里面所使用的方式，它们只被关联到一个菜单项，所以就不需要进行额外的判断，因为你明确知道是谁调用了监听器。尽管这种方式产生了更多的类，但是类内部的代码会更短，整个处理过程也更安全。

你会发现，有关菜单的代码很快就变得冗长而且凌乱了。这时，使用 GUI 构造工具才是明智的选择。好的工具还可以对菜单进行维护。

弹出式菜单（Pop-up menu）

要实现一个JPopupMenu，最直接的方法就是创建一个继承自 MouseAdapter 的内部类，然后对每个希望具有弹出式行为的组件，都添加一个该内部类的对象：

```
//: c14:Popup.java
// Creating popup menus with Swing.
// <applet code=Popup width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Popup extends JApplet {
    private JPopupMenu popup = new JPopupMenu();
    private JTextField t = new JTextField(10);
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText(((JMenuItem)e.getSource()).getText());
            }
        };
        JMenuItem m = new JMenuItem("Hither");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Yon");
        m.addActionListener(al);
    }
}
```



```

        popup.add(m);
        m = new JMenuItem("Afar");
        m.addActionListener(al);
        popup.add(m);
        popup.addSeparator();
        m = new JMenuItem("Stay Here");
        m.addActionListener(al);
        popup.add(m);
        PopupListener pl = new PopupListener();
        addMouseListener(pl);
        t.addMouseListener(pl);
    }
    class PopupListener extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            maybeShowPopup(e);
        }
        public void mouseReleased(MouseEvent e) {
            maybeShowPopup(e);
        }
        private void maybeShowPopup(MouseEvent e) {
            if(e.isPopupTrigger())
                popup.show(((JApplet)e.getComponent())
                    .getContentPane(), e.getX(), e.getY());
        }
    }
    public static void main(String[] args) {
        Console.run(new Popup(), 300, 200);
    }
} ///:~

```

同一个ActionListener被添加到了每一个JmenuItem上，所以它要从菜单标签中抓取文本，然后插入到JTextField中。

绘图

如果使用好的GUI框架，绘图应该会非常简单，Swing库正是如此。对于任何绘图程序，其问题在于决定绘图位置的计算通常比对绘图功能的调用要复杂得多，并且这些计算程序常常与绘图程序混在一起，所以看起来程序的接口比实际需要的要更复杂。

为了简化问题，考虑一个在屏幕上表示数据的问题，这里的数据将由内置的Math.sin()方法提供，它可以产生数学上的正弦函数。为了使事情变得更有趣一些，也为了进一步演示Swing组件使用起来有多么的简单，我们在窗体底部放置了一个滑块，用来动态控制被显示的正弦波周期的个数。此外，如果调整了视窗的大小，你会发现正弦波能够自动调整，以适应新的视窗。

尽管在任何JComponent上都可以绘图,而且正因为如此,可以把它们当作画布(canvas),但是,要是你只是想有一个可以直接绘图的平面的话,典型的做法是从JPanel继承。唯一需要重载的方法就是paintComponent(),它将在组件必须被重新绘制的时候被调用(通常你不必为此担心,因为何时调用由Swing决定)。当此方法被调用时,Swing将传入一个Graphics对象,然后你就可以使用这个对象并在组件上绘制了。

在下面的例子中,所有与绘制动作相关的代码都在 SineDraw 类中; SineWave 类只是配置程序并控制 slider 组件。在 SineDraw 中, setCycles()方法提供了一个钩子(hook),它允许其它对象(这里就是 slider 组件)控制周期的个数。

```
//: c14:SineWave.java
// Drawing with Swing, using a JSlider.
// <applet code=SineWave width=700 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class SineDraw extends JPanel {
    private static final int SCALEFACTOR = 200;
    private int cycles;
    private int points;
    private double[] sines;
    private int[] pts;
    public SineDraw() { setCycles(5); }
    public void setCycles(int newCycles) {
        cycles = newCycles;
        points = SCALEFACTOR * cycles * 2;
        sines = new double[points];
        for(int i = 0; i < points; i++) {
            double radians = (Math.PI/SCALEFACTOR) * i;
            sines[i] = Math.sin(radians);
        }
        repaint();
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int maxWidth = getWidth();
        double hstep = (double)maxWidth/(double)points;
        int maxHeight = getHeight();
        pts = new int[points];
        for(int i = 0; i < points; i++)
            pts[i] =
                (int)(sines[i] * maxHeight/2 * .95 + maxHeight/2);
        g.setColor(Color.RED);
    }
}
```

```

        for(int i = 1; i < points; i++) {
            int x1 = (int)((i - 1) * hstep);
            int x2 = (int)(i * hstep);
            int y1 = pts[i-1];
            int y2 = pts[i];
            g.drawLine(x1, y1, x2, y2);
        }
    }
}

public class SineWave extends JApplet {
    private SineDraw sines = new SineDraw();
    private JSlider adjustCycles = new JSlider(1, 30, 5);
    public void init() {
        Container cp = getContentPane();
        cp.add(sines);
        adjustCycles.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                sines.setCycles(
                    ((JSlider)e.getSource()).getValue());
            }
        });
        cp.add(BorderLayout.SOUTH, adjustCycles);
    }

    public static void main(String[] args) {
        Console.run(new SineWave(), 700, 400);
    }
} ///:~

```

在计算正弦波上的点的过程中用到了所有的域和数组；**cycles**表示所希望的完整的正弦波个数，**points**是将要绘制的点的总数，**sines**包含了正弦函数的值，**pts**包含将要绘制在JPanel上点的y坐标。**SetCycles()**方法先根据所需点的数目创建数组，然后为数组里的每个元素计算相应的正弦函数值。它通过调用**repaint()**方法，迫使**paintComponent()**得到调用，这样，余下的计算和重绘动作就会发生。

当你重载**paintComponent()**方法的时候，必须先调用该方法的基类版本。然后你才可以做想做的事情。通常，这意味着要使用你在**java.awt.Graphics**的文档（在**java.sun.com**的JDK文档中）中可以找到的**Graphics**方法向JPanel上绘制像素点。这里，几乎所有的代码都和计算有关；实际上只有**setColor()**和**drawLine()**这两个方法和操纵屏幕有关。在你创建自己的用于显示图形数据的程序时，可能会有类似的经历：你会把大部分时间用在计算要绘制的内容上，而真正的绘制过程则非常简单。

在我创建此程序的时候，把大量时间用在显示正弦波上。做完这些之后，我觉得如果是能够动态改变周期的个数，那么它的效果会更好。当我准备这么做的时候，我在别的语言中的编程经验使我觉得，这并不容易实现，但是结果却显示，这是整个程序中最容易的部

分。我先创建了一个JSlider（其构造器参数分别是最左边的值、最右边的值和初始值；它还有其它的构造器），然后把它加入到JApplet中。接下来，我参考了JDK文档，发现它仅有的监听器是addChangeListener，它将在滑块的变动足以产生新值的时候被触发。唯一能够处理此事件的方法显然就是stateChanged()，在调用它时为其提供了一个ChangeEvent对象，这样就可以找到变动源并找出新值。通过调用sines对象的setCycles()，这个新值将被设置，JPanel也将被重绘。

通常情况下，你会发现在Swing程序中的遇到的问题，大部分可以通过相似的过程得到解决，而且这个过程通常非常简单，甚至对于从未使用过的组件，也是如此。

如果要解决更复杂的问题，可以使用更高级的绘图库，包括第三方提供的JavaBean组件或者Java 2D API。这些内容超出了本书的范围，不过要是你的绘图代码确实变得过于复杂了，你就应该查找这些内容的相关资源。

对话框（Dialog box）

对话框是从视窗弹出的另一个窗口。它的目的是处理一些具体问题，同时又不会使这些具体细节与原先窗口的内容混在一起。对话框在视窗编程环境下被大量使用，不过在applet中比较少见。

如果要编写一个对话框，就需要从JDialog继承，它只不过是另一种类型的Window，与JFrame类似。JDialog具有一个布局管理器（缺省情况下为BorderLayout），并且你要添加事件监听器来处理事件。它与别的Window相比，有一个明显不同的地方，就是当你关闭对话框窗口的时候，并不希望关闭整个应用程序。这时候的做法是通过调用dispose()方法来释放对话框窗口所占用的资源。下面是个简单的例子：

```
//: c14:Dialogs.java
// Creating and using Dialog Boxes.
// <applet code=Dialogs width=125 height=75></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class MyDialog extends JDialog {
    public MyDialog(JFrame parent) {
        super(parent, "My dialog", true);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(new JLabel("Here is my dialog"));
        JButton ok = new JButton("OK");
        ok.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dispose(); // Closes the dialog
            }
        });
    }
}
```

```

    }
    });
    cp.add(ok);
    setSize(150, 125);
}
}

public class Dialogs extends JApplet {
    private JButton b1 = new JButton("Dialog Box");
    private MyDialog dlg = new MyDialog(null);
    public void init() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dlg.show();
            }
        });
        getContentPane().add(b1);
    }
    public static void main(String[] args) {
        Console.run(new Dialogs(), 125, 75);
    }
} ///:~

```

创建JDialog以后，要显示和激活它就必须调用show()方法。而要关闭对话框则必须调用dispose()。

你将发现，任何从applet弹出的窗体，包括对话框在内，都是“不受信任”的。也就是说，你将从被弹出的窗体中得到警告信息。这是因为至少从理论上讲，弹出的窗体有可能欺骗用户，使他们认为自己在操作一个常规的本地程序，比如让用户输入他们的信用卡号码，从而使这个号码被散布到网络上。Applet总是嵌入在某个网页内的，并且在浏览器中是可视的，而applet又可以弹出对话框，所以从理论上讲，这是有可能的。因此，使用对话框的applet并不常见。

下面的例子更加复杂；对话框由一个网格构成（使用GridLayout），并且添加了一种特殊按钮，它由ToeButton类定义。按钮将先在自己周围画一个边框，然后根据状态的不同，在中央画“空白”，“x”或者“o”。开始时的按钮状态为“空白”，然后根据每一轮单击，变成“x”或者“o”。而且，当你在非“空白”的按钮上单击的时候，它将在“x”和“o”之间翻转（这采用了三值翻转(tic-tac-toe)的概念，只是为了使程序比目前更复杂）。此外，通过改变在主应用视窗中的数字，可以为对话框设置任意的行数和列数。

```

///: c14:TicTacToe.java
// Dialog boxes and creating your own components.
// <applet code=TicTacToe width=200 height=100></applet>
import javax.swing.*;
import java.awt.*;

```

```

import java.awt.event.*;
import com.bruceeckel.swing.*;

public class TicTacToe extends JApplet {
    private JTextField
        rows = new JTextField("3"),
        cols = new JTextField("3");
    private static final int BLANK = 0, XX = 1, OO = 2;
    class ToeDialog extends JDialog {
        private int turn = XX; // Start with x's turn
        ToeDialog(int cellsWide, int cellsHigh) {
            setTitle("The game itself");
            Container cp = getContentPane();
            cp.setLayout(new GridLayout(cellsWide, cellsHigh));
            for(int i = 0; i < cellsWide * cellsHigh; i++)
                cp.add(new ToeButton());
            setSize(cellsWide * 50, cellsHigh * 50);
            setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        }
        class ToeButton extends JPanel {
            private int state = BLANK;
            public ToeButton() { addMouseListener(new ML()); }
            public void paintComponent(Graphics g) {
                super.paintComponent(g);
                int
                    x1 = 0, y1 = 0,
                    x2 = getSize().width - 1,
                    y2 = getSize().height - 1;
                g.drawRect(x1, y1, x2, y2);
                x1 = x2/4;
                y1 = y2/4;
                int wide = x2/2, high = y2/2;
                if(state == XX) {
                    g.drawLine(x1, y1, x1 + wide, y1 + high);
                    g.drawLine(x1, y1 + high, x1 + wide, y1);
                }
                if(state == OO)
                    g.drawOval(x1, y1, x1 + wide/2, y1 + high/2);
            }
        }
        class ML extends MouseAdapter {
            public void mousePressed(MouseEvent e) {
                if(state == BLANK) {
                    state = turn;
                    turn = (turn == XX ? OO : XX);
                }
            }
        }
    }
}

```

```

        }
        else
            state = (state == XX ? 00 : XX);
        repaint();
    }
}
}
}
}
class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JDialog d = new ToeDialog(
            Integer.parseInt(rows.getText()),
            Integer.parseInt(cols.getText()));
        d.setVisible(true);
    }
}
public void init() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(2,2));
    p.add(new JLabel("Rows", JLabel.CENTER));
    p.add(rows);
    p.add(new JLabel("Columns", JLabel.CENTER));
    p.add(cols);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    JButton b = new JButton("go");
    b.addActionListener(new BL());
    cp.add(b, BorderLayout.SOUTH);
}
public static void main(String[] args) {
    Console.run(new TicTacToe(), 200, 100);
}
} ///:~

```

因为static关键字只能处于类的外层，所以内部类不能包含静态的数据或者嵌套类。

paintComponent()方法先在面板周围绘制正方形，然后在中间画“x”或“o”。这里充满了乏味的计算，但是却很直接明了。

MouseListener被用来处理鼠标单击：首先，它检查面板上是否为空白，如果不是空白，就向父窗体查询现在是哪一轮，这样就得到了ToeButton的状态。通过内部类机制，ToeButton可以操作其外部类，更新当前的轮次；如果按钮已经显示为“x”或“o”，那么就翻转它。在这个计算中，你可以看到第3章学习的if-else三元运算符的习惯用法。在状态改变之后，ToeButton将被重绘。

ToeDialog的构造器非常简单；它向网格中添加你所要求数目的按钮，然后调整窗体大小，使得每个按钮的长和宽均为 50 个像素。

TicTacToe通过创建两个TextField（用来输入按钮网格的行数和列数）和带有ActionListener的“go”按钮，完成整个程序的设置工作。当按钮被按下时，将获取JTextField里面的数据，因为它们是字符串，所以使用静态的Integer.parseInt()方法把它们解析成整数。

文件对话框（File dialog）

某些操作系统具有大量特殊的内置对话框，它们可以处理诸如选择字体、颜色、打印机等操作。基本上所有的图形操作系统都支持打开和保存文件，所以Java提供了JFileChooser，它封装了这些操作，使文件操作变得更加方便。

下面的程序演练了两个 JFileChooser 对话框，一个用来打开文件，一个用来保存文件。大多数代码你现在应该都已经很熟悉了，所有有趣的行为都集中在处理两个按钮单击的事件监听器中：

```
///  
// Demonstration of File dialog boxes.  
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
import com.bruceeckel.swing.*;  
  
public class FileChooserTest extends JFrame {  
    private JTextField  
        filename = new JTextField(),  
        dir = new JTextField();  
    private JButton  
        open = new JButton("Open"),  
        save = new JButton("Save");  
    public FileChooserTest() {  
        JPanel p = new JPanel();  
        open.addActionListener(new OpenL());  
        p.add(open);  
        save.addActionListener(new SaveL());  
        p.add(save);  
        Container cp = getContentPane();  
        cp.add(p, BorderLayout.SOUTH);  
        dir.setEditable(false);  
        filename.setEditable(false);  
        p = new JPanel();  
        p.setLayout(new GridLayout(2, 1));
```



```

        p.add(filename);
        p.add(dir);
        cp.add(p, BorderLayout.NORTH);
    }
    class OpenL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser c = new JFileChooser();
            // Demonstrate "Open" dialog:
            int rVal = c.showOpenDialog(FileChooserTest.this);
            if(rVal == JFileChooser.APPROVE_OPTION) {
                filename.setText(c.getSelectedFile().getName());
                dir.setText(c.getCurrentDirectory().toString());
            }
            if(rVal == JFileChooser.CANCEL_OPTION) {
                filename.setText("You pressed cancel");
                dir.setText("");
            }
        }
    }
    class SaveL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser c = new JFileChooser();
            // Demonstrate "Save" dialog:
            int rVal = c.showSaveDialog(FileChooserTest.this);
            if(rVal == JFileChooser.APPROVE_OPTION) {
                filename.setText(c.getSelectedFile().getName());
                dir.setText(c.getCurrentDirectory().toString());
            }
            if(rVal == JFileChooser.CANCEL_OPTION) {
                filename.setText("You pressed cancel");
                dir.setText("");
            }
        }
    }
    public static void main(String[] args) {
        Console.run(new FileChooserTest(), 250, 110);
    }
} ///:~

```

注意JFileChooser的使用有很多种变化，包括使用过滤器来缩小可供选择的文件名范围等。

对于“open file”对话框，你调用 `showOpenDialog()` 方法，对于“save file”对话框，调用 `showSaveDialog()`。这些方法直到对话框关闭的时候才会返回。此时JFileChooser

对象仍旧存在，`getSelectedFile()`和`getCurrentDirectory()`是用来查询操作返回结果的两种方法。如果它们返回为空，就表示用户已经取消操作并关闭了对话框。

Swing 组件上的 HTML

任何能接受文本的组件都可以接受 HTML 文本，且能根据 HTML 的规则来重新格式化文本。也就是说，你可以很容易地在 Swing 组件上加入漂亮的文本。例如：

```
//: c14:HTMLButton.java
// Putting HTML text on Swing components.
// <applet code=HTMLButton width=250 height=500></applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class HTMLButton extends JApplet {
    private JButton b = new JButton(
        "<html><b><font size=+2>" +
        "<center>Hello!<br><i>Press me now!");
    public void init() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                getContentPane().add(new JLabel("<html>" +
                    "<i><font size=+4>Kapow!"));
                // Force a re-layout to include the new label:
                validate();
            }
        });
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b);
    }
    public static void main(String[] args) {
        Console.run(new HTMLButton(), 200, 500);
    }
} ///:~
```

你必须使文本以“<html>”标记开始，然后你就可以使用普通的HTML标记了。注意，你不会被强制要求添加结束标记。

`ActionListener`将一个新的`JLabel`添加到了窗体中，它也包含了HTML文本。不过，这个标签不是在`init()`中添加的，所以你必须调用容器的`validate()`方法来强制对组件进行重新布局（这样就能显示该新标签了）。

你还可以在JTabbedPane, JMenuItem, JToolTip, JRadioButton以及JCheckBox中使用HTML文本。

滑块与进度条 (Slider & progress bar)

滑块（已经在 SineWave.java 中使用过了）能令用户通过前后移动滑点来输入数据，在某些情况下这显得很直观（比如音量控制）。进度条能以从“无”到“有”的动态方式显示数据，这也能给用户以直观的感受。我最喜欢的例子是把滑块与进度条关联在一起，这样当你移动滑块的时候，进度条就可以跟着作相应的改变：

```
/// c14:Progress.java
// Using progress bars and sliders.
// <applet code=Progress width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Progress extends JApplet {
    private JProgressBar pb = new JProgressBar();
    private JSlider sb =
        new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(2, 1));
        cp.add(pb);
        sb.setValue(0);
        sb.setPaintTicks(true);
        sb.setMajorTickSpacing(20);
        sb.setMinorTickSpacing(5);
        sb.setBorder(new TitledBorder("Slide Me"));
        pb.setModel(sb.getModel()); // Share model
        cp.add(sb);
    }
    public static void main(String[] args) {
        Console.run(new Progress(), 300, 200);
    }
} ///~
```

把两个组件联系到一起的关键在于让它们共享一个模型，就像下面这一行一样：

```
pb.setModel(sb.getModel());
```

当然，你也可以使用监听器进行控制，不过在简单的情况下这种方法更直接。

JProgressBar相当简单，而JSlider就有许多可选项，比如放置方向，主次标记等等。你应该能够注意到，添加一个带标题的边框是多么地直截了当。

树 (Tree)

使用 Jtree 的方式可以像下面这样简单：

```
add(new JTree(new Object[] { "this", "that", "other" }));
```

这将显示一棵原始的树。树的API非常多，肯定是Swing里最大的组件之一。它似乎能帮你作任何事情，只是如果要用它来完成更复杂的任务，你还需要对它作更多的研究和试验。

幸运的是，在类库中已经提供了一个半成品：“缺省”的一些树组件，一般来说它就能够满足你的要求。所以，在大多数时间内，你可以就使用这些组件，只有在需要实现特殊功能的时候，你才需要对树组件有更深入的理解。

在下面的例子中，使用了“缺省”的树组件来在一个applet中显示一棵树。当你按下按钮的时候，将在当前被选择的节点（如果没有节点被选择的话，就使用根节点）上添加一棵子树：

```
//: c14:Trees.java
// Simple Swing tree. Trees can be vastly more complex.
// <applet code=Trees width=250 height=250></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.tree.*;
import com.bruceeckel.swing.*;

// Takes an array of Strings and makes the first
// element a node and the rest leaves:
class Branch {
    private DefaultMutableTreeNode r;
    public Branch(String[] data) {
        r = new DefaultMutableTreeNode(data[0]);
        for(int i = 1; i < data.length; i++)
            r.add(new DefaultMutableTreeNode(data[i]));
    }
    public DefaultMutableTreeNode node() { return r; }
}

public class Trees extends JApplet {
```

```

private String[][] data = {
    { "Colors", "Red", "Blue", "Green" },
    { "Flavors", "Tart", "Sweet", "Bland" },
    { "Length", "Short", "Medium", "Long" },
    { "Volume", "High", "Medium", "Low" },
    { "Temperature", "High", "Medium", "Low" },
    { "Intensity", "High", "Medium", "Low" },
};

private static int i = 0;
private DefaultMutableTreeNode root, child, chosen;
private JTree tree;
private DefaultTreeModel model;
public void init() {
    Container cp = getContentPane();
    root = new DefaultMutableTreeNode("root");
    tree = new JTree(root);
    // Add it and make it take care of scrolling:
    cp.add(new JScrollPane(tree), BorderLayout.CENTER);
    // Capture the tree's model:
    model = (DefaultTreeModel) tree.getModel();
    JButton test = new JButton("Press me");
    test.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if(i < data.length) {
                child = new Branch(data[i++]).node();
                // What's the last one you clicked?
                chosen = (DefaultMutableTreeNode)
                    tree.getLastSelectedPathComponent();
                if(chosen == null)
                    chosen = root;
                // The model will create the appropriate event.
                // In response, the tree will update itself:
                model.insertNodeInto(child, chosen, 0);
                // Puts the new node on the chosen node.
            }
        }
    });
    // Change the button's colors:
    test.setBackground(Color.BLUE);
    test.setForeground(Color.WHITE);
    JPanel p = new JPanel();
    p.add(test);
    cp.add(p, BorderLayout.SOUTH);
}

```

```

        public static void main(String[] args) {
            Console.run(new Trees(), 250, 250);
        }
    } ///:~

```

第一个类Branch是一个工具，它接受一个字符串数组，然后用第一个字符串作为根创建一个DefaultMutableTreeNode对象，数组中的其余字符串将用于创建叶子。调用node()方法能得到这个“子树”的根。

Trees类包含了一个二维字符串数组，它被用来创建“子树”，静态整型变量i用来遍历这个数组。DefaultMutableTreeNode对象能持有节点，但屏幕上的表示是由JTree及其关联的模型DefaultTreeModel所控制的。注意，当把Jtree添加到applet的时候，它被包装进了一个JScrollPane中，这是为了使其能够提供自动滚动的功能。

可以通过JTree的模型对它进行控制。当你修改模型时，它将产生一个事件，JTree将根据这个事件对树的视觉表现作出必要的更新。在init()中，通过调用getModel()得到对模型的引用。当按钮按下的时候，新的“子树”被创建。然后查找当前被选择的节点（没有节点被选择的话，就使用根节点），insertNodeInto()方法用来作剩下的工作，包括修改树的结构和更新树的外观。

像上面这样的例子可能已经能够满足你的需求了。不过，树组件的功能十分强大，只要是你想得到的，它几乎都能做到。上面的例子中只要出现“default”的地方，你都可以自己编写类来替换，以得到不同的行为。不过要当心：几乎所有这些类的接口都非常大，所以要花大量时间和努力才能理解它的复杂行为。尽管如此，它的设计仍然非常优秀，某些类似的实现比它要糟糕得多。

表格（Table）

与树组件类似，Swing提供的表格组件功能也很强。它的基本目标是通过Java数据库连接（JDBC，在《Thinking in Enterprise Java》中讨论）为数据库提供流行的“网格”式接口。所以，表格组件要有极大的灵活性，当然也带来了与之相应的复杂度。要想创建一个具有完整功能的电子表格应用程序，可能需要整本书来讨论。不过，如果你理解了JTable的基本概念，创建一个相对简单的表格组件也是可能的。

JTable 用来控制数据如何显示，而 TableModel 则控制数据本身。所以要创建一个 JTable 对象，你要先创建一个 TableModel。你可以完整实现 TableModel 接口，不过从 AbstractTableModel 这个半成品继承则更容易：

```

///: c14:JTableDemo.java
// Simple demonstration of JTable.
// <applet code=Table width=350 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

```

```

import javax.swing.table.*;
import javax.swing.event.*;
import com.bruceeckel.swing.*;

public class JTableDemo extends JApplet {
    private JTextArea txt = new JTextArea(4, 20);
    // The TableModel controls all the data:
    class DataModel extends AbstractTableModel {
        Object[][] data = {
            {"one", "two", "three", "four"},
            {"five", "six", "seven", "eight"},
            {"nine", "ten", "eleven", "twelve"},
        };
        // Prints data when table changes:
        class TML implements TableModelListener {
            public void tableChanged(TableModelEvent e) {
                txt.setText(""); // Clear it
                for(int i = 0; i < data.length; i++) {
                    for(int j = 0; j < data[0].length; j++)
                        txt.append(data[i][j] + " ");
                    txt.append("\n");
                }
            }
        }

        public DataModel() { addTableModelListener(new TML()); }
        public int getColumnCount() { return data[0].length; }
        public int getRowCount() { return data.length; }
        public Object getValueAt(int row, int col) {
            return data[row][col];
        }

        public void setValueAt(Object val, int row, int col) {
            data[row][col] = val;
            // Indicate the change has happened:
            fireTableDataChanged();
        }

        public boolean isCellEditable(int row, int col) {
            return true;
        }
    }

    public void init() {
        Container cp = getContentPane();
        JTable table = new JTable(new DataModel());
        cp.add(new JScrollPane(table));
        cp.add(BorderLayout.SOUTH, txt);
    }
}

```

```

    }
    public static void main(String[] args) {
        Console.run(new JTableDemo(), 350, 200);
    }
} ///:~

```

`DataModel`包含了一个`data`数组，不过你也可以从其它数据源得到数据，比如数据库。构造器中添加了一个`TableModelListener`，每当表格变化的时候，它用来打印数组内容。其它方法遵循了Bean的命名规则（使用“`get`”和“`set`”方法，将在本章后面介绍），当`JTable`要显示`DataModel`里面的内容时，会调用这些方法。`AbstractTableModel`为`setValueAt()`和`isCellEditable()`方法提供了缺省实现，用来防止改变数据。所以你要是希望能编辑数据，就得重载这些方法。

一旦有了`TableModel`，你只需把它传递给`JTable`的构造器。有关显示，编辑，更新的所有细节将自动完成。本例中把`JTable`也包装进了一个`JScrollPane`中。

选择外观（Look & Feel）

“可插拔外观”（Pluggable Look & Feel）使你的程序能够模仿不同的操作系统的外观。你甚至可以得到各种奇妙的功能，比如在程序运行期间动态改变程序的外观。不过，通常你只会在以下二者中选择一个：要么选择“跨平台”的外观（即Swing的“金属”外观）；要么选择程序当前所在系统的外观，这样你的Java程序看起来就好像是为该系统专门设计的（在大多数情况下，这的确是最好的选择，而且可以避免误导用户）。实现这两种行为的代码都很简单，不过你要确保在创建任何可视组件之前先调用这些代码，这是因为组件是根据当前的外观而创建的，而且创建之后就不会改变（你很少会在程序运行的时候改变程序的外观，这个过程相当复杂且不常见，可以参考专门研究Swing的书）。

实际上，如果你要使用跨平台的“金属”外观（它是Swing程序的特征），你什么都不需要做，因为它是缺省外观。不过你要想使用当前操作系统的外观，加入下列代码即可，一般把这些代码添加在`main()`开始的地方，至少也要在添加任何组件之前：

```

try {
    UIManager.setLookAndFeel(UIManager.
        getSystemLookAndFeelClassName());
} catch(Exception e) {
    throw new RuntimeException(e);
}

```

在`catch`子句中你什么也不用做，因为在缺省情况下，如果你的设置代码失败了，`UIManager`将设置成跨平台的外观。不过，在调试的时候异常非常有用，所以你也许希望通过`catch`子句看看发生的问题。

下面的程序能通过命令行参数选择外观，这里选择了几种组件，以在不同的外观下演示：


```

//: c14:LookAndFeel.java
// Selecting different looks & feels.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class LookAndFeel extends JFrame {
    private String[] choices = {
        "eeny", "meeny", "Minnie", "Mickey", "Moe", "Larry", "Curly"
    };
    private Component[] samples = {
        new JButton("JButton"),
        new JTextField("JTextField"),
        new JLabel("JLabel"),
        new JCheckBox("JCheckBox"),
        new JRadioButton("Radio"),
        new JComboBox(choices),
        new JList(choices),
    };
    public LookAndFeel() {
        super("Look And Feel");
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i = 0; i < samples.length; i++)
            cp.add(samples[i]);
    }
    private static void usageError() {
        System.out.println(
            "Usage:LookAndFeel [cross|system|motif]");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length == 0) usageError();
        if(args[0].equals("cross")) {
            try {
                UIManager.setLookAndFeel(UIManager.
                    getCrossPlatformLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace();
            }
        } else if(args[0].equals("system")) {
            try {

```

```

        UIManager.setLookAndFeel(UIManager.
            getSystemLookAndFeelClassName());
    } catch (Exception e) {
        e.printStackTrace();
    }
} else if (args[0].equals("motif")) {
    try {
        UIManager.setLookAndFeel("com.sun.java." +
            "swing.plaf.motif.MotifLookAndFeel");
    } catch (Exception e) {
        e.printStackTrace();
    }
} else usageError();
// Note the look & feel must be set before
// any components are created.
Console.run(new LookAndFeel(), 300, 200);
}
} ///:~

```

你可以观察到，一种选择是明确地使用一个字符串来明确指定一个外观，比如 **MotifLookAndFeel** 外观。而且，只有这个外观和缺省的“金属”外观能够合法地在所有平台上使用；尽管有针对 **Windows** 和 **Macintosh** 外观的字符串，但它们只能在各自的平台上使用才合法（当你在这些平台上调用 `getSystemLookAndFeelClassName()` 方法时，可以得到相应的字符串）。

自己编写一种外观也是可能的，比如，你在为某个公司编写框架代码，他们要求有独特的外观。这是个大工程，远远超出了本书的范围（事实上，你会发现这也超出了许多专业 **Swing** 书的范围！）。

剪贴板（Clipboard）

Java基本类库（**JFC**）对系统剪贴板提供了有限的支持（以 `java.awt.datatransfer` 包的形式）。你可以把字符串对象作为文本复制到剪贴板，或者从剪贴板粘贴文本到一个字符串对象中。当然，剪贴板被设计成可以持有任何数据，不过剪贴板上的数据如何表示则取决于实现剪切和粘贴动作的程序。**Java**剪贴板的API通过“风味”的概念提供了扩展性。当数据离开剪贴板时，它就关联了一个相关的风味集，表明它可以被转换成的格式（比如，图形可能以数字字符串表示，也可以用图像表示），你可以查看这个剪贴板数据是否支持你所感兴趣的“风味”。

下面的程序简单地演示了 **JTextArea** 组件内对字符串数据的剪切、复制和粘贴。你要注意的一点是，你在剪切、复制和粘贴时通常使用的快捷键序列仍然有效。但你如果在其它程序中查看 **JTextField** 或者 **JTextArea**，就会发现它们也支持这些键盘快捷键。这个例子只是为了编程控制剪贴板，当你要把剪贴板数据捕获到 **JtextComponent** 以外的组件中的时候，就要用到这些技术。

```

//: c14:CutAndPaste.java
// Using the clipboard.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import com.bruceeckel.swing.*;

public class CutAndPaste extends JFrame {
    private JMenuBar mb = new JMenuBar();
    private JMenu edit = new JMenu("Edit");
    private JMenuItem
        cut = new JMenuItem("Cut"),
        copy = new JMenuItem("Copy"),
        paste = new JMenuItem("Paste");
    private JTextArea text = new JTextArea(20, 20);
    private Clipboard clipbd =
        getToolkit().getSystemClipboard();
    public CutAndPaste() {
        cut.addActionListener(new CutL());
        copy.addActionListener(new CopyL());
        paste.addActionListener(new PasteL());
        edit.add(cut);
        edit.add(copy);
        edit.add(paste);
        mb.add(edit);
        setJMenuBar(mb);
        getContentPane().add(text);
    }
    class CopyL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String selection = text.getSelectedText();
            if(selection == null)
                return;
            StringSelection clipString =
                new StringSelection(selection);
            clipbd.setContents(clipString, clipString);
        }
    }
    class CutL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String selection = text.getSelectedText();
            if(selection == null)
                return;

```

```

        StringSelection clipString =
            new StringSelection(selection);
        clipbd.setContents(clipString, clipString);
        text.replaceRange("", text.getSelectionStart(),
            text.getSelectionEnd());
    }
}

class PasteL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Transferable clipData =
            clipbd.getContents(CutAndPaste.this);
        try {
            String clipString = (String)clipData.
                getTransferData(DataFlavor.stringFlavor);
            text.replaceRange(clipString,
                text.getSelectionStart(), text.getSelectionEnd());
        } catch (Exception ex) {
            System.err.println("Not String flavor");
        }
    }
}

public static void main(String[] args) {
    Console.run(new CutAndPaste(), 300, 200);
}
} ///:~

```

创建并添加菜单和JTextArea的工作现在应该已经是很熟悉了。这里不同的只是对Clipboard域clipbd的创建，它从通过Toolkit来实现的。

所有的动作都发生在监听器内部。CopyL 和 CutL 监听器基本相同，但是 CutL 的最后一行例外，因为这里需要把复制过的文本删除。这里比较特殊的两行是：先用字符串创建一个StringSelection对象，然后把这个对象传递给setContents()。要把字符串放进剪贴板，这样做就够了。

在PasetL中，先使用getContents()把数据从剪贴板中取出。这里得到的是一个相当隐匿的Transferable对象，你并不知道它的确切内容。要判断它的格式，一个方法是调用getTransferDataFlavors()，这将返回一个类型为DataFlavor对象的数组，表示此对象所支持的“风味”。你还可以直接调用isDataFlavorSupported()，传入你所感兴趣的“风味”，以进行判断。不过，这里使用了一个大胆的方法：假设它支持String“风味”，直接调用getTransferData()，如果它不支持的话，问题将在异常处理程序中显示。

在Java的未来版本中，你可以期望它支持更多的“风味”。

把 applet 打包进 JAR 文件

JAR工具的一个重要用途是对applet的载入过程进行优化。在Java 1.0 版中，人们倾向于把所有代码都塞进一个applet类中，这样用户只需要一次服务器访问就可以下载applet的所有代码。这种做法不仅混乱，使程序难以阅读（维护），而且.class文件仍然没有经过压缩，所以下载速度并没有太大提高。

JAR文件通过把所有的.class文件压缩进一个文件，然后由浏览器去下载的方式，解决了这个问题。现在你就可以进行合理地设计了，而不用担心这会产生多少个.class文件，用户的下载时间也大大缩短了。

考虑一下 TicTacToe.java 程序。它看起来只有一个类，但它实际上包含了五个内部类，所以总共有六个类。一旦编译完程序，你可以用下列命令把它们打包进 JAR 文件：

```
jar cf TicTacToe.jar *.class
```

这里假定当前目录下的所有.class文件都是从TicTacToe.java编译而来（否则，就会加入不必要的文件）。

现在就可以编写 HTML 页面，使用新的 archive 标记来表示 JAR 文件的名字。下面就是简单的 applet 标记：

```
<head><title>TicTacToe Example Applet
</title></head>
<body>
<applet code=TicTacToe.class
        archive=TicTacToe.jar
        width=200 height=100>
</applet>
</body>
```

要使上面的例子能够工作，你需要用随JDK一起发布的HTMLConverter程序来处理这个文件。

为 applet 签名

由于Java的沙盒安全模型的存在，使得未签名的applet在客户端被禁止执行某些操作，比如读写文件或者连接局域网¹⁰。经过签名的applet能够向用户验证applet的编写者：他是否真的编写了这个applet；此JAR文件离开服务器后，内容是否被篡改过。如果没有这些最基本的保证，applet将不被允许执行任何有可能破坏用户机器，或者侵犯用户隐私的操作。要在互联网上安全地使用applet，这些限制非常重要，不过相对来说，这也限制了applet的能力。

¹⁰ 本节和下一节是由Jeremy Meyer编写的。

自从Java Plugin发布以来，为applet签名的过程已经变得更加简单和标准化了，这使得applet成为了一种更加可行的部署你的应用的方式。使用标准的Java工具，为applet签名已经成为了相当简单的工作。

在plugin出现以前，要为applet签名的话，你不得不为Netscape用户使用Netscape的工具签名一个.jar文件，为Internet Explorer用户使用微软的工具签名一个.cab文件，在HTML文件里为这两种平台编写applet标记。最后，用户还得在浏览器上安装相应的证书（certificate），这样applet才能被信任。

plugin不仅为applet的签名和部署提供了标准方法，而且还通过自动安装证书使得最终用户使用起来更方便。

考虑一个需要访问用户文件系统的applet，它要对一些文件进行读写。这与FileChooserTest.java程序很相似，不过这是一个applet，所以只有在这个applet是从一个签名过的JAR文件中运行的情况下，才能打开Swing的JFileChooser对话框；否则showOpenDialog()将抛出SecurityException异常。

```
///  
// Demonstration of File dialog boxes.  
package c14.signedapplet;  
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
import java.io.*;  
import com.bruceeckel.swing.*;  
  
public class FileAccessApplet extends JApplet {  
    private JTextField  
        filename = new JTextField(),  
        dir = new JTextField();  
    private JButton  
        open = new JButton("Open"),  
        save = new JButton("Save");  
    private JEditorPane ep = new JEditorPane();  
    private JScrollPane jsp = new JScrollPane();  
    private File file;  
    public void init() {  
        JPanel p = new JPanel();  
        open.addActionListener(new OpenL());  
        p.add(open);  
        save.addActionListener(new SaveL());  
        p.add(save);  
        Container cp = getContentPane();  
        jsp.getViewport().add(ep);  
        cp.add(jsp, BorderLayout.CENTER);  
    }  
}
```

```

cp.add(p, BorderLayout.SOUTH);
dir.setEditable(false);
save.setEnabled(false);
ep.setContentType("text/html");
filename.setEditable(false);
p = new JPanel();
p.setLayout(new GridLayout(2, 1));
p.add(filename);
p.add(dir);
cp.add(p, BorderLayout.NORTH);
}

class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser();
        c.setFileFilter(new TextFileFilter());
        // Demonstrate "Open" dialog:
        int rVal = c.showOpenDialog(FileAccessApplet.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {
            file = c.getSelectedFile();
            filename.setText(file.getName());
            dir.setText(c.getCurrentDirectory().toString());
            try {
                System.out.println("Url is " + file.toURL());
                ep.setPage(file.toURL());
                // ep.repaint();
            } catch (IOException ioe) {
                throw new RuntimeException(ioe);
            }
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
            filename.setText("You pressed cancel");
            dir.setText("");
        } else {
            save.setEnabled(true);
        }
    }
}

class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser(file);
        c.setSelectedFile(file);
        // Demonstrate "Save" dialog:
        int rVal = c.showSaveDialog(FileAccessApplet.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {

```

```

        filename.setText(c.getSelectedFile().getName());
        dir.setText(c.getCurrentDirectory().toString());
        try {
            FileWriter fw = new FileWriter(file);
            ep.write(fw);
        } catch (IOException ioe) {
            throw new RuntimeException(ioe);
        }
    }
    if(rVal == JFileChooser.CANCEL_OPTION) {
        filename.setText("You pressed cancel");
        dir.setText("");
    }
}
}

public class TextFileFilter extends
    javax.swing.filechooser.FileFilter {
    public boolean accept(File f) {
        return f.getName().endsWith(".txt")
            || f.isDirectory();
    }
    public String getDescription() {
        return "Text Files (*.txt)";
    }
}

public static void main(String[] args) {
    Console.run(new FileAccessApplet(), 500, 500);
}
} ///:~

```

看起来这只是一个普通的applet。不过，正如它所表现的，这个applet不能打开和关闭客户机系统上的文件。要使这个applet作为一个已签名的applet运行，你需要把它打包进一个JAR文件（参考本章前面有关jar工具的小节），然后为JAR文件签名。

一旦做好了JAR文件，你还需要一个与签名一起的证书或密钥。如果你是个大公司，你可以向Verisign或Thawte这样的权威部门申请，他们能向你提供证书。证书用来为代码签名，以向用户保证他们下载的确是提供的代码，并且这些被部署的代码在你签名之后没有被改动过。本质上，数字签名就是一些比特位，当用户下载签名的时候，由权威部门为你作出担保。

从权威部门获得证书需要付费，而且需要定期更新。在我们的例子中，我们可以只制作一个自己的签名。它们需要存放在某个文件中（通常称为“键链”（keychain））。如果你输入：

```
keytool -list
```


这将试图访问缺省的文件。如果没有这个文件，那么你就需要创建一个，或者指定一个已存在的文件。你可能要搜索一个名为“cacerts”的文件，然后试着：

```
keytool -list -file <path/filename>
```

缺省位置一般是：

```
{java.home}/lib/security/cacerts
```

这里的 `java.home` 属性指的是 JRE 的路径。

为了测试目的，你可以使用 `keytool` 工具创建一个自签名的证书。如果 Java 的“bin”目录已经在可执行目录之中，你可以输入：

```
keytool -genkey -alias <keyname> -keystore <url>
```

这里的 `keyname` 是你要给出的密钥别名，比如“mykeyname”，`url` 是存放密钥的文件位置，通常就是如上所述的 `cacerts` 文件。

现在将提示你输入密码。除非你已经改动了缺省值，否则它将是“changeit”（提示你去修改）。下面将提示你输入姓名、部门、单位、城市、州、国家等信息。这些信息将存放在证书文件内。最后，将询问你这个密钥的密码。如果你真的关心安全问题，你可以另外设置密码，不过缺省密码与证书密码相同，一般这样就足够了。以上信息可以使用像 `Ant` 这样的创建工具在命令行指定。

在命令行提示下，如果你不使用参数调用 `keytool` 工具，它将显示一个包含了许多选项的列表。比如，你可能会乐于使用 `-valid` 选项，这将使你可以指定密钥合法的天数。

要确定你的密钥现在是否已经存储在 `cacerts` 文件中了，可以输入：

```
keytool -list -keystore <url>
```

然后输入前面设定的密码。在屏幕上，你的密钥也许藏身于已经在证书文件中的其它密钥之中了。

你的新证书是自己签的名，所以并不能为某个权威部门所信任。如果你使用这个证书来为 JAR 文件签名，最终用户将得到警告信息，它会强烈建议用户不要使用你的软件。你和你的用户不得不忍受这些，直到你出于商业目的为受信任的证书付费为止。

要为你的 JAR 文件签名，可以使用标准 Java 工具 `jarsigner`：

```
jarsigner -keystore <url> <jarfile> <keyname>
```

这里 `url` 表示 `cacerts` 文件的位置，`jarfile` 是 JAR 文件的名称，`keyname` 是密钥的别名。这里你将再次被提示输入密码。

现在你就拥有了一个JAR文件，它被标识成是采用你提供的密钥进行签名的，这样就能保证在你签名之后它没有被篡改过（比如，没有文件被修改、添加、删除等等）。

现在你要做的就是确保HTML文件中的applet标签包含了“archive”属性，它用来指定JAR文件的名称。

对于 plugin, applet 标签要稍微复杂一些，不过你可以创建一个像下面这样的简单标记：

```
<APPLET
  CODE=package.AppletSubclass.class
  ARCHIVE = my.jar.jar
  WIDTH=300
  HEIGHT=200>
</APPLET>
```

然后运行HTMLConverter工具（可随JDK一起免费下载），它可以为你生成正确的applet标签。

现在，当有客户下载你的applet时，他们将被告知，正在载入一个已签名的applet，并提示是否信任签名人。如前所述，你这个处于测试目的的证书不具有很高的可信度，所以用户会得到警告。如果他们选择信任你的applet，此时applet就好像一个普通应用程序一样，能完全访问客户的系统。

本书的源代码，可以从www.BruceEckel.com下载，里面包含了完整的运行配置文件，以及一个Ant构建脚本，用来正确编译和构建这个项目。

JNLP 与 Java Web Start

经过签名的applet功能强大，能够有效取代应用程序，不过它们只能在浏览器中运行。这就需要在客户机上运行浏览器，从而增加了额外的开销；同时，它也限制了applet的用户界面，常常带来视觉上的混乱。因为Web浏览器有自己的菜单和工具条，它们会显示在applet的上方。

Java网络发布协议（JNLP, Java Network Launch Protocol）在不牺牲applet优点的前提下，解决了这个问题。通过一个JNLP程序，你可以在客户机上下载和安装单机版的Java应用程序。你可以通过命令行，桌面图标，或者与你的JNLP实现一起安装的应用程序管理器来执行这个程序。应用程序甚至可以从其被下载的网站运行。

一个JNLP应用程序可以在运行时刻从互联网上动态下载资源，并且能自动检查程序的版本（如果用户连接到互联网上的话）。也就是说，它能把applet的所有优点和应用程序的优点结合起来。

与applet类似，客户机系统也会小心对待JNLP应用程序的。JNLP程序是基于Web的，很容易下载，但它也可能是恶意程序。有鉴于此，JNLP程序遵循与applet一样的沙盒安全限

制。与applet类似，它们也能被部署到签名的JAR文件中，这能让用户选择是否信任签名人。与applet不同的是，如果它们被部署到未签名的JAR文件中，它们通过JNLP API提供的服务仍然能够请求访问客户系统上的特定资源（在程序运行期间，用户必须同意此请求）。

因为JNLP描述的是一个协议，而不是实现，所以要使用JNLP，你需要它的一个实现。Java Web Start，或者简称为JAWS，是由Sun免费提供的官方参考实现。你要做的就是下载并安装它，如果把它用于开发，那么你就确保它的JAR文件处于classpath路径中。如果从Web服务器上部署你的JNLP应用程序，必须确保服务器能够识别MIME类型application/x-java-jnlp-file。你用的如果是最新版的Tomcat服务器

（<http://jakarta.apache.org/tomcat>），那么它已经帮你配置好了。如果你用的是别的服务器，就要参考一下用户指南。

创建一个JNLP应用程序并不困难。你要先编写一个标准应用程序，然后把它打包到一个JAR文件中。这时，你需要提供一个启动文件，它是一个简单的XML文件，用来告诉客户下载和安装这个程序所需的所有信息。如果你不准备为JAR文件签名，那么对于你将来在客户机上访问的每种类型的资源，必须使用JNLP API提供的服务进行访问。

下面是另一种使用JFileChooser对话框的例子，不过这次是使用JNLP服务来打开对话框，所以这个类可以被打包进未经签名的JAR文件，然后作为JNLP应用程序部署。

```
//: c14:JnlpFileChooser.java
// Opening files on a local machine with JNLP.
// {Depends: javaws.jar}
package c14.jnlp;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.jnlp.*;

public class JnlpFileChooser extends JFrame {
    private JTextField filename = new JTextField();
    private JButton
        open = new JButton("Open"),
        save = new JButton("Save");
    private JEditorPane ep = new JEditorPane();
    private JScrollPane jsp = new JScrollPane();
    private FileContents fileContents;
    public JnlpFileChooser() {
        JPanel p = new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
```

```

Container cp = getContentPane();
jsp.getViewPort().add(ep);
cp.add(jsp, BorderLayout.CENTER);
cp.add(p, BorderLayout.SOUTH);
filename.setEditable(false);
p = new JPanel();
p.setLayout(new GridLayout(2,1));
p.add(filename);
cp.add(p, BorderLayout.NORTH);
ep.setContentType("text");
save.setEnabled(false);
}

class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileOpenService fs = null;
        try {
            fs = (FileOpenService)ServiceManager.lookup(
                "javax.jnlp.FileOpenService");
        } catch (UnavailableServiceException use) {
            throw new RuntimeException(use);
        }
        if(fs != null) {
            try {
                fileContents = fs.openFileDialog(".",
                    new String[]{"txt", "*"});
                if(fileContents == null)
                    return;
                filename.setText(fileContents.getName());
                ep.read(fileContents.getInputStream(), null);
            } catch (Exception exc) {
                throw new RuntimeException (exc);
            }
            save.setEnabled(true);
        }
    }
}

class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileSaveService fs = null;
        try {
            fs = (FileSaveService)ServiceManager.lookup(
                "javax.jnlp.FileSaveService");
        } catch (UnavailableServiceException use) {
            throw new RuntimeException(use);
        }
    }
}

```

```

    }
    if(fs != null) {
        try {
            fileContents = fs.saveFileDialog(".",
                new String[]{"txt"},
                new ByteArrayInputStream(
                    ep.getText().getBytes()),
                fileContents.getName());
            if(fileContents == null)
                return;
            filename.setText(fileContents.getName());
        } catch (Exception exc) {
            throw new RuntimeException (exc);
        }
    }
}

}

}

public static void main(String[] args) {
    JnlpFileChooser fc = new JnlpFileChooser();
    fc.setSize(400, 300);
    fc.setVisible(true);
}
} ///:~

```

注意，FileOpenService和FileCloseService类是从javax.jnlp包引入的，在代码中没有一处是JFileChooser对话框直接的。这里使用的两个服务必须使用ServiceManager.lookup()方法进行请求，客户系统上的资源只能通过此方法返回的对象进行访问。这里，客户系统中的文件通过JNLP提供的FileContent接口进行读写，任何通过诸如File或FileReader对象直接访问资源的企图都将导致抛出SecurityException异常，这与在未经签名的applet中执行这些操作得到的结果相似。如果你想用这些类，却不愿为JNLP的服务接口所限制，那么就必须对JAR文件签名（有关对JAR文件签名，请参考上一节）。

现在我们已经有了一个可运行的类，它使用了JNLP服务，接下来就要把类打包进JAR文件，然后编写一个启动文件。这里有一个为上面的例子准备的适合的启动文件。

```

<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec = "1.0+"
    codebase="file:///C:\TIJ3code\c14\jnlp"
    href="filechooser.jnlp">
    <information>
        <title>FileChooser demo application</title>
        <vendor>Mindview Inc.</vendor>
        <description>
            Jnlp File choose Application

```

```

</description>
<description kind="short">
    A demonstration of opening, reading and
    writing a text file
</description>
<icon href="images/tijicon.gif"/>
<offline-allowed/>
</information>
<resources>
    <j2se version="1.3+"/>
    <jar href="jnlpfilechooser.jar" download="eager"/>
</resources>
<application-desc
    main-class="c14.jnlp.JnlpFileChooser"/>
</jnlp>

```

这个启动文件需要以扩展名.jnlp保存，这里命名为filechooser.jnlp，它与JAR文件在同一个目录。

你可以发现，这是一个XML文件，包含了一个“<jnlp>”标签。它有一些子元素，其涵义大多不言自明。

Jnlp 元素的 spec 属性可以告诉客户系统此 JNLP 程序所遵循规范的版本。codebase 属性指向可以找到启动文件和资源的目录。通常，它指向 Web 服务器的 URL，不过这里它指向的是本地机器上的目录，这是一个不错的测试程序的方法。href 属性必须指定这个启动文件的名称。

information 标签也有几个子元素，它们提供了有关应用程序的信息。这些信息将用于 Java Web Start 的管理控制台或者类似程序，它们可以安装 JNLP 程序，并且可以让用户从命令行运行程序，使其更快捷等等。

resources 标签与 HTML 文件中的 applet 标记功能很相似。j2se 子元素指定程序运行时需要的 j2se 版本，jar 元素的 href 属性指定包含了 .class 文件的 JAR 文件。jar 元素还有一个 download 属性，它的值可以是“eager”或者“lazy”，这可以告诉 JNLP 的实现在程序运行之前，是否需要下载整个 JAR 文件。

application-desc 元素的 main-class 属性能告诉 JNLP 实现 JAR 文件中的哪个类是可执行的类，即指定程序的入口。

Jnlp 标签的另一个有用的子元素是 security 标签，这里并没有演示它。下面是一个 security 标签的例子：

```

<security>
    <all-permissions/>
</security>

```

当你把程序部署在一个已签名的JAR文件中后，就可以使用security标签了。上面的例子不需要这个标签是因为所有本地资源都是通过JNLP服务进行访问的。

还有其它一些标签可用，其细节部分可以参考规范

<http://java.sun.com/products/javawebstart/download-spec.html>。

现在已经写好了.jnlp文件，下面你要在HTML页面里添加上指向这个文件的超链接。这个页面将是它的下载页面。你可能会有一个布局复杂的、对你的应用的详细介绍，不过只要把下列内容：

```
<a href="filechooser.jnlp">click here</a>
```

放进你的HTML页面，之后，你就可以通过单击链接来安装JNLP应用程序了。一旦程序下载完毕，你可以使用管理控制台对它进行配置。如果你在Windows系统上使用Java Web Start，你将被提示是否为程序创建一个快捷方式供下次使用。这个行为是可以配置的。

本书的源代码，可以从www.BruceEckel.com下载，里面包含了全部可以运作的配置文件，以及一个Ant构建脚本，用来正确编译和构建这个项目。

这里只介绍了两种JNLP服务，当前版本的JNLP规范包含了七种服务。每个服务都被设计用来执行特定的任务，比如打印、以及和剪贴板有关的剪切和粘贴等。对这些服务的深入讨论，已经超出了本书的范围。

编程技巧

Java的GUI编程是一项不断进化的技术，在从Java 1.0/1.1 到Java 2 的Swing类库的演化过程中，他产生过某些非常明显的变化，你也许曾经在某些Swing例子中见到过这些过时的编程技巧。此外，Swing使你得以采用比老式模型更好的方式来编程。在本节中，我们将通过引入和分析一些编程技巧来演示这方面的问题。

动态绑定事件

Swing 事件模型的好处之一就是灵活。通过单一的方法调用就可以添加或移除事件行为。以下例子演示了这一点：

```
//: c14:DynamicEvents.java
// You can change event behavior dynamically.
// Also shows multiple actions for an event.
// <applet code=DynamicEvents
// width=250 height=400></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```

import java.util.*;
import com.bruceeckel.swing.*;

public class DynamicEvents extends JApplet {
    private java.util.List list = new ArrayList();
    private int i = 0;
    private JButton
        b1 = new JButton("Button1"),
        b2 = new JButton("Button2");
    private JTextArea txt = new JTextArea();
    class B implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            txt.append("A button was pressed\n");
        }
    }
    class CountListener implements ActionListener {
        private int index;
        public CountListener(int i) { index = i; }
        public void actionPerformed(ActionEvent e) {
            txt.append("Counted Listener " + index + "\n");
        }
    }
    class B1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            txt.append("Button 1 pressed\n");
            ActionListener a = new CountListener(i++);
            list.add(a);
            b2.addActionListener(a);
        }
    }
    class B2 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            txt.append("Button2 pressed\n");
            int end = list.size() - 1;
            if(end >= 0) {
                b2.removeActionListener(
                    (ActionListener)list.get(end));
                list.remove(end);
            }
        }
    }
    public void init() {
        Container cp = getContentPane();
        b1.addActionListener(new B());
    }
}

```



```

        b1.addActionListener(new B1());
        b2.addActionListener(new B());
        b2.addActionListener(new B2());
        JPanel p = new JPanel();
        p.add(b1);
        p.add(b2);
        cp.add(BorderLayout.NORTH, p);
        cp.add(new JScrollPane(txt));
    }
    public static void main(String[] args) {
        Console.run(new DynamicEvents(), 250, 400);
    }
} ///:~

```

程序中引入的新技巧有：

1. 每个按钮都关联了不只一个监听器。通常，组件以“多路”（**multicast**）方式处理事件，即你能为一个事件注册多个监听器。也有一些特殊组件，只能以“单路”（**unicast**）方式处理事件，你要是为这种组件注册多个事件，会得到 **TooManyListenersException** 异常。
2. 在程序的执行期间，按钮 **b2** 的监听器被动态添加和移除。添加动作的方法前面已经见过，每个组件还有一个 **removeXXXListener()** 方法，用来移除相应类型的监听器。

这种灵活性为编写程序时提供了更强的能力。

你要注意，不能保证“事件监听器被调用的顺序”与“它们被添加的顺序”相同（尽管实际上大部分实现能做到这一点）。

将业务逻辑与用户界面逻辑分离

一般来说，你要对类进行设计，使得每个类“只作一件事”。当牵涉到有关用户界面的代码时，这一点尤其重要，因为“正在做的”和“怎样显示”很容易混在一起。这种耦合阻碍了代码的重用。应该把“业务逻辑”从GUI代码中分离出来。这样，你不仅能更容易地重用业务逻辑代码，而且重用GUI代码也变得更加轻松。

另外一个方面和“多层”系统有关，在这种系统中，“业务对象”位于某台完全隔离的机器。这种以业务规则为中心的系统，能够有效地把规则的变更即时应用于所有新启动的事务中，所以这也成了一种用来建立系统的引入注目的方式。这些业务对象可以应用于许多不同的程序，所以它们不应该与某个特定的显示方式捆绑在一起。它们只执行与业务相关的操作就足够了¹¹。

¹¹ 《Thinking in Enterprise Java》（可以从www.BruceEckel.com得到）一书对此概念进行了详细讨论。

下面的例子演示了把业务逻辑从 GUI 代码中分离出来是多么地简单：

```
///  
// Separating GUI logic and business objects.  
// <applet code=Separation width=250 height=150></applet>  
import javax.swing.*;  
import java.awt.*;  
import javax.swing.event.*;  
import java.awt.event.*;  
import java.applet.*;  
import com.bruceeckel.swing.*;  
  
class BusinessLogic {  
    private int modifier;  
    public BusinessLogic(int mod) { modifier = mod; }  
    public void setModifier(int mod) { modifier = mod; }  
    public int getModifier() { return modifier; }  
    // Some business operations:  
    public int calculation1(int arg) { return arg * modifier; }  
    public int calculation2(int arg) { return arg + modifier; }  
}  
  
public class Separation extends JApplet {  
    private JTextField  
        t = new JTextField(15),  
        mod = new JTextField(15);  
    private JButton  
        calc1 = new JButton("Calculation 1"),  
        calc2 = new JButton("Calculation 2");  
    private BusinessLogic bl = new BusinessLogic(2);  
    public static int getValue(JTextField tf) {  
        try {  
            return Integer.parseInt(tf.getText());  
        } catch (NumberFormatException e) {  
            return 0;  
        }  
    }  
  
    class Calc1L implements ActionListener {  
        public void actionPerformed(ActionEvent e) {  
            t.setText(Integer.toString(  
                bl.calculation1(getValue(t))));  
        }  
    }  
  
    class Calc2L implements ActionListener {
```

```

        public void actionPerformed(ActionEvent e) {
            t.setText(Integer.toString(
                bl.calculation2(getValue(t))));
        }
    }
    // If you want something to happen whenever
    // a JTextField changes, add this listener:
    class ModL implements DocumentListener {
        public void changedUpdate(DocumentEvent e) {}
        public void insertUpdate(DocumentEvent e) {
            bl.setModifier(getValue(mod));
        }
        public void removeUpdate(DocumentEvent e) {
            bl.setModifier(getValue(mod));
        }
    }
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        calc1.addActionListener(new Calc1L());
        calc2.addActionListener(new Calc2L());
        JPanel p1 = new JPanel();
        p1.add(calc1);
        p1.add(calc2);
        cp.add(p1);
        mod.getDocument().addDocumentListener(new ModL());
        JPanel p2 = new JPanel();
        p2.add(new JLabel("Modifier:"));
        p2.add(mod);
        cp.add(p2);
    }
    public static void main(String[] args) {
        Console.run(new Separation(), 250, 100);
    }
} ///:~

```

BusinessLogic是执行某些操作的简单类，它只负责执行自己的操作，没有任何迹象能表明这个类会在GUI环境中被用到。

Separation负责所有用户界面的细节部分，它只通过**public**接口访问**BusinessLogic**。所有操作都以用户界面和**BusinessLogic**对象之间的信息交换为中心。所以，**Seperation**是只负责用户界面的类。既然**Seperation**只知道它要访问一个**BusinessLogic**对象（即，这里不是高耦合），所以它同样也可以访问别的类型的对象，而不会带来太多的麻烦。

当你修改遗留代码，以适应Java的时候，遵循“用户界面和业务逻辑相分离”的原则，会让你的工作更容易。

典型方式

内部类、Swing事件模型，以及过时的AWT事件模型仍旧被支持的事实，再加上新库的功能依赖于某些过时的程序，所有这些都为代码设计过程引入了新的混乱。现在你有了更多的花样使你编写出的代码变得一塌糊涂。

除了特殊情况以外，你应该使用最简单和最清晰的方式：用监听器类（通常以内部类实现）来解决事件处理问题。本章中绝大多数例子都采用了这种方式。

采用这种方式，你能够在程序中减少判断事件来源的语句。每段事件处理代码应该只与“动作”有关，而不是检查事件类型。这是编写代码的最好方式；这样不仅容易理解，而且更容易阅读和维护。

Swing 与并发

使用Swing编写程序的时候，会很容易忘记你其实也是在编写多线程程序。不用明确创建Thread对象的事实同时也意味着，线程问题会让你大吃一惊。典型地，当你编写一个Swing程序，或者任何与视窗显示有关的GUI程序时，程序的大部分动作由事件驱动，也就是说，只有用户使用鼠标在GUI组件上点击，或者是按下键盘的时候，才会产生事件，否则程序什么也不作。

要记住的是，始终存在着一个Swing事件调度线程，它用来依次对Swing的所有事件进行调度。如果你要确保程序不会遭受死锁或者竞争条件带来的麻烦，你就需要考虑这些问题。

本节将讨论几个问题，在Swing下使用线程的时候，它们很值得注意。

Runnable 回顾

在第 13 章，我建议你在实现Runnable接口之前要三思。当然，如果你必须从某个类继承，同时还要使它具有线程行为的话，实现Runnable是正确的选择。下面的例子探讨了这种情况，编写了一个实现Runnable接口的JPanel，它可以在其自身上绘制不同的颜色。程序启动的时候将从命令行接受参数，以决定颜色块的数目和颜色变化的时间间隔（线程休眠的时间）。尝试用不同的值运行程序，你会发现一些有趣的问题，以及线程的一些难以言表的特性：

```
//: c14:ColorBoxes.java
// Using the Runnable interface.
// <applet code=ColorBoxes width=500 height=400>
```

```

// <param name=grid value="12">
// <param name=pause value="50"></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

class CBox extends JPanel implements Runnable {
    private Thread t;
    private int pause;
    private static final Color[] colors = {
        Color.BLACK, Color.BLUE, Color.CYAN,
        Color.DARK_GRAY, Color.GRAY, Color.GREEN,
        Color.LIGHT_GRAY, Color.MAGENTA,
        Color.ORANGE, Color.PINK, Color.RED,
        Color.WHITE, Color.YELLOW
    };
    private static Random rand = new Random();
    private static final Color newColor() {
        return colors[rand.nextInt(colors.length)];
    }
    private Color cColor = newColor();
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(cColor);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
    public CBox(int pause) {
        this.pause = pause;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        while(true) {
            cColor = newColor();
            repaint();
            try {
                t.sleep(pause);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

```

```

    }
}

public class ColorBoxes extends JApplet {
    private boolean isApplet = true;
    private int grid = 12;
    private int pause = 50;
    public void init() {
        // Get parameters from Web page:
        if(isApplet) {
            String gsize = getParameter("grid");
            if(gsize != null)
                grid = Integer.parseInt(gsize);
            String pse = getParameter("pause");
            if(pse != null)
                pause = Integer.parseInt(pse);
        }
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(grid, grid));
        for(int i = 0; i < grid * grid; i++)
            cp.add(new CBox(pause));
    }
    public static void main(String[] args) {
        ColorBoxes applet = new ColorBoxes();
        applet.isApplet = false;
        if(args.length > 0)
            applet.grid = Integer.parseInt(args[0]);
        if(args.length > 1)
            applet.pause = Integer.parseInt(args[1]);
        Console.run(applet, 500, 400);
    }
} ///:~

```

ColorBoxes是一个普通的applet/应用程序，它在init()中设置了GUI。这里使用了GridLayout布局管理器，以得到二维的网格单元。然后加入适当数目的CBox对象来填充网格，并为这些对象传入pause值。在main()中可以发现，pause和grid都有缺省值，通过命令行传入的值或者applet参数可以修改这些值。

所有的工作由CBox完成。它从JPanel继承，并且实现了Runnable接口，所以它既是JPanel，同时也具有线程行为。记住，当你实现Runnable的时候，你得到的不是一个Thread，而只是一个具有run()方法的类。所以，你必须把Runnable对象传递给Thread构造器，才能明确创建一个Thread对象，之后才能调用start()(在CBox的构造器中调用)。在Cbox中，这个线程以t表示

注意，`colors`数组列出了`Color`类里面的所有颜色。它被用于`newColor()`方法，以随机选择一种颜色。组件当前的颜色由`cColor`表示。

`paintComponent()`方法非常简单；先把填充颜色设置成`cColor`指定的颜色，然后填充整个`JPanel`。

在`run()`方法的无穷循环中，先把`cColor`设置成新的随机颜色，然后调用`repaint()`进行显示。接着调用`sleep()`，使线程休眠一段时间，这个时间可以从命令行指定。

这个设计很灵活，并且线程与每个`Jpanel`都联系到了一起，所以你可以试着创建任意多的线程。（事实上，这个数目受你的JVM能够自如地处理的线程数目所限制。）

这个程序还能作一个有趣的基准测试，因为可以针对不同的JVM线程实现，用它来演示这些实现的动态性能以及行为上的差异。

管理并发

当你在类的`main()`方法中，或者在一个独立线程中，准备修改任何Swing组件属性的时候，要注意，Swing的事件调度线程可能会与你竞争同一资源¹²。

下面的例子将向你演示，如果你不注意事件调度线程的话，可能会得到意料之外的结果：

```
//: c14:EventThreadFrame.java
// Race Conditions using Swing Components.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.Console;

public class EventThreadFrame extends JFrame {
    private JTextField statusField =
        new JTextField("Initial Value");
    public EventThreadFrame() {
        Container cp = getContentPane();
        cp.add(statusField, BorderLayout.NORTH);
        addWindowListener(new WindowAdapter() {
            public void windowOpened(WindowEvent e) {
                try { // Simulate initialization overhead
                    Thread.sleep(2000);
                } catch (InterruptedException ex) {
                    throw new RuntimeException(ex);
                }
            }
        });
    }
}
```

¹² 本节由Jeremy Meyer编写。

```

        statusField.setText("Initialization complete");
    }
});
}
public static void main (String[] args) {
    EventThreadFrame etf = new EventThreadFrame();
    Console.run(etf, 150, 60);
    etf.statusField.setText("Application ready");
    System.out.println("Done");
}
} ///:~

```

很容易看出程序期望得到的效果。在`main()`中，先创建一个`EventThreadFrame`，然后使用`Console.run()`运行它。在窗体被创建和运行之后，把窗体文本域的内容设置成“Application ready”，然后在退出`main()`之前，向控制台显示“Done”。

在创建窗体的构造器内，先向窗体添加了一个文本域，内容为“Initial Value”；然后添加了一个事件监听器，用来监听窗体打开的事件。一旦窗体的`setVisible(true)`方法被调用（在`Console.run()`中被调用），`JFrame`就会收到这个事件，所以对于任何会影响窗体外观的初始化代码，很适合在这里执行。在本例中，调用`sleep()`来模拟那些会耗费时间的初始化动作。完成这些之后，文本域的内容被设置成“Initialization complete”。

你可能会认为，文本域内容的显示顺序应该先是“Initial Value”，然后是“Initialization complete”，接着是“Application Ready”。最后“Done”将显示在控制台上。但是实际上：在`EventThreadFrame`有机会处理事件之前，`main`线程已经在`TextField`上调用了`setText()`方法，也就是说，“Application ready”可能先于“Initialization complete”之前显示。实际的过程甚至也不一定是这个顺序。根据你的系统速度，`Swing`事件调度线程可能已经忙于处理`windowOpened`事件了，所以在此事件之后文本域才会显示内容，而那时内容已经被修改成为“Initialization Complete”。因为文本域的内容已经被设置成最后一个，所以“Application ready”不会被显示。更糟糕的是，在任何动作发生之前，“Done”已经出现在控制台上了！

这种不恰当并且有些难以预测的效果原因很简单，出现了两个线程，它们之间需要一些同步。这表明`Swing`中有时会遇到一些线程问题。要解决整个问题，你必须确保在任何情况下，只能在事件调度线程里修改`Swing`组件的属性。

实际上这很简单，可以使用`Swing`提供的两种机制：`SwingUtilities.invokeLater()`和`SwingUtilities.invokeAndWait()`。它们可以作大部分的工作，也就是说，你不必自己去那些很复杂的同步和线程编程。

它们都接受`Runnable`对象作为参数，并且在`Swing`的事件处理线程中，只有当事件队列中的任何未处理事件都被处理完毕之后，它们才会调用`Runnable`对象的`run()`方法。

```

///: c14:InvokeLaterFrame.java
// Eliminating race Conditions using Swing Components.

```



```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.Console;

public class InvokeLaterFrame extends JFrame {
    private JTextField statusField =
        new JTextField("Initial Value");
    public InvokeLaterFrame() {
        Container cp = getContentPane();
        cp.add(statusField, BorderLayout.NORTH);
        addWindowListener(new WindowAdapter() {
            public void windowOpened(WindowEvent e) {
                try { // Simulate initialization overhead
                    Thread.sleep(2000);
                } catch (InterruptedException ex) {
                    throw new RuntimeException(ex);
                }
                statusField.setText("Initialization complete");
            }
        });
    }
    public static void main(String[] args) {
        final InvokeLaterFrame ilf = new InvokeLaterFrame();
        Console.run(ilf, 150, 60);
        // Use invokeAndWait() to synchronize output to prompt:
        // SwingUtilities.invokeLater(new Runnable() {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                ilf.statusField.setText("Application ready");
            }
        });
        System.out.println("Done");
    }
} ///:~

```

`SwingUtilities.invokeLater()` 被传入一个实现了 `Runnable` 接口的匿名内部类对象，它负责调用文本域的 `setText()` 方法。这使得 `Runnable` 对象被作为一个事件而放进了事件调度线程的队列，所以事件调度线程是在首次处理完所有未决事件之后才调用了 `setText()` 方法。也就是说，`windowOpening` 事件被处理完毕之后，文本域才会显示 “Application ready”，这也是我们想要的结果。

`invokeLater()` 是异步方法，所以它立刻返回。这很有用，因为它不会阻塞，所以你的代码运行流畅。不过，它不能解决 “Done” 的问题，这个字符串仍旧会在发生任何动作之前，就出现在控制台上。

要解决这个问题，应该使用`invokeAndWait()`，而不是`invokeLater()`来把文本域的内容设置成“Application Ready”。它是同步方法，也就是说，它将一直阻塞，直到事件处理完毕之后才会返回。只有在文本域的内容被设置之后，才会运行`System.out.println("Done")`这一行，所以它是最后执行的语句。这就得到了完全可以预测的正确行为。

使用`invokeAndWait()`将会带来一个产生死锁的必要条件，所以要使用`invokeAndWait()`，必须小心控制共享资源，尤其是在多个线程中调用此方法的时候。

与`invokeAndWait()`相比，使用`invokeLater()`更常见，不过要记住，如果你要在初始化之后的任何时刻设置Swing组件的属性，你应该通过这些方法进行。

可视化编程与 JavaBean

本书到目前为止，你已经看到了Java在编写可重用代码方面具有的价值。类是“最可重用”的代码单元，因为它把性质（字段）和行为（方法）聚合成一个单元，所以它既可以被直接重用，也可以通过组合或继承得到重用。

继承和多态是面向对象编程的关键部分，不过在大多数情况下，当你把程序放在一起时，你真正希望的是能够精确地满足你的需求的组件。你乐于把这些部分集成到你的设计中，就好像电子工程师把芯片放在电路板上一样。看来，应该有某种方法能加速这种“模块化装配”形式的编程。

“可视化编程”是首先成功的方式，而且非常成功。首先是微软公司的Visual BASIC (VB)，接着是Borland公司的Delphi（JavaBean的设计主要就是受到了它的启发），它属于第二代产品。伴随着这些编程工具，组件也以可视的形式表现，因为它们通常表现为一些诸如按钮或文本域的可视组件，所以这种可视化很有意义。实际上可视化表示通常就是组件在运行的程序中可被观察到的方式。所以可视化编程的部分过程就包括了从选用区选择组件，然后把它拖放到窗体上。在你拖放的时候，应用程序构造工具会帮你生成相应的代码，这些代码将在程序实际运行的时候创建相应的组件。

简单地把组件拖放到窗体上一般还不足以完成程序。通常，你必须改变组件的特征，比如颜色、文本、所连接的数据库等等。可以在设计期间被修改的特征被称为属性。你可以使用应用程序构建器工具对组件的属性进行设置，在你创建的程序时候，这些配置信息将被保存，这样就可以在程序运行的时候恢复这些信息。

现在你可能已经习惯于对象并不仅仅包含了特征；它还包括一组行为。在设计期间，可视化组件的行为部分由事件表示，意思是“这是可以在组件上发生的动作”。通常，你通过为事件编写代码来决定事件发生的时候该如何处理。

关键在于：应用程序构建工具能够使用反射机制来动态地向组件查询，以找出组件具有的属性和支持的事件。一旦查询完毕，它将显示属性并且允许你进行修改（在你构建程序的时候会把状态保存起来），此外它还能显示事件。通常，你需要在事件上双击或者作出类

似操作，应用程序构建工具会为你生成一个与此事件关联的代码框架。这时你要做的就是编写事件发生时将要执行的代码。

所有这些加起来就是应用程序构建工具能够帮你完成的大量工作。这样，你就能集中精力于程序的外观和功能，然后依赖应用程序构建工具为你管理相关的细节。可视化编程工具如此成功的原因就在于，它们极大地加速了程序的构建过程（当然包括了用户界面部分，而且常常对程序其它部分的编写也有帮助）。

JavaBean 是什么？

在剥去层层包裹之后，组件只不过就是一段代码，通常以类的形式出现。对于组件来说，关键在于要具有“能够被应用程序构建工具侦测其属性和事件的”能力。要编写一个VB组件，程序员不得不根据某些固定规则编写相当复杂的代码，以暴露出组件的属性和方法。Delphi作为第二代可视化编程工具，语言本身就是围绕着可视化编程而设计，所以编写可视化组件要容易很多。然而，通过引入JavaBean，Java把可视化组件的编写带到了最高的层次，因为Bean就是一个类。要编写一个Bean，你不必添加任何特殊代码或者使用任何特殊的语言功能。实际上你唯一要做的就是，对方法的名称作少许修改。通过方法的名称就能告诉应用程序构建工具，这是一个属性、一个事件还是只是一个普通方法。

在JDK文档中，命名规则使用了错误的术语“设计模式（design pattern）”。这是不恰当的，因为设计模式（参考 www.BruceEckel.com 网站上的《Thinking in Patterns (with Java)》）即使不与这些概念混在一起，其本身也具有足够的挑战性。这不是设计模式，这只是一个命名规则，而且非常简单：

1. 对于一个名称为 **xxx** 的属性，通常你要写两个方法：**getXxx()**和**setXxx()**。注意，把“**get**”或“**set**”后面的第一个字母换成小写就能得到属性的名称。“**get**”方法返回的类型要与“**set**”方法里参数的类型相同。属性的名称与“**get**”和“**set**”所依据的类型毫无关系。
2. 对于布尔型属性，你可以使用以上“**get**”和“**set**”的方式，不过你也可以把“**get**”替换成“**is**”。
3. Bean的普通方法不必遵循以上的命名规则，不过它们必须是公共方法。
4. 对于事件，你要使用Swing中处理监听器的方式。这与你前面见到的完全相同：
addBounceListener(BounceListener) 和
removeBounceListener(BounceListener)用来处理**BounceEvent**事件。大多数情况下，内置的事件和监听器就能够满足你的需求，不过你还是可以自己编写事件和监听器接口。

把过时代码与新代码进行比较，你可能会注意到这样的问题：很多方法的名称有微小的、明显无意义的变化。上面第一点回答了这个问题。现在你就能明白这些变化绝大多数是为了遵守“**get**”和“**set**”的命名规则，这样就能使组件成为JavaBean。

我们可以使用这些规则编写一个简单的 Bean：

```
//: frogbean:Frog.java
```

```

// A trivial JavaBean.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
    private int jumps;
    private Color color;
    private Spots spots;
    private boolean jmpr;
    public int getJumps() { return jumps; }
    public void setJumps(int newJumps) {
        jumps = newJumps;
    }
    public Color getColor() { return color; }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Spots getSpots() { return spots; }
    public void setSpots(Spots newSpots) {
        spots = newSpots;
    }
    public boolean isJumper() { return jmpr; }
    public void setJumper(boolean j) { jmpr = j; }
    public void addActionListener(ActionListener l) {
        //...
    }
    public void removeActionListener(ActionListener l) {
        // ...
    }
    public void addKeyListener(KeyListener l) {
        // ...
    }
    public void removeKeyListener(KeyListener l) {
        // ...
    }
    // An "ordinary" public method:
    public void croak() {
        System.out.println("Ribbet!");
    }
} ///:~

```

首先，你可以发现这只是一个类。通常，所有的域都是私有的，只能通过方法进行访问。根据命名规则，Bean的属性是jumps, color, spots和jumper（注意属性名称第一个字母的大小写变化）。对于前三个属性，其名称与其内部标识符的名称相同，不过通过jumper你会发现，属性名称并不要求你为内部变量使用任何特定的标识符（或者，实际上甚至不需要有任何内部变量与属性对应）。

根据“add”和“remove”方法对相关监听器的命名可以看出，这个Bean所处理的事件是ActionEvent和KeyEvent。最后，你可以发现普通方法croak()也是Bean的一部分，这只是因为它是公共方法，而不是因为它遵循了任何命名规则。

使用内省器（Introspector）来抽取出 BeanInfo

JavaBean模型最关键部分之一，发生在当你从选用区拖动一个Bean，然后把它放置到窗体上的时候。应用程序构建工具必须能够创建这个Bean（如果有缺省构造器就可以创建），然后在不访问Bean的源代码的情况下，抽取出所有必要信息，以创建属性和事件处理器的列表。

部分解决方案在第 10 章就出现了：Java的反射机制能发现未知类的所有方法。对于解决JavaBean的这个问题，这是个完美的方案，你不用像其它可视化编程语言那样使用任何语言附加的关键字。实际上，Java语言里加入反射机制的主要原因之一就是支持JavaBean（尽管反射也支持对象序列化和远程方法调用）。所以，你也许会认为应用程序构建工具的编写者将使用反射来抽取Bean的方法，然后在方法里面查找出Bean的属性和事件。

这当然是可行的，不过Java的设计者希望提供一个标准工具，不仅要使Bean用起来简单，而且对于创建更复杂的Bean，能够提供一个标准方法。这个工具就是Introspector类，这个类最重要的就是静态的getBeanInfo()方法。你向这个方法传递一个Class对象引用，它能够完全侦测这个类，然后返回一个BeanInfo对象，你可以通过这个对象得到Bean的属性、方法和事件。

通常，你不用关心这些问题；你使用的大多数 Bean 可能从供应商那里直接购买，你不必知道底层的魔幻般的细节。你只需把 Bean 拖动到窗体上，配置它们的属性，然后为感兴趣的事件编写处理程序即可。不过，使用 Introspector 来显示 Bean 的信息是个既有趣，又值得学习的练习。下面就是这个工具：

```
//: c14:BeanDumper.java
// Introspecting a Bean.
import java.beans.*;
import java.lang.reflect.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
```

```

public class BeanDumper extends JFrame {
    private JTextField query = new JTextField(20);
    private JTextArea results = new JTextArea();
    public void print(String s) { results.append(s + "\n"); }
    public void dump(Class bean) {
        results.setText("");
        BeanInfo bi = null;
        try {
            bi = Introspector.getBeanInfo(bean, Object.class);
        } catch(IntrospectionException e) {
            print("Couldn't introspect " + bean.getName());
            return;
        }
        PropertyDescriptor[] properties =
            bi.getPropertyDescriptors();
        for(int i = 0; i < properties.length; i++) {
            Class p = properties[i].getPropertyType();
            if(p == null) continue;
            print("Property type:\n " + p.getName() +
                "Property name:\n " + properties[i].getName());
            Method readMethod = properties[i].getReadMethod();
            if(readMethod != null)
                print("Read method:\n " + readMethod);
            Method writeMethod = properties[i].getWriteMethod();
            if(writeMethod != null)
                print("Write method:\n " + writeMethod);
            print("=====");
        }
        print("Public methods:");
        MethodDescriptor[] methods = bi.getMethodDescriptors();
        for(int i = 0; i < methods.length; i++)
            print(methods[i].getMethod().toString());
        print("=====");
        print("Event support:");
        EventSetDescriptor[] events =
            bi.getEventSetDescriptors();
        for(int i = 0; i < events.length; i++) {
            print("Listener type:\n " +
                events[i].getListenerType().getName());
            Method[] lm = events[i].getListenerMethods();
            for(int j = 0; j < lm.length; j++)
                print("Listener method:\n " + lm[j].getName());
            MethodDescriptor[] lmd =
                events[i].getListenerMethodDescriptors();

```

```

        for(int j = 0; j < lmd.length; j++)
            print("Method descriptor:\n "
                + lmd[j].getMethod());
        Method addListener= events[i].getAddListenerMethod();
        print("Add Listener Method:\n " + addListener);
        Method removeListener =
            events[i].getRemoveListenerMethod();
        print("Remove Listener Method:\n " + removeListener);
        print("=====");
    }
}

class Dumper implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String name = query.getText();
        Class c = null;
        try {
            c = Class.forName(name);
        } catch(ClassNotFoundException ex) {
            results.setText("Couldn't find " + name);
            return;
        }
        dump(c);
    }
}

public BeanDumper() {
    Container cp = getContentPane();
    JPanel p = new JPanel();
    p.setLayout(new FlowLayout());
    p.add(new JLabel("Qualified bean name:"));
    p.add(query);
    cp.add(BorderLayout.NORTH, p);
    cp.add(new JScrollPane(results));
    Dumper dmpr = new Dumper();
    query.addActionListener(dmpr);
    query.setText("frogbean.Frog");
    // Force evaluation
    dmpr.actionPerformed(new ActionEvent(dmpr, 0, ""));
}

public static void main(String[] args) {
    Console.run(new BeanDumper(), 600, 500);
}
} ///:~

```

`BeanDumper.dump()`方法执行了所有工作。首先，它试图创建一个`BeanInfo`对象，成功的话，就调用`BeanInfo`的方法得到有关其属性、方法和事件的信息。你会发现`Introspector.getBeanInfo()`方法有第二个参数，它用来告诉`Introspector`在哪个继承层次上停止查询。因为我们不关心来自`Object`的方法，所以这里的参数让`Introspector`在解析来自`Object`的所有方法前停止查询。

对于属性来说，`getPropertyDescriptors()`返回类型为`PropertyDescriptor`的数组，你可以针对每一个`PropertyDescriptor`都调用`getPropertyType()`来得到“通过属性方法设置和返回的对象”类型。然后，针对每个属性，你可以通过`getName()`方法得到它的别名(从方法名中抽取)，通过`getReadMethod()`方法得到读方法，通过`getWriteMethod()`方法得到写方法。后两个方法返回`Method`对象，它们能够用来在对象上调用相应的方法(这是反射的一部分)。

对于公共方法(包括属性方法)，`getMethodDescriptors()`方法返回类型为`MethodDescriptor`的数组。对于数组的每个元素，你可以得到相关联的`Method`对象，并显示它们的名称。

对于事件，`getEventSetDescriptors()`方法返回类型为`EventSetDescriptor`的数组(或是别的什么?)。可以对数组中的每一个元素进行查询，以得到监听器的类型、监听器类的方法，以及添加和移除监听器的方法。`BeanDumper`程序显示了所有这些信息。

在启动之后，程序强制评估 `frogbean.Frog`。下面是程序输出，这里移除了不必要的额外细节：

```
class name: Frog
Property type:
    Color
Property name:
    color
Read method:
    public Color getColor()
Write method:
    public void setColor(Color)
=====
Property type:
    Spots
Property name:
    spots
Read method:
    public Spots getSpots()
Write method:
    public void setSpots(Spots)
=====
Property type:
    boolean
```


Property name:

jumper

Read method:

```
public boolean isJumper()
```

Write method:

```
public void setJumper(boolean)
```

=====

Property type:

int

Property name:

jumps

Read method:

```
public int getJumps()
```

Write method:

```
public void setJumps(int)
```

=====

Public methods:

```
public void setJumps(int)
```

```
public void croak()
```

```
public void removeActionListener(ActionListener)
```

```
public void addActionListener(ActionListener)
```

```
public int getJumps()
```

```
public void setColor(Color)
```

```
public void setSpots(Spots)
```

```
public void setJumper(boolean)
```

```
public boolean isJumper()
```

```
public void addKeyListener(KeyListener)
```

```
public Color getColor()
```

```
public void removeKeyListener(KeyListener)
```

```
public Spots getSpots()
```

=====

Event support:

Listener type:

KeyListener

Listener method:

keyTyped

Listener method:

keyPressed

Listener method:

keyReleased

Method descriptor:

```
public void keyTyped(KeyEvent)
```

Method descriptor:

```
public void keyPressed(KeyEvent)
```

```

Method descriptor:
    public void keyReleased(KeyEvent)
Add Listener Method:
    public void addKeyListener(KeyListener)
Remove Listener Method:
    public void removeKeyListener(KeyListener)
=====
Listener type:
    ActionListener
Listener method:
    actionPerformed
Method descriptor:
    public void actionPerformed(ActionEvent)
Add Listener Method:
    public void addActionListener(ActionListener)
Remove Listener Method:
    public void removeActionListener(ActionListener)
=====

```

这里列出的是Introspector能够观察到的从你的Bean中产生的BeanInfo对象中的大多数信息。你可以观察到属性的类型和名称相互独立。注意属性名称使用小写字母（唯一例外的情况是，属性名称以连续多个大写字母开头）。记住，你在这里看到的方法名称（比如读和写方法）是从Method对象中得到的，它可以用来在对象上调用相关联的方法。

公共方法列表中既包括了那些与属性或事件无关的方法，比如croak()，也包括了那些有关的方法。这些就是你可以通过编程在Bean上调用的所有方法，为了让你的工作更容易，应用程序构建工具可以在你编写方法调用的时候，显示这个列表。

最后，你可以发现所有事件都被完全地解析了出来，包括相关的监听器、它的方法，以及添加和移除监听器所用的方法。基本上，一旦你获得了BeanInfo对象，你就可以得到Bean的所有重要信息。你还能够调用Bean上的方法，甚至在你除了对象以外（这里又是反射的功能）再没有其它任何信息的情况下，也能够这么做。

一个更复杂的 Bean

下面的例子稍微复杂了一些，虽然它并不是很重要。它是一个JPanel，当鼠标移动的时候，可以在鼠标周围绘制小圆圈。当你按下鼠标，单词“Bang!”将出现在屏幕的中央，并且触发一个动作监听器。

你可以改变的属性包括了圆圈的大小、当你按下鼠标时所显示单词的颜色、大小和文本。BangBean 类还具有 addActionListener() 和 removeActionListener()，所以你可以自己编写监听器并与之关联，当用户在 BangBean 上单击的时候，你的监听器就会被触发。你应该能够识别它们支持的属性和事件：

```

//: bangbean:BangBean.java
// A graphical Bean.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class
BangBean extends JPanel implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Circle size
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.RED;
    private ActionListener actionListener;
    public BangBean() {
        addMouseListener(new ML());
        addMouseMotionListener(new MML());
    }
    public int getCircleSize() { return cSize; }
    public void setCircleSize(int newSize) {
        cSize = newSize;
    }
    public String getBangText() { return text; }
    public void setBangText(String newText) {
        text = newText;
    }
    public int getFontSize() { return fontSize; }
    public void setFontSize(int newSize) {
        fontSize = newSize;
    }
    public Color getTextColor() { return tColor; }
    public void setTextColor(Color newColor) {
        tColor = newColor;
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLACK);
        g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
    }
    // This is a unicast listener, which is

```

```

// the simplest form of listener management:
public void addActionListener(ActionListener l)
throws TooManyListenersException {
    if(actionListener != null)
        throw new TooManyListenersException();
    actionListener = l;
}

public void removeActionListener(ActionListener l) {
    actionListener = null;
}

class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font("TimesRoman", Font.BOLD, fontSize));
        int width = g.getFontMetrics().stringWidth(text);
        g.drawString(text, (getSize().width - width) / 2,
            getSize().height/2);
        g.dispose();
        // Call the listener's method:
        if(actionListener != null)
            actionListener.actionPerformed(
                new ActionEvent(BangBean.this,
                    ActionEvent.ACTION_PERFORMED, null));
    }
}

class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}

public Dimension getPreferredSize() {
    return new Dimension(200, 200);
}
} ///:~

```

你能首先注意到的是，BangBean实现了Serializable接口。这就意味着应用程序构建工具能够在程序设计者调整属性之后，通过对象序列化机制“保存”BangBean的所有信息。在程序运行，需要创建这个Bean的时候，这些“保存”过的属性能够被恢复，这样你就得到了与程序设计时一致的Bean。

你可以发现所有的域都是私有的，你通常都应该这么做，即使用“属性”模式，也只允许通过方法去访问域。

当你观察addActionListener()方法的签名时，就会发现它可以抛出TooManyListenersException异常。这表明它只支持单路的事件处理方式，即事件发生的时候它只能通知一个监听器。通常你会使用支持多路方式的组件，这样一个事件可以通知给多个监听器。不过，这样会牵涉到线程问题，我们将在本章的后面标题为“JavaBean与同步”的小节研究这个问题。然而，使用单路方式就可以回避线程的问题。

当你单击鼠标时，文字将显示在BangBean的中央，此时如果actionListener域非空，它的actionPerformed()方法将被调用，调用之前要创建一个新的ActionEvent对象。当鼠标移动的时候，新的坐标将被捕获，并且重新绘制窗体(如你所见，擦除窗体上的任何文字)。

下面是测试 Bean 的 BangBeanTest 类：

```
//: c14:BangBeanTest.java
import bangbean.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BangBeanTest extends JFrame {
    private JTextField txt = new JTextField(20);
    // During testing, report actions:
    class BBL implements ActionListener {
        private int count = 0;
        public void actionPerformed(ActionEvent e) {
            txt.setText("BangBean action " + count++);
        }
    }
    public BangBeanTest() {
        BangBean bb = new BangBean();
        try {
            bb.addActionListener(new BBL());
        } catch (TooManyListenersException e) {
            txt.setText("Too many listeners");
        }
        Container cp = getContentPane();
        cp.add(bb);
        cp.add(BorderLayout.SOUTH, txt);
    }
    public static void main(String[] args) {
```

```
        Console.run(new BangBeanTest(), 400, 500);
    }
} ///:~
```

当Bean处于开发环境之中时，不必使用这个类，不过为你的每个Bean提供一种快速的测试方法会很有帮助。BangBeanTest将把一个BangBean对象放置在applet上，给BangBean对象关联一个简单的动作监听器，当ActionEvent事件发生的时候，监听器能把事件数目显示到JTextField。当然，通常应用程序构建工具会帮助你创建使用Bean的大部分代码。

当你通过BeanDumper查询BangBean，或者把BangBean放置在支持Bean的开发环境中时，显示出的属性和动作将比上面代码中编写的要多许多。这是因为BangBean继承自JPanel，而JPanel也是一个Bean，所以你会看到所有的属性和事件。

JavaBean 与同步

当你创建 Bean 的时候，你必须要假设它可能会在多线程环境下运行。也就是说：

1. 尽可能地让Bean中的所有公共方法都是synchronized的。当然，这将导致synchronized的运行时开销（在近几个版本的JDK中，这个开销已经大大降低了）。如果这么做会有问题，那么对那些不会导致临界区域问题的方法，可以考虑不同步。但要记住，判断这一点并不容易。进行同步的方法应该尽可能短（比如下面例子中的getCircleSize()方法）和/或“原子的”，原子性是指，在调用含有这一小段代码的方法时，对象不能被改变。不同步这样的方法也许不会对程序的执行速度有明显的效果。所以你还不如同步Bean的所有公共方法，只有在确实必要并且会有明显效果的时候，才移除synchronized关键字。
2. 当一个多路事件触发了一组对该事件感兴趣的监听器时，你必须假定，在你遍历列表进行通知的同时，监听器可能会被添加或移除。

第一点很容易处理，但第二点就需要更多的思考。前面版本的 BangBean.java 通过忽略synchronized关键字并使用单路事件，回避了多线程问题。下面是修改后的版本，它可以在多线程环境下工作，而且使用了多路事件：

```
///: c14:BangBean2.java
// You should write your Beans this way so they
// can run in a multithreaded environment.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import com.bruceeckel.swing.*;
```

```

public class BangBean2 extends JPanel
implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Circle size
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.RED;
    private ArrayList actionListeners = new ArrayList();
    public BangBean2() {
        addMouseListener(new ML());
        addMouseMotionListener(new MM());
    }
    public synchronized int getCircleSize() { return cSize; }
    public synchronized void setCircleSize(int newSize) {
        cSize = newSize;
    }
    public synchronized String getBangText() { return text; }
    public synchronized void setBangText(String newText) {
        text = newText;
    }
    public synchronized int getFontSize() { return fontSize; }
    public synchronized void setFontSize(int newSize) {
        fontSize = newSize;
    }
    public synchronized Color getTextColor() { return tColor; }
    public synchronized void setTextColor(Color newColor) {
        tColor = newColor;
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLACK);
        g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
    }
    // This is a multicast listener, which is more typically
    // used than the unicast approach taken in BangBean.java:
    public synchronized void
    addActionListener(ActionListener l) {
        actionListeners.add(l);
    }
    public synchronized void
    removeActionListener(ActionListener l) {
        actionListeners.remove(l);
    }
    // Notice this isn't synchronized:

```

```

public void notifyListeners() {
    ActionEvent a = new ActionEvent(BangBean2.this,
        ActionEvent.ACTION_PERFORMED, null);
    ArrayList lv = null;
    // Make a shallow copy of the List in case
    // someone adds a listener while we're
    // calling listeners:
    synchronized(this) {
        lv = (ArrayList)actionListeners.clone();
    }
    // Call all the listener methods:
    for(int i = 0; i < lv.size(); i++)
        ((ActionListener)lv.get(i)).actionPerformed(a);
}

class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font("TimesRoman", Font.BOLD, fontSize));
        int width = g.getFontMetrics().stringWidth(text);
        g.drawString(text, (getSize().width - width) / 2,
            getSize().height/2);
        g.dispose();
        notifyListeners();
    }
}

class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}

public static void main(String[] args) {
    BangBean2 bb = new BangBean2();
    bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("ActionEvent" + e);
        }
    });
    bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("BangBean2 action");
        }
    });
}

```



```

    }
  });
  bb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.out.println("More action");
    }
  });
  Console.run(bb, 300, 300);
}
} ///:~

```

给方法加上synchronized关键字很容易。不过要注意，addActionListener()和removeActionListener()方法现在要在ArrayList中添加或移除ActionListener，所以你可以添加任意多的监听器。

你可以观察到notifyListeners()方法没有被同步。这个方法可以同时被多个线程调用。在调用notifyListeners()期间，也可能有别的线程在对addActionListener()或removeActionListener()进行调用，这将遍历actionListeners这个ArrayList，所以就可能发生冲突。为了解决这个问题，先在一个同步子句里对ArrayList进行克隆，然后对克隆的对象进行遍历（克隆的详细信息请参考附录A）。这样，就可以操作原来那个ArrayList而不会对notifyListeners()过程有影响。

paintComponent()方法也没有被同步。判断是否同步一个被重载方法并不像同步一个自己写的方法那么明显。在本例中表明，无论是否同步，paintComponent()方法看起来工作都正常。不过你必须考虑这些问题：

1. 这个方法会修改对象中“关键”变量的状态吗？要弄清楚变量是否“关键”，你必须判断它们是否被程序中的其它线程读写。（在本例中，读写操作基本上是通过同步方法进行的，所以你检查这些就可以了。）在paintComponent()方法中，就没有进行任何修改操作。
2. 这个方法依赖于那些“关键”变量吗？如果有某个同步方法会修改此方法所使用的变量，那么你应该把这个方法也同步。根据这一点，你会发现cSize被同步方法所改变，所以paintComponent()方法应该被同步。不过，这里你还要问自己，“如果cSize在调用paintComponent()的过程中被改变，最坏的结果是什么？”要是觉得问题不大，只是有个瞬时效果，你就可以作出不同步paintComponent()方法的决定，以避免进行同步方法调用时所产生的开销。
3. 第三个线索是查看基类版本的paintComponent()是否同步，在这里并没有被同步。这不是个严密的判断方法，只是一个线索。比如在本例中，通过同步方法被改变的域（比如cSize）已经混在paintComponent()的公式中了，在这种情况下它有可能被改变。不过，请注意，同步不会继承；也就是说，如果基类方法是同步的，继承类中的重载版本不一定要同步。
4. paint()和 paintComponent()的执行必须尽可能快。要尽量把处理的开销移到方法外面，所以你要是发现需要同步这些方法，那么你的设计可能就存在问题。

与BangBeanTest相比，main()里面的测试代码已经被修改过了，它通过添加额外的监听器，来演示BangBean2 的多路事件处理能力。

把 Bean 打包

在你把 JavaBean 加入到某个支持 Bean 的可视化构建工具之前，必须把它打包进一个标准的 Bean 容器，也就是一个 JAR 文件，它里面包含了 Bean 的所有.class 文件以及能表明“这是一个 Bean”的“清单（Manifest）”文件。清单文件是一个文本文件，它遵循特定的格式。对于 BangBean，它的清单文件看起来像这样：

```
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
```

第一行显示了清单文件格式的版本，目前Sun公司发布的是 1.0。第二行（忽略空行）为 BangBean.class文件命名，第三行表明“这是一个Bean”。没有第三行，程序构建工具将不能把类识别成Bean。

唯一需要技巧的部分是，你要确定在“Name:”字段写上正确的路径。如果你看看前面的 BangBean.java，就会发现它处于 bangbean 包中（也就是一个名为“bangbean”的子目录，它不包括在 classpath 中），清单文件的名称字段必须含有这个包信息。此外，你必须把清单文件放在包的根目录的上层目录中，这里就是把清单文件放在“bangbean”目录的父目录中。然后你需要在清单文件所在的目录下调用 jar 工具，如下：

```
jar cfm BangBean.jar BangBean.mf bangbean
```

这里假定你要把生成的JAR文件被命名为BangBean.jar，并且清单文件的名称为 BangBean.mf。

你可能会奇怪：“当我编译完 BangBean.java 之后，生成的其它.class 文件在哪呢？”其实，它们都在 bangbean 子目录下，上面 jar 命令的最后一个参数就是 bangbean 目录名。当你把目录名传递给 jar 工具时，它将把整个目录打包进 JAR 文件（在本例中，包括了 BangBean.java 源文件，你也许不会选择在自己的 Bean 中包括源代码）。此外，如果你把刚才生成的 JAR 文件解包，就会发现里面并没有你指定的清单文件，jar 工具创建了自己的清单文件（部分根据你提供的信息）MANIFEST.MF，这个文件放在“META-INF”（元信息）目录下。打开这个文件，就会看到 jar 为每个文件都添加了数字签名信息，如下：

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/00=
MD5-Digest: 04NcS1hE3Smnzlp2hj6qeg==
```

通常，你不必考虑这些，如果改变了程序，你只要修改原来的清单文件，然后重新调用jar工具来为Bean创建一个新的JAR文件即可。通过把相关信息加入清单文件，你还能把其它Bean也添加到这个JAR文件中。

要注意的一点是，你可能希望把每个Bean都放进专门的子目录中，因为在创建JAR文件的时候，你把子目录的名称传递给了jar工具，它将把子目录里面的所有文件都打包进JAR文件。你可以发现Frog和BangBean都在它们各自的子目录中。

一旦你把Bean正确地打包成JAR文件，就可以把它导入支持Bean的程序构建环境中了。导入的方式可能根据不同的工具而有所不同，不过Sun公司在它们的“Bean Builder”里提供了一个免费使用的测试工具（可以从java.sun.com/beans下载）。你只要把JAR文件复制到正确的目录下，就可以把你的Bean导入到Bean Builder中。

对 Bean 更高级的支持

你已经看到了制作一个Bean是多么的简单，不过你并没有被局限于目前所看到的功能。JavaBean架构提供的门槛很低，但也可以扩展到更复杂的情况。这些情形超出了本书的范围，不过这里可以做一些简要介绍。你可以在java.sun.com/beans找到更多的细节。

你可以针对属性提供高级功能。前面的例子只演示了单一属性，但也可以使用数组来表示多重属性。这称为索引属性。你只要提供恰当的方法（也就是根据命名规则给方法命名），Introspector将识别出索引属性，这样你的应用程序构建工具就可以正确工作。

属性可以被绑定，即它们能通过PropertyChangeEvent事件通知其它对象。这些被通知的对象可以根据Bean上的变化来决定如何改变自己。

属性可以被约束，即如果属性的改变是不可接受的，其它对象可以否决这个改变。这些对象也是通过PropertyChangeEvent事件得到通知的，而且能抛出PropertyVetoException异常来阻止属性的改变，然后恢复属性的旧值。

你还可以改变Bean在设计阶段的表示方式：

1. 你可以为自己的Bean提供一个自定义的属性表。对于其它Bean将使用通常的属性表，当你的Bean被选中的时候，将自动激活你提供的表。
2. 你还可以为特定的属性提供自定义的编辑器，对于其它属性，将使用普通编辑器，但是当你的特殊属性被编辑的时候，将自动激活你提供的编辑器。
3. 你可以为你的Bean提供一个自定义的BeanInfo类，它可以产生与缺省情况下由Introspector所提供的信息不同的信息。
4. 还可以把每一个FeatureDescriptor里的“专家”模式打开或者关闭，这样就能够区分简单功能和复杂功能。

Bean 的更多信息

有很多关于JavaBean的书籍;比如,《JavaBeans 》,Rusty Harold著 (IDG出版, 1998).

总结

从Java 1.0 到Java 2, 在所有类库中GUI库的变化最大。Java 1.0 的AWT被批评为最差劲的设计之一, 它允许你编写可移植的程序, 不过写出来的图形界面也是“在所有平台上都表现一般”。与其它为特定平台量身定做的应用开发工具相比, 它限制很多, 使用笨拙, 让人难以接受。

随着Java 1.1 引入了新的事件模型和JavaBean规范, 新的时代开始了。在可视化应用程序构建工具中可以很容易地通过拖放组件来生成程序。此外, 事件模型和JavaBean的设计清楚地表现出, 设计者在易于编程和维护代码方面(Java 1.0 的AWT里看不出这些)作了充分考虑。但是直到JFC/Swing出现, 这个工作才算完成。随着Swing组件的引入, 跨平台的GUI编程才成为大众化的技术。

当然, 它唯一没有提供的就是应用程序构建工具, 这也是真正出现革命性进步的地方。微软的Visual BASIC和Visual C++需要微软出品的应用程序构建工具, Borland的Delphi和C++ Builder也是如此。要是你希望得到更好的应用程序构建工具, 你就只能寄希望于提供商来满足你的要求。但是, Java作为一个开放的环境, 它不仅允许应用程序构建工具之间的竞争, 而且鼓励这种竞争。对于那些正规的工具, 它们必须支持JavaBean。这就意味着一个公平的竞争环境;如果有更好的工具, 你不必拘泥于现有工具。你可以采用这个新工具以提高生产率。对于应用程序构建工具而言, 这种新式的竞争环境以前从未出现过, 结果也必将对程序员的生产率产生积极的影响。

本章的目的仅仅是向你介绍Swing的功能, 让你能够入门, 这样就可以在使用库的过程中体会到Swing的方便。目前所介绍的内容, 对于设计一个比较好的用户界面可能已经足够了。不过, Swing还有很多内容;它的设计目标是要成为一个功能完整的UI设计工具包。只要你能想到, 都有可能用它来实现。

要是在本章中没有发现需要的内容, 你可以去钻研Sun公司的JDK文档或者在Web上搜索, 如果还不够的话, 就去找一本专门研究Swing的书籍。《The JFC Swing Tutorial》Walrath & Campione著, (Addison Wesley, 1999). 是个好的起点。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到, 你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 使用本章提供的Console类编写一个applet/应用程序, 包括一个文本域和三个

按钮。按下每个按钮的时候，在文本域中显示不同的文字。

2. 向练习 1 编写的applet添加一个检查框，捕获其事件，并在事件处理程序中向文本域插入不同的文字。
3. 使用Console编写一个applet/应用程序。在java.sun.com的JDK文档中，查找JPasswordField，并把它添加到程序中。如果用户输入了正确的密码，使用JOptionPane向用户显示成功的消息。
4. 使用Console编写一个applet/应用程序，添加所有具有addActionListener()方法的Swing组件。（在java.sun.com的JDK文档中查找这些组件。提示：使用索引功能。）针对每个组件，捕获其事件并且在一个文本域中显示相应的信息。
5. 使用Console编写一个applet/应用程序，添加一个JButton和一个JTextField。编写恰当的监听器：如果按钮获得了焦点，键入的字符将出现在JTextField里面。
6. 使用Console编写一个applet/应用程序，在主窗体里添加本章介绍的所有组件，包括菜单和对话框。
7. 修改TextFields.java，使得t2 里面的字符保持原来输入时候的大小写，而不要自动转换成大写。
8. 在互联网上查找并下载几个免费的GUI构建工具，或者购买一个商业产品。研究一下要把BangBean导入工具的环境，需要哪些必要步骤。
9. 把Frog.class加入本章中的清单文件，执行jar工具以创建包含了Frog和BangBean的JAR文件。然后从Sun下载并安装Bean Builder，或者使用已有的支持Bean的程序构建工具，把JAR文件导入开发环境，以测试这两个Bean。
10. 自己编写一个JavaBean，取名为Valve。它有两个属性：一个布尔型的“on”，一个整形的“level”。写一个清单文件，使用jar为Bean打包，在Bean Builder或者支持Bean的程序构建工具里面导入这个Bean，然后进行测试。
11. 修改MessageBoxes.java，使它的每个按钮拥有单独的ActionListener（而不是通过按钮文字的匹配来共享）。
12. 通过加入处理新事件的代码，在TrackEvent.java中监听新的事件。你要自己决定监听的事件类型。
13. 从JButton继承，编写一个新的按钮。每当按钮按下时候，将为按钮随机选择一种颜色。随机生成颜色的方法，请参考ColorBoxes.java。
14. 修改TextPane.java，使用JTextArea而不是JTextPane。
15. 修改Menus.java，在菜单上使用单选按钮而不是单选框。
16. 通过传递数组给JList构造器，以及移除动态地向列表框添加元素的代码，来简化List.java。
17. 修改SineWave.java，加入相应的“getter”和“setter”，使SineDraw成为一个JavaBean。
18. 还记得“绘图板”这个玩具吗？它有两个调节器，一个用来控制绘图点垂直方向的运动，一个用来控制水平方向。以SineWave.java程序为基础，编写一个具有类似功能的程序。这里调节器可以使用滑块来实现。添加一个可以擦除整个图形的按钮。
19. 在SineWave.java的基础上编写程序（一个使用Console的applet/应用程序），在观察窗口画一条动态正弦波，它可以像示波器那样向后卷动，使用一个线程来控制动画。动画的速度由java.swing.JSlider组件进行控制。

20. 修改练习 19，在程序里创建多个显示正弦波的面板。面板的数目可以通过HTML标记或命令行参数进行控制。
21. 修改练习 19，使用`java.swing.Timer`类来控制动画。注意它与`java.util.Timer`类的区别。
22. 编写一个“渐进的进度表示器”，当接近结束的时候，它的进度越来越慢。加入一些随机的行为，使它能够不时地表现出加速的效果。
23. 修改`Progress.java`，不要共享模型（`model`），而是使用一个监听器来关联滑块和进度条。
24. 根据“把applet打包进JAR文件”这一节的步骤，把`TicTacToe.java`打包进一个JAR文件。编写一个HTML页面，尽量使用简单的applet标签，包括用`archive`指定JAR文件。运行`HTMLConverter`处理这个文件，以得到能工作的HTML文件。
25. 使用`Console`编写一个applet/应用程序。它有三个滑块，每一个分别表示`java.awt.Color`类型的红绿兰颜色值，窗体的其它部分是一个`JPanel`，用来显示由滑块决定的颜色。加入一个不可编辑的文本域，显示当前的RGB值。
26. 在`javax.swing`的JDK文档中，查找`JColorChooser`。写一个程序，加入一个按钮，它可以弹出选择颜色的对话框。
27. 几乎所有的Swing组件都继承自`Component`，它有一个`setCursor()`方法。在JDK中查找这些信息。写一个applet，把当前光标改变成`Cursor`类中内置的光标之一。
28. 在`ShowAddListeners.java`的基础上编写程序，完全实现第 10 章`ShowMethods.java`程序的功能。
29. 把第 12 章的`TestRegularExpression.java`程序改写成一个交互式Swing程序，它允许你把输入字符串放进文本区域，把正则表达式放进文本域。结果应该显示在另一个文本区域里。
30. 修改`InvokeLaterFrame.java`以使用`invokeAndWait()`方法。

第十五章 发现问题

在 C 被驯化到 ANSI C 之前，我们有一个小玩笑是这样说的：“我的代码编译过了，因此它应该可以运行了！”（哈哈！）

一旦你理解了 C 语言就会发现这很是好笑，因为在那时，C 编译器几乎可以接受任何东西；C 语言是作为一种真正“可移植的汇编语言”而被创建的，它被用来发现是否有这种可能性：可以开发出一种可移植的操作系统（Unix），以便使其从一种机器体系迁移到另一种机器体系上时，不需要用新机器的汇编语言从头重新编写。因此，C 语言实际上是作为构建 Unix 的副带效应之一，而不是作为通用目的的程序设计语言而被创建的。

因为 C 语言针对的是那些用汇编语言编写操作系统的编程人员，因此，它隐含地假设了这些编程人员知道他们正在做什么并且不需要安全保证。例如，汇编语言的编程人员不需要编译器去检查参数的类型和用法，而且如果他们决定按照某种不同的方式而不是其固有的方式来使用某种数据类型，那么他们一定有很好的理由去这样做，而且编译器也不会对这种做法有任何妨碍。因此，在开发无 bug 程序的漫长过程中，编译 pre-ANSI C 程序只是其中的第一步。

在许多人除了将 C 语言用于编写操作系统而且也用于项目工程之后，ANSI C 的发展也就伴随着关于“编译器应该接受什么”这样更严格的规则的产生而到来了，并且在 C++ 语言出现之后，也大大提高了程序一旦经过编译之后就可以正确运行的机率。这种改善来自于“强静态类型检查”：“强”是因为编译器防止我们滥用数据类型，“静态”是因为 ANSI C 和 C++ 是在编译时执行类型检查的。

对于很多人（包括我自己）来说，这种改善如此的引人注目以至于强静态类型检查似乎是我们大部分问题的解决答案。确实，设计 Java 的动机之一就是 C++ 的类型检查不够强大（主要是因为 C++ 不得不向后兼容 C，因此也就被约束限制住了）。因此，Java 可以更进一步充分利用类型检查的好处，这样一来 Java 在运行时也具有语言检查机制（而 C++ 并没有，在运行时刻，只能依赖于基本的汇编语言了——它很快，但是没有任何自我意识），它没有被限制为只能进行静态类型检查¹。

然而，似乎只有语言检查机制能够我们将我们带到追求开发可以正确运行的程序的境界。C++ 程序运行起来比 C 程序快了很多，但是它仍然会常常出现问题，例如内存漏洞和其它一些难于发现的隐藏错误。Java 在向解决这些问题的目标而迈进时走了很长的一段路，而且至今编写出含有令人生厌的 Bug 的 Java 程序的机率仍然很高。另外（尽管令人惊异的性能要

¹ 不过，它主要面向的是静态检测。然而，我们有另一个可供选择的系统，称作潜式类型检查或动态类型检查或弱类型检查，其中的对象类型仍是强制检查的，但是它是在运行时，而不是编译时强迫检查的。用这样的语言——Python 是一种极好的例子（<http://www.python.org>）——编写代码给程序员带来了更多的灵活性并且只需要更少的措辞就可以满足编译器的要求，而且它仍能确保对象被正确使用。然而，对于深信强的静态类型检查是唯一明智的解决方案的程序员来说，潜式类型检查是被诅咒的事，而且导致在对这两种方式进行比较时人们产生了激烈的争论。作为一个总是追求更大的生产率的人，我发现潜式类型检查的价值非常引人注目。另外，我相信，思考潜式类型检查问题的能力有助于你去解决在强静态类型检查语言中遇到的难题。

求经常被 Sun 高层人士吹捧)，Java 中所有的安全措施都增加了额外的开销，因此我们有时会遇到这样的挑战：让我们的 Java 程序运行得足够快以便达到某个特定速度（尽管通常情况下，拥有可运行的程序比以特定速度运行的程序更重要）。

本章将讲述一些工具，用来解决那些编译器无法解决的问题。在某种意义上，我们承认只有编译器能够带领我们创建健壮的程序，因此，我们正在超越编译器去创建一个能够更多地了解程序应该做什么以及不应该做什么的构建系统及其代码。

最大的进步是合并了“自动单元测试”。这意味着要编写测试并将这些测试合并到一个构建系统中，该系统在每一次运行时都会对你的代码进行编译并且运行这些测试，就好像这些测试是编译过程的一部分一样（你很快就会开始依赖于它们所起的作用了）。为了这本书，我们开发了一个订制的测试系统，以确保程序输出的正确性（而且直接在代码列表中显示输出结果），不单是，实际上在恰当的时候我们也会用到标准的 Junit 测试系统。为了确保测试是自动产生的，要使用 Ant（一种开放源码的工具，已经成为 Java 开发中的事实标准）以及 CVS（另一种开放源码的工具，用来维护一个包含了某一特定工程的所有源码的存储池（Repository））将测试作为构建过程的一部分来运行。

JDK1.4 引入了断言（assertion）机制，可以在运行时辅助代码验证。有关断言的一个更为引人注目的用法是 DBC（Design by Contract，按契约设计）：一种描述类的正确性的形式化方法。通过与自动测试搭配使用，DBC 会成为一个强有力的工具。

仅仅使用单元测试还是不够的，有时我们需要追踪正在运行（但不能正确运行）的程序的问题。在 JDK1.4，引入了 logging API。可以让我们很容易地记录下程序的相关信息。这是对通过添加和删除 `println()` 语句来追踪问题的一次重大改进，这部分的介绍将十分详细，以期使你对此 API 有一个全面的认识。本章还会介绍调试、展示典型的调试器所提供的能够帮助我们发现细小问题的信息。最后，你将会学习有关剖析（profiling）的知识以及如何去发现引起程序运行速度缓慢的瓶颈。

单元测试

人们在编程实践中的新近认识是单元测试的巨大价值。也就是将测试集成到我们创建的所有代码中并且在我们每次执行构建时运行这些测试的过程。这样，在构建过程中，就不是只进行语法错误检查了，因为我们同时还告诉了它怎样进行语义检查。C 风格的编程语言，尤其是 C++，在编程安全上具有典型的宝贵性能。用 Java 开发程序比用 C++ 要快得多（根据大多数报道，大约快一倍）的原因是因为 Java 的安全措施：像垃圾回收器以及改善的类型检测机制这样的特性。通过将单元测试集成到我们的构建过程中来，我们就可以延伸这种安全措施，从而加速开发过程。并且当我们发现设计或实现中有纰漏时，也可以更大胆地进行修改，以及更容易地对我们的代码进行重新分解，而且一般情况下会更快地产生更好的产品。

单元测试在开发中所起的作用是相当重要的，所以在这本书中从头到尾都用到了它，它不但被用来验证本书中的代码，而且还被用来显示所期望的输出。我自己开始尝试使用单元测试是因为我意识到：为了保证书中代码的正确性，必须自动地抽取书中的所有程序，将它们构成一个源代码树，并提供一个恰当的构建系统。本书中使用的构建系统是 Ant（本章稍后将

会讲述)，安装完毕后，我们只需键入 `ant` 就可以构建本书中的所有代码。自动抽取和编译过程对本书中代码质量所起的作用是如此直接和生动，以至于它（在我的思维中）迅速成为所有程序设计书籍的必需品——我们怎么会去信任没有编译过的代码呢？我还发现如果我想进行大规模的修改，可以通过对整本书进行查询替换来实现，或者只是对某段代码周围的代码进行深度的修改。因为我知道，如果我的修改会引发某个错误，那么代码抽取器和构建系统就会将其冲掉。

然而，随着程序的日益复杂，我也发现在我的系统当中存在着一个严重的漏洞。能够成功编译程序无疑是重要的第一步，并且对于一本出版的书籍，它似乎是一个具有相当的创新性的特点；通常由于出版的压力，使得随机打开一本编程用书，就会发现一个编码纰漏的情况变得很常见。但是，我可以不断地获取读者报告的有关代码中的语义问题的消息。这些问题只有在运行代码时才能发现。当然，我了解这一点并且尽早地采取了一些艰难的步骤向实现一个可以自动执行测试的系统进发，但是我还是屈从于出版社的时间进度，我始终都知道自己的代码中肯定存在着某些错误，它们会以令人窘迫的Bug报告的形式送回给我，并噬咬着我（在开放源码的世界中²，窘迫是提高代码质量的主要动力因素之一！）

另外一个问题是我缺乏一种测试系统的构成组织。最后，我开始采用单元测试和 Junit，它们提供了测试构成组织的基础。我发现 Junit 的初始版本让人难于忍受，因为它们要求程序员即使是编写最简单的测试包也要编写过多的代码。更近的一些版本利用反射技术已经大大减少了所需的代码，因此，它们现在令人满意得多了。

然而，我还需要解决另外一个问题，也就是验证程序输出的有效性，并且展示本书中被验证过的输出。我经常收到关于本书中没有展示足够的程序输出的抱怨。我的观点是：读者应该在阅读本书的时候运行这些程序，许多读者这样做了，而且也受益匪浅。不过我持有该观点的另一个潜藏原因是我没办法测试本书所展示的输出是正确的。凭经验，我知道经过一段时间也许就会发生什么事导致这些输出不再正确（或者，首先是我无法保证它是正确的）。这里展示了一个简单测试的框架，不但能够捕获程序的控制台输出——本书中大部分程序都产生控制台输出，而且也可以与书中作为开放源码清单的一部分打印出来的、所期望的输出进行比较，因此读者可以看到输出是什么，而且还可以知道此输出已经由构建过程校验过，并且它们可以进行自我校验。

我想知道这个测试系统是否使用起来更容易、更简单，于是我将极限编程原则（Extreme Programming principle）“做可以运转的最简单的事物”作为出发点，然后按使用要求演化该系统。（另外，我想尽力精简测试代码的数量，以尝试在更少的代码中为屏幕显示设置更多的功能。）我这么做的成果³就是在下面将要描述的一个简单的测试框架。

一个简单的测试框架

这个框架的主要目的⁴是为了校验本书中的例子的输出结果。我们已经看到如下所示的几行：

```
private static Test monitor = new Test();
```

² 尽管本书的电子版可以免费得到，但是它不是开放源码的。

³ 无论如何，首先要尝试。我发现第一次构建某事物的过程最终会产生一些见识和新的观念。

⁴ 受Python的doctest模型的激发。

大多数类的开始部分都包含一个 `main()` 方法。`monitor` 对象的任务是拦截以及保存一个标准输出及标准错误的备份到一个文本文件中。接着利用此文件，通过将文件的内容和所期望的输出进行比较来验证一个示例程序的输出。

我们首先定义在本测试系统中将要被抛出的异常。该类库的通用异常是其他异常的基类。注意它扩展了 **`RuntimeException`**，因此不会涉及已经检测过的异常：

```
//: com:bruceeckel:simpletest:SimpleTestException.java
package com.bruceeckel.simpletest;

public class SimpleTestException extends RuntimeException {
    public SimpleTestException(String msg) {
        super(msg);
    }
} ///:~
```

一个基本测试被用来验证由程序送往控制台的行数与所期望的行数相同：

```
//: com:bruceeckel:simpletest:NumOfLinesException.java
package com.bruceeckel.simpletest;

public class NumOfLinesException
extends SimpleTestException {
    public NumOfLinesException(int exp, int out) {
        super("Number of lines of output and "
            + "expected output did not match.\n" +
            "expected: <" + exp + ">\n" +
            "output:   <" + out + "> lines)");
    }
} ///:~
```

或者，行数可能是正确的，但是有一行或多行并不匹配：

```
//: com:bruceeckel:simpletest:LineMismatchException.java
package com.bruceeckel.simpletest;
import java.io.PrintStream;

public class LineMismatchException
extends SimpleTestException {
    public LineMismatchException(
        int lineNum, String expected, String output) {
        super("line " + lineNum +
            " of output did not match expected output\n" +
            "expected: <" + expected + ">\n" +
            "output:   <" + output + ">");
    }
}
```

```

    }

    } ///:~

```

这个测试系统通过使用 **TestStream** 类替换标准控制台输出及控制台错误来拦截控制台输出，从而得以运行的。

```

///: com:bruceeckel:simpletest:TestStream.java
// Simple utility for testing program output. Intercepts
// System.out to print both to the console and a buffer.
package com.bruceeckel.simpletest;
import java.io.*;
import java.util.*;
import java.util.regex.*;

public class TestStream extends PrintStream {
    protected int numOfLines;
    private PrintStream
        console = System.out,
        err = System.err,
        fout;
    // To store lines sent to System.out or err
    private InputStream stdin;
    private String className;
    public TestStream(String className) {
        super(System.out, true); // Autoflush
        System.setOut(this);
        System.setErr(this);
        stdin = System.in; // Save to restore in dispose()
        // Replace the default version with one that
        // automatically produces input on demand:
        System.setIn(new BufferedInputStream(new InputStream(){
            char[] input = ("test\n").toCharArray();
            int index = 0;
            public int read() {
                return
                    (int)input[index = (index + 1) % input.length];
            }
        }));
        this.className = className;
        openOutputFile();
    }
    // public PrintStream getConsole() { return console; }
    public void dispose() {
        System.setOut(console);
    }

```

```

    System.setErr(err);
    System.setIn(stdin);
}
// This will write over an old Output.txt file:
public void openOutputFile() {
    try {
        fout = new PrintStream(new FileOutputStream(
            new File(className + "Output.txt")));
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }
}
// Override all possible print/println methods to send
// intercepted console output to both the console and
// the Output.txt file:
public void print(boolean x) {
    console.print(x);
    fout.print(x);
}
public void println(boolean x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
}
public void print(char x) {
    console.print(x);
    fout.print(x);
}
public void println(char x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
}
public void print(int x) {
    console.print(x);
    fout.print(x);
}
public void println(int x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
}
public void print(long x) {
    console.print(x);

```

```

        fout.print(x);
    }
    public void println(long x) {
        numOfLines++;
        console.println(x);
        fout.println(x);
    }
    public void print(float x) {
        console.print(x);
        fout.print(x);
    }
    public void println(float x) {
        numOfLines++;
        console.println(x);
        fout.println(x);
    }
    public void print(double x) {
        console.print(x);
        fout.print(x);
    }
    public void println(double x) {
        numOfLines++;
        console.println(x);
        fout.println(x);
    }
    public void print(char[] x) {
        console.print(x);
        fout.print(x);
    }
    public void println(char[] x) {
        numOfLines++;
        console.println(x);
        fout.println(x);
    }
    public void print(String x) {
        console.print(x);
        fout.print(x);
    }
    public void println(String x) {
        numOfLines++;
        console.println(x);
        fout.println(x);
    }
    public void print(Object x) {

```

```

        console.print(x);
        fout.print(x);
    }
    public void println(Object x) {
        numOfLines++;
        console.println(x);
        fout.println(x);
    }
    public void println() {
        if(false) console.print("println");
        numOfLines++;
        console.println();
        fout.println();
    }
    public void
    write(byte[] buffer, int offset, int length) {
        console.write(buffer, offset, length);
        fout.write(buffer, offset, length);
    }
    public void write(int b) {
        console.write(b);
        fout.write(b);
    }
} ///:~

```

TestStream 的构造器在调用完基类的构造器之后，首先存储指向标准输出和标准错误的引用，然后将两个流都重定向到 TestStream 对象。静态方法 setOut() 和 setErr() 都接受一个 PrintStream 参数，将 System.out 和 System.err 引用从它们的常规对象中抽出，然后插入到 TestStream 对象中去，因此 TestStream 必须也是 PrintStream（或者与此等同地是继承自 PrintStream 的事物）。初始的标准输出 PrintStream 引用被 TestStream 内部的控制台引用所捕获，而且每一次控制台输出被拦截时，都会被发送到初始控制台和一个输出文件。dispose() 方法用来在 TestStream 完成对标准 I/O 引用的使用之后，将其设置回到它们初始的对象。

对于需要用户从控制台输入的示例自动测试，构造器将这些调用重定向到标准输入。当前的标准输入被存储到一个引用中，以便 dispose() 可以将其恢复到它的初始状态。通过使用 System.setIn()，一个匿名内部类被设置对所有由被测试程序的输入所产生的请求进行处理。此内部类的 read() 方法产生字符串“test”，后面跟随着一个换行符。

TestStream 重载了 PrintStream 的为每一种类型所设置的各种各样的 print() 和 println() 方法。这些方法中的每一个都既写向标准输出又写向输出文件。接着使用 expect() 方法测试由程序产生的输出是否匹配提供给 expect() 作为参数的期望输出。

在 Test 类中使用到这些工具：

```

//: com:bruceeckel:simpletest:Test.java
// Simple utility for testing program output. Intercepts
// System.out to print both to the console and a buffer.
package com.bruceeckel.simpletest;
import java.io.*;
import java.util.*;
import java.util.regex.*;

public class Test {
    // Bit-shifted so they can be added together:
    public static final int
        EXACT = 1 << 0, // Lines must match exactly
        AT_LEAST = 1 << 1, // Must be at least these lines
        IGNORE_ORDER = 1 << 2, // Ignore line order
        WAIT = 1 << 3; // Delay until all lines are output
    private String className;
    private TestStream testStream;
    public Test() {
        // Discover the name of the class this
        // object was created within:
        className =
            new Throwable().getStackTrace()[1].getClassName();
        testStream = new TestStream(className);
    }
    public static List fileToList(String fname) {
        ArrayList list = new ArrayList();
        try {
            BufferedReader in =
                new BufferedReader(new FileReader(fname));
            try {
                String line;
                while((line = in.readLine()) != null) {
                    if(fname.endsWith(".txt"))
                        list.add(line);
                    else
                        list.add(new TestExpression(line));
                }
            } finally {
                in.close();
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return list;
    }
}

```

```

    }
    public static List arrayToList(Object[] array) {
        List l = new ArrayList();
        for(int i = 0; i < array.length; i++) {
            if(array[i] instanceof TestExpression) {
                TestExpression re = (TestExpression)array[i];
                for(int j = 0; j < re.getNumber(); j++)
                    l.add(re);
            } else {
                l.add(new TestExpression(array[i].toString()));
            }
        }
        return l;
    }

    public void expect(Object[] exp, int flags) {
        if((flags & WAIT) != 0)
            while(testStream.numOfLines < exp.length) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        List output = fileToList(className + "Output.txt");
        if((flags & IGNORE_ORDER) == IGNORE_ORDER)
            OutputVerifier.verifyIgnoreOrder(output, exp);
        else if((flags & AT_LEAST) == AT_LEAST)
            OutputVerifier.verifyAtLeast(output,
                arrayToList(exp));
        else
            OutputVerifier.verify(output, arrayToList(exp));
        // Clean up the output file - see c06:Detergent.java
        testStream.openOutputFile();
    }

    public void expect(Object[] expected) {
        expect(expected, EXACT);
    }

    public void expect(Object[] expectFirst,
        String fname, int flags) {
        List expected = fileToList(fname);
        for(int i = 0; i < expectFirst.length; i++)
            expected.add(i, expectFirst[i]);
        expect(expected.toArray(), flags);
    }

```



```

    public void expect(Object[] expectFirst, String fname) {
        expect(expectFirst, fname, EXACT);
    }
    public void expect(String fname) {
        expect(new Object[] {}, fname, EXACT);
    }
} ///:~

```

为方便起见，上面的程序中提供了多个 `expect()` 的重载版本（这样，客户端程序员就可以提供包含期望输出的文件，而不是包含期望输出行的一个数组）。所有这些重载方法都是调用主 `expect()` 方法，该方法接收一个包含期望输出行的对象数组以及一个包含多种标志的整数。标志是通过按位切换来实现的，每位与一个在 `Test.java` 开始部分已经定义的特定标志相对应。

`expect()` 方法首先检查 `flags` 参数，看看它是否应该延迟处理以便允许运行缓慢的程序能够跟上节奏。接着它调用静态方法 `fileToList()`，该方法将程序产生的输出文件的内容装换成一个 `List` 对象。`fileToList()` 方法还将每个字符串对象包装成了 `OutputLine` 对象；你很快就会清楚这样做的原因。最后，`expect()` 根据标志参数调用适合的 `verify()` 方法。

有三种验证方式：`verify()`、`verifyIgnoreOrder()` 和 `verifyAtLeast()`，分别对应与 `EXACT`、`IGNORE_ORDER` 和 `AT_LEAST` 模式。

```

///: com:bruceeckel:simpletest:OutputVerifier.java
package com.bruceeckel.simpletest;
import java.util.*;
import java.io.PrintStream;

public class OutputVerifier {
    private static void verifyLength(
        int output, int expected, int compare) {
        if((compare == Test.EXACT && expected != output)
            || (compare == Test.AT_LEAST && output < expected))
            throw new NumOfLinesException(expected, output);
    }
    public static void verify(List output, List expected) {
        verifyLength(output.size(), expected.size(), Test.EXACT);
        if(!expected.equals(output)) {
            //find the line of mismatch
            ListIterator it1 = expected.listIterator();
            ListIterator it2 = output.listIterator();
            while(it1.hasNext()
                && it2.hasNext()
                && it1.next().equals(it2.next()));

```

```

        throw new LineMismatchException(
            it1.nextIndex(), it1.previous().toString(),
            it2.previous().toString());
    }
}

public static void
verifyIgnoreOrder(List output, Object[] expected) {
    verifyLength(expected.length, output.size(), Test.EXACT);
    if(!(expected instanceof String[]))
        throw new RuntimeException(
            "IGNORE_ORDER only works with String objects");
    String[] out = new String[output.size()];
    Iterator it = output.iterator();
    for(int i = 0; i < out.length; i++)
        out[i] = it.next().toString();
    Arrays.sort(out);
    Arrays.sort(expected);
    int i = 0;
    if(!Arrays.equals(expected, out)) {
        while(expected[i].equals(out[i])) {i++;}
        throw new SimpleTestException(
            ((String) out[i]).compareTo(expected[i]) < 0
            ? "output: <" + out[i] + ">"
            : "expected: <" + expected[i] + ">");
    }
}

public static void
verifyAtLeast(List output, List expected) {
    verifyLength(output.size(), expected.size(),
        Test.AT_LEAST);
    if(!output.containsAll(expected)) {
        ListIterator it = expected.listIterator();
        while(output.contains(it.next())) {}
        throw new SimpleTestException(
            "expected: <" + it.previous().toString() + ">");
    }
}
} ///:~

```

这些“验证”方法可以测试程序生成的输出是否与由特定模式指定的期望输出相匹配。如果不能匹配，“验证”方法则会引起异常，放弃构建过程。

每个“验证”方法都使用 `verifyLength()` 测试输出的行数。`EXACT` 模式要求输出数组与期望的输出数组具有相同的大小，而且每个输出行与期望的输出数组中的对应行相等。

IGNORE_ORDER 也要求两个数组大小相同，但是这些行出现的实际顺序可以不予考虑（这两个输出数组必须可以置换的）。IGNORE_ORDER 模式用来测试线程实例，由于 JVM 不确定的线程调度机制，程序生成的输出行顺序可能无法预测。AT_LEAST 模式不需要这两个数组大小相同，但是，期望输出的每行必须包含在由程序生成的实际输出中，不管次序如何。这个特性对于测试那些包含可能会被打印也可能不会被打印的输出行的程序示例尤其有用，正如大多数处理垃圾回收问题的示例的情形一样。注意这三种模式是规范的；也就是说，如果一个测试通过了 IGNORE_ORDER 模式验证，那么它也会通过 AT_LEAST 模式验证；如果它通过了 EXACT 模式验证，那么它也一定会通过其他两种模式的验证。

请注意“验证”方法的实现是多么的简单。例如 `verify()`，直接调用由 `List` 类提供的 `equals()` 方法，`verifyAtLeast()` 调用的是 `List.containsAll()`。记着这两个输出列表可以包含 `OutputLine` 或 `RegularExpression` 对象。现在将简单的字符串对象包装成 `OutputLines` 的理由应该变得很清楚了：此办法可以让我们重载 `equals()` 方法，这对于利用 Java Collections API 是必需的。

在 `expect()` 数组中的对象不是 `String` 就是 `TestExpression`，它们会被封装成正则表达式（在第 12 章已经叙述过），这对于测试产生随机输出的示例是很有用的。`TestExpression` 类会对一个用来表示某个特定的正则表达式的字符串进行封装。

```
//: com:bruceeckel:simpletest:TestExpression.java
// Regular expression for testing program output lines
package com.bruceeckel.simpletest;
import java.util.regex.*;

public class TestExpression implements Comparable {
    private Pattern p;
    private String expression;
    private boolean isRegex;
    // Default to only one instance of this expression:
    private int duplicates = 1;
    public TestExpression(String s) {
        this.expression = s;
        if(expression.startsWith("%% ")) {
            this.isRegex = true;
            expression = expression.substring(3);
            this.p = Pattern.compile(expression);
        }
    }
    // For duplicate instances:
    public TestExpression(String s, int duplicates) {
        this(s);
        this.duplicates = duplicates;
    }
    public String toString() {
```

```

        if(isRegex) return p.pattern();
        return expression;
    }
    public boolean equals(Object obj) {
        if(this == obj) return true;
        if(isRegex) return (compareTo(obj) == 0);
        return expression.equals(obj.toString());
    }
    public int compareTo(Object obj) {
        if((isRegex) && (p.matcher(obj.toString()).matches()))
            return 0;
        return
            expression.compareTo(obj.toString());
    }
    public int getNumber() { return duplicates; }
    public String getExpression() { return expression;}
    public boolean isRegex() { return isRegex; }
} ///:~

```

TestExpression 可以将正则表达式模式与纯粹的字符串文字区分开来。为了方便起见，第二个构造器允许将多个相同的表达式行包装成一个单一对象。

这个测试系统已经相当有用了，而且创建它和将其付诸应用的练习也具有难以估量的价值。不过，最终我对它并不感到满意，并且还有一些想法，可能会在本书的下一版中（或者尽快）实现。

JUnit

虽然刚刚讲述的测试构架可以让我们简单而容易地验证程序输出，但在有些情况下，我们可能想对一个程序进行功能更广泛的测试。JUnit（可以 www.junit.org 从获得）是一种迅速出现的、为 Java 程序编写可重复测试的标准，它既可以提供简单测试也可以提供复杂测试。

最初的JUnit可能是基于JDK1.0的，因此不能使用Java的反射工具。因此，用老的JUnit版本写单元测试是一件相当忙碌而冗长的事情，而且我发现其设计也不令人满意。基于上述原因，我编写了自己的Java测试框架⁵，并且走了另一极端，“做可以运作的最简单的事情。”⁶此后，JUnit作出了修改并且使用反射大大简化了编写单元测试代码的过程。尽管我们可以选择使用带有测试包及其他所有复杂细节的“旧”方法来编写代码，但是我相信绝

⁵ 最初放在www.BruceEckel.com站点上的Thinking in Patterns (with Java)中。不过，随着JUnit中反射方法的增加，我的框架结构不再具有太多的意义了，并且将来可能会被删除。

⁶ 来自于极限编程（Extreme Programming, XP）的关键词语。具有讽刺意味的是，JUnit的作者之一（Kent Beck）也是Extreme Programming Explained（Addison-Wesley 2000）的作者之一和XP的倡导者之一。

大多数情况下你会采用这里所展示的简单方法（这样做会使我们的生活更加舒适愉快）。

在最简单的使用 JUnit 的方式中，我们将所有的测试放到 `TestCase` 的某个子类中。每个测试必须是 `public`，不接受任何参数，返回 `void`，并且具有一个以“test”开头的方法名。JUnit 的反射会把这些方法标识为单个的测试，然后一次一个地创建和运行它们，并且采取措施避免这些测试之间的负面影响。

传统情况下，`setUp()` 方法创建一个对象的通用集合并将其初始化，这些对象在所有的测试中都将用到；不过，我们也可以将所有这些初始化放到测试类的构造器中。JUnit 为每个测试创建一个对象以确保在测试运行之间没有不利的影响。不过，所有测试的所有对象都是同时被创建的（而不是正好在测试前才创建对象），因此，使用 `setUp()` 和构造器的唯一区别是：`setUp()` 是在测试之前直接被调用的。大多数情况下，这不是问题，而且为了简单起见，你可以使用构造器方法。

如果我们需要在每次测试后执行清除工作，那么无论它是任何清除（如果我们修改了任何需要恢复的静态数据，打开了需要关闭的文件，打开网络连接等等），我们都要写一个 `tearDown()` 方法。这也是可选的。

下面这个示例使用了上述这种简单的方式来创建 JUnit 测试，以演练标准的 Java `ArrayList` 类。为了跟踪 JUnit 怎样创建及清除它的测试对象，`CountedList` 继承了 `ArrayList`，并且添加了跟踪信息。

```
//: c15:JUnitDemo.java
// Simple use of JUnit to test ArrayList
// {Depends: junit.jar}
import java.util.*;
import junit.framework.*;

// So we can see the list objects being created,
// and keep track of when they are cleaned up:
class CountedList extends ArrayList {
    private static int counter = 0;
    private int id = counter++;
    public CountedList() {
        System.out.println("CountedList #" + id);
    }
    public int getId() { return id; }
}

public class JUnitDemo extends TestCase {
    private static com.bruceeckel.simpletest.Test monitor =
        new com.bruceeckel.simpletest.Test();
    private CountedList list = new CountedList();
    // You can use the constructor instead of setUp():
```

```

public JUnitDemo(String name) {
    super(name);
    for(int i = 0; i < 3; i++)
        list.add("" + i);
}
// Thus, setUp() is optional, but is run right
// before the test:
protected void setUp() {
    System.out.println("Set up for " + list.getId());
}
// tearDown() is also optional, and is called after
// each test. setUp() and tearDown() can be either
// protected or public:
public void tearDown() {
    System.out.println("Tearing down " + list.getId());
}
// All tests have method names beginning with "test":
public void testInsert() {
    System.out.println("Running testInsert()");
    assertEquals(list.size(), 3);
    list.add(1, "Insert");
    assertEquals(list.size(), 4);
    assertEquals(list.get(1), "Insert");
}
public void testReplace() {
    System.out.println("Running testReplace()");
    assertEquals(list.size(), 3);
    list.set(1, "Replace");
    assertEquals(list.size(), 3);
    assertEquals(list.get(1), "Replace");
}
// A "helper" method to reduce code duplication. As long
// as the name doesn't start with "test," it will not
// be automatically executed by JUnit.
private void compare(ArrayList lst, String[] strs) {
    Object[] array = lst.toArray();
    assertTrue("Arrays not the same length",
        array.length == strs.length);
    for(int i = 0; i < array.length; i++)
        assertEquals(strs[i], (String)array[i]);
}
public void testOrder() {
    System.out.println("Running testOrder()");
    compare(list, new String[] { "0", "1", "2" });
}

```

```

}
public void testRemove() {
    System.out.println("Running testRemove()");
    assertEquals(list.size(), 3);
    list.remove(1);
    assertEquals(list.size(), 2);
    compare(list, new String[] { "0", "2" });
}
public void testAddAll() {
    System.out.println("Running testAddAll()");
    list.addAll(Arrays.asList(new Object[] {
        "An", "African", "Swallow"}));
    assertEquals(list.size(), 6);
    compare(list, new String[] { "0", "1", "2",
        "An", "African", "Swallow" });
}
public static void main(String[] args) {
    // Invoke JUnit on the class:
    junit.textui.TestRunner.run(JUnitDemo.class);
    monitor.expect(new String[] {
        "CountedList #0",
        "CountedList #1",
        "CountedList #2",
        "CountedList #3",
        "CountedList #4",
        // '.' indicates the beginning of each test:
        ".Set up for 0",
        "Running testInsert()",
        "Tearing down 0",
        ".Set up for 1",
        "Running testReplace()",
        "Tearing down 1",
        ".Set up for 2",
        "Running testOrder()",
        "Tearing down 2",
        ".Set up for 3",
        "Running testRemove()",
        "Tearing down 3",
        ".Set up for 4",
        "Running testAddAll()",
        "Tearing down 4",
        "",
        "% Time: .*",
        "",
    });
}

```

```

        "OK (5 tests)",
        "",
    });
}
} ///:~

```

为了建立单元测试，我们仅须导入 `junit.framework.*` 包，然后扩展 `TestCase`，正如 `JUnitDemo` 所做的一样。另外，我们必须创建一个构造器，可以接收一个字符串参数并将其传递给它的 `super` 构造器。

对于每个测试，将会创建一个新的 `JUnitDemo` 对象，因而也将创建其所有的非静态属性。这意味着，每次测试都会创建和初始化一个新的 `CountedList` 对象 (`list`)，因为它是 `JUnitDemo` 的一个属性域。另外，构造器在每一次的测试中都会被调用，因此，`list` 在每次测试运行之前，将会被初始化为字符串“0”、“1”和“2”。

为了观察 `setUp()` 和 `tearDown()` 的行为，创建了一些方法用来显示关于那些正在被初始化或清除的测试的信息。注意，因为基类方法是 `protected` 的，因此重载的方法可以是 `protected` 或 `public` 的。

`testInsert()` 和 `testReplace()` 示范了典型的测试方法，因为它们遵循了所要求的方法签名和命名规则。`JUnit` 使用反射发现这些方法，然后把每一个都作为一项测试来运行。在这些方法内部，我们可以执行任何想要的操作，然后利用 `JUnit` 的断言 (`assertion`) 方法（它们都以名字“`assert`”开头）来验证我们测试的正确性（完整全面地的“`assert`”语句可以在 `JUnit` 中的 `junit.framework.Assert.java` 帮助文档中找到）。如果断言 (`assertion`) 失败，引起失败的表达式和值将会被显示出来。通常，这已经足够了，但是我们也可以使用每个 `JUnit` 断言语句的重载版本，让它们包含一个在断言失败时会被打印显示出来的字符串。

断言 (`assertion`) 语句并非是必要的；我们也可以只运行没有断言的测试，而且如果没有抛出异常就可以认为测试是成功的。

`compare()` 方法是一个“助手”方法的例子，该方法不是由 `JUnit` 执行而是被类中的其它测试所使用的。只要方法名不是以“`test`”开头，`JUnit` 就不会运行它或者期望它具有特定的方法签名。这里，`compare()` 之所以是 `private` 的，是为了强调它仅能在测试类内部被用到，但是它也可以是 `public` 的。余下的测试方法通过将此功能分解到 `compare()` 方法中来消除重复的代码。

为了执行 `JUnit` 测试，要在 `main()` 中调用静态方法 `TestRunner.run()`。该方法被传入了包含测试集合的类作为其调用参数，而且它会自动建立并运行所有的测试。从 `expect()` 的输出中，我们可以看到运行所有测试所需的所有对象一开始就在一个批处理中被创建了——这正是构造发生的地方⁷。在运行每个测试前，都要调用 `setUp()` 方法，然后运行测试，随后是 `tearDown()` 方法。`JUnit` 是用“.”划分每个测试的。

⁷ Bill Venners和我曾经对此进行过详尽的讨论，而且我们无法明白它为什么是按照这种方式而不是恰在测试运行之前创建每个对象。可能仅仅是因为它是 `JUnit` 最初被实现时的产物。

尽管仅使用如上述示例所展示的 JUnit 的最简单使用方式，我们大概也可以很容易地实现我们的目的；但是 JUnit 最初设计时具有一个功能几乎显得过剩的复杂结构。如果你很好学，你可以很容易地学到关于它们的更多知识，因为可以从 www.JUnit.org 下载到 JUnit，以及文档和指南。

利用断言提高可靠性

断言 (assertion)，我们已经看到在本书中前面的例子中曾经用到过，它被添加到 Java 的 JDK1.4 版本中用来帮助程序员提高他们的程序的可靠性。如果能被正确地使用，断言可以通过验证在程序执行期间满足的特定条件来增强程序的鲁棒性。例如，假设在某个对象中有一个数字类型的域，代表公历中的月份。我们知道这个值一定总是在 1-12 这个范围内，可以用一个断言对此进行检查，并且如果它不知何故落在这个范围之外，将会报错。如果是在一个方法的内部，我们可以使用断言来检查参数的合法性。这些是确保程序正确性的重要测试，但是却不能由编译时刻的检查来实现，并且也不属于单元测试的范畴。在这部分，我们将会着眼于断言机制的结构，以及利用断言局部实现 DBC (design by contract) 概念的方法。

断言语法

既然我们可以使用其它的编程结构来仿真断言的效用，这表明将断言添加到 Java 的关键之处在于它们很容易编写。断言语句有两种常用的形式：

```
assert boolean-expression;
assert boolean-expression: information-expression;
```

这两个语句都表明“我断言这个布尔表达式将会产生 true 值。”如果不是这种情形，该断言将会产生 `AssertionError` 异常。这是 `Throwable` 的一个子类，而且也同样不需要异常说明。

遗憾的是，第一种形式的断言不产生任何包含有关由失败断言所产生的异常中的布尔表达式的信息（与其他大多数编程语言的断言机制形成了对比）。下面示例展示了第一种形式的用法：

```
//: c15:Assert1.java
// Non-informative style of assert
// Compile with: javac -source 1.4 Assert1.java
// {JVMArgs: -ea} // Must run with -ea
// {ThrowsException}

public class Assert1 {
```

```

    public static void main(String[] args) {
        assert false;
    }
} ///:~

```

在 JDK1.4 中，缺省情况下断言是关闭的（这很令人烦恼，不过设计者们却竭力使他们自己相信这是一个好主意）。为了防止编译时的错误，我们必须带下面的标志进行编译：

```
-source 1.4
```

如果我们没有使用该标志，将会获得一个友善的消息说在 JDK1.4 中 `assert` 是关键字，不能再用作标识符。

如果我们只是按照常规的方式运行程序，没有任何特定的断言标志，那么什么也不会发生。在运行程序的时候，我们必须使能断言。最简单的使能方式是用 `-ea` 标志，不过我们也可以把它们拼写全：`-enableassertions`。这样就会运行程序并且执行所有的断言语句。因此，我们将得到：

```

Exception in thread "main" java.lang.AssertionError
    at Assert1.main(Assert1.java:8)

```

我们可以看到输出包含的有用信息并不是很多。另一方面，如果我们使用信息表达式，当断言失败时，我们将会产生一条有用的消息。

为了使用第 2 种形式，我们要提供一个信息表达式，它将作为异常堆栈追踪的一部分被显示出来。这个信息表达式可以生成任何数据类型。不过，最有用的信息表达式通常是带有文本的字符串，这对程序员很有帮助，下面是一个例子：

```

///: c15:Assert2.java
// Assert with an informative message
// {JVMArgs: -ea}
// {ThrowsException}

public class Assert2 {
    public static void main(String[] args) {
        assert false: "Here's a message saying what happened";
    }
} ///:~

```

现在，输出是：

```

Exception in thread "main" java.lang.AssertionError:
Here's a message saying what happened at
Assert2.main(Assert2.java:6)

```

尽管我们在这里看到的仅仅是一个简单的 **String** 对象，但是实际上信息表达式可以产生任何类型的对象，因此，我们通常构建更复杂的字符串，例如包含失败断言所涉及的对象值。

因为从失败断言查看有用信息的唯一办法就是使用信息表达式，这也是本书一再用到的形式，而第 1 种形式则被认为是一种较差的选择。

我们也可以基于类名或包名（）来决定是打开还是关闭断言（也就是说，我们可以在整个包内使能或者禁止断言）。你可以在 JDK1.4 的文档中找到关于断言的细节。如果我们有一个很大的、装备断言的工程，而且我们想将其中的一些断言关掉，那么这个特性对我们可能就很有帮助。不过，日志机制和调试机制（都会在本章稍后部分讲到）可能都会是捕获那种类型信息的更好的工具。本书仅是在需要的时候打开所有的断言，因此，我们会忽略对断言的细粒度的控制。

还有另一种控制断言的方法：通过钩住 `ClassLoader` 对象来编程实现。JDK1.4 为 `ClassLoader` 添加了几个新方法，允许动态使能和禁止断言，包括 `setDefaultAssertionStatus()`，它为所有随后载入的类设置断言的状态。因此，我们可以认为我们几乎可以像下面这样悄无声息地打开所有断言。

```
//: cl5:LoaderAssertions.java
// Using the class loader to enable assertions
// Compile with: javac -source 1.4 LoaderAssertions.java
// {ThrowsException}

public class LoaderAssertions {
    public static void main(String[] args) {
        ClassLoader.getSystemClassLoader()
            .setDefaultAssertionStatus(true);
        new Loaded().go();
    }
}

class Loaded {
    public void go() {
        assert false: "Loaded.go()";
    }
} ///:~
```

尽管这样可以在程序运行时，不必在命令行使用 **-ea** 标志；但这并不是一个彻底的解决方案，因为我们仍必须使用 **-source 1.4** 标志编译这一切。使用命令行参数使能断言可能只是为了简单直接；当交付一个单机版产品时，我们可能不得不为用户建立一个执行脚本来启动程序，以便配置其他的启动参数。

不过，在程序运行时，决定你想要所需的断言使能是有意义的。我们可以使用下面这个放在

系统主类里面的 **static** 语句来实现:

```
static {
    boolean assertionsEnabled = false;
    // Note intentional side effect of assignment:
    assert assertionsEnabled = true;
    if (!assertionsEnabled)
        throw new RuntimeException("Assertions disabled");
}
```

一旦使能了断言, **assert** 语句就会被执行, 而且 **assertionsEnabled** 也会被设置为 **true**。断言永远不会失败, 因为任务的返回值就是指定值。如果断言未被使能, 就不会执行 **assert** 语句, **assertionsEnabled** 也将保持为 **false**, 从而引起异常。

为 DBC 使用断言

DBC (Design by Contract) 是由 Bertrand Meyer (Eiffel 编程语言的创建者) 所阐明的一个概念, 它通过确保对象遵循特定的、不能被编译时的类型检查所验证的规则, 来帮助建立健壮的程序⁸。这些规则由待解决问题的性质所决定, 这些待解决问题处于编译器可以了解和测试的范围之外。

尽管断言不能直接实现 DBC (就像 Eiffel 语言那样去实现), 但是它们可以用来创建非正式的 DBC 编程格式。

DBC 的基本思想是: 有一个明确指定的契约, 该契约存在于服务的提供者和服务的消费者或客户之间。在面向对象的编程中, 服务通常是由对象提供, 而且对象的边界——供应者和消费者之间的划分——是该对象类的接口。当客户调用特定的公共方法时, 他们总是期望能从那个调用中会产生某种行为: 对象中状态的改变, 以及一个可预测的返回值。Meyer 的理论是:

1. 这种行为可以被明确指定, 就好像是一个契约。
2. 这种行为可通过执行某个运行时刻的检查来确保, 他检查调用先验条件, 后验条件以及不变量。

无论你是否同意第一点总是对的, 对于很多情形来说, 它确实看起来是正确的, 这使得 DBC 成为了一种有趣的方法。(我相信, 像其他任何解决方案一样, 它也存在有效性边界。但是如果你知道这些边界, 就会知道在什么时候应该设法应用它。)特别是, 在设计过程中极其有价值的一个部分就是作用对特定类的 DBC 限制条件的表达式; 如果我们不能指定限制条件, 我们可能就不是很清楚自己正在努力构建什么?

⁸ DBC在面向对象的软件构架 (第二版, Bertrand Meyer, Prentice Hall 著, 1997) 中的第 11 章有详细介绍。

检测指令

在深入探讨 DBC 工具之前，先考虑一下断言的最简单用法，Meyer 称之为检测指令。一个检测指令表示我们确信在代码的某个位置上某个特定属性将会被满足。检测指令的思想是为了表示代码中的非显而易见的结论，这么做不但为了验证测试，而且还可以作为给代码的后续读者的文档。

例如，在化学过程中，我们可以将一种清澈液体滴定到另一种液体中，并且当我们达到某一点时，所有液体都变成蓝色。从两种液体的颜色来看，这种结果并不是显而易见的；它是复杂反应的一部分。在全部滴定过程中，一个很有用的检测指令是断言作为结果而产生的液体是蓝色的。

在 JDK1.4 中引入的另一个例子是方法 `Thread.holdsLock()`。它被用于复杂线程的情形（例如，以线程安全方式迭代一个集合）。这里，我们必须依赖客户程序员或者在我们系统中正确使用类库的其他类，而不是单单依赖于关键字 `synchronized`。为了确保代码遵循我们类库设计的规定，我们可以断言当前的线程确实持有锁。

```
assert Thread.holdsLock(this);
```

检测指令对我们的代码来说，是颇有价值的添加物。既然断言可以被禁止，那么检测指令应该可以被应用于我们还没有明确了解对象或程序状态的任何时候。

先验条件

先验条件（precondition）是一种测试，以确保客户端（调用此方法的代码）已经履行了它那部分契约。这几乎意味着要刚好在方法调用前检测参数（在我们在那个方法中作任何事情之前），以确保这些参数是适合用于这个方法的。既然我们永远不知道客户端打算传递给我们的是什么东西，那么先验条件检测总是一个好主意。

后验条件

后验条件（postcondition）测试检测我们在方法中所做事情的结果。这部分代码置于方法调用的结尾部分，`return` 语句（如果有的话）之前。对于在返回计算结果之前应该对其进行验证的运行时间长的复杂方法（也就是说，在一些情形下，出于某些原因我们不能总是相信结果），后验条件检测是必不可缺的，但是任何时候我们都可以对方法的结果进行限制条件描述，所以，明智的做法是将代码中的这些限制条件表示成后验条件。在 Java 中，它们被编码为断言，不过断言语句会随着方法的不同而有所不同。

不变量

不变量（invariant）对方法调用之间需要被维护的对象状态提供保障。不过，在一个方法执行期间，它不会阻止该方法暂时偏离这些保证。它仅仅是说对象的状态信息将总是在下面的节点处遵守这些规则：

1. 在方法入口之上。
2. 在方法离开之前。

另外，不变量还是一种对构建后的对象状态的保障。

根据这些描述，一个有效的不变量应该定义成一个方法，比如有可能被命名为 `invariant()`，它将在构建之后以及在每个方法的开始和结束部分被调用。该方法的调用如下：

```
assert invariant();
```

这样，如果由于性能原因，我们选择禁止断言，那么它就不会产生任何开销。

DBC 的放宽

尽管 Meyer 强调了能够表示先验条件、后验条件和不变量的重要性以及在开发期间使用这些东西的好处，但是他还是承认将所有的 DBC 代码都囊括到打包的产品中的做法并不总是很实际。我们可以基于在特定位置上放置代码的信任度来放宽 DBC 检测。下面是从最安全的到最不安全的放宽的顺序：

1. 首先，在每个方法开始部分的不变量检测可以被禁止，因为每个方法结束部分的不变量检测将会保证在每个方法调用的初始时刻对象的状态是有效的。也就是说，我们通常可以相信对象的状态在方法调用之间不会被改变。这是一种相当安全的假设，以至于我们可以选择只带有在方法结束时进行不变量检测的方式来编写代码。
2. 接下来，后验条件可以被禁止，只要我们拥有适当的单元测试来验证我们的方法正在返回恰当的值。既然不变量检测一直在监视着对象的状态，而后验条件检测只验证方法运行期间的计算结果，因此，为了利于单元测试，可以将其丢弃。单元测试并不像运行时空后验条件检测那样安全，不过用它可能已经足够了，特别是在我们对代码有足够信心的情况下更是如此。
3. 如果我们足够确信方法体不会将对象置于非法状态，那么方法调用结束部分的不变量检测就可以被禁止。可能用白盒单元测试可以验证对象的状态（也就是说，单元测试能够访问私有属性，因此它们可以验证对象的状态）。因此，尽管它不可能完全像调用 `invariant()` 那样健壮，但是可以将不变量检测从运行时测试“迁移”到“构建时”测试（经由单元测试），正像用后验条件一样。
4. 最后，作为最后的手段，我们可以禁止先验条件测试，这是最不安全、最不明智的做法，因为虽然我们了解并且控制着自己的代码，但是我们不能控制客户端传递给方法的会是

什么参数。然而，如果在这样的情况下：(a) 极想获得高性能，而且分析结果指出先验条件检测已经成为瓶颈，(b) 我们有某种适当的保证，确信客户将不会违反先验条件（像我们已经自己编写了客户端代码这种情形），那么我们就可以接受禁止先验条件测试。

当我们将检测禁止时，我们不应该移除这里所讲述的执行检测的代码。如果发现 bug，我们可以很容易地打开测试以便快速发现问题。

实例：DBC+白盒单元测试

下面的例子演示了用单元测试联合来自 DBC 的概念所产生的潜力。它展示了一个小的先进先出（FIFO）队列类，该队列被用来实现一个“循环”数组——也就是说，一个按照循环方式使用的数组。当到达数组末尾时，这个类再向后环绕到数组的开头位置。

我们可以对这个队列做出若干个契约性的定义：

1. 先验条件（应用于 put ()）：不允许将空元素添加到队列中。
2. 先验条件（应用于 put ()）：向满队列添加元素是非法的。
3. 先验条件（应用于 put ()）：从空队列取元素是非法的。
4. 后验条件（应用于 get ()）：不能从数组中产生空元素。
5. 不变量：数组中包含对象的区域不能有任何空元素。
6. 不变量：数组中不包含对象的区域必须仅有空值。

下面是实现这些规则的一种方法，它为每种类型的 DBC 元素都使用了显示的方法调用：

```
//: c15:Queue.java
// Demonstration of Design by Contract (DBC) combined
// with white-box unit testing.
// {Depends: junit.jar}
import junit.framework.*;
import java.util.*;

public class Queue {
    private Object[] data;
    private int
        in = 0, // Next available storage space
        out = 0; // Next gettable object
    // Has it wrapped around the circular queue?
    private boolean wrapped = false;
    public static class
        QueueException extends RuntimeException {
        public QueueException(String why) { super(why); }
    }
    public Queue(int size) {
```

```

    data = new Object[size];
    assert invariant(); // Must be true after construction
}
public boolean empty() {
    return !wrapped && in == out;
}
public boolean full() {
    return wrapped && in == out;
}
public void put(Object item) {
    precondition(item != null, "put() null item");
    precondition(!full(), "put() into full Queue");
    assert invariant();
    data[in++] = item;
    if(in >= data.length) {
        in = 0;
        wrapped = true;
    }
    assert invariant();
}
public Object get() {
    precondition(!empty(), "get() from empty Queue");
    assert invariant();
    Object returnVal = data[out];
    data[out] = null;
    out++;
    if(out >= data.length) {
        out = 0;
        wrapped = false;
    }
    assert postcondition(
        returnVal != null, "Null item in Queue");
    assert invariant();
    return returnVal;
}
// Design-by-contract support methods:
private static void
precondition(boolean cond, String msg) {
    if(!cond) throw new QueueException(msg);
}
private static boolean
postcondition(boolean cond, String msg) {
    if(!cond) throw new QueueException(msg);
    return true;
}

```



```

}
private boolean invariant() {
    // Guarantee that no null values are in the
    // region of 'data' that holds objects:
    for(int i = out; i != in; i = (i + 1) % data.length)
        if(data[i] == null)
            throw new QueueException("null in queue");
    // Guarantee that only null values are outside the
    // region of 'data' that holds objects:
    if(full()) return true;
    for(int i = in; i != out; i = (i + 1) % data.length)
        if(data[i] != null)
            throw new QueueException(
                "non-null outside of queue range: " + dump());
    return true;
}
private String dump() {
    return "in = " + in +
        ", out = " + out +
        ", full() = " + full() +
        ", empty() = " + empty() +
        ", queue = " + Arrays.asList(data);
}
// JUnit testing.
// As an inner class, this has access to privates:
public static class WhiteBoxTest extends TestCase {
    private Queue queue = new Queue(10);
    private int i = 0;
    public WhiteBoxTest(String name) {
        super(name);
        while(i < 5) // Preload with some data
            queue.put("" + i++);
    }
    // Support methods:
    private void showFullness() {
        assertTrue(queue.full());
        assertFalse(queue.empty());
        // Dump is private, white-box testing allows access:
        System.out.println(queue.dump());
    }
    private void showEmptiness() {
        assertFalse(queue.full());
        assertTrue(queue.empty());
        System.out.println(queue.dump());
    }
}

```

```

}
public void testFull() {
    System.out.println("testFull");
    System.out.println(queue.dump());
    System.out.println(queue.get());
    System.out.println(queue.get());
    while(!queue.full())
        queue.put(" " + i++);
    String msg = "";
    try {
        queue.put("");
    } catch(QueueException e) {
        msg = e.getMessage();
        System.out.println(msg);
    }
    assertEquals(msg, "put() into full Queue");
    showFullness();
}
public void testEmpty() {
    System.out.println("testEmpty");
    while(!queue.empty())
        System.out.println(queue.get());
    String msg = "";
    try {
        queue.get();
    } catch(QueueException e) {
        msg = e.getMessage();
        System.out.println(msg);
    }
    assertEquals(msg, "get() from empty Queue");
    showEmptiness();
}
public void testNullPut() {
    System.out.println("testNullPut");
    String msg = "";
    try {
        queue.put(null);
    } catch(QueueException e) {
        msg = e.getMessage();
        System.out.println(msg);
    }
    assertEquals(msg, "put() null item");
}
public void testCircularity() {

```

```

        System.out.println("testCircularity");
        while(!queue.full())
            queue.put(" " + i++);
        showFullness();
        // White-box testing accesses private field:
        assertTrue(queue.wrapped);
        while(!queue.empty())
            System.out.println(queue.get());
        showEmptiness();
        while(!queue.full())
            queue.put(" " + i++);
        showFullness();
        while(!queue.empty())
            System.out.println(queue.get());
        showEmptiness();
    }
}

public static void main(String[] args) {
    junit.textui.TestRunner.run(Queue.WhiteBoxTest.class);
}
} ///:~

```

in 计数器表示数组中下一个将要置入数组的对象的位罝，而 **out** 计数器表示下一对象将来自于数组中的何处。**wrapped** 标志表示 **in** 已经“环绕了一圈了”，现在正要从后面靠上 **out** 所指的位置了。当 **in** 和 **out** 一致时，表明队列是空的（如果 **wrapped** 为 **false**）或满的（如果 **wrapped** 为 **true**）。

我们可以看到 **put()** 和 **get()** 方法调用了 **precondition()**、**postcondition()** 和 **invariant()** 方法，它们是在类的更下方定义的 **private** 方法。**precondition()** 和 **postcondition()** 都是为了使代码清楚明了而设计的助手方法。注意，**precondition()** 的返回为 **void**，因为它没有使用 **assert**。如先前提到的，我们通常想要在代码中保持先验条件；不过，如果你怯于将它们关闭所带来的令人头痛的迁移操作，那么通过将它们封装在一个 **precondition()** 方法调用内的做法，就是你可以得到的一个更好的选择。

postcondition() 和 **invariant()** 返回一个布尔值，所以它们可以被用于 **assert** 语句中。因而，如果由于性能原因而禁止断言，将不会再进行任何方法调用。

invariant() 对对象执行内部有效性检查。我们可以看到正如 Meyer 所建议的那样，如果在每个方法调用的开始和结尾部分都执行这样的操作，代价会是很大的。不过，在代码中将其明确地表现出来还是很有价值的，有助于我获得正确的执行。另外，如果我们对实现作任何，**invariant()** 都将确保我们不会破坏代码。但是，我们会发现将不变量测试从方法调用移至单元测试代码中，是一件相当琐碎的事情。如果我们的单元测试相当彻底，我们可以对“不变量将得到遵守”这一点置于相当高的相信度。

注意：`dump()` 助手方法返回包含所有数据的字符串而不是直接将数据打印出来。这种方案为关于怎样使用此信息提供了更多的选择。

`TestCase` 的子类 `WhiteBoxTest` 是作为一个内部类而创建的，所以它可以访问 `Queue` 的 `private` 元素，因此能够验证底层的具体实现，而不是像白盒测试那样仅测试类的行为。其构造器中添加一些数据，所以对每个测试来说，`Queue` 是部分填充满的。调用支持方法 `showFullness()` 和 `showEmptiness()` 分别来验证 `Queue` 是慢的还是空的。四个测试方法的每个方法确保 `Queue` 操作的不同方面能够正确地运行。

注意，将 DBC 和单元测试结合起来，我们不仅可以两全其美，而且还得到了一个迁移路径——可以将 DBC 测试移到单元测试中，而不是直接禁止它们，这样我们仍具有某种级别的测试。

用 Ant 构建

我的职业生涯开始于编写控制实时设备的汇编语言。这些程序通常适合用单一文件，因此当别人向我介绍 `make` 工具时，我并不是很兴奋，因为我曾经要干的最复杂的事情也就只是在几个代码文件之上运行汇编程序或者是 C 编译器。在那之前，构建一个项目算不上是我的任务中的困难部分，手工运行所有这些事物也不是太麻烦。

随着时间的流失，发生了两件事情。首先，我开始创建包含更多文件的复杂项目。弄明白哪些文件需要编辑已经超出了我的想象（或者是希望）。再者，因为这种复杂性，我开始意识到无论构建过程多么简单，如果我们想多次做某事，我们也会开始变得杂乱不堪，而且处理过程的有些部分也开始因为错误而失败。

自动化所有事物

我开始意识到要为一个系统创建健壮而可靠的模式，就必须使进入构建过程的任何事物都要自动化。这需要专注于重要部分，正像编写程序需要专注一样，但是这样做的代价是我们要“一次性”地解决所有问题，而且从此刻起，我们就要依赖于我们的构建配置来管理具体的细节了。它是“抽象”这一基本编程原则的一种变体：通过将细节隐藏于过程内部并赋予该过程一个名字，我们将我们自己从碾碎的细节中抽身而出。多年后，这个处理过程的名字被叫做 `make`。

`make` 应用程序与 C 语言一起都是作为创建 Unix 操作系统的一种工具而出现的。`make` 的主要功能是比较两个文件的日期，并且执行某些使这两个文件彼此反映最新的变化量的操作。在我们的项目工程中，所有文件之间的关系以及使它们彼此反映最新的变化量的必要规则（此规则通常是在一个源文件上运行 C/C++ 编译器）包含在一个生成文件（`makefile`）中。程序员要创建这个包含了怎样构建该系统的描述的生成文件。当我们想使系统反映最新的变换，我们仅需在命令行键入 `make`。到目前为止，安装 Unix/Linux 程序的过程由将它们解包及键入 `make` 命令所构成。

make 的问题

make 观念无疑是一个好观念，而且这个观念还在扩散，从而产生了许多 **make** 版本。C 和 C++ 编译器的提供商通常连同他们自己的编译器一起提供他们自己的 **make** 变体——这些变体通常提供了自由让人们选择什么是标准的生成文件规则，因此作为结果的生成文件彼此之间无法运行。这个问题最后是通过一个曾经是而且现在仍旧是优于其他 **make** 的 **make: GNU make**⁹ 来解决的，并且它也是免费的，因此使用它不存在任何阻碍。这个工具拥有比其他 **make** 版本显著优异的性能设置，而且在任何平台上都可以使用。

在“Java 编程思想 (*Thinking in Java*)”的前两个版本中，我使用生成文件来构建书本上源代码树中的所有代码。我可以自动地生成这些生成文件——一个目录一个，而且在根目录下是一个主生成文件，可以调用其他的生成文件——使用的是一个我最初用 C++ 为“Java 编程思想 (*Thinking in C++*)”而编写的一个工具 (大概用了两星期)。后来我又用 Python (大概半天时间吧) 重写了它，并命名为 **MakeBuilder.py**¹⁰。它可以在 Windows 和 Linux/Unix 上运行，但是我必须写一些额外的代码才能这么做，并且我从来没有在 Macintosh 上试过。它存在着 **make** 的第一个问题：我们可以让它在多平台上运转，但它本来并不是跨平台的。因此，对于一个假定“写一次，在任何地方都可以运行” (也就是 Java) 的语言来说，如果我们要使用 **make**，我们就要花费很大的努力才能在构建系统中获得同样的行为。

make 的其他问题也许能够总结为：它与其它的为 Unix 而开发的众多工具一样；创建工具的人不能抵制创建他们自己的语言语法的诱惑，因此，Unix 中充满了全都明显不同的工具，而且同样地难于理解。也就是说，**make** 语法很难被整体理解——我曾经学过几年——而且会遇见很多恼人的事情，例如支持制表符而不支持空格符¹¹。

虽然这么说，但是请注意，我仍旧发现对于我所创建的许多工程来说，GNU **make** 还是必不可少的。

Ant: 事实上的标准

make 的所有这些问题激怒了一位名叫 James Duncan Davidson 的 Java 程序员，这些问题太多了，于是他创建了一种开放源码的工具——Ant，该工具被迁移到了 Apache 的企业网站 <http://jakarta.apache.org/ant> 上。这个站点包含了可执行的 Ant 及其文档的完全下载。Ant 在不断地成长和提高，到现在它已经成为了 Java 工程事实上的标准构

⁹ 除了个别公司，由于无法理解的原因，仍旧相信封闭源码的工具无论怎样都是比较好或者优于的技术支持。我所见到的这种理由成立的唯一情形是工具只有很少的使用者基础，但是尽管这样，更安全的作法是雇佣专业人士修改开放源码的工具，这样可以利用先前的成果而且可以确保我们的付出所得到的成果不会变成无法利用了 (并且还会使我们更容易地发现其他的专业人士已经很熟悉这些程序了)。

¹⁰ 它无法在网上得到，因为它太客户化了以至于无法通用。

¹¹ 其他的工具仍在开发中，它们尝试修补 **make** 的问题，但是没有向 Ant 的妥协。例如，见 www.a-a-p.org 或在网上搜索“bjam”。

建工具。

为了使 Ant 跨平台,用于工程描述文件的格式是 XML(在 Thinking in Enterprise Java 中讲述)。我们不是创建一个生成文件,而是创建一个构建文件,这个文件的缺省命名为 **build.xml** (这样我们可以在命令行中只用键入“ant”。如果我们将构建文件命名为其他名字,那么我们必须用一个命令行标志来指定该名字。)

对我们的构建文件唯一严格的要求是它必须是一个有效的 XML 文件。Ant 弥补了与平台相关的问题,像行末字符以及目录路径的分隔符等。我们可以在构建文件中使用我们喜欢的制表符或者空格符。另外,在构建文件中所使用的语法和标签名字可以产生易读、可理解的(因此也是可维护的)代码。

在这之上, Ant 被设计成可扩展的,它带有一个标准接口,允许我们编写自己的任务以满足 Ant 自带的任务(task)所不能满足的需求(不过,它们通常如此,而且其类库是在定期地扩展)。

与 **make** 不同, Ant 的学习曲线相当平缓。我们并不需要了解很多就可以创建一个对在某一目下的 Java 代码进行编译的构建文件。例如,下面是本书第 2 章的一个非常基础的 **build.xml** 文件。

```
<?xml version="1.0"?>

<project name="Thinking in Java (c02)"
  default="c02.run" basedir=".">
  <!-- build all classes in this directory -->
  <target name="c02.build">
    <javac
      srcdir="${basedir}"
      classpath="${basedir}/.."
      source="1.4"
    />
  </target>

  <!-- run all classes in this directory -->
  <target name="c02.run" depends="c02.build">
    <antcall target="HelloDate.run"/>
  </target>

  <target name="HelloDate.run">
    <java
      taskname="HelloDate"
      classname="HelloDate"
      classpath="${basedir};${basedir}/.."
      fork="true"
    />
  </target>
</project>
```

```

        failonerror="true"
    />
</target>

<!-- delete all class files -->
<target name="clean">
    <delete>
        <fileset dir="${basedir}" includes="**/*.class"/>
        <fileset dir="${basedir}" includes="**/*Output.txt"/>
    </delete>
    <echo message="clean successful"/>
</target>

</project>

```

第一行声明该文件遵循 XML1.0 版本。除了我们可以标记自己的标签，以及要严格遵守 XML 规则外，XML 看上去很像 HTML（注意注释语法是一样的）。例如，一个像 **<project>** 这样的开放操作标签要么以连同斜线一起的关闭三角括号这样的标签 (**</>**) 而结束，要么以像我们在文件末尾看到的那种相匹配的关闭标签 (**</project>**) 而结束。在标签内部可以有属性，不过属性值必须要用引号引起来。XML 允许任意排版格式，但通常采用的是像在这里看到的这种缩排格式。

每个构建文件都可以管理由它的 **<project>** 标签所描述的一个单一工程。该工程有一个可选择的在显示关于构建信息的时候用到的 **name** 属性，。当我们仅在命令行键入 **ant** 而没有给出特定目标名时，**default** 属性就是必需的，而且要指向要构建的目标 (**target**)。引用路径 **basedir** 可以被构建文件的其他地方使用。

target 具有 **dependencies** 和 **tasks**。**Dependencies** 要表明“其他哪些 **target** 必需在此 **target** 构建之前被构建？”我们注意到创建的 **default** 目标是 **c02.run**，而且 **c02.run target** 表明它在次序上要依赖 **c02.build**。因此，**c02.build** 目标必须在 **c02.run** 可以执行之前被执行。按照这种方式对构建文件进行划分，不仅可以使它更易于理解，而且也使我们可以通过 **Ant** 命令行来选择我们想要做的事情；如果我们指定 **'ant c02.build'**，那么它将只会编译代码，但是如果我们声明 **'ant c02.run'**（或者，由于它是缺省 **target**，所以也可以仅只声明 **'ant'**），那么将首先确定它是否已经被构建过，然后再运行示例。

因此，为了使工程成功，那么按照上述顺序，目标 **c02.build** 和 **c02.run** 必须首先成功。**c02.build** 目标包含一个单一任务 (**task**)，该任务是一个命令，它真正去执行使事物反映最新变化的工作。此任务对当前基目录下的所有 Java 文件运行 **javac** 编译器；注意语法 **\${}** 用来产生前面定义过的变量的值，而且目录路径中的斜线方向并不重要，因为 **Ant** 抵消了对我们所运行的操作系统的依赖。**classpath** 属性提供一个要添加到 **Ant** 的类路径中的目录列表，**source** 属性指定所使用的编译器（这实际上这个属性只有 **JDK 1.4** 及以上版本才会用到）。注意 Java 编译器负责对类自身之间的依赖进行排序整理，因此我们不必像用 **make** 和 **C/C++** 那样，不得不明确声明文件之间的依赖（这样节约了很多功夫）。

为了运行目录中的程序（在这种情况下，仅是一个单一的 **HelloDate** 程序），该构建用到了一个名为 **antcall** 的任务，它递归调用 Ant 的另外一个任务，在上例中，被递归调用的任务仅仅是使用 **java** 来执行程序。注意 **java** 任务有一个 **taskname** 属性；这个属性实际上能够被所有的任务访问，并且会在 Ant 输出日志信息时被用到。

正如我们大概所期望的，**java** 标签还可以选择设置要执行的类名，以及类路径。另外，

```
fork="true"
failonerror="true"
```

属性告诉 Ant 分开一个新的进程来运行这个程序，并且如果程序失败了，那么 Ant 构建过程也就失败了。我们可以从和 Ant 一起被下载的文档中查看到不同的任务及它们的属性。

最后一个目标通常在每个构建文件都可以发现：它允许我们通过声明 **ant clean** 来删除所有为了执行此构建过程而创建的文件。无论何时，只要我们创建了一个构建文件，我们就应该谨慎地包含一个 **clean** 目标，因为我们通常是最清楚什么可以删除、什么应该保留的人。

clean 目标引入了某些新语法。我们可以使用此任务的一行脚本的版本来删除单一的项，像这样：

```
<delete file="${basedir}/HelloDate.class"/>
```

此任务的多行版本允许我们指定一个文件集，这是一个更复杂的对文件集合的描述，而且利用通配符可以包含和排除指定的文件。在这个例子中，删除的文件集包含在这个目录下的所有文件，所有具有 **.class** 扩展名的子目录，以及以 **Output.txt** 结束的当前目录下的所有文件。

这里展示的构建文件相当简单；在本书的源代码树里面（可以从 www.BruceEckel.com 下载），我们将会发现更多更复杂的构建文件。同样，Ant 能够做的比本书使用到的更多。对于它的能力的全部细节，见和安装程序在一起的文档。

Ant 的扩展

Ant 带有一个扩展 API，以便我们可以用 Java 编写自己的任务。我们可以在官方的 Ant 文档中以及关于 Ant 的出版书籍中找到其全部细节。

因为是一个可供选择的方法，因此我们可以直接编写一个 Java 程序，并且从 Ant 中调用它；这样，我们就不必学习扩展 API 了。例如，编译本书中的代码，我们需要验证用户正在使用的 Java 版本是 JDK 1.4 还是更高的版本，因此我们创建了如下程序：

```
//: com:bruceeckel:tools:CheckVersion.java
// {RunByHand}
```



```

package com.bruceeckel.tools;

public class CheckVersion {
    public static void main(String[] args) {
        String version = System.getProperty("java.version");
        char minor = version.charAt(2);
        char point = version.charAt(4);
        if(minor < '4' || point < '1')
            throw new RuntimeException("JDK 1.4.1 or higher " +
                "is required to run the examples in this book.");
        System.out.println("JDK version "+ version + " found");
    }
} ///:~

```

这里直接用 **System.getProperty()** 来获得 Java 版本，获得的版本至少要是 1.4，否则抛出异常。当 Ant 发现这个异常时，它将会停止运行。现在我们可以将下面部分纳入任何我们想检测版本号的构建文件中了。

```

<java
    taskname="CheckVersion"
    classname="com.bruceeckel.tools.CheckVersion"
    classpath="${basedir}"
    fork="true"
    failonerror="true"
/>

```

如果我们使用这种方法添加工具，我们可以很快地对它们进行编写和测试，如果证明它是正确的，我们可以投入更多的功夫去编写一个 Ant 扩展了。

用 CVS 进行版本控制

版本控制系统是一个工具类，它已经被开发出了多年了，可以我们帮助管理大型团队参与的程序设计工程。它还被证明是所有真正的开放源码工程成功的基础，因为开放源码团队大多总是通过 Internet 分布全球的。因此即使仅有两个人做某一个工程，他们也会从版本控制系统的使用中受益。

对于开放源码工程，事实上的标准修改控制系统叫做并发版本系统（CVS，Concurrent Versions System），可以从 www.cvshome.org 获得。因为它是开放源码的，因此很多人知道该怎么使用它，而且 cvs 同时也是不开放工程的一个通用选择。有些工程甚至使用 cvs 作为分发系统的一种方式。cvs 具有流行的开放源码工程通常都具有的好处：代码已经彻底复审过，它可用于我们的复审和修改，并且错误会被迅速修正。

cvs 将我们的代码保存在一个服务器上的仓库（Repository）中。这个服务器可能在本地

的局域网内，但是通常情况是通过 Internet 访问它的，这样团队中的人不需要处在特定位置就可以获得更新。为了连接到 cvs，我们必须分配用户名和密码，因此它具有适度的安全性；想得到更高的安全性，我们可以使用 **ssh** 协议（尽管这些是 Linux 工具，但是通过使用 Cygwin——见 www.cygwin.com，可以很容易地在 Windows 环境下获得）。一些图形化的开发环境（像免费的 Eclipse 编辑器；见 www.eclipse.org）提供了优秀的 cvs 集成环境。

一旦系统管理员初始化了这个仓库，那么团队成员就可以通过对它进行检出（check out）来获取代码树的备份。例如，一旦我们的机器登录到恰当的 cvs 服务器（这里忽略具体细节），我们就可以使用像这样的命令行来执行初始初始的检出：

```
cvsv -z5 co TIJ3
```

这条命令将与 cvs 服务器连接，并且处理称作 **TIJ3** 的代码仓库的检出（**'co'**）。**'-z5'** 参数告诉 cvs 程序在通信的两端都使用级别为 5 的 gzip 压缩来加速在网络上的传输。

一旦完成这个命令，在我们自己的本地机器上，将会有一份该代码仓库的副本。另外，我们将会看到该仓库中的每个代码目录下都有一个新添加的、名为 cvs 的子目录，它存储着有关代码目录下的文件的所有 cvs 信息。

现在我们就拥有了自己的 cvs 仓库副本，我们就可以修改这些文件去进行工程开发了。通常，这些修改包括所作的修正和增加的特征，以及编译和运行测试所必需的测试代码和修改过的构建文件。我们将会发现检入（check in）那些不能成功运行其所有测试的代码是很不受大家欢迎的，因为这样做之后团队的其他所有人都将得到这些不完整的代码（因此导致他们的构建失败）。

当我们已经对我们的代码做过改进，并且准备检入它们的时候，我们必须经过两步处理，这是 cvs 代码同步的关键所在。首先，我们更新我们的本地仓库使它的主 cvs 仓库同步，这是通过进入我们本地的代码仓库的根目录并且运行下面这条命令来实现的：

```
cvsv update -dP
```

此时，我们不需要再次登录，因为 cvs 子目录保留了登录远程仓库的信息，而且远程仓库也保留了关于我们机器的签名信息，它们被用来进行验证我们身份的双向检查。

'-dP' 标志是可选的，**'-d'** 告知 cvs 在本地机器上创建所有可能已经添加到主仓库中的新目录，**'-P'** 则告知 cvs 在本地机器上剪除所有已经在主仓库中被清空的目录。在缺省情况下，这两项操作都不会被执行。

然而，**update** 的主要行为却十分有趣。我们实际上应该周期性有规律地运行 **update**，而不仅仅是在我们检入之前，因为它将我们的本地仓库与主仓库进行同步。如果它发现任何在主仓库中的文件比我们的本地仓库上的文件要新，那么它就会将这种变化引入到我们的本地机器上。不过，它不仅仅只是复制这些文件，而是逐行地比较这些文件，并且将这些变化从主仓库中补缀到我们的本地版本中。如果我们已经对一个文件进行了修改，而且其他人也已

经对同一文件进行了修改，那么只要改变不同时发生在代码的同一行（**cv**s 匹配的是各行的内容，而不仅仅是行号，因此，即使行号变化了，它也能够正确同步），那么 **cv**s 将会把这些改变补缀到一块。因此，我们可以与其他人同时处理同一文件，而且，当我们执行 **update** 时，任何其他入提交给主仓库的修改都会与我们所做的修改合并在一起。

当然，可能两个人会对同一文件的同一行进行修改。这是由于缺乏交流而导致的一种意外；通常地，我们会互相告知我们正在处理的是什么，以防互相覆盖了彼此的代码（同样地，如果文件很大，以至于有必要让两个不同的人处理同一文件的不同部分，那么为了更容易地进行工程管理，我们可以考虑将大文件分解为更小的文件）。如果发生了这种情形，**cv**s 仅仅提示冲突并且通过锁定冲突行方式来强制我们解决这个问题。

注意在一次 **update** 期间，没有任何来自于我们机器的文件被移入主仓库。**Update** 只是从主仓库中将修改过的文件拷贝到我们的机器上，然后将我们所作的任何修改中补缀到其中。那么我们的修改怎样才能进入主仓库呢？这就是第二步：提交（**commit**）。

当我们键入

```
cv s commit
```

cvs 将会启动缺省的编辑器并且要求编写对我们所做修改的描述。此描述将会加入到仓库中，以便让其他人了解到什么被修变了。在这以后，我们修改过的文件将会被放入到主仓库中，因此其他所有人在他们下次进行 **update** 时可以得到这些修改过的文件。

cvs 还有其他一些功能，但是大多数时间我们所做的都是检出、更新以及提交。关于 **cv**s 的更多细节，可以从书上得到，**cv**s 的主网站 www.cvshome.org 上也有其完整的文档。另外，我们还可以到 Internet 上用 Google 或其他搜索引擎进行搜索；这里有一些对 **cv**s 很棒的精简介绍，它们可以使我们在起步阶段不致于因过多的细节而陷入困境（由 Daniel Robbins 编的“Gentoo Linux CVS Tutorial”尤为简单易懂）。

每日构建

通过将编译和测试合并到我们的构建文件中，我们可以进行由极限编程人员和其他一些人所提倡的“执行每日构建（*daily build*）”的实践了。无论我们当前已经实现了多少系统特性，我们总是要保持系统处于一个可以被正确构建的状态，以便在有人执行检出操作并且运行 **Ant** 时，构建文件都可以无误地执行所有的编译并无误地运行所有的测试。

这是一项强大的技术。这意味着我们总是有一个作为基线的系统来编译并通过其所有测试。在任何时刻，通过检查在运行系统中实际被实现的特性，我们总是可以观察到开发过程的真实状态。这种方法节省时间的因素之一是：没有人须要浪费时间去产生一个报告用来解释系统正处于什么状态；每个人都可以通过检出当前的构建并运行程序来亲眼看看。

如果某人（我们假定意外地）检入了会引起测试失败的修改，那么每日（或者更频繁地）构建同样能确保在这些bug有机会在系统中引起更大的问题之前，我们就可以迅速地了解到这

一点。Ant 甚至还有一个发送email的任务 (task)，因为很多团队将他们自己的构建文件设置成为一个**cron**¹²任务来每天甚至一天几次地自动运行，而且如果失败则发送email。这里还有一个开放源码的工具，可以自动执行构建，而且提供了一个显示工程状态的网页；详见<http://cruisecontrol.sourceforge.net>。

记录日志

记录日志是报告有关运行程序信息的处理过程。在被调试程序中，这些信息可以是普通的、描述程序进度的状态数据（例如，如果我们有一个安装程序，我们可能将在安装期间所采取的步骤、我们存储文件的目录以及程序的启动值等都记录到日志中。）。

记录日志在调试期间也很有用。没有记录日志，我们可能会设法通过插入 `println()` 语句来解释程序的行为。本书中很多示例使用正是这种技术，而且在缺少调试器时（该主题将会很快予以介绍），它就是我们所能利用的全部。但是，一旦我们判定程序正在正确运行，我们可能会把 `println()` 语句去掉。这之后，如果我们运行出来了更多的 bug，那么我们可能需要将这些 `println()` 语句放回原处。如果我们能够插入某种输出语句，它们只能在需要时被使用，那该有多好呀。

在可以在 JDK1.4 中获得记录日志 API 之前，程序员通常会使用这样一项技术：该技术依赖于 Java 编译器将会优化掉那些从不会被调用的代码这个事实。如果 `debug` 是 `static final boolean`，而且我们声明：

```
if(debug) {
    System.out.println("Debug info");
}
```

那么当 `debug` 的值是 `false` 时，编译器将彻底移除括号里面的代码（因此这些代码在不会被用到的时候，根本就不会引起任何运行时的开销）。使用这项技术，我们可以在程序各处放置追踪代码，而且很容易地打开和关闭它。不过，这种技术的一个缺点是：为了打开或关闭追踪语句，我们必须重编译代码。但是，如果通过使用我们可以修改的配置文件（修改其记录日志属性），就能够打开追踪而不需要重新编译程序，那就会更方便了。

JDK1.4 中记录日志 API 提供了一种更成熟完善的工具，它可以被用来报告关于我们程序的信息，并且在技术上，它具有与前面的例子几乎相同的功效。对于非常简单的信息性的记录日志，我们可以像下面这样做：

```
//: c15:InfoLogging.java
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
import java.io.*;
```

¹² **Cron**是在Unix下开发的、按照指定次数运行程序的一个程序。不过，在Windows下也可以得到其免费版本，而且它被当为了一项Windows NT/2000 的服务：[http:// www.kalab.com/freeware/cron/ cron.htm](http://www.kalab.com/freeware/cron/cron.htm) 。

```

public class InfoLogging {
    private static Test monitor = new Test();
    private static Logger logger =
        Logger.getLogger("InfoLogging");
    public static void main(String[] args) {
        logger.info("Logging an INFO-level message");
        monitor.expect(new String[] {
            "%% .* InfoLogging main",
            "INFO: Logging an INFO-level message"
        });
    }
} ///:~

```

在某次运行上面程序时所产生的输出如下：

```

Jul 7, 2002 6:59:46 PM InfoLogging main
INFO: Logging an INFO-level message

```

注意记录日志系统已经探测到了产生日志的类名和方法名。它不能保证这些名字都是正确的，因此我们不应该依赖于它们所能提供的精确性。如果我们想确保打印出正确的类名和方法，那么我们就可以使用更复杂的方法将消息记录到日志中，像下面这样：

```

///: c15:InfoLogging2.java
// Guaranteeing proper class and method names
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
import java.io.*;

public class InfoLogging2 {
    private static Test monitor = new Test();
    private static Logger logger =
        Logger.getLogger("InfoLogging2");
    public static void main(String[] args) {
        logger.logp(Level.INFO, "InfoLogging2", "main",
            "Logging an INFO-level message");
        monitor.expect(new String[] {
            "%% .* InfoLogging2 main",
            "INFO: Logging an INFO-level message"
        });
    }
} ///:~

```

logp()方法接受记录日志级别（下面我们将学习到）、类名、方法名以及要记入日志的字

串作为其参数。在记录日志是非临界的情况下，我们可以看到如果报送了类名和方法名，那么只依赖于自动方法会显得更简单。

记录日志级别

记录日志 API 提供了多个报告级别，以及在程序执行期间改变报告级别的能力。因此，我们可以将记录日志级别动态地设置为下面的任何状态：

级别	作用	数值
OFF	不报告任何日志消息。	Integer.MAX_VALUE
SEVERE	只报告 SEVERE 级别上的日志消息。	1000
WARNING	报告 WARNING 和 SEVERE 级别上的日志消息。	900
INFO	报告 INFO 及以上级别的日志消息。	800
CONFIG	报告 CONFIG 及以上级别的日志消息。 .	700
FINE	报告 FINE 及以上级别的日志消息。	500
FINER	报告 FINER 及以上级别的日志消息。	400
FINEST	报告 FINEST 及以上级别的日志消息。	300
ALL	报告所有的日志消息。	Integer.MIN_VALUE

我们甚至可以继承 `java.util.Logging.Level`（它有 `protected` 的构造器），来定义我们自己的级别。例如，可以有一个小于 300 的值，因此这个级别低于 `FINEST`。那么当级别是 `FINEST` 时，我们新级别上的日志消息就不会出现。

在下面的例子中，我们可以看到不同级别日志试验的效果：

```
//: c15:LoggingLevels.java
import com.bruceeckel.simpletest.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.Handler;
import java.util.logging.LogManager;

public class LoggingLevels {
```

```

private static Test monitor = new Test();
private static Logger
    lgr = Logger.getLogger("com"),
    lgr2 = Logger.getLogger("com.bruceeckel"),
    util = Logger.getLogger("com.bruceeckel.util"),
    test = Logger.getLogger("com.bruceeckel.test"),
    rand = Logger.getLogger("random");
private static void logMessages() {
    lgr.info("com : info");
    lgr2.info("com.bruceeckel : info");
    util.info("util : info");
    test.severe("test : severe");
    rand.info("random : info");
}
public static void main(String[] args) {
    lgr.setLevel(Level.SEVERE);
    System.out.println("com level: SEVERE");
    logMessages();
    util.setLevel(Level.FINEST);
    test.setLevel(Level.FINEST);
    rand.setLevel(Level.FINEST);
    System.out.println("individual loggers set to FINEST");
    logMessages();
    lgr.setLevel(Level.SEVERE);
    System.out.println("com level: SEVERE");
    logMessages();
    monitor.expect("LoggingLevels.out");
}
} ///:~

```

main()开始的几行还是必要的，因为要报告的日志消息的缺省级别是 **INFO** 及更高级的别（更严格）。如果我们不改变这一点，那么 **CONFIG** 及其以下级别的消息将不会被报告（试试去掉这几行看看会发生什么）。

在我们的程序中，可以有多个日志记录器对象，而且这些日志记录器被组织成一个层次树，可以通过编程方式将它和包的名字空间关联起来。子日志记录器紧跟着它们的直接父日志记录器，而且缺省地将日志的记录向上传递给父日志记录器。

“根”日志记录器对象总是缺省创建的，而且它是日志记录器对象树的基础。我们通过调用静态方法 **Logger.getLogger(“”)** 获得指向根日志记录器的引用。注意它接受一个空字符串而不是不带任何参数。

每个**Logger**对象可以有一到多个与之相关联的**Handler**对象。每个**Handler**对象提供一种

用于发布日志消息的策略（strategy）¹³，该消息包含在**LogRecord**对象中。为了创建一种新类型的**Handler**，我们仅仅从**Handler**类继承，重载其**publish()**方法（连同**flush()**和**e()**一起，来处理我们在**Handler**中可能使用到的所有流）。

根日志记录器缺省地总是与一个 **handler** 相关联，将输出发送到控制台。为了访问到 **handler**，我们可以在日志记录器对象之上调用 **getHandlers()**。在前面的例子中，我们知道只存在一个 **handler**，因此，在技术上我们不必迭代整个列表，不过一般情况下这样做更安全，因为其他人可能会向根日志记录器添加其它的 **handler**。每个 **handler** 的缺省级别都是 **INFO**，因此为了看到所有的消息，我们将级别设为 **ALL**（与 **FINEST** 相同）。

levels 数组使得我们可以很容易地测试所有的 **Level** 值。每个日志记录器都被设置了一个值，并且所有不同的记录日志级别都被测试了。在输出结果中，我们可以看到只有在当前选择的记录日志级别上的消息以及那些更严格的消息才会被报告。

日志记录

LogRecord是一个信使（Messenger）对象¹⁴，它的任务仅仅是将信息从一个地方传送到另一个地方。**LogRecord**中的所有方法都是获取器（getter）和设置器（setter），下面这个例子使用获取器方法卸出了所有存储在**LogRecord**中的信息。

```
//: c15:PrintableLogRecord.java
// Override LogRecord toString()
import com.bruceeckel.simpletest.*;
import java.util.ResourceBundle;
import java.util.logging.*;

public class PrintableLogRecord extends LogRecord {
    private static Test monitor = new Test();
    public PrintableLogRecord(Level level, String str) {
        super(level, str);
    }
    public String toString() {
        String result = "Level<" + getLevel() + ">\n"
            + "LoggerName<" + getLoggerName() + ">\n"
            + "Message<" + getMessage() + ">\n"
            + "CurrentMillis<" + getMillis() + ">\n"
            + "Params";
        Object[] objParams = getParameters();
        if(objParams == null)
```

¹³ 一种可插拔的算法。策略让我们可以很容易地改变一个解决方案中的一部分，而余下部分不变。它们通常作为允许客户端程序员提供解决特定问题所必需的一部分代码的方式来使用（如本例中）。更多细节，见www.BruceEckel.com站点上的Thinking in Patterns (in Java)。

¹⁴ 是Bill Venners制造出的一个术语。这可能是也可能不是一种设计模式。


```

        result += "<null>\n";
    else
        for(int i = 0; i < objParams.length; i++)
            result += "  Param # <" + i + " value " +
                objParams[i].toString() + ">\n";
    result += "ResourceBundle<" + getResourceBundle()
        + ">\nResourceBundleName<" + getResourceBundleName()
        + ">\nSequenceNumber<" + getSequenceNumber()
        + ">\nSourceClassName<" + getSourceClassName()
        + ">\nSourceMethodName<" + getSourceMethodName()
        + ">\nThread Id<" + getThreadID()
        + ">\nThrown<" + getThrown() + ">";
    return result;
}

public static void main(String[] args) {
    PrintableLogRecord logRecord = new PrintableLogRecord(
        Level.FINEST, "Simple Log Record");
    System.out.println(logRecord);
    monitor.expect(new String[] {
        "Level<FINEST>",
        "LoggerName<null>",
        "Message<Simple Log Record>",
        "% CurrentMillis<.+>",
        "Params<null>",
        "ResourceBundle<null>",
        "ResourceBundleName<null>",
        "SequenceNumber<0>",
        "SourceClassName<null>",
        "SourceMethodName<null>",
        "Thread Id<10>",
        "Thrown<null>"
    });
}
} ///:~

```

PrintableLogRecord 是对 **LogRecord** 的简单扩展，它重载了 **toString()** 用以调用所有在 **LogRecord** 中可以得到的获取器方法。

Handler

如前面提到的，通过继承 **Handler** 和定义 **publish()** 来执行我们所期望的操作，我们可以很容易地创建自己的处理器（handler）。不过，有一些事先定义好了的处理器，可能会满足我们的需要而不需要 we 做任何额外的工作。

StreamHandler	将格式化的记录写到 OutputStream 中
ConsoleHandler	将格式化的记录写到 System.err 中
FileHandler	将格式化的日志记录写到一个单一文件, 或者一个交替的记录文件集合中。
SocketHandler	将格式化的日志记录写到远程 TCP 端口中
MemoryHandler	在内存中缓存日志记录。

例如, 我们通常想将日志输出存储到一个文件中。**FileHandler** 使这变得很容易:

```
//: c15:LogToFile.java
// {Clean: LogToFile.xml,LogToFile.xml.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class LogToFile {
    private static Test monitor = new Test();
    private static Logger logger =
        Logger.getLogger("LogToFile");
    public static void main(String[] args) throws Exception {
        logger.addHandler(new FileHandler("LogToFile.xml"));
        logger.info("A message logged to the file");
        monitor.expect(new String[] {
            "%% .* LogToFile main",
            "INFO: A message logged to the file"
        });
    }
} ///:~
```

当我们运行该程序时, 会注意到两件事情。首先, 即使我们正在把输出发送给一个文件, 我们仍会看到控制台输出。这是因为每条消息被转换成 **LogRecord**, 它首先由本地的 **logger** 对象使用, 这个对象将它传递给它自己的处理器。此时, **LogRecord** 被传递给拥有其自己的处理器的父对象。这个过程将持续直到达到根日志记录器为止。根日志记录器有一个缺省的 **ConsoleHandler**, 因此消息既在显示器上出现, 也在在日志文件中出现 (我们可以通过调用 **setUseParentHandlers(false)** 来关闭这个动作)。

我们将会注意到的第二件事情是日志文件的内容是 XML 格式的, 它看上去像下面这样:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2002-07-08T12:18:17</date>
```

```

<millis>1026152297750</millis>
<sequence>0</sequence>
<logger>LogToFile</logger>
<level>INFO</level>
<class>LogToFile</class>
<method>main</method>
<thread>10</thread>
<message>A message logged to the file</message>
</record>
</log>

```

FileHandler 的缺省输出格式是 XML。如果我们想改变这种格式，我们必须给处理器附加上一个不同的 **Formatter** 对象。一个 **SimpleFormatter** 被用于该文件，以使输出为纯文本格式。

```

//: c15:LogToFile2.java
// {Clean: LogToFile2.txt,LogToFile2.txt.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class LogToFile2 {
    private static Test monitor = new Test();
    private static Logger logger =
        Logger.getLogger("LogToFile2");
    public static void main(String[] args) throws Exception {
        FileHandler logFile= new FileHandler("LogToFile2.txt");
        logFile.setFormatter(new SimpleFormatter());
        logger.addHandler(logFile);
        logger.info("A message logged to the file");
        monitor.expect(new String[] {
            "%% .* LogToFile2 main",
            "INFO: A message logged to the file"
        });
    }
} ////:~

```

LogToFile2.txt 文件的内容像下面这样：

```

Jul 8, 2002 12:35:17 PM LogToFile2 main
INFO: A message logged to the file

```

多重处理器

我们可以用每个 **Logger** 对象注册多个处理器。当日志请求到达 **Logger** 时，它告知所有向它注册的处理器¹⁵，只要 **Logger** 的日志级别高于或等于处理器的日志请求级别。这些处理器每一个依次拥有其自己的日志级别；如果 **LogRecord** 的级别高于或等于某个处理器的级别，那么该处理器就会发布报告。

下面例子向 **Logger** 对象添加了一个 **FileHandler** 和一个 **ConsoleHandler**:

```
//: c15:MultipleHandlers.java
// {Clean: MultipleHandlers.xml, MultipleHandlers.xml.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class MultipleHandlers {
    private static Test monitor = new Test();
    private static Logger logger =
        Logger.getLogger("MultipleHandlers");
    public static void main(String[] args) throws Exception {
        FileHandler logFile =
            new FileHandler("MultipleHandlers.xml");
        logger.addHandler(logFile);
        logger.addHandler(new ConsoleHandler());
        logger.warning("Output to multiple handlers");
        monitor.expect(new String[] {
            "%% .* MultipleHandlers main",
            "WARNING: Output to multiple handlers",
            "%% .* MultipleHandlers main",
            "WARNING: Output to multiple handlers"
        });
    }
} //:~
```

当我们运行这个程序时，我们会注意到控制台输出发生了两次；这是因为根日志记录器的缺省动作仍处于激活状态。如果想将其关闭，请调用 **setUseParentHandlers(false)**:

```
//: c15:MultipleHandlers2.java
// {Clean: MultipleHandlers2.xml, MultipleHandlers2.xml.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class MultipleHandlers2 {
```

¹⁵ 这是观察者设计模式（如前所述）

```

private static Test monitor = new Test();
private static Logger logger =
    Logger.getLogger("MultipleHandlers2");
public static void main(String[] args) throws Exception {
    FileHandler logFile =
        new FileHandler("MultipleHandlers2.xml");
    logger.addHandler(logFile);
    logger.addHandler(new ConsoleHandler());
    logger.setUseParentHandlers(false);
    logger.warning("Output to multiple handlers");
    monitor.expect(new String[] {
        "%% .* MultipleHandlers2 main",
        "WARNING: Output to multiple handlers"
    });
}
} ///:~

```

现在看到的只有一条控制台消息了。

编写自己的处理器

通过继承 **Handler** 类，我们可以很容易地编写定制的处理程序。为了能够实现此目的，我们不但要实现 **publish()** 方法（它执行实际上的报送），还要实现 **flush()** 和 **close()**，它们确保用于报送的流被完全清空。下面这个例子将来自于 **LogRecord** 的信息存储到了另一个对象中（一个字符串数组）。在这个程序的最后，将此对象打印输出到了控制台：

```

//: c15:CustomHandler.java
// How to write custom handler
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
import java.util.*;

public class CustomHandler {
    private static Test monitor = new Test();
    private static Logger logger =
        Logger.getLogger("CustomHandler");
    private static List strHolder = new ArrayList();
    public static void main(String[] args) {
        logger.addHandler(new Handler() {
            public void publish(LogRecord logRecord) {
                strHolder.add(logRecord.getLevel() + ":");
                strHolder.add(logRecord.getSourceClassName()+"");
                strHolder.add(logRecord.getSourceMethodName()+"");
            }
        });
    }
}

```

```

        strHolder.add("<" + logRecord.getMessage() + ">");
        strHolder.add("\n");
    }
    public void flush() {}
    public void close() {}
});
logger.warning("Logging Warning");
logger.info("Logging Info");
System.out.print(strHolder);
monitor.expect(new String[] {
    "%.* CustomHandler main",
    "WARNING: Logging Warning",
    "%.* CustomHandler main",
    "INFO: Logging Info",
    "[WARNING:, CustomHandler:, main:, " +
    "<Logging Warning>, ",
    ", INFO:, CustomHandler:, main:, <Logging Info>, ",
    "]"
});
}
} //::~

```

控制台输出来自于根日志记录器。当打印 **ArrayList** 时，我们可以看到仅有被选中的信息被捕获到了对象中。

过滤器

当我们要编写代码将日志记录消息发送给 **Logger** 对象时，那么就在编写代码的时候，我们通常就要决定日志记录消息应该是什么级别（日志记录 API 当然允许我们设计更复杂的系统，在其中消息级别可以动态决定，但是在实际中并不普遍）。**Logger** 对象有一个可以设置的级别，以便能够决定它可以接收什么级别的消息；而其它所有消息都将被忽略掉。这可以被看作是一种基本的过滤功能，但是通常这也就是我们所需要的全部了。

不过，有时我们需要更复杂的过滤，以便可以让我们基于其他某些原则而不是当前的级别来决定是接受还是拒绝一条消息。为了实现这一点，我们可以编写自定义的 **Filter** 对象。**Filter** 是一个接口，只包含一个单一方法：**boolean isLoggable(LogRecord record)**，用来决定这个特定的 **LogRecord** 是否能够引起足够注意将其报送。

一旦我们创建了一个 **Filter**，就可以通过使用 **setFilter()** 方法，将它注册到一个 **Logger** 或一个 **Handler** 上。在下面的例子中，假定我们把关于 **Duck** 的信息记录到日志：

```

//: c15:SimpleFilter.java

```

```

import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class SimpleFilter {
    private static Test monitor = new Test();
    private static Logger logger =
        Logger.getLogger("SimpleFilter");
    static class Duck {};
    static class Wombat {};
    static void sendLogMessages() {
        logger.log(Level.WARNING,
            "A duck in the house!", new Duck());
        logger.log(Level.WARNING,
            "A Wombat at large!", new Wombat());
    }
    public static void main(String[] args) {
        sendLogMessages();
        logger.setFilter(new Filter() {
            public boolean isLoggable(LogRecord record) {
                Object[] params = record.getParameters();
                if(params == null)
                    return true; // No parameters
                if(record.getParameters()[0] instanceof Duck)
                    return true; // Only log Ducks
                return false;
            }
        });
        logger.info("After setting filter..");
        sendLogMessages();
        monitor.expect(new String[] {
            "%.* SimpleFilter sendLogMessages",
            "WARNING: A duck in the house!",
            "%.* SimpleFilter sendLogMessages",
            "WARNING: A Wombat at large!",
            "%.* SimpleFilter main",
            "INFO: After setting filter..",
            "%.* SimpleFilter sendLogMessages",
            "WARNING: A duck in the house!"
        });
    }
} ///:~

```

在设置 **Filter** 前，有关 **Duck** 和 **Wombat** 的消息都会被报送。**Filter** 是作为一个匿名内部类被创建的，用来检查 **LogRecord** 的参数，查看是否有一个 **Duck** 作为一个额外参数

被传递给了 `log()` 方法。如果是这样的话，它会返回 `true`，表明这个消息应该被处理。

注意 `getParameters()` 方法签名表明它将返回一个 `Object[]`。不过，如果没有额外参数传递给 `log()` 方法，`getParameters()` 将返回 `null`（违反了它的方法签名——这是一种不好的编程习惯）。因此我们必须要进行 `null` 检测，而不是假定返回一个数组（如所期望的）并检查它的长度是否为 0。如果我们不能这样正确地实现，那么对 `logger.info()` 的调用将会抛出异常。

格式器（Formatter）

Formatter 是一种向 **Handler** 的处理步骤中插入格式化操作的方式。如果我们一个 **Handler** 注册了一个 **Formatter** 对象。那么在通过 **Handler** 发布 **LogRecord** 之前，它首先会被送到 **Formatter**。在被格式化之后，**LogRecord** 被返回给 **Handler**，接着把它发布出去。

为了编写一个自定义的 **Formatter**，要扩展 **Formatter** 类并且重载 `format(LogRecord record)`。然后，通过使用 `setFormatter()` 调用将 **Formatter** 注册到 **Handler**，如在这里看到的那样：

```
//: c15:SimpleFormatterExample.java
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
import java.util.*;

public class SimpleFormatterExample {
    private static Test monitor = new Test();
    private static Logger logger =
        Logger.getLogger("SimpleFormatterExample");
    private static void logMessages() {
        logger.info("Line One");
        logger.info("Line Two");
    }
    public static void main(String[] args) {
        logger.setUseParentHandlers(false);
        Handler conHdlr = new ConsoleHandler();
        conHdlr.setFormatter(new Formatter() {
            public String format(LogRecord record) {
                return record.getLevel() + " : "
                    + record.getSourceClassName() + " -:- "
                    + record.getSourceMethodName() + " -:- "
                    + record.getMessage() + "\n";
            }
        });
    }
}
```



```

        logger.addHandler(conHdlr);
        logMessages();
        monitor.expect(new String[] {
            "INFO : SimpleFormatterExample -:- logMessages "
                + "-:- Line One",
            "INFO : SimpleFormatterExample -:- logMessages "
                + "-:- Line Two"
        });
    }
} ///:~

```

记住：像 **myLogger** 这样的日志记录器有一个从父记录器（在本例中是根日志记录器）那里获得的缺省处理器。这里，通过调用 **setUseParentHandlers(false)**，我们将缺省处理器关闭，然后添加一个控制台处理器以供使用。在 **setFormatter()** 语句中创建了一个作为内部匿名类的新的 **Formatter**。被重载的 **format()** 语句仅仅从 **LogRecord** 中抽取出一些信息，然后将其格式化为一个字符串。

例子：发送 email 来报告日志消息

我们实际上可以让一个记录日志的处理器发送给我们一封 email，以便对于重要问题，我们能够被自动地通知到。随后的例子中使用 JavaMail API 来开发了一个用户代理来发送 email。

JavaMail API 是一个类集合，作为与底层的邮件协议（IMAP，POP，SMTP）的接口。通过注册一个附加的用来发送 email 的 **Handler**，我们可以在运行的代码中设置一个基于某些异常条件的通告机制。

```

///: c15:EmailLogger.java
// {RunByHand} Must be connected to the Internet
// {Depends: mail.jar,activation.jar}
import java.util.logging.*;
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class EmailLogger {
    private static Logger logger =
        Logger.getLogger("EmailLogger");
    public static void main(String[] args) throws Exception {
        logger.setUseParentHandlers(false);
        Handler conHdlr = new ConsoleHandler();
        conHdlr.setFormatter(new Formatter() {

```

```

        public String format(LogRecord record) {
            return record.getLevel() + " : "
                + record.getSourceClassName() + ":"
                + record.getSourceMethodName() + ":"
                + record.getMessage() + "\n";
        }
    });
    logger.addHandler(conHdlr);
    logger.addHandler(
        new FileHandler("EmailLoggerOutput.xml"));
    logger.addHandler(new MailingHandler());
    logger.log(Level.INFO,
        "Testing Multiple Handlers", "SendMailTrue");
}
}

// A handler that sends mail messages
class MailingHandler extends Handler {
    public void publish(LogRecord record) {
        Object[] params = record.getParameters();
        if(params == null) return;
        // Send mail only if the parameter is true
        if(params[0].equals("SendMailTrue")) {
            new MailInfo("bruce@theunixman.com",
                new String[] { "bruce@theunixman.com" },
                "smtp.theunixman.com", "Test Subject",
                "Test Content").sendMail();
        }
    }
    public void close() {}
    public void flush() {}
}

class MailInfo {
    private String fromAddr;
    private String[] toAddr;
    private String serverAddr;
    private String subject;
    private String message;
    public MailInfo(String from, String[] to,
        String server, String subject, String message) {
        fromAddr = from;
        toAddr = to;
        serverAddr = server;
    }
}

```

```

        this.subject = subject;
        this.message = message;
    }
    public void sendMail() {
        try {
            Properties prop = new Properties();
            prop.put("mail.smtp.host", serverAddr);
            Session session =
                Session.getDefaultInstance(prop, null);
            session.setDebug(true);
            // Create a message
            Message mimeMsg = new MimeMessage(session);
            // Set the from and to address
            Address addressFrom = new InternetAddress(fromAddr);
            mimeMsg.setFrom(addressFrom);
            Address[] to = new InternetAddress[toAddr.length];
            for(int i = 0; i < toAddr.length; i++)
                to[i] = new InternetAddress(toAddr[i]);
            mimeMsg.setRecipients(Message.RecipientType.TO,to);
            mimeMsg.setSubject(subject);
            mimeMsg.setText(message);
            Transport.send(mimeMsg);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
} ///:~

```

MailingHandler 是一个注册到日志记录器的 **Handler**。为了发送 email，**MailingHandler** 使用了 **MailInfo** 对象。当一个记录日志消息伴随一个附加的参数 **"SendMailTrue"** 被发送时，**MailingHandler** 就发送 email。

MailInfo 对象包含了必需的状态信息，例如接收地址、发送地址以及发送 email 所必需的主题信息等。当实例化 **MailInfo** 对象时，这些状态信息通过构造器传递给了它。

为了发送 email，我们首先必须用简单邮件传输协议 (SMTP, Simple Mail Transfer protocol) 服务器建立一个 **Session**。它是通过传输在 **Properties** 对象中的服务器地址来实现的，该地址在一个名叫 **mail.smtp.host** 的属性中。我们通过调用 **Session.getDefaultInstance()** 建立一个会话，向它传送 **Properties** 对象作为第一个参数。第二个参数是能用来鉴别用户身份的 **Authenticator** 的一个实例。当传递给 **Authenticator** 一个 **null** 值时，特指无需任何鉴别。如果设置了 **Properties** 对象中的调试标记，关于 **SMTP** 服务器和程序之间的交流信息将会被打印出来。

MimeMessage 是 Internet email 消息（扩展了 **Message** 类）的一个抽象。它构造了

一条遵守 MIME (Multipurpose Internet Mail Extension, 多用途 Internet Mail 扩展) 格式的消息。通过向 **MimeMessage** 传递一个 **Session** 实例来构造它。我们通过创建一个 **InternetAddress** 类 (**Address** 的一个子类) 的实例来设置发送和接收地址。我们使用来自于抽象类 **Transport** 的静态调用 **Transport.send** 来发送消息。一个 **Transport** 的实现会使用一种具体的协议 (通常是 SMTP) 与服务器进行通信来发送消息。

通过名字空间控制记录日志级别

尽管不是强制的, 但是将要使用日志记录器的类名传递给记录器是明智的。这使得我们可以操纵在相同的包层次中 (按包目录结构的粒度) 的日志记录器组的记录日志级别。例如, 我们可以修改在 **com** 中的所有包的所有记录日志级别, 或者仅在 **com.bruceeckel** 中, 或者仅在 **com.bruceeckel.util** 中的所有包的所有记录日志级别, 如随后的例子所示:

```
//: c15:LoggingLevelManipulation.java
import com.bruceeckel.simpletest.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.Handler;
import java.util.logging.LogManager;

public class LoggingLevelManipulation {
    private static Test monitor = new Test();
    private static Logger
        lgr = Logger.getLogger("com"),
        lgr2 = Logger.getLogger("com.bruceeckel"),
        util = Logger.getLogger("com.bruceeckel.util"),
        test = Logger.getLogger("com.bruceeckel.test"),
        rand = Logger.getLogger("random");
    static void printLogMessages(Logger logger) {
        logger.finest(logger.getName() + " Finest");
        logger.finer(logger.getName() + " Finer");
        logger.fine(logger.getName() + " Fine");
        logger.config(logger.getName() + " Config");
        logger.info(logger.getName() + " Info");
        logger.warning(logger.getName() + " Warning");
        logger.severe(logger.getName() + " Severe");
    }
    static void logMessages() {
        printLogMessages(lgr);
        printLogMessages(lgr2);
        printLogMessages(util);
        printLogMessages(test);
        printLogMessages(rand);
    }
}
```

```

    }
    static void printLevels() {
        System.out.println(" -- printing levels -- "
            + lgr.getName() + " : " + lgr.getLevel()
            + " " + lgr2.getName() + " : " + lgr2.getLevel()
            + " " + util.getName() + " : " + util.getLevel()
            + " " + test.getName() + " : " + test.getLevel()
            + " " + rand.getName() + " : " + rand.getLevel());
    }
    public static void main(String[] args) {
        printLevels();
        lgr.setLevel(Level.SEVERE);
        printLevels();
        System.out.println("com level: SEVERE");
        logMessages();
        util.setLevel(Level.FINEST);
        test.setLevel(Level.FINEST);
        rand.setLevel(Level.FINEST);
        printLevels();
        System.out.println(
            "individual loggers set to FINEST");
        logMessages();
        lgr.setLevel(Level.FINEST);
        printLevels();
        System.out.println("com level: FINEST");
        logMessages();
        monitor.expect("LoggingLevelManipulation.out");
    }
} ///:~

```

正如我们在代码中看到的那样，如果我们传递一个代表名字空间的字符串给 `getLogger()`，作为结果的 **Logger** 将会控制那个名字空间的记录日志的精确级别；也就是，所有在那个名字空间内的所有包都将受到日志记录器精确级别的改变所带来的影响。

每个 **Logger** 都追踪其现有的祖先 **Logger**。如果一个子 **logger** 已经具有了一个记录日志级别的设置，那么它将使用该级别而不是其父 **logger** 的记录日志级别。一旦孩子具有它自己的记录日志级别，改变其父亲的记录日志级别不会影响到该孩子的记录日志级别。

尽管单个记录器的级别被设置为 **FINEST**，但是只有记录日志级别等于或者更严格于 **INFO** 的消息才会被打印出来，因为我们正在使用根日志记录器的 **ConsoleHandler**，它处于 **INFO** 级别。

当日志记录器 **com** 或 **com.bruceeckel** 的记录日志级别被改变时，**random** 的记录日志级别并没有受到影响，因为它没有处在同一名字空间内。

大型工程的记录日志实践

乍一看，Java 记录日志 API 对大多数编程问题来说，可能都显得过于工程化了。直到我们开始创建更大的工程时，这些额外的特性和性能才能派上用场。在这一节我们将会着眼于这些特性以及推荐的使用它们的方法。如果我们仅在较小的工程上使用日志记录，那么我们可能不需要用到这些特性。

配置文件

下面的文件展示了我们怎样在一个工程中使用属性文件来配置日志记录器：

```
//:! c15:log.prop
#### Configuration File ####
# Global Params
# Handlers installed for the root logger
handlers= java.util.logging.ConsoleHandler
java.util.logging.FileHandler
# Level for root logger—is used by any logger
# that does not have its level set
.level= FINEST
# Initialization class—the public default constructor
# of this class is called by the Logging framework
config = ConfigureLogging

# Configure FileHandler
# Logging file name - %u specifies unique
java.util.logging.FileHandler.pattern = java%g.log
# Write 100000 bytes before rotating this file
java.util.logging.FileHandler.limit = 100000
# Number of rotating files to be used
java.util.logging.FileHandler.count = 3
# Formatter to be used with this FileHandler
java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter

# Configure ConsoleHandler
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
java.util.logging.SimpleFormatter
```

```
# Set Logger Levels #
com.level=SEVERE
com.bruceeckel.level = FINEST
com.bruceeckel.util.level = INFO
com.bruceeckel.test.level = FINER
random.level= SEVERE
///  
~
```

配置文件允许我们将处理器与根日志记录器关联起来。属性 `handler` 指定用逗号隔开的、我们想注册到根日志记录器的处理器列表。这里，我们将 **FileHandler** 和 **ConsoleHandler** 注册到根日志记录器。`.level` 属性为日志记录器指定缺省级别。这个级别可用于所有根日志记录器的孩子日志记录器，而且这些孩子日志记录器没有指定它们自己的级别。注意，在没有属性文件时，缺省记录日志级别是 `INFO`。这是因为，缺少自定义的配置文件，虚拟机将使用来自于 `JAVA_HOME\jre\lib\logging.properties` 文件的配置。

交替日至文件

前面的配置文件产生了交替日志文件 (rotating log file)，用于防止任何日志文件变得太大。通过设置 **FileHandler.limit** 的值，我们就可以给定在下一个日志文件开始被填充之前，在一个日志文件中所允许的最大字节数。**FileHandler.count** 决定能够使用的交替日志文件的数目；这里展示的配置文件指定了 3 个文件。如果所有这 3 个文件都被填充到了最大值，那么第一个文件再重新开始被填充，并覆盖掉旧的内容。

另一个选择是，通过给 **FileHandler.count** 赋值为 1，可以将所有的输出都提交到一个单一文件中。（**FileHandler** 参数在 JDK 文档中有详细说明）。

为了让下面的程序使用前面的配置文件，我们必须在命令行指定参数 **java.util.logging.config.file**：

```
java -Djava.util.logging.config.file=log.prop ConfigureLogging
```

配置文件只能修改根日志记录器。如果我们想为其他的日志记录器添加过滤器和处理器，那么我们就必须在 Java 文件中编写代码去实现，正如在构造器中看到的那样：

```
///  
c15:ConfigureLogging.java  
// {JVMArgs: -Djava.util.logging.config.file=log.prop}  
// {Clean: java0.log,java0.log.lck}  
import com.bruceeckel.simpletest.*;  
import java.util.logging.*;
```

```

public class ConfigureLogging {
    private static Test monitor = new Test();
    static Logger lgr = Logger.getLogger("com"),
        lgr2 = Logger.getLogger("com.bruceeckel"),
        util = Logger.getLogger("com.bruceeckel.util"),
        test = Logger.getLogger("com.bruceeckel.test"),
        rand = Logger.getLogger("random");
    public ConfigureLogging() {
        /* Set Additional formatters, Filters and Handlers for
           the loggers here. You cannot specify the Handlers
           for loggers except the root logger from the
           configuration file. */
    }
    public static void main(String[] args) {
        sendLogMessages(lgr);
        sendLogMessages(lgr2);
        sendLogMessages(util);
        sendLogMessages(test);
        sendLogMessages(rand);
        monitor.expect("ConfigureLogging.out");
    }
    private static void sendLogMessages(Logger logger) {
        System.out.println(" Logger Name : "
            + logger.getName() + " Level: " + logger.getLevel());
        logger.finest("Finest");
        logger.finer("Finer");
        logger.fine("Fine");
        logger.config("Config");
        logger.info("Info");
        logger.warning("Warning");
        logger.severe("Severe");
    }
} ///:~

```

这个配置将使输出发送到程序执行目录下名为 **java0.log**、**java1.log** 和 **java2.log** 的文件中。

建议惯例

尽管不是强制的，但是我们通常应该考虑为每个类都使用一个日志记录器，以遵循将日志记录器名字设置为与类的完整标识名字相同的标准。正如前面展示的那样，因为具备基于名字空间来将日志记录打开和关闭的能力，所以这使得我们可以进行细粒度的日志记录控制。

如果我们没有为那个包中的单个类设置记录日志级别，那么这些单个类缺省地使用包设置的记录日志级别（假定我们根据它们的包和类来命名日志记录器）。

如果我們是在一个配置文件中控制记录日志级别，而不是在我们的代码中动态地进行改变，那么我们就可以在不需重新编译代码的情况下修改记录日志级别。当系统已经被部署时，我们并不总是选择重新编译；通常，我们只是将类文件传載到目标环境中。

有时，需要运行一些代码来完成初始化动作，例如向日志记录器添加 **Handler**、**Filter** 和 **Formatter**。这可以通过在属性文件中设置 **config** 属性来实现。我们可以有若干个类，它们的初始化都可以通过使用 **config** 属性来实现。这些类应该使用像下面这样的空格界定符来指定：

```
config = ConfigureLogging1 ConfigureLogging2 Bar Baz
```

按照这种方式指定的类将会被调用的是其缺省构造器。

小结

尽管已经相当全面地介绍了记录日志 API，但是仍然没有涉及所有方面。例如，我们没有探讨 **LogManager** 以及各种各样的内置处理器的细节，例如 **MemoryHandler**、**FileHandler**、**ConsoleHandler** 等。我们应该到 JDK 文档上查阅更多的细节。

调试

尽管恰当使用 **System.out** 语句或者日志记录信息可以产生对程序行为非常有价值的洞察¹⁶，但对于困难问题这种方法就变得笨拙而费时了。另外，我们可能需要比允许打印语句要更深入地洞察程序。正因为如此，我们需要调试器。

除了可以更快更容易地显示用打印语句产生的信息，调试器还可以设置断点（breakpoint），然后当程序到达这些断点时会停止运行。调试器还可以显示任何时刻的程序状态，查看我们感兴趣的变量值，逐行地运行程序，连接在远程运行的程序，以及其它更多的功能。尤其当我们开始创建较大的系统时（在这里 bug 可能更容易被隐藏起来），去熟悉调试器是很值得的。

使用 JDB 调试

Java 调试器（JDB，Java Debugger）是一个装載在 JDK 中的命令行调试器。根据 JDB

¹⁶ 我主要是通过打印信息来学习 C++ 的，因为在我学习的时候没有任何调试器可用。

的调试指令以及它的命令行界面，JDB 至少从概念上讲是 Gnu 调试器（Gnu Debug，GDB，是从最初的 Unix DB 中得到启发的）的后代。JDB 对于学习如何进行调试和执行较简单的调试任务来说很有用，而且有助于你去了解无论 JDK 安装在哪里 JDB 总是可用的。不过，对于较大的工程，我们可能希望能够使用一个图形化调试器，这在稍后会讲述。

假设我们已经编写了下面的程序：

```
//: c15:SimpleDebugging.java
// {ThrowsException}
public class SimpleDebugging {
    private static void foo1() {
        System.out.println("In foo1");
        foo2();
    }
    private static void foo2() {
        System.out.println("In foo2");
        foo3();
    }
    private static void foo3() {
        System.out.println("In foo3");
        int j = 1;
        j--;
        int i = 5 / j;
    }
    public static void main(String[] args) {
        foo1();
    }
} ///:~
```

如果我们看一下 `foo3()`，就会发现问题很明显：我们正在除 0。但是假设这部分代码隐藏在一个大的程序中（例如一个调用序列调用到了这里），而且我们不知道该从哪里开始去查找问题，那么会怎么样呢？。正如它所显示的那样，抛出的异常会为我们提供足够的信息来定位问题（这正是异常的好处之一）。不过，让我们仅仅假设一下，如果问题比这更困难，而且我们需要更进一步地训练它以获得比异常所能提供的还要多的消息，那么又会怎么样呢？

要运行 JDB，我们必须告诉编译器通过使用 `-g` 标志来编译 `SimpleDebugging.java` 文件以产生调试信息。

```
jdb SimpleDebugging
```

这会调出来 JDB 并且给我们一个命令提示符。我们可以通过在提示符处键入 `?` 查看可用的 JDB 命令列表。

下面是一个对交互调试的跟踪，展示了该怎样找出问题：

```
Initializing jdb ...
> catch Exception
```

>表明 JDB 正在等待一个命令，用户键入的命令用粗体显示。**catch Exception** 命令会在任何抛出异常的地方设置一个断点（不过，调试器无论如何都会停止，即使我们没有显式地给出这个注释——在 JDB 中，异常就是缺省的断点）。

```
Deferring exception catch Exception.
It will be set after the class is loaded.
> run
```

此刻，程序将会运行直到下一个断点，在上面这种情况下就是异常发生的位置。下面是运行 **run** 命令后的结果：

```
run SimpleDebugging
>
VM Started: In foo1
In foo2
In foo3
Exception occurred: java.lang.ArithmeticException
(uncaught)"thread=main", SimpleDebugging.foo3(), line=18
bci=15
18      int i = 5 / j;
```

程序运行直到第 18 行，在这里产生了异常，不过在到达异常时，JDB 并不退出。调试器还显示了引起意外的代码行。通过 **list** 命令，我们可以列出像下面所示的在程序源中执行被中止的位置：

```
main[1] list
14      private static void foo3() {
15          System.out.println("In foo3");
16          int j = 1;
17          j--;
18 =>     int i = 5 / j;
19      }
20
21      public static void main(String[] args) {
22          foo1();
23      }
```

列表中的指针 ("=>") 表明当前位置是执行将被恢复的地方。我们可能通过 **cont** (continue, 继续) 命令恢复执行。不过这样做将会使 JDB 在异常处退出，并打印堆栈追

踪信息。

locals 命令用来导出所有局部变量的值:

```
main[1] locals
Method arguments:
Local variables:
j = 0
```

我们可以看到就是 **j=0** 这个值引起了异常。

wherei 命令将压入到当前线程的方法栈中的堆栈帧 (stack frames) 打印出来:

```
main[1] wherei
[1] SimpleDebugging.foo3 (SimpleDebugging.java:18), pc = 15
[2] SimpleDebugging.foo2 (SimpleDebugging.java:11), pc = 8
[3] SimpleDebugging.foo1 (SimpleDebugging.java:6), pc = 8
[4] SimpleDebugging.main (SimpleDebugging.java:22), pc = 0
```

在 **wherei** 命令后面的每一行都代表了一个方法调用, 以及调用将要返回的位置 (由程序计数器 **pc** 的值指示)。这里的调用序列是 **main()**、**foo1()**、**foo2()** 和 **foo3()**。我们可以用 **pop** 命令将调用 **foo3()** 时被压入的堆栈帧弹出栈:

```
main[1] pop
main[1] wherei
[1] SimpleDebugging.foo2 (SimpleDebugging.java:11), pc = 8
[2] SimpleDebugging.foo1 (SimpleDebugging.java:6), pc = 8
[3] SimpleDebugging.main (SimpleDebugging.java:22), pc = 0
```

我们可以使用 **reenter** 命令, 使 JDB 再次步入到对 **foo3()** 的调用:

```
main[1] reenter
>
Step completed: "thread=main", SimpleDebugging.foo3(), line=15
bci=0
System.out.println("In foo3");
```

list 命令向我们展示从 **foo3()** 起始部分开始的执行序列:

```
main[1] list
11      foo3();
12      }
13
14      private static void foo3() {
```

```

15 =>    System.out.println("In foo3");
16      int j = 1;
17      j--;
18      int i = 5 / j;
19    }
20

```

JDB 还允许我们修改局部变量的值。通过改变 `j` 的值，我们可以避免在上次执行这部分程序时所引起的被零除问题。我们可以在调试器中直接这样做，因此我们可以继续调试程序而不需要回溯和改变源文件。在我们设置 `j` 的值之前，我们必须执行第 25 行，因为 `j` 是在那里被声明的。

```

main[1] step
> In foo3

Step completed: "thread=main", SimpleDebugging.foo3(), line=16
bci=8
16      int j = 1;

main[1] step
>
Step completed: "thread=main", SimpleDebugging.foo3(), line=17
bci=10
17      j--;

main[1] list
13
14      private static void foo3() {
15          System.out.println("In foo3");
16          int j = 1;
17 =>      j--;
18          int i = 5 / j;
19      }
20
21      public static void main(String[] args) {
22          foo1();

```

在这里，`j` 被定义而且我们可以设置它的值以避免异常。

```

main[1] set j=6
j=6 = 6
main[1] next
>

```

```

Step completed: "thread=main", SimpleDebugging.foo3(), line=18
bci=13
18      int i = 5 / j;
main[1] next
>
Step completed: "thread=main", SimpleDebugging.foo3(), line=19
bci=17
19      }
main[1] next
>
Step completed: "thread=main", SimpleDebugging.foo2(), line=12
bci=11
12      }
main[1] list
8
9      private static void foo2() {
10          System.out.println("In foo2");
11          foo3();
12 =>    }
13
14      private static void foo3() {
15          System.out.println("In foo3");
16          int j = 1;
17          j--;
main[1] next
>
Step completed: "thread=main", SimpleDebugging.foo1(), line=7
bci=11
7      }
main[1] list
3      public class SimpleDebugging {
4          private static void foo1() {
5              System.out.println("In foo1");
6              foo2();
7 =>      }
8
9          private static void foo2() {
10              System.out.println("In foo2");
11              foo3();
12          }
main[1] next
>
Step completed: "thread=main", SimpleDebugging.main(), line=23
bci=3

```

```

23      }

main[1] list
19      }
20
21      public static void main(String[] args) {
22          fool();
23 =>    }
24    } ///:~
main[1] next
>
The application exited

```

Next 命令每次执行一行。我们可以看到异常被避免了而且我们可以继续逐行地执行程序。使用 **list** 可以显示在程序中将要继续执行下去的位置。

图形化调试器

使用像 JDB 这样的命令行调试器很不方便。我们必须使用显式命令进行查看变量值的状态（局部变量，转存变量），列出源代码执行位置（列表），找出系统中的线程（线程），设置断点（断入方法内部，断在方法外部），等等诸如此类的操作。图形化的调试器使得我们不需要显式命令，通过使用几下点击就能完成这些事情，而且还可以查看正在被调试的程序细节。

因此，尽管我们可能想尝试使用 JDB 开始进行调试，不过我们会发现学习使用图形化调试器来快速跟踪到 bug 显得更有效率。在本书此版本的编写过程中，我们开始使用 IBM 的 *Eclipse* 编辑器和开发环境，它包含一个很好的 Java 图形化调试器。*Eclipse* 的设计与实现都很优良，而且我们可以从 www.Eclipse.org 免费下载它（这是一个免费的工具，不是实验版或共享软件——感谢 IBM 投入资金、时间和努力使它可以供每个人使用）。

其他的免费开发工具也有图形化的调试器，例如 Sun 的 *Netbeans* 和 Borland 的 *JBuilder* 免费版。

剖析和优化

“我们应该忽略较小的效率，在 97% 的时间里我们都应该说：不成熟的优化是万恶之源。”——*Donald Knuth*

尽管我们应该始终紧记这段话，尤其当我们发现自己正处在不成熟优化的危险境地时，但是有时候我们确实需要确定程序正在将它的所有时间花费到了什么地方，以此来看看我们是否可以提高这些部分的性能。

剖析器 (Profiler) 负责收集信息, 使得我们可以查看程序的那些部分使用了内存以及那些方法花费了最多的时间。某些剖析器甚至允许我们中止垃圾回收器以帮助确定内存分配模式。

剖析器还是一个可以在我们的程序中发现线程死锁的有用工具。

追踪内存消费

下面是剖析器为说明内存使用目的而能够展示的数据类型:

- 为某一特定类型分配的对象个数。
- 对象分配发生的地方。
- 在该类实例的分配中涉及到的方法。
- 闲散对象: 已分配的, 但是没有被使用过的, 也没有被垃圾回收的对象。这些对象会持续增加 JVM 堆的大小而且会表现为内存泄漏, 这可能会引起内存溢出的错误或者在垃圾回收器方面过多的开销。
- 过度分配的临时对象, 它们增加了垃圾收集器的工作量, 因此降低了应用的性能。
- 在释放那些添加到垃圾收集器但是没有被移除掉的实例时所导致的失败 (这是闲散对象的特殊情况)。

追踪 CPU 的使用

剖析器还能追踪到 CPU 在代码的不同部分花费了多少时间。它们能告诉我们:

- 方法被调用的次数。
- 每个方法占用 CPU 时间的百分比。如果某个方法调用了其他的方法, 那么剖析器可以告诉我们花费到这些其他方法上的时间总量。
- 每个方法花费的绝对时间, 包括它等待 I/O 的时间, 加锁时间等。这些时间依赖于系统的可用资源。

这样我们就能够确定代码的那些部分需要优化了。

覆盖测试

覆盖测试 (coverage testing) 可以显示在测试期间没有被执行的代码行。这样就会吸引我们注意那些没有被使用到的代码, 因此它们应该成为被移除或重构的候选对象。

为了获取对 `SimpleDebugging.java` 的覆盖测试, 我们可以使用命令:


```
java -Xrunjcov:type=M SimpleDebugging
```

实验一下，我们试着将没被执行的代码放入 `SimpleDebugging.java` 中（作这件事时，我们必须机智一点，因为编译器可以探测到不可到达的代码行）。

JVM 剖析接口

剖析代理（profiler agent）对那些它感兴趣的事件与 JVM 进行通信。Java 虚拟机的剖析接口支持以下事件：

- 进入和退出方法
- 分配、移动和释放一个对象
- 创建和删除一个堆区域
- 开始和结束一次垃圾收集循环
- 分配和释放一个 JNI 全局引用
- 分配和释放一个 JNI 弱全局引用
- 载入和卸载某个编译过的方法
- 开始和结束某个线程
- 为设备准备的类文件数据
- 载入和卸载某个类文件
- 处于竞争状态的 Java 监视器的事件：等待进入，进入及退出
- 处于竞争状态的 raw 监视器的事件：等待进入，进入及退出
- 未处于竞争状态的 Java 监视器的事件：等待和已完成等待
- 丢弃监视器
- 丢弃堆
- 丢弃对象
- 请求导空或复位剖析数据
- JVM 初始化和关闭

在进行剖析时，Java 虚拟机将这些事件发送到剖析代理，接着剖析代理将期望的信息传送给剖析器前端，如果需要的话，剖析器前端可以是在另一台机器上的运行进程。

使用 HPROF

这部分的一个实例展示了我们怎样才能够运行装载在 JDK 中的剖析器。尽管从剖析器而来的消息是以略显粗糙的文本文件形式表示的，而不是像一般的商业剖析器那样产生图形化的表示，但是在判定我们程序的特性方面，它仍然能够提供很有价值的帮助。

当我们调用程序时，通过向 Java 虚拟机传送一个额外参数来运行剖析器。这个参数必须是一个单一字符串，逗号后面没有任何空格，像这样（尽管它应该在一个单一的行中，但是在

书中被缠绕表示了，因为书页面不够宽)：

```
java
-Xrunhprof:heap=sites,cpu=samples,depth=10,monitor=y,thread=y,
doe=y ListPerformance
```

- **heap=sites** 告知剖析器编写在堆上的内存使用信息，指示被分配在什么地方。
- **cpu=samples** 告知剖析器进行统计抽样来确定 CPU 的使用情况。
- **depth=10** 指示线程追踪的深度。
- **thread=y** 告诉剖析器去标识在堆栈序列中的线程。
- **doe=y** 告知剖析器在退出时清空剖析数据。

下面的列表仅包含 HPROF 所产生的文件的一部分。输出文件被创建在当前目录下并且被命名为 **java.hprof.txt**。

java.hprof.txt 开始部分描述了文件中其余部分的细节。由剖析器产生的数据处于不同部分；例如，TRACE 表示文件中的追踪部分。我们将会看到许多 TRACE 部分，每个都编了号，以便可以在后面进行引用。

SITES 部分展示了内存分配的位置。这部分有几行，它们按照被分配和被引用的字节（活动着的字节）数排序。内存以字节列出。*Self* 列代表该位置占据内存的百分比，下一列 *accum*，代表累积的内存百分比。*live bytes* 和 *live objects* 列代表在该位置上的活动的字节数和所创建的、占用这些字节的对象个数。*allocated bytes* 和 *objects* 代表实例的对象总数和字节总数，包括那些正在被使用的和没有被使用的。在 *allocated* 和 *live* 中列出的字节数之差代表可以被垃圾收集的字节数。*Trace* 列实际上引用了文件中的一个 TRACE。第一行引用了下面显示的 668 追踪。*name* 代表被创建实例所属的类。

```
SITES BEGIN (ordered by live bytes) Thu Jul 18 11:23:06 2002
      percent      live      alloc'ed stack class
rank  self accum  bytes objs  bytes objs trace name
  1 59.10% 59.10% 573488   3 573488   3 668 java.lang.Object
  2  7.41% 66.50%  71880  543  72624  559   1 [C
  3  7.39% 73.89%  71728   3  82000  10 649 java.lang.Object
  4  5.14% 79.03%  49896  232  49896  232   1 [B
  5  2.53% 81.57%  24592  310  24592  310   1 [S
TRACE 668: (thread=1)
    java.util.Vector.ensureCapacityHelper(Vector.java:222)
    java.util.Vector.insertElementAt(Vector.java:564)
    java.util.Vector.add(Vector.java:779)
    java.util.AbstractList$Itr.add(AbstractList.java:495)
    ListPerformance$3.test(ListPerformance.java:40)
    ListPerformance.test(ListPerformance.java:63)
    ListPerformance.main(ListPerformance.java:93)
```

这个追踪展示了分配内存的方法调用序列。如果我们进入由行号所指示的追踪，我们将发现有一个对象分配动作发生在 **Vector.java** 的 222 行：

```
elementData = new Object[newCapacity];
```

这有助于我们发现程序中使用掉相当大的内存数量（在这种情形下是 59.10 %）的部分。

注意在位置 1 的 **[C** 表示基本数据类型 **char**。这是 Java 虚拟机中对基本数据类型的内部表示。

线程性能

CPU SAMPLES 部分展示了 CPU 的使用情况。下面是在这个部分中的一个追踪的一部分。

```
SITES END
CPU SAMPLES BEGIN (total = 514) Thu Jul 18 11:23:06 2002
rank  self  accum  count trace method
   1  28.21% 28.21%   145  662 java.util.AbstractList.iterator
   2  12.06% 40.27%    62  589 java.util.AbstractList.iterator
   3  10.12% 50.39%    52  632 java.util.LinkedList.listIterator
   4   7.00% 57.39%    36  231 java.io.FileInputStream.open
   5   5.64% 63.04%    29  605 ListPerformance$4.test
   6   3.70% 66.73%    19  636 java.util.LinkedList.addBefore
```

这个列表的组织方式相似于 SITES 列表的组织方式。这些行是按照 CPU 的使用情况分类的。最上面的行具有最大的 CPU 使用率，正如在 *self* 列指示的那样。*Accum* 列列出了累积的 CPU 利用率。*count* 域列出了这个追踪被激活的次数。接下来的两列列出了追踪号及此次调用的方法。

看一下 CPU SAMPLES 部分的第一行。在方法 **java.util.AbstractList.iterator()** 中使用的总的 CPU 时间为 28.12%，而且被调用了 145 次。该调用的细节可以在 662 号追踪中查看到：

```
TRACE 662: (thread=1)
    java.util.AbstractList.iterator(AbstractList.java:332)
    ListPerformance$2.test(ListPerformance.java:28)
    ListPerformance.test(ListPerformance.java:63)
    ListPerformance.main(ListPerformance.java:93)
```

我们可以推断在列表中进行迭代花费了相当多的时间。

对于大型工程，具有以图形化形式表示的信息通常会更有帮助。许多剖析器都可以产生图形化的显示，但是对这一点的论述不在本书的范围之内。

最优化指南

- 避免为性能而牺牲代码的可读性。
- 不能孤立地考虑性能。要权衡所需付出的努力与能够得到的利益之间的关系。
- 性能是大型工程要关心的问题，但是它通常不是小型工程要考虑的问题。
- 使程序可以运转应该比钻研程序的性能有更高的优先权。一旦我们拥有了可运转的程序，我们就可以使用剖析器来使其更为有效。仅当性能被确定为是一个关键因素的时候，在初始设计/开发过程期间才应该予以考虑。
- 不要假设瓶颈在什么地方。而是应该运行剖析器来获得数据。
- 在任何可能的情况下，尽量通过将对象设置为 `null` 从而显式地将其销毁。有时这可能是对垃圾回收器的一种很有帮助的提示。
- 程序大小的问题。仅当程序是大型的、运行时间长而且速度也是一个问题时，性能优化才有价值。
- **static final** 变量可以通过 Java 虚拟机进行优化以提高程序的速度。因此程序常量也应该被声明为 **static** 和 **final**。

Doclets

虽然为文档支持开发一种工具，作为帮助我们在程序中追踪问题的想法有一点不可思议，但是 doclets 就具备这种令人吃惊的用处。因为 doclet 探入了 Javadoc 解析器的内部，它能够得到 javadoc 解析器可以获得的信息。有了它，我们就可以编程检验代码中的类名、域名和方法签名，并且标记潜在的问题。

从 Java 源文件中产生 JDK 文档的过程涉及到使用标准的 doclet 进行解析源文件和格式化这个被解析文件的操作。我们可以编写定制的 doclet 来定制我们的 javadoc 注释的格式。但是，doclet 允许我们做的比对注释进行格式化要多得多，因为 doclet 可以获取被解析的源文件的很多信息。

我们可以提取类中所有的成员信息：域、构造器、方法以及与每个成员变量相关的注释（唉，方法体的代码无法获得）。关于成员变量的细节被封装到了特殊对象的内部，它包含关于成员变量的属性的信息（`private`、`static` 和 `final` 等）。这些信息对于探测编写得很拙劣的代码会有所帮助，这些代码包括应该是私有的但却是公有的成员变量，没有注释的方法参数以及没有遵守命名规则的标志符等等。

Javadoc 不能捕获所有的编译错误。它可以探查语法错误，例如不匹配的括号，但是它不能捕捉语义错误。最安全的方法是在试图使用基于 doclet 的工具之前，运行 Java 编译器。

由 Javadoc 提供的解析机制会解析整个源文件，并且将其存储到内存中的一个 **RootDoc** 类的对象中。提交到 Javadoc 的 doclet 的入口点是 **start(RootDoc doc)**。它等同于通常 Java 程序的 **main(String[] args)**。我们可以通过 **RootDoc** 对象抽取出必要的信息。下面的例子展示了怎样编写一个简单的 doclet；它只是打印出每个被解析类的全体

成员变量:

```
//: c15:PrintMembersDoclet.java
// Doclet that prints out all members of the class.
import com.sun.javadoc.*;

public class PrintMembersDoclet {
    public static boolean start(RootDoc root) {
        ClassDoc[] classes = root.classes();
        processClasses(classes);
        return true;
    }
    private static void processClasses(ClassDoc[] classes) {
        for(int i = 0; i < classes.length; i++) {
            processOneClass(classes[i]);
        }
    }
    private static void processOneClass(ClassDoc cls) {
        FieldDoc[] fd = cls.fields();
        for(int i = 0; i < fd.length; i++)
            processDocElement(fd[i]);
        ConstructorDoc[] cons = cls.constructors();
        for(int i = 0; i < cons.length; i++)
            processDocElement(cons[i]);
        MethodDoc[] md = cls.methods();
        for(int i = 0; i < md.length; i++)
            processDocElement(md[i]);
    }
    private static void processDocElement(Doc dc) {
        MemberDoc md = (MemberDoc)dc;
        System.out.print(md.modifiers());
        System.out.print(" " + md.name());
        if(md.isMethod())
            System.out.println("(" + md.parameters() + ")");
        else if(md.isConstructor())
            System.out.println();
    }
} ////:~
```

我们可以这样使用 doclet 来打印出成员变量:

```
javadoc -doclet PrintMembersDoclet -private PrintMembersDoclet.java
```

它将在这行调用命令的最后一个参数所指定的文件上调用 javadoc, 这意味着它将解析

`PrintMembersDoclet.java` 文件。`-doclet` 选项告诉 javadoc 使用自定义的 doclet——`PrintMembersDoclet`。`-private` 标记指示 javadoc 还要打印 `private` 成员变量（缺省情况是只打印 `protected` 和 `public` 类型的成员变量）。

`RootDoc` 包含一个 `ClassDoc` 的集合来保存有关这个类的所有信息。像 `MethodDoc`、`FieldDoc` 和 `ConstructorDoc` 这样的类分别包含关于方法、域和构造器的信息。`processOneClass()` 方法抽取这些成员的信息并将它们打印出来。

我们也可以创建 taglet，它允许我们实现自定义的 javadoc 标志。JDK 文档中展示了一个实现了 `@todo` 标记的例子，该标记在作为结果的 Javadoc 输出中以黄色显示它的文本。查询 JDK 文档中的“taglet”可以得到更多的细节。

总结

本章介绍了我已经开始意识到的可能是程序设计中已经取代了语言语法和设计问题而成为最重要话题的一个问题：怎样确保你的代码是正确的，并且怎样让它保持总是对的？

最近的实验表明，最有帮助和最实用的工具就是有些陈旧的单元测试，正如在本章中所展示的那样，它可以非常有效地和 DBC 结合起来。同样，还有一些其他类型的测试，例如验证我们的用况/用户要求已经全部实现的一致性测试。但是由于某种原因，我们会进入过去已经提交的将在后面由其他人再次去执行的测试。极限编程主张编写单元测试应该在编写代码之前；我们先为类创建测试框架，然后才是类本身（在偶尔的情况下，我成功地这样做过，但是我通常还是喜欢让测试出现在最初的编码处理过程中的某处）。也有一些人反对测试，通常是那些从未这样尝试过并且相信他们可以不用测试就能编写出良好代码的人。但是我实战经验越多，我就越反复地这样做：

如果它没有被测试过，那么它就是拙劣的。

这是一句很有价值的格言，尤其是在我们想抄近路的时候。我们越多地发现自己的 bug，附着在内置测试上的安全性的提高也就越多。

构建系统（尤其是 Ant）和版本控制（CVS）也在本章给予了介绍，因为它们为我们的工程及其测试提供了组织结构。对于我来说，极限编程的首要目标是速度——向前快速推进我们的工程的能力（不过要以可信赖的形式），而且当我们意识到它可以被提高改善时，可以快速对它进行重构。速度需要一个支持它的组织结构，当我们开始对我们的工程进行大的改动时，它可以给我们以信心。这包括一个可靠的仓库，它可以允许我们回退到先前的任何版本，以及一个自动构建系统，一旦经过配置，它能够确保工程可以在一个单一步骤中被编译和测试。

一旦我们有理由相信我们的程序是“健康”的，记录日志就可以提供一种可以用来监视它的“脉搏”的途径，而且如果有什么不对，它甚至可以（正如在本章看到的那样）自动向我们发送 email。在这么做时，调试和剖析可以帮助我们追踪 bug 及性能问题。

想获得一个简单的、清晰的和具体的答案可能是计算机编程的本质特性。毕竟，我们是在用一些零和一来执行操作，它们没有模糊的边界（它们实际上是有的，不过电子工程师已经尽一切可能的手段给我们提供我们想要的模型）。当提到解决方案时，应该坚信肯定存在一个答案。但是我发现任何技术都有边界，并且知道这些边界在什么地方比任何单一方法的能力都要更为强大得多，因为这样的话，对于一种方法，我们可以在最能体现它的强大能力的地方去使用它，而在它并不是那么强大的地方去和其它方法结合起来使用。例如，在本章中 DBC 是与白盒单元测试结合起来而出现的，正如我创建的例子那样，我发现两个一块使用比任何一个单独使用都要有用得多。

我还发现这种思想不仅在发现问题方面是正确的，而且也包括在处于首要地位的构建系统中。例如，使用单一编程语言或工具来解决我们的问题最吸引人的地方在于其一致性，但是我发现通常对工程的总的利益而言，我使用 Python 编程语言而不用 Java 来解决特定问题会更加迅速和有效。我们可能还会发现 Ant 适用于一些地方，而在其他地方 make 则更有用。或者，如果我们的客户端使用的是 Windows 平台，那么作出使用 Delphi 或 Visual BASIC 开发客户端程序的极端决定可能更为明智，因为这比我们用 Java 去实现要快得多的。重要的是要保持开放的头脑并且记住我们正在努力获得结果，没必要使用某一特定工具或技术。这样可能比较困难，但是如果想到工程的失败率很高而我们获得成功的比率很低，那么就应该让我们的头脑对那些可能更高产的解决方案更开放一些。在极限编程中我最喜欢的名句（并且是我发现我经常会因为一些愚蠢的原因而违背的一点）是“做可以运转的最简单的事物。”大多数情况下，最简单和最有用的方法，如果我们能够发现，那么它将会是最好的一个方法。

练习

1. 创建一个包含 **static** 语句的类，如果断言没有被激活，则该语句抛出异常。验证该测试运转正确。
2. 修改上一个作业，使用 **LoaderAssertions.java** 中的方法，来打开断言而不是抛出异常。验证该测试运转正确。
3. 在 **LoggingLevels.java** 中，注释掉设置根日志记录器的处理器的严格级别的代码，并且验证 **CONFIG** 级别及其以下的消息不予报送。
4. 从 **java.util.Logging.Level** 继承，并定义我们自己的、其值低于 **FINEST** 的级别。修改 **LoggingLevels.java**，使用我们自己的新级别；并且展示当日志记录级别是 **FINEST** 时，在我们自己级别上的消息不会显现。
5. 将 **FileHandler** 与根日志记录器关联起来。
6. 修改 **FileHandler**，可以将输出格式化到一个简单文本文件中。
7. 修改 **MultipleHandlers.java**，产生纯文本形式而不是 XML 形式的输出。
8. 修改 **LoggingLevels.java**，为那些和根日志记录器相关连的处理器设置不同的日志记录级别。
9. 编写一个简单程序，用来设置基于命令行参数的根日志记录器的日志记录级别。
10. 编写一个例子，运用 **Formatter** 和 **Handler** 将日志文件以 **HTML** 格式输出。
11. 编写一个例子，运用 **Formatter** 和 **Handler** 将任何高于 **INFO** 严格级别的消息记录到一个文件中，将任何等于和低于 **INFO** 严格级别的消息记录到另一个文件中。这些文件都应该被写为简单文本形式。
12. 修改 **log.prop** 类，添加另外一个初始化类，用来为 **com** 的日志记录器初始化定

制的 **Formatter**。

13. 在 **SimpleDebugging.java** 上运行 JDB，但是不给出 **catch Exception** 命令。证明它仍能捕获异常。
14. 给 **SimpleDebugging.java**（我们必须以编译器不捕获错误的方式去实现它！）添加一个未被初始化的引用，并使用 JDB 来跟踪问题。
15. 执行在“覆盖测试”部分中所描述的实验。
16. 创建一个 doclet，它通过检测大写字母是怎样被用于标识符的，来显示未遵循 Java 命名规则的标识符。

第十六章 分析与设计

面向对象是一种全新的编程思考方式。

第一次了解如何处理一个 OOP 工程时，多数人都会感到困难。现在你已经理解了对象的概念，那么在学会以面向对象的方式思考问题之后，你就能够充分利用 OOP 带来的所有优势，创造出“好”的设计了。本章将介绍分析与设计的思想，以及一些方法，以帮助你在适度的时间内开发出良好的面向对象的程序。

方法学

方法学（methodology，有时直接称之为“方法（method）”）是一套用以降解编程问题复杂性的过程与启发。自从面向对象编程面世以来，已经系统地提出了许多种 OOP 方法论。本节将让你体会在采用某种方法论时你将能够达到的目标。

方法论是由许多经验积累而成的，特别是在 OOP 中尤为如此。所以，在采用某种方法论之前，弄清楚它能解决什么问题就显得十分重要。对于 Java 而言，这尤其正确，Java（与 C 相比）是用来降低程序表达时的复杂度的编程语言。这实际上降低了对复杂的方法论的需求程度。对比过程型编程语言，在使用 Java 时，简单的方法论就足以解决大多数问题。

正确理解“方法论”这个术语也很重要，它通常显得太隆重了，并且总是给人以太多的希望。其实设计与编程时所做的事情，就是方法论。它可能是你自己的方法论，也许你根本就是无意识的，但是你所经历的处理过程就是你创建的方法论。如果它很有效，那么只需要对它作小小的调整，就能将它应用于 Java。但如果你对自己的生产率与程序的开发方式不满意，那么你就应该考虑采用某种形式化的方法论，或者从多种形式化的方法论中选择多个部分加以应用。

在你经历整个开发过程时，最重要的一点是：不要迷失。这很容易做到。大多数分析与设计方法论都试图去解决最大型的问题。但是请记住，多数工程并属于这一范畴，我们通常只需采用某种方法论所推荐的一个相对较小的子集就可以得到成功的分析和设计¹。但是某些类型的处理过程，无论它们怎样小或者怎样受限制，通常也都应该采用某种比直接开始编码要好得多的方式。

这很容易使人迷失，从而陷入“分析瘫痪”，使你感觉无法取得进展，因为你无法确定当前阶段的每一个细节。记住，无论你了做多少分析，总有些问题不到设计阶段是无法发现的，而更多的问题直到编码阶段，或者程序运行时才会暴露出来。因此，快速地完成分析与设计阶段，并且对开发的系统进行测试才是至关重要的。

这一点很值得强调。过去使用的是过程型编程语言，在设计与实现某个系统之前，开发团队会仔细地进行分析，理解每一个细节。当然，开发数据库管理系统(DBMS)时，必须彻底

¹ Martin Fowler 的《UML Distilled, 2nd edition》(Addison-Wesley 2000)中有极佳的例子，可以说明如何将有时难以掌控的 UML 处理过程降低为可管理的子集。

理解用户的需求。但是DBMS这一类问题，我们已经描述与理解得很好了。在许多这样的程序中，数据库的结构就是问题的核心。本章讨论的编程问题，我称之为“外卡(wild-card)”的多样性。要解决此类问题，可不是简单地修改某个著名的解决方案就可以的。其中包含一个或多个“外卡因素”，这些因素并没有已知的很好的解法，因此对它们的研究就很有必要²。如果在进入设计与实现阶段之前，你试图彻底分析外卡问题，那么其结果只能陷入分析瘫痪。因为在分析阶段，你没有足够的信息来解决此类问题。解决这样的问题需要迭代重复整个过程，需要采取冒险的行为（这是有意义的，因为你正在尝试解决一些新问题，它们具有更高的潜在回报）。匆忙地进行初步开发，看起来似乎增加了风险，但是对于外卡型项目，却是降低了它们的风险，因为你可以尽早地发现某种方法是否可行。产品开发其实就是风险管理。

我们常常听人建议“构建一个，然后扔掉”。使用 OOP，你可能仍需丢弃一部分，但是由于代码以类的形式进行了封装，所以在第一次开发过程中肯定会产生一些有用的类，以及一些对于系统设计有价值的想法，这些可不该被丢弃掉。因此，第一次快速开发，不仅生成了对下一次分析、设计与开发都有用的信息，同时也生成了代码的根基。

也就是说，如果你选择的方法论包含大量的细节，并且建议采用多个步骤与文档，那么就仍然难以了解何时应该停止。请铭记于心，你要尝试去发现的是：

1. 要使用对象是什么？（如何将项目划分为各个构件？）
2. 这些对象的接口是什么？（你需要发送什么消息给每个对象？）

如果确定了对象和接口，那么就可以开始编写程序了。由于各种各样的原因，你可能还需要更多的描述与文档才能开始编程，但是这两者缺一不可。

这个过程可以分为 5 个阶段，以及仅仅是对所采用的结构做最初的确定的阶段 0。

阶段 0：制定计划

首先你得决定，在开发过程中需要哪些步骤。听起来这很简单（事实上，所有的事情听起来都很简单），然而在开始编码之前，人们经常无法做出这样的决定。如果你的计划是“直接开始编码”，那也可以（当你对问题很清楚时，这样的决定也是合适的），至少这也算是有计划。

在这一阶段，你可能还要决定采用某些必要的附加处理结构，但这并不是完整的结构。有些程序员喜欢以“度假的方式”工作，他们对开发过程不限制任何结构，这是可以理解的。“能完成的时候，它自然就完成了。”这自然讨人喜欢，但是我发现，如果在开发过程中划分一些里程碑，将有助于集中精力。并且每达到一个里程碑，都是对你所获得的一点成就的激励，使你不至于被“完成整个项目”这样一个单一的目标给弄得晕头转向。此外，将项目划分为许多可以一口吃掉的小块，也可以使得项目不那么吓人（增加里程碑提供了更多庆祝的机会）。

² 我评估此类项目的首要原则是：如果存在不止一个外卡，那么在建立能够运作的原型之前，不要急着计划整个项目会花费多少时间，耗用多少成本。因为它有太多的自由度。

在我开始研究小说结构的时候（因为我希望某天我能写一本小说），我对于规划小说结构的思想有些抗拒，我觉得，直接把故事写到纸上就是最好的方法。但是，后来我认识到，当我撰写有关计算机的文章时，由于我已经很清楚它的结构了，所以才不需要考虑很多。但我还是将我的工作结构化了，虽然只是出于半自觉状态。所以，即便你的计划是直接开始编码，其实在对某些问题的问答之间，你已经不知不觉地经历了此阶段。

任务描述

开发任何系统，无论多复杂，都有一个基本目的：它所处的行业、它所需要满足的基本需求。只有抛开用户界面、与硬件相关或系统相关的细节、算法、效率等问题，你才能找出简单而直接的系统核心。就像好莱坞电影的“中心思想”，只用一两句话来描述。这种纯粹的描述就是项目的起点。

“中心思想”很重要，它为项目设定了基调，也就是任务描述。不必强求第一次就要描述得正确无误（在完全将问题弄清楚之前，你可能已经进入项目的下一阶段了），但是应该不断努力，直到你感觉正确了为止。例如对于一个空中交通控制系统，你可能将“中心思想”锁定在你要开发的系统本身：“航管塔台记录飞机行踪”。但是当你将系统缩小，应用到很小的机场时，会发生什么呢？也许那里只有人工控制，根本没有航管塔台。有一种更为实用的模型，它并不关心你的解决方案，只关心问题的描述：“飞机抵达、卸客、检修、登机、然后离港。”

阶段 1：做什么？

在上一代的程序设计中（所谓过程型设计 *procedural design*），将此阶段称为“生成需求分析与系统说明书”。这正是令人不知所措的地方。那些名头吓人的文档，本身就是一项大工程。当然，这些文档的用意是好的。需求分析就是“列出指导方针，使我们知道项目何时完成，并且使用户满意”³。系统说明书就是要“描述程序需要做什么（不是如何做）才能满足需求”。需求分析是你与客户之间真正的契约（即使客户就是你公司的一员，或者是别的某个对象或系统）。系统说明书是从顶层探讨问题，在某种意义上，它说明此项目是否可以做到，要多久。这两件工作需要开发人员意见一致（因为他们的意见经常会随着时间而改变），所以我认为它们越少越好，从而可以节约时间。理想情况是只需将其以列表方式列出，配合基本的图片即可。（极限编程 *Extreme Programming* 包含此观点，它建议使用尽量小的文档，虽然这是针对中小型项目提出的）。有些限制可能需要你扩充这些文档，但是初始文档应该保持小而简练。只需几次头脑风暴会议，团队领导当场即可生成这些文档。这样做不仅征求了每个人的意见，也促使了团队达成一致。最重要的是，这样可以使大家充满激情地开始项目的开发工作。

此阶段我们要完成什么任务呢？注意力必须集中于问题的核心：确定系统要作什么。对此，最有用的工具就是一组“用例”（*use cases*），也就是极限编程中的“用户故事”（*user*

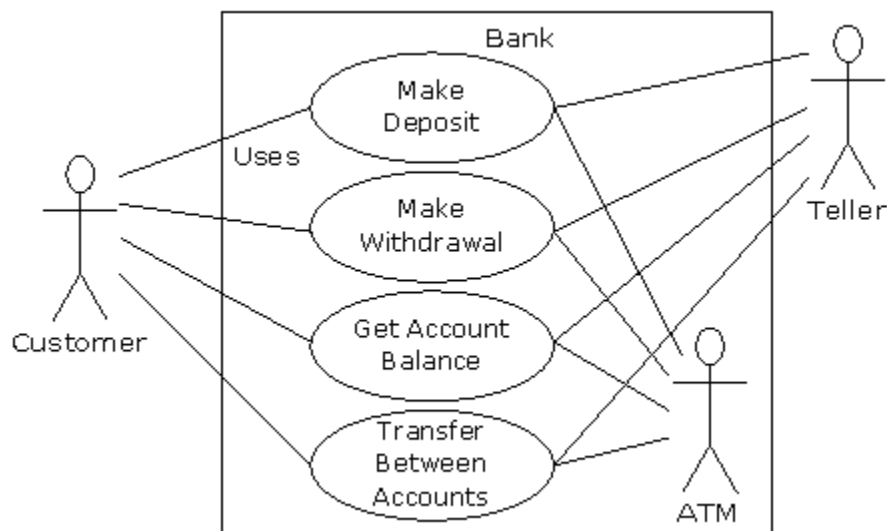
³ Gause & Weinberg 所著的《*Exploring Requirements: Quality Before Design*》(Dorset House 1989)是需求分析方面极佳的资源。

stories)。用例能够找出系统的关键特征，揭示出一些将要用到的基本的类。这需要对如下问题，做出本质性的描述式解答⁴：

- 谁将使用该系统？
- 系统参与者能使用该系统做什么？
- 参与者如何使用该系统？
- 如果别的人正在做这件事，或者同一个参与者有不同的目的，那么它会以什么方式运作？（用以揭示变异情况）
- 使用该系统做某件事时，可能会发生什么问题？（用以揭示异常）

例如你正在设计银行的自动出纳系统，系统功能各个方面的用例应该描述自动出纳系统在各种可能的情况下的行为。这些“情况”都被称作“场景（scenario）”，用例则被看作场景的集合。你可以把一个场景看作如下的问题：“如果……系统会做什么？”例如，“如果用户在 24 小时内存入一张支票，该支票尚未过户，账户中没有足够的余款可供提取，此时自动出纳系统如何处理？”

用例图应该尽量简单，以免你过早陷入系统实现的细节：

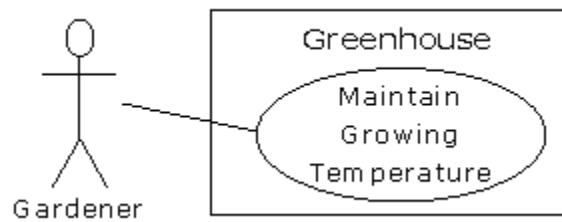


每个棒状小人代表一个“参与者”，通常指一个真正的人或某个其它种类的自由代理（可以是其他计算机系统，例如这里的“ATM”）。方框表示系统边界。椭圆代表用例，描述了系统可以执行的有价值的工作。参与者与用例之间的连线表示交互。

系统具体是如何实现的并不重要，对于用户来说，只要系统看起来像是这样就可以了。

即使底层系统确实十分复杂，用例也不需要很复杂。用例只是用来展示用户所看到的系统的那一面。例如：

⁴ 感谢James H Jarrett的帮助。



通过确定用户与系统间的所有交互，用例可以生成需求说明。找出系统所有的用例，也就找出了系统的核心。专注于用例的好处在于，它总可以令你回到问题的本质，不至于转到对工作不重要的事情上。也就是说，只要有了一套完整的使用例，就能够描述系统，然后进入下一阶段的工作。你可能无法一次就理清所有事情，但是没关系，所有事情都会及时显露出来。如果在此阶段你就想获得完美的系统功能说明书，那么你就会陷入困境。

如果你真的陷入了困境，可以使用粗略的近似工具，来启动此阶段的工作：用很少的段落来描述系统，然后找出名词与动词。名词通常是参与者、用例的环境（例如上例的“大厅”）、或者用例中被操作的人工制品。动词可以表示角色与用例间的交互，并说明用例中的步骤。你还会发现这里找到的名词和动词，将在设计阶段生成对象和消息（注意，用例描述的是子系统间的交互，而“名词动词”技术并不生成用例，所以它只能作为头脑风暴的工具）⁵。

用例与参与者之间的边界能够指出用户接口之存在，但它并不定义任何用户接口。关于定义与生成用户接口的过程，可以参考Larry Constantine 和 Lucy Lockwood所著的 *Software for Use*, (Addison-Wesley Longman, 1999) 或者看看 www.ForUse.com。

虽然进度安排根本就是一种魔术，但是此时，基本的进度安排还是很重要的。现在，你对你要做的东西应该有了一定的了解，因此你能够大致猜到这项工作要做多久。但是这其中还有许多其他因素。如果你把进度估计的过长，那么公司可能会决定不做此项目（这样，公司的资源可以用在更合理的地方）。或者，公司经理可能已经决定了此项目要做多长时间，并试图影响你的看法。不过，最好一开始就能有一个诚实的进度安排，并尽早做出这个困难的决定。许多人试图建立能精确安排进度的技术（这很像预测股票市场的技术），但或许最好的方法就是凭借你的经验与直觉。先凭直觉来估计所需时间，然后乘以两倍，最后再加上百分之十。你的直觉可能正确，让你可以及时完成项目。乘上两倍是为了把事情做得更像样，最后百分之十则用在润饰处理上⁶。无论你想如何解释，无论你作出的进度安排引来怎样的抱怨，最终的结果似乎就是如此⁷。

⁵ 关于用例，更多的信息可在Rosenberg 所著的《*Use Case Driven Object Modeling with UML*》(Addison-Wesley 1999) 中找到。Beck 与Fowler合著的《*Planning Xtreme Programming*》(Addison-Wesley 2001)中有关于用户故事的很好的观点。

⁶ 对此，我的观点后来有了变化。乘以两倍再加上百分之十，虽然可以得到合理的估计（假设不存在太多的外卡），但是你仍然得辛勤工作，才能按时完成工作。如果你需要时间使系统更优雅，而且你很享受这个过程，那么我认为正确的乘数可能应该是三倍或四倍。可以参考DeMarco 与 Lister的《*PeopleWare*》(Dorset House 1999)，学习如何根据生产力与“Parkinson法则”估算进度。

⁷ 《*Planning Xtreme Programming*》中有一些关于计划与时间估算的有价值的观点。

阶段 2：如何构建？

在这个阶段，你必须完成一个设计，它描述了各个类的模样，以及它们之间交互。在确定类及其交互的时候，CRC 卡（类—职责—协作对象 *Class-Responsibility-Collaboration*）是一个非常管用的工具。它的价值在部分程度上在于它没什么技术：做一些 3×5 的卡片，然后在上面填写。每张卡片表示一个单独的类，卡片上应该记录：

1. 类的名字。这很重要，应该做到让人望文生义，一眼就能猜到类的功能。
2. 类的职责：它应该做的事情。通常可以从类的方法名中总结获得（因为在好的设计中，那些方法的名字应该是描述性的），但也不排除还有别的成分。如果你还想有所收获，从一个懒惰的程序员的观点来看，这个问题就是：你期待什么样的对象神奇般地出现以帮你解决问题？
3. 类的“协作者”：它与哪些类交互？“交互”是很宽泛的词，可以表示聚合，或者只是存在某些对象，提供服务给此类的对象。协作关系还应该考虑这个类的听众。例如，如果你创建 **Firecracker**（鞭炮）类，谁需要观察它，**Chemist**（化学家）还是 **Spectator**（观众）？前者想知道它的化学构造，后者关心鞭炮爆炸时的颜色和形状。

你也许觉得卡片应该更大一些，因为你想在上面记录所有的信息。它们是故意被做得很小的，这不仅可以使你的类保持短小，还可以令你不至过早深入细节。如果对于某个类，一张卡片的信息无法满足你的需要，那么这个类就太复杂了（或者是你考虑的太仔细了，或者是应该多创建几个类）。理想的类应该让人一眼就能理解。CRC 卡的思想就是帮助你形成最初的设计，对问题有全面的了解，然后再改进你的设计。

CRC 卡最大的优点之一就是“交流”。尤其是在团体中不用电脑进行实时交流的时候。每个人负责几个类（开始还没有名字或其他信息）。通过模拟场景，每次解决一个“场景”，决定要向哪些不同的对象发送哪些消息。走完这个过程，就找出了类、类的职责和类的协作者，同时也填写好了 CRC 卡。所有的用例都分析过之后，你就有了一份相当完整的初始设计。

在使用 CRC 卡做初始设计之前，我经历过的最成功的经验是，站在团队前面，在白板上画出所有的对象。那是个从未做过 OOP 项目的团队。我们讨论对象之间应该怎样相互通信，白板上的对象可以被擦掉，可以被替换。我在白板上管理所有的“CRC 卡”，这很有效。而开发团队（知道项目要做什么）实际上已经在设计了；他们“拥有”这份设计，而不是将做好的设计丢给他们。我所做的只是引导这个过程，问恰当的问题，提取出所有的假设，再基于开发团队的反馈去修改这些假设。这个过程最漂亮的地方在于，开发团队学会了如何做面向对象的设计。而且不是通过抽象的示例，而是通过自己做设计学会的，在那一刻，这对他们才是最感兴趣的，因为这样的设计是属于他们的。

当你完成了一组 CRC 卡，你可能会想用 UML 更形式化地描述你的设计⁸。你可以不用 UML，不过它确实很有帮助，特别是如果你想在墙上挂一张图，让大家专注于对它进行思考的

⁸ 对于新手而言，我推荐《UML Distilled, 2nd edition》。

时候。这是个好主意（有太多的UML画图工具可用）。UML的替代方案是，以文字描述对象和接口，或者依赖程序语言，即代码本身⁹。

UML 还提供了用以描述系统动态模型的图形符号。有时候它们很管用。例如，当系统或子系统的状态迁移很重要时（例如控制系统），需要专门的图进行表现。对于数据为主导因素的系统或子系统（例如数据库），可能还需要描述数据结构的图。

描述完了对象和接口，也就完成了第二阶段的工作。然而总会有遗漏，要到第三阶段才能发现。不过没关系。重要的是，你最终能找出所有的对象。在这个过程中，早点找出所有的对象和接口当然最好，但是 OOP 有完整的结构，所以即使晚一些才找出它们也不算很糟糕。事实上，对象的设计可划分为五个阶段，贯穿整个程序开发过程。

对象设计的五个阶段

对象的设计并不只限于写程序的那段时间，而是呈现为一系列阶段。这种观点很有帮助，因为，当你不再期望立刻就能达到完美，反而可以认识到：对对象行为和外貌的理解，会随着时间的流逝而发生。这种观点适用于各种类型的程序设计。针对特定类型程序的模式都是在一遍遍与问题对抗的过程中浮现出来的（这是《*Thinking in Patterns (with Java)*》一书的主题，可由 www.BruceEckel.com 取得）。对象也是如此，在理解、使用、重复运用的过程中，对象的模式也显现了出来。

1. **发现对象。**这发生在开始分析程序的时候。通过找出外部因素和系统边界、重复元素、以及最小的概念单元，就能够发现对象。如果已经有了一套类库，那么某些对象就很明显了。通过类之间的共性，可以确定基类。类的继承体系也许立刻就能显现出来，也有可能要推迟到设计阶段。
2. **对象的组合。**当你创建对象的时候，不一定能发现对象需要的新成员。对象内部的需求可能需要其他类的支持。
3. **系统的构建。**如前所述，更多的需求要在稍后的阶段才显现出来。就像学习过程一样，你的对象会逐步演化。与其它对象相互通信和互联的需求，可能会改变你所需要的类或者需要新的类。例如，你可能发现需要辅助类或助手类，例如链表，它们不包含或只包含少量状态信息，并且只是辅助其他类的运作。
4. **系统扩展。**当你向系统添加新特性的时候，可能会发现先前的设计不便扩展。有了这些信息，就可以通过添加新的类或者类的层次，来重新构造部分的系统。这也是一个思考从项目中剔除一些功能的好时机。
5. **对象重用。**对类而言，这才是真正的重点。如果某人在全新的场景中重用某个类，可能会发现该类的某些缺陷。因为你要修改这个类，使它适应更多的程序，所以这也使得类的更通用的原则变得越来越清晰了，直至它真正可以被重复使用。无论如何，别指望系统中的多数对象都被设计为可重用的，因为完全可以接受大多数的对象是系统相关的。可重用的类型不是很常见，而且为了可重用，它们必须解决更通用的问题。

⁹ Python (www.Python.org) 通常被用来作为“可执行的伪代码(pseudocode)”。

对象开发指南

在思考如何开发类的时候，下列步骤是一些具体的指导：

1. 为一个特定问题生成一个类，然后在解决其他问题的过程中，让这个类成长并且成熟。
2. 记住，发现你所需要的类（和接口）是系统设计的主要工作。如果这些类已经有了，那么这将是简单的项目。
3. 不要强迫自己在开始阶段就明白所有事情。在学习的过程中，随时可能了解某些事情。
4. 开始编程。做出可以运行的某些东西，才可以证明或者反驳你的设计。不要担心你的代码会变成意大利面条似的过程型代码。类可以将问题分解，并有助于对无序和混乱的状态进行控制。不好的类并不会破坏好的类。
5. 保持简单。功能明确、短小而清楚的对象，好过庞大而复杂的接口。需要做决定的时候，可以使用**Ockham**的剃刀法则¹⁰：考虑最简单的选择，因为简单的类总是最好的。由短小而简单开始，在更好的理解问题之后，可以扩展类或接口。向类中添加方法总是容易的，但随着时间推移，从类中删除方法可就困难了。

阶段 3：构建系统核心

这是从原始设计到可测试的代码的最初的转换，编译并执行此代码，可以证明你的系统架构是否正确。这不是一次就能通过的过程，到第四阶段你就会明白，这只是迭代构建系统的一系列步骤的开始。

你的目标是找出系统体系结构的核心，为了生成可运行的系统，无论系统在此初始阶段如何不完整，此核心都必须实现。你开发的是一个框架，更多的迭代开发都是基于它进行的。你同时也是在执行第一次系统集成与测试，并将其反馈给系统的投资者，让他们了解系统的样子，以及进展如何。理想情况下，你还可以揭示一些风险，还能够发现原始的架构需要改变和提高之处，这些事情只有到具体实现系统的时候才能了解。

系统开发包含对其实现性的检查，也就是依据需求分析与系统说明（无论它们以何种方式存在）进行测试。要确保你的测试验证了所有需求和用例。当系统核心稳定下来以后，你就可以进入下一阶段，为系统添加更多的功能了。

阶段 4：迭代用例

在核心框架运行后，向其添加的每个功能，本身就是一个小型项目。每次迭代，都会添加一个功能集，它只占相当短的开发时间。

¹⁰ 使用较少的资源就可以实现的事情…如果是较多的资源实现的，那么这么做就是徒劳的…若无必要不应该一心多用。” 参见Ockham的《William》，1290—1349 页。

一次迭代规模多大？理想情况下，每次迭代需要一至三周（这要基于实现所采用的语言）。在此阶段结束的时候，你的系统应该集成妥当、通过测试，并且功能更多。不过，最有趣的是迭代的基础：单个的用例。每个用例就是一组相关的功能，一次迭代就将其全部集成到系统中。这有助于你更好的理解用例的活动范围，并且能够验证你的理解。因为在完成分析与设计之后，用例也没有被丢弃，在整个软件开发过程中，它都是基本的开发单元。

当目标达成，或者截至日期已到，而且用户对当前的版本还满意时，你才可以停止迭代。（记住，软件是“预订”的生意。）因为开发过程是迭代式的，所以你有许多机会可以交付产品，而不是只有一个终点。开放源代码的项目更是完全采用迭代式开发，具有高反馈的环境，这正是它们成功的原因。

许多原因使得迭代式的开发过程具有非凡的价值。你可以尽早发现并解除风险；用户有充足的机会改变主意；程序员可以获得更高的成就感；能够更准确地掌握项目。但是，更重要的好处是对投资方的反馈，让每个人都可以看到项目各个方面当前的进展。这可以减少令人昏昏欲睡的会议，提高投资方的信心与支持。

阶段 5：演化

这一阶段，传统上称为“维护”阶段。这个术语包含广阔，能够表示内容从“令系统按设计的方式工作”到“添加用户忘记了的功能”，还有更传统的“修订程序错误”和“根据新的需要添加新的功能”。对“维护”这个术语的众多误解，使它带有一定的欺骗性。部分原因是，它暗示已经生成了原始程序，需要做的只是改变某些部分、加点油、防止生锈。或许应该用一个更好的术语描述实际发生的事情。

我使用“演化（*evolution*）”这个术语¹¹。它表示“由于不是一次就能做对，所以要给自己留有余地，能够学习、返工、并做出改变。”随着你的学习，在更深刻地理解问题之后，可能需要作出许多改变。只要你不断演化，直到把事情做对，就能够作出优雅的产品。无论长期或短期，这样的付出总会有回报。“演化”是让你的程序从“好”到达“非常好”，也是将开始并不了解的问题，变得清楚的过程，同时也是将你的类，从仅适合单个项目使用，“演化”到成为能够可复用资源的过程。

“把事情做对”并不仅仅表示程序按照需求与用例工作，还表示代码的内部结构对你来说要有意义，看起来它们确实适合被组织到一起，并且组合良好、没有笨拙的语法和过大的对象，也没有丑陋的代码。此外，你必须意识到，在开发过程中，变化是不可避免的。但是，如果程序有良好的结构，就不会随之变化，并且可以容易并清楚地应对变化。这可不是小技巧。你不仅要明白自己在开发什么，更得了解程序如何演化（我称之为“改变向量（*vector of change*）”）。幸运的是，面向对象的编程语言特别擅长支持此类持续的修改，由对象确定的边界，会主动防止结构被破坏。它们还允许你做出改变，而不会像过程型程序那样，引发危及所有代码的激烈地震。事实上，“演化机制”可能是 OOP 最为重要的优势。

¹¹ Martin Fowler的《*Refactoring: Improving the Design of Existing Code*》(Addison-Wesley 1999)完全采用Java的示例，至少涵盖了“演化”的一个方面。

有了演化机制，你可以先建立某些与你所想近似的东西，稍事休息，然后将它与需求分析做比较，看看哪里有缺陷。回过头来，再重新设计并实现工作不正确的那部分程序¹²。要解决某个问题，或者问题的一个方面，在找到正确的解决方案之前，实际上可能需要做多次尝试。（学习设计模式对此会有所帮助。你可以从www.BruceEckel.com的《*Thinking in Patterns (with Java)*》中找到需要的信息）

系统开发也需要演化。例如，当你检查它是否符合需求时，发现它并不是你实际想要的。当系统运行起来，你可能会发现，你真正想解决是另一个问题。如果你认为这种演化总会发生，那么你应该尽快完成第一个版本，这样才能够判断它是否就是你想要的东西。

也许需要记住的最重要的事情是，缺省情形下，如果你修改某个类的定义，其父类与子类仍然可以正常运作。你不必害怕修改（特别是，如果你有了一套内置的单元测试集合，就能够验证修改的正确性）。修改并不一定会破坏程序，任何改变都应该限制在子类和/或所修改的类的特定合作者之间。

成功的计划

如果没有很仔细做好的计划，你肯定不会开始建造房屋。如果你只是要建一个平台，或者狗屋，那就无需计划得精心详尽，不过你仍然需要画一些草图，用来引导自己。软件开发则走向了极端。很长一段时间内，在开发过程中，人们不做结构性规划，于是大型项目纷纷失败。针对这个问题，我们在大型项目中采用了另一种方法论，它有令人望而生畏的繁多的结构和细节。这些方法论都太吓人了，看起来会耗尽你所有的时间用来写文档，根本没时间编程（事情经常如此）。我希望这里为你展示了一条中庸之道：可变的尺度。使用适合你的需要（和你的性格）的方法。相对与完全没有计划，无论你的计划有多么简单，都能令你的项目有很大的改善。别忘了，据估计，超过百分之五十的项目都失败了（有些评估甚至到了百分之七十！）。

你应该遵循计划——最好是简单而短小的计划，在编码之前好好设计结构。你会发现，比起你直接埋头硬干，这样做使得将事物组合起来变得容易多了，你也能得到更多的满足感。我的经验是，优雅的解决方案令你获得的满足感，完全超过以往的经验。这不仅是技术，更近于艺术了。优雅，不是轻佻的追求，它是有回报的。它不仅令你的程序更易于开发与调试，并且更易于理解与维护，而这正是其经济价值所在。

极限编程

从我在研究生院学习的时候开始，我就断断续续地研究了分析与设计技术。极限编程 *Extreme Programming* (XP) 是其中最激进、最令人愉快的一种观点。Kent Beck 所著的《*Extreme Programming Explained*》(Addison-Wesley, 2000) 记录了它的发展，它的网站是 www.xprogramming.com。每隔一两个月，Addison-Wesley 出版社就会

¹² 这就像是“快速建立原型”，你可以快速开发出不健全的版本，从中可以更了解系统。然后抛掉这个原型，开发健全的系统。快速建立原型的问题在于，人们不愿抛掉开发出来的原型，而是基于它进行开发。由于其中包含了过程型编程中“结构不足”的缺点，因此经常导致系统混乱，带来高昂的维护成本。

出版一本 XP 系列的新书。其目的似乎是要令每个人都相信，应该使用这些书（不过这些书通常都很小而且读起来很愉快）。

XP 既是编程工作的哲学，也是一套实践准则。其中的一些原则，近来也反映在其他方法论中，但是按我的观点，其中最重要而独特的两个贡献是“优先编写测试(write tests first)”和“结对编程(pair programming)”。虽然 Beck 强烈支持整个过程都采用 XP，但是他也指出，只要采用了这两条实践准则，就能大幅提高生产率和可靠性。

优先编写测试

传统上，将测试视为项目的最后部分，你已经“令所有东西都工作了，只是需要确认”。它隐含地具有较低优先级，专门做测试的人员也没有被给予什么地位，甚至常常被隔离在地下室，远离“真正的程序员”。作为回应，测试团队就像是穿着黑袍的人，弄坏了东西就高兴得咯咯笑（老实说，当令编译器出错时，我也有这种感觉）。

XP 完全颠覆了测试的概念，给予其与代码相等（甚至更高）的优先级。事实上，它要求在写代码前先写出对其要做的测试，并且将这些测试与代码永远放在一起。每完成项目的一个版本，必须成功通过测试（通常一天一次，甚至多次）。

优先编写测试有两个非常重要的影响。

首先，它要求类的接口的定义必须清楚。我总是建议人们，在设计系统的时候“为解决特定问题而专门设计一个类”。XP 的测试策略则更进一步，它确切地向类的用户说明，类应该是什么样子、应该如何运作。这里没有不确定的术语。你可以写成散文，或画图来描述类应该如何运作，以及它的样子，但没有什么比一套测试集合更真实。前者是愿望的列表，而测试是一个契约，由编译器与测试框架保证。很难想象有工具能比测试更具体地描述一个类。

在创建测试时，你会被强制去彻底思考类，并且常会因此而发现一些需要加入的功能，这些功能在使用 UML 图，CRC 卡、用例等工具时被遗漏了。

优先编写测试的第二个重要的影响是，开发出的每一版软件都要进行测试。此时编译器帮你完成了一半的测试工作。如果从这个角度观察编程语言的演化，就会发现，技术上最实际的改善其实就是以测试为中心。汇编语言只能检查语法，而 C 做了一些语义约束，可以避免你使用错误的类型。OOP 语言则有更强的语义约束，实际上，你可以将其视为某种形式的测试。“这个数据的类型用得正确吗”，“这个方法是否调用得正确”，这都是由编译器在运行期做的测试。我们已经看到了将这类测试内置于程序语言中的效果：人们能够写出更复杂的系统，花费更少的时间和精力就能令系统正确工作。我曾其原因而感到迷惑，但是现在我认识到，原因就是测试：如果某件事做错了，内置的测试安全措施会通知你出了问题，并指出问题在哪儿。

但是编程语言内置的测试也只能做到这么多。某些时候，你必须介入其中，添加余下的测试（与编译器和运行期的系统相互配合），从而产生完整的能够验证整个程序的测试包。而且，难道你不希望像编译器从旁照看着你一样，让这些测试从一开始就帮你走得正确吗？

这就是为什么要优先编写测试，并且每开发一版系统都会自动运行测试的原因。你写的测试将成为语言提供的安全措施扩展。

使用那些越来越具威力的编程语言时，我发现了一件事情，由于我知道这些语言使我不必浪费时间来查错，所以我可以大胆的做一些试验。对整个项目来说，XP 的测试体系正在在做同样的事情。因为你知道这些测试能够捕捉到任何你所引入的问题（只要你想到了，就应该添加测试），所以你可以做大幅度地修改，而不必担心会令整个项目陷入混乱。这真是太有威力了。

本书到了第三版，我更认识到测试是如此地重要，以至于必须应用到书里的实际示例中。在 2002 年 Crested Butte 夏令营实习生的协助下，我们开发了一套测试系统，它们被应用于全书。第十五章是其代码与描述。此系统大大增强了本书示例代码的强壮性。

结对编程

我们向来被灌输个人主义，无论是学校（我们成败都靠自己，与邻座合作被视为“作弊”）或是媒体。尤其是好莱坞电影，其中的英雄总是反抗盲目的顺从¹³。“结对编程”反对这种丑陋的个人主义。程序员被视作个人主义的典范，Larry Constantine 喜欢称之为“编码牛仔”。然而与传统思想作战的 XP 认为，应该两个人合用一部机器写代码。而且应该在一个公共区域内，放一组机器，不要在其中使用室内设计师最喜欢的隔板。实际上，Beck 就认为转向 XP 的第一步是拿起螺丝刀和扳手，把所有碍事的东西都拆掉¹⁴。（这可就需要一个能够平息设备部门的愤怒的经理。）

结对编程的价值在于，在某个人思考时，另一个人就实际编码。思考者要把握系统全局，不仅是手边的问题，还有 XP 的所有准则。如果两个人都在工作，就不太可能会有人说：“我可不想优先编写测试”。如果编码的人陷入了困境，两个人可以交换位置。如果两人都陷入了困境，那么他们发呆的样子，可能会吸引工作区域内的其他人来帮忙。结对工作可以让事情按计划、更平顺的进行。更重要的可能是，它使得编程工作更具互动性，并且令人愉快。

我在某些研讨会中，要求学员结对编程做练习，这似乎在相当程度地提高了每个人的经验。

过渡策略

如果你的公司决定采用 OOP，你的下一个问题可能是：“如何让我的经理、同事、部门、搭档都开始使用对象呢？”回顾一下你作为一个独立程序员的时候，是如何学习新的编程语言和新的程序设计思维的。首先是读书和学习示例，然后是开发实验性的项目，感受一些基本原理，此时不要做容易陷入混乱的项目。接下来就是一个“真实世界”的项目，让它确实做些有用的工作。在第一个项目中，要不断地通过阅读、向专家请教、和朋友交流

¹³ 虽然有点大美国主义，但是好莱坞的故事的确流传各地。

¹⁴ （特别是）包括扩音装置。我曾经工作的一个公司，使用广播通知主管人员的每一个电话，这经常打断我们的工作（但是管理者并不会想到，扩音装置这么重要的设备有多让人受不了）。最终，在没人注意的时候，我剪掉了扩音器的电线。

经验等方式去学习。对于转向 Java，许多经验丰富的程序员都推荐此方法。要转换整个公司，必然会造成一定的团体变动，但是记住个人所采用的方法，对每个步骤都有帮助。

指南

要过渡到 OOP 和 Java，可以参考以下指南：

1. 训练

第一步便是某种形式的教育。记住公司在代码上的投资，当每个人都对不熟悉的功能感到困惑的时候，尽量别让任何东西处于混乱状态超过六到九个月。选择一小组人进行培训，最好是一组求知欲强、合作无间、能够在学习 Java 的时候相互帮助的人。

另一种方法是整个公司一起学习，包括针对决策高层的课程纵览，针对项目开发人员的设计与编程的课程。这种方法特别适合较小的公司进行完全转变，或者在大型公司的部门层次上采用。不过，由于代价比较高，有些公司选择从项目层次开始训练，做一个引导性项目（可能有外聘的讲师），然后再让此项目的开发团队指导公司的其他成员。

2. 低风险项目

先尝试一个低风险的项目，并允许错误发生。获得了一些经验以后，就可以将最初的团队成员分散到其他项目中去，或者将团队成员作为 OOP 技术支持。第一个项目可能无法一次就成功，所以不应该选择对公司有重大影响的项目。应该选择简单的、能自我控制、具有教育性的项目。这是指此项目应该生成一些类，它对公司其他程序员学习 Java 时有指导意义。

3. 向成功案例借鉴

在手忙脚乱之前，应该找一些好的面向对象设计的例子。很可能已经有人解决了你遇到的问题，即便没有完全解决，你也能够应用所学，按需要修改别人已有的设计。这正是设计模式的一般概念。《*Thinking in Patterns (with Java)*》一书涵盖此内容，可以在 www.BruceEckel.com 找到。

4. 使用现成的类库

转换到 OOP 的一个重要的经济动机是，可以轻松使用现成的类库（特别是 Java 标准类库，本书涵盖此内容）。当你能够利用现成的类库创建与使用对象时，你就能达到最短的程序开发周期。然而，一些新程序员没有认识到这一点，不清楚现成的类库，或者对语言太入迷了，想自己撰写那些已经有了的类。如果在过渡到 OOP 的过程中，能尽快找到别人的代码并重用之，就能通过 OOP 和 Java 取得最大的成功。

5. 不要用 Java 重写现成的程序

使用 Java 重写那些现成的、功能性的代码，通常不是使用时间的最佳方式。如果必须将其转换为对象，你可以使用 Java 原生接口（Java Native Interface）或者 XML 技术，与 C 或 C++ 程序接轨。这样做的好处，会逐渐体现，特别是需要重复使用那些程序代码的时候。不过这么一来，在刚开始的几个项目中，你就无法看到生产率的惊人提升，除非是一个全新的项目。当一个项目从概念到实现都采用 Java 和 OOP 时，它们才能显现出最佳的效果。

管理上的障碍

如果你是管理者，你的工作便是为你的团队争取资源，排除影响团队成功的障碍，提供最具生产力，最让人愉快的工作环境，如此才可能达成你经常要求的“奇迹”。迁移至 Java 要做三方面的工作，如果你无需为此付出任何代价，那可真是件美事。有些 OOP 的替代方案，是专为 C 程序员（和别的过程型语言的程序员）准备的。虽然转到 Java 付出的代价可能小些，（这依赖于你本身的限制）因为 Java 是免费的，但是在向公司推广 Java 前，仍然有一些障碍需要当心。

启动成本

过渡到 Java 的成本不仅仅是取得 Java 编译器（Sun 的 Java 编译器是免费的，所以这称不上障碍）。投资在训练上，并且买入可以解决你的问题的类库（不要自己开发），这才是令你的中长期成本最小化的正途。这些需要花费金钱的成本，在实际提案中必须考虑。此外，学习新的程序语言和新的编程环境时损失的生产率，也是隐藏的成本。训练与指导能够将这些成本降到最低，但是团队成员也必须克服在学习新技术时的自我斗争。在这个过程中，他们会犯更多的错误（这是一种特色，因为认识错误是最快的学习方式），并且生产率也会下降。尽管如此，对于某些类型的问题，如果有合适的类，合适的开发环境，即使你还处在学习阶段（虽然所犯的错误更多，每天能完成的编码更少），比起使用 C，还是可能具有更高的生产率。

性能问题

一个常见的问题：“OOP 不会将我的程序变得又大又慢吗？”答案是：“视情况而定。”Java 提供了格外安全的特性，传统上是牺牲了诸如 C++ 语言所具备的性能。但是“hotspot”等技术与编译技术的进步，使得 Java 的速度显著提高，而且还在向更高的性能不断努力。

如果你关注于快速原型开发，则可以忽略效率问题，尽快将构件拼凑在一起。如果使用第三方类库，通常厂商已经对其做过优化。因此在进行快速开发时，性能并不是问题。如果你的系统已经够小够快，那你的工作就可以结束了。否则，你应该使用性能检查工具，看看能否只改写小部分代码，以提高系统速度。如果这还不够，就需要修改底层系统，但尽

量不要改动上层代码。除非没有别的解决方法了，你才需要完全改变设计。如果性能在设计中占有重要地位，那它必须成为基本的设计标准之一。采用快速开发的优点是，你可以在早期就发现问题。

第十五章介绍的性能检查工具，能够帮助你发现需要优化的系统瓶颈（有了 hotspot 技术，Sun 公司不再推荐使用原生函数优化性能）。优化工具也就派上用场了。

常见的设计错误

当你的团队开始转向 OOP 和 Java 时，程序员会犯一些常见的错误。早期的项目在设计与实现时，通常缺乏专家的反馈意见。这可能是公司内没有培养出专家，也可能是对固定的顾问产生了抵触。在整个开发过程中，很容易发生因过早地了解 OOP 而走偏了方向的事情。熟悉此语言的人觉得很显然的事情，对新手而言可能是十分重大的议题。雇用经验丰富的专家进行训练和指导，可以避免许多此类争论。

总结

本章介绍了 OOP 方法论的概念，以及公司转向 OOP 和 Java 时可能遇到的问题。更多的关于对象设计的内容参见 MindView 的“Designing Objects and Systems”研讨会（在 www.MindView.net 上查找“Seminars”）。

附录 A:对象的传递与返回

现在,你应该已经相当适应这样的概念了:当你“传递”对象的时候,实际上是在传递对象的引用。

对许多编程语言而言,只需按“正规”方式传递对象,就能处理绝大多数情况。然而总有些时候,需要以不太正规的方式处理。于是,事情突然变得复杂起来(在 C++ 中,会变得相当复杂),Java 也不例外。因此,确切理解在传递对象的时候发生了什么,就显得很重要了。本附录将带你深入其中。

如果你有其他语言的编程经验,即可从另一个角度看待附录 A 探讨的问题:“Java 中是否有指针”?有些人认为,指针既难掌握又很危险,所以指针很糟糕。既然 Java 全部都是优点,还能减轻你的编程负担,所以它不可能包含指针。然而,确切地说,Java 有指针。事实上,Java 中(除了基本类型)每个对象的标识符就是一个指针。但是它们受到了限制,有编译器和运行期系统监视着它们。或者换个说法,Java 有指针,但是没有指针的相关算法。我一直称之为“引用(reference)”,你也可以将它们看作“安全的指针”,就像小学生的安全剪刀,它不尖锐,不故意使劲通常不会伤着人,但是使用起来不但慢而且麻烦。

传引用

将一个引用传入某个方法之后,它仍然指向原来的对象。可以用一个简单的试验证明:

```
//: appendixa:PassReferences.java
// Passing references around.
import com.bruceeckel.simpletest.*;

public class PassReferences {
    private static Test monitor = new Test();
    public static void f(PassReferences h) {
        System.out.println("h inside f(): " + h);
    }
    public static void main(String[] args) {
        PassReferences p = new PassReferences();
        System.out.println("p inside main(): " + p);
        f(p);
        monitor.expect(new String[] {
            "%% p inside main\\(\\):
PassReferences@[a-z0-9]+",
            "%% h inside f\\(\\): PassReferences@[a-z0-9]+"
        });
    }
} ///:~
```


打印语句自动调用 `toString()` 方法，而继承自 `Object` 的 `PassReferences` 类没有重新定义 `toString()` 方法。所以使用的是 `Object` 的 `toString()` 方法，它打印出类名以及存储对象的地址（不是引用，而是实际的对象）。输出看起来像这样：

```
p inside main(): PassReferences@ad3ba4
h inside f(): PassReferences@ad3ba4
```

可以看到，`p` 和 `h` 都指向同一个对象。像这样只是发送一个参数给方法，比复制一个新的 `PassReferences` 对象效率要高很多。但它也引出了一个重要的话题。

别名效应

“别名效应”是指，多个引用指向同一个对象，参见前例。当某人修改那个对象时，别名带来的问题就会显现出来。例如，如果此对象还有其他的引用指向它，而使用那些引用的人，根本没想到对象会有变化，定然会对此感到十分诧异。可以用一个简单的例子做演示：

```
//: appendixa: Alias1.java
// Aliasing two references to one object.
import com.bruceeckel.simpletest.*;

public class Alias1 {
    private static Test monitor = new Test();
    private int i;
    public Alias1(int ii) { i = ii; }
    public static void main(String[] args) {
        Alias1 x = new Alias1(7);
        Alias1 y = x; // Assign the reference
        System.out.println("x: " + x.i);
        System.out.println("y: " + y.i);
        System.out.println("Incrementing x");
        x.i++;
        System.out.println("x: " + x.i);
        System.out.println("y: " + y.i);
        monitor.expect(new String[] {
            "x: 7",
            "y: 7",
            "Incrementing x",
            "x: 8",
            "y: 8"
        });
    }
} ///:~
```

看这一行：

```
Alias1 y = x; // Assign the reference
```

它生成了一个新的 **Alias1** 引用，但是并没有赋予其用 **new** 创建的新对象，而是赋值为已存在的引用。既是将 **x** 的内容（**x** 指向的对象的地址）赋给 **y**，因此 **x** 和 **y** 指向同一个对象。所以，**x** 中的 **i** 增加时：

```
x.i++;
```

y 中的 **i** 也会受到影响。这可以从输出中看到：

```
x: 7
y: 7
Incrementing x
x: 8
y: 8
```

对此有一个很好的解决方案，很简单，就是不要这样做；不要在相同作用域内生成同一个对象的多个引用。这样的代码更易于理解与调试。然而，当你将引用作为参数传递时，它会自动被别名化（这是 **Java** 的工作方式）。因为，在方法内创建的局部引用能够修改“外部的对象”（在方法作用域以外创建的对象）。参见下例：

```
//: appendixa: Alias2.java
// Method calls implicitly alias their arguments.
import com.bruceeckel.simpletest.*;

public class Alias2 {
    private static Test monitor = new Test();
    private int i;
    public Alias2(int ii) { i = ii; }
    public static void f(Alias2 reference) { reference.i++; }
    public static void main(String[] args) {
        Alias2 x = new Alias2(7);
        System.out.println("x: " + x.i);
        System.out.println("Calling f(x)");
        f(x);
        System.out.println("x: " + x.i);
        monitor.expect(new String[] {
            "x: 7",
            "Calling f(x)",
            "x: 8"
        });
    }
} ///: ~
```

方法 `f()` 改变了它的参数，也就是 `f()` 外面的对象。出现这种情况时，你必须考虑清楚，这样做是否有意义，是否如用户所愿，会不会引起其他问题。

一般而言，调用方法是为了产生返回值，或者为了改变被调用者（某对象）的状态。通常不会为了处理其参数而调用方法，那被称作“为了副作用而调用方法”。因此，如果你创建的方法会修改其参数，你必须清楚地向用户说明它的使用方式，以及潜在危险。考虑到别名带来的困扰和缺点，我们最好能避免修改参数。

如果确实要在方法调用中修改参数，但又不希望修改外部参数，那么就应该在方法内部制作一份参数的副本，以保护原参数。这就是附录 A 的主题。

制作局部拷贝

做个复习：Java 中所有的参数传递，执行的都是引用传递。也就是说，当你传递“对象”时，真正传递的只是一个引用，指向存活于方法外的“对象”。所以，对此引用做的任何修改，都是在修改方法外的对象。此外：

- 别名效应在参数传递时自动发生。
- 方法内没有局部对象，只有局部引用。
- 引用有作用域，对象则没有。
- 在 Java 中，不需要为对象的生命周期操心。
- 没有提供语言级别的支持（例如“常量”）以阻止对象被修改，或者消除别名效应的负面影响。不能简单地使用 `final` 关键字来修饰参数，它只能阻止你将当前引用指向其他对象而已。

如果你只是从对象中读取信息，而不修改对象，那么“传引用”就是传递参数最高效的方式。缺省的方式就是最高效的方式，这当然最好。但是，有时必须将参数对象视为“局部对象”，才能使方法内的修改只影响局部的副本，从而不会改变方法外的对象。很多编程语言支持这种可以自动为参数对象创建一份方法内的局部拷贝的能力¹。Java 虽然不支持此能力，但是它允许你产生同样的效果。

传值

这引出了一个术语问题，对它的争论总是有益的。“传值”这个术语，及其含义依赖于你如何看待程序的操作。它通常的意义是，对于你传递的东西，得到它的一份局部拷贝。但真正的问题是，如何看待你传递的东西。对于“传值”的含义，有截然不同的两派观点：

1. Java 中传递任何东西都是传值。如果传入方法的是基本类型的东西，你就得到此基本类型元素的一份拷贝。如果是传递引用，就得到引用的拷贝。因此，所有东西都是传值。当然，前提是你认为（并关心）传递的是引用（而不是对象）。但是，Java 的设计是为了帮助你（在大多数时候）忽略你是在

¹ 在 C 中，通常处理的是很少的数据，它缺省的是传值。C++ 不得不沿用此模式，但是通过传值来传递对象则不太有效率。此外，在 C++ 中编写支持传值的类是个令人头疼的问题。

使用引用。也就是说，它希望你将引用看作是原本的“对象”，因为 Java 会在你调用方法的时候隐式地解除引用。

2. 对于基本类型而言，Java 是传值（对此双方没有异议），但是对于对象，则是传引用。大部分人的观点是，引用是对象的另一种称呼罢了，所以不会想到是“传引用”，而是认为“就是在传递对象。”既然向方法传递对象时，不会得到局部复制，所以传递对象很明显不是传值。Sun 公司似乎比较支持此观点，因此，曾经有一个“保留的、但没有实现的”关键字“byvalue”（可能永远也不会实现）。

两派观点都陈列出来了，我认为“这依赖于你如何看待引用”。此后我将回避此问题。最终你会明白，这个争论并没有那么重要。真正重要的是，你要理解，传引用使得（调用者的）对象的修改变得不可预期。

克隆对象

需要使用对象的局部拷贝的最可能的原因是：你必须修改那个对象，但又不希望改动调用者的对象。如果你决定要制做一份局部拷贝，可以使用 clone() 方法。这是定义在 Object 类中的 protected 方法。如果要使用它，必须在子类中以 public 方式重载此方法。例如，标准类库中的 ArrayList 类就重载了 clone()，所以我们才能由 ArrayList 调用 clone() 方法：

```
//: appendixA:Cloning.java
// The clone() operation works for only a few
// items in the standard Java library.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Int {
    private int i;
    public Int(int ii) { i = ii; }
    public void increment() { i++; }
    public String toString() { return Integer.toString(i); }
}

public class Cloning {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new Int(i));
        System.out.println("v: " + v);
        ArrayList v2 = (ArrayList)v.clone();
        // Increment all v2's elements:
        for(Iterator e = v2.iterator();
```

```

        e.hasNext(); )
        ((Int)e.next()).increment();
// See if it changed v's elements:
System.out.println("v: " + v);
monitor.expect(new String[] {
    "v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]",
    "v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]"
});
    }
} ///: ~

```

`clone()`方法只能生成 `Object`，之后必须将其转型为合适的类型。此例演示了 `ArrayList` 的 `clone()`方法，它并不自动克隆 `ArrayList` 中包含的每个对象。克隆的 `ArrayList` 只是将原 `ArrayList` 中的对象别名化。这通常称为**浅层拷贝**（*shallow copy*），因为它只复制对象“表面”的部分。实际的对象由以下几部分组成：对象的“表面”，由对象包含的所有引用指向的对象，再加上这些对象又指向的对象，等等。通常称之为“对象网”。将这些全部复制即为**深层拷贝**（*deep copy*）。

可以由输出看到浅层拷贝的效果，对 `v2` 的操作影响了 `v`：

```

v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

不自动对 `ArrayList` 包含的所有对象执行 `clone()` 是正确的，因为不能保证那些对象都是可克隆的（`cloneable`）²。

使类具有克隆能力

虽然是在所有类的基类 `Object` 中定义了克隆方法，但也不是每个类都自动具有克隆能力³。这似乎违反直觉，基类的方法在其子类中不是应该自动可用吗？Java 的克隆操作确实违背了此思想。如果一个类需要具有克隆能力，你必须专门添加一些代码，它才能够克隆。

² 这个词不是字典中的拼法，但Java类库就是这样用的，所以我也这么用，希望可以减少混淆。

³ 针对这句话，可以写一个简单的反例，像这样：

```

public class Cloneit implements Cloneable {
    public static void main (String[] args)
        throws CloneNotSupportedException {
        Cloneit a = new Cloneit();
        Cloneit b = (Cloneit)a.clone();
    }
}

```

然而，它能够运行只是因为 `main()` 是 `Cloneit` 的方法，它有权调用基类的 `protected clone()` 方法。如果你从别的类调用它，则无法通过编译。

使用 `protected` 的技巧

为防止所有类缺省地就具备克隆能力，基类中的 `clone()` 方法被声明为 `protected`。这意味着，对使用（而非继承）此类的客户端程序员，克隆方法不再是缺省地可用。而且，也不能通过指向基类的引用调用 `clone()` 方法。（虽然这在某些情形下似乎很有用，例如以多态方式克隆大量 `Object` 型对象。）实际上，此方法让你在编译期就知道，你的对象不具备克隆能力。而且，标准 Java 类库中的大多数类都不可克隆，这够奇怪吧。因此，如果你这样写：

```
Integer x = new Integer(1);
x = x.clone();
```

在编译时就会得到错误信息，说明 `clone()` 不可访问（因为 `Integer` 没有重载 `clone()`，它缺省是 `protected` 方法）。

不过，因为 `Object.clone()` 是 `protected`，所以在 `Object` 的子类（等同于所有类）内部，你有权限去调用它。基类的 `clone()` 方法很实用，它在“位”（bitwise）级别上复制子类的对象，如同通常的克隆操作一样。然而，你必须将你的克隆操作声明为 `public`，它才可以被访问。所以，克隆对象时有两个关键问题：

- 调用 `super.clone()`
- 将自己的克隆方法声明为 `public`

你可能需要重载所有子类的 `clone()` 方法；否则，你的（现在为 `public`）`clone()` 虽然能用，其行为却不一定正确（即使由于 `Object.clone()` 复制了实际的对象，令其行为有可能正确）。`protected` 的技巧只能用一次：在第一次继承无克隆能力的类，而又希望它的子类具有克隆能力时。继承自你的类的任何子类，其 `clone()` 方法都可用，因为 Java 中，继承无法削减方法的访问权。既是说，如果某个类是可克隆的，它的继承者也都是可克隆的，除非你使用某种机制（稍后会描述）“关闭”克隆能力。

实现 `Cloneable` 接口

要完善一个对象的克隆能力，还需要做一件事：实现 `Cloneable` 接口。该接口有点奇怪，因为它是空的！

```
interface Cloneable {}
```

实现空接口的原因显然不是为了类型转换，然后调用 `Cloneable` 接口的方法。这样使用的接口称为“标记接口（*tagging interface*）”，因为它像是一种贴在类身上的标记。

`Cloneable` 接口的存在有两个理由。第一，如果某个引用向上类型转换为基类后，你就不知道它是否能克隆。此时，可以使用 `instanceof` 关键字（第十章对此有描述）检查该引用是否指向一个可克隆的对象。

```
if(myReference instanceof Cloneable) // ...
```

第二个原因与克隆能力的设计有关，这是考虑到也许你不愿意所有类型的对象都是可克隆的。所以 `Object.clone()` 会检查当前类是否实现了 `Cloneable` 接口。如果没有，它抛出 `CloneNotSupportedException` 异常。所以，作为实现克隆能力的一部分，通常你必须实现 `Cloneable` 接口。

成功的克隆

只要你明白了 `clone()` 方法的实现细节，就能够让你的类很容易地复制本身，以提供一个局部副本：

```
//: appendixA: LocalCopy.java
// Creating local copies with clone().
import com.bruceeckel.simpletest.*;
import java.util.*;

class MyObject implements Cloneable {
    private int n;
    public MyObject(int n) { this.n = n; }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {
            System.err.println("MyObject can't clone");
        }
        return o;
    }
    public int getValue() { return n; }
    public void setValue(int n) { this.n = n; }
    public void increment() { n++; }
    public String toString() { return Integer.toString(n); }
}

public class LocalCopy {
    private static Test monitor = new Test();
    public static MyObject g(MyObject v) {
        // Passing a reference, modifies outside object:
        v.increment();
        return v;
    }
    public static MyObject f(MyObject v) {
        v = (MyObject)v.clone(); // Local copy
    }
}
```

```

        v.increment();
        return v;
    }
    public static void main(String[] args) {
        MyObject a = new MyObject(11);
        MyObject b = g(a);
        // Reference equivalence, not object equivalence:
        System.out.println("a == b: " + (a == b) +
            "\na = " + a + "\nb = " + b);
        MyObject c = new MyObject(47);
        MyObject d = f(c);
        System.out.println("c == d: " + (c == d) +
            "\nc = " + c + "\nd = " + d);
        monitor.expect(new String[] {
            "a == b: true",
            "a = 12",
            "b = 12",
            "c == d: false",
            "c = 47",
            "d = 48"
        });
    }
} ///: ~

```

首先，为了想 `clone()` 方法可以被访问，必须将其声明为 `public`。其次，作为你的 `clone()` 操作的初始化部分，应该调用基类的 `clone()`。该 `clone()` 是在 `Object` 中定义的，它是 `protected` 方法，在子类中也能访问，所以可以调用。

`Object.clone()` 能够按对象大小创建足够的内存空间，从“旧”对象到“新”对象，复制所有比特位。这被称为“逐位复制 (*bitwise copy*)”，它正是你期望的 `clone()` 方法的典型行为。但是，`Object.clone()` 在执行操作前，会先检查此类是否可克隆，即检查它是否实现了 `Cloneable` 接口。如果没有实现此接口，`Object.clone()` 会抛出 **`CloneNotSupportedException`** 异常，说明它不能被克隆。因此，你必须将 `super.clone()` 置于 `try` 块内，以捕获不应该发生的异常（因为你已经实现了 `Cloneable` 接口）。

在 `LocalCopy` 中，方法 `g()` 和 `f()` 演示了两种参数传递的区别。方法 `g()` 是传引用，它会修改外部的对象，并返回指向此外部对象的引用。方法 `f()` 则克隆参数，切断了与原对象的关联，然后就可以随意使用它了，甚至是返回新对象的引用，也不会对原对象有任何影响。注意下面这行有点古怪的语句：

```
v = (MyObject)v.clone();
```

这是在创建局部副本。为避免被这种语句迷惑，请记住，这种奇怪的编码习惯在 `Java` 中非常合宜，因为对象标识符实际就是引用。`v` 使用 `clone()` 克隆出它所指对象的副本，并返回

副本对象的 `Object` 引用（`Object.clone()` 就是这样定义的），因此必须对返回的引用做适当的类型转换。

在 `main()` 中，测试了两种参数传递方式的不同效果。请注意，重要的是 `Java` 比较对象相等的等价测试并未深入对象的内部。`==` 和 `!=` 操作符只是简单地比较引用。如果引用代表的内存地址相同，则它们指向同一个对象，因此视为“相等”。所以，该操作符测试的其实是：不同的引用是否是同一个对象的别名。

`Object.clone()` 的效果

调用 `Object.clone()` 时实际会发生什么，致使你重载 `clone()` 时必须调用 `super.clone()` 呢？`Object` 类的 `clone()` 方法负责创建正确容量的存储空间，并作“逐位复制”，由原对象复制到新对象的存储空间中。也就是说，它并不是仅仅创建存储空间，然后复制一个 `Object`。它实际是计算出即将复制的真实对象的大小（不只是基类对象，还包括源于它的对象）。由于这都发生在根类定义的 `clone()` 方法中（它并不知道谁会继承它），可以猜到，是 RTTI 机制确定要被克隆的实际对象。通过这种方式，`clone()` 方法能够创建合适的存储空间，正确地“逐位复制”你的类型。

无论怎样做，克隆过程的第一步通常都是调用 `super.clone()`。它制作出完全相同的副本，为克隆操作建立了基础。在此基础上，你可以执行对完成克隆必要的其他操作。

要确切了解“其他操作”是指什么，你首先需要知道 `Object.clone()` 为你做了什么。特别是，它是否自动将所有引用克隆至目的地？下面的例子是个试验：

```
//: appendixA:Snake.java
// Tests cloning to see if destination
// of references are also cloned.
import com.bruceeckel.simpletest.*;

public class Snake implements Cloneable {
    private static Test monitor = new Test();
    private Snake next;
    private char c;
    // Value of i == number of segments
    public Snake(int i, char x) {
        c = x;
        if(--i > 0)
            next = new Snake(i, (char)(x + 1));
    }
    public void increment() {
        c++;
        if(next != null)
            next.increment();
    }
}
```

```

public String toString() {
    String s = ":" + c;
    if(next != null)
        s += next.toString();
    return s;
}
public Object clone() {
    Object o = null;
    try {
        o = super.clone();
    } catch(CloneNotSupportedException e) {
        System.err.println("Snake can't clone");
    }
    return o;
}
public static void main(String[] args) {
    Snake s = new Snake(5, 'a');
    System.out.println("s = " + s);
    Snake s2 = (Snake)s.clone();
    System.out.println("s2 = " + s2);
    s.increment();
    System.out.println("after s.increment, s2 = " + s2);
    monitor.expect(new String[] {
        "s = :a:b:c:d:e",
        "s2 = :a:b:c:d:e",
        "after s.increment, s2 = :a:c:d:e:f"
    });
}
} ///: ~

```

`Snake` 由许多节组成，每节也是 `Snake` 类型。因此，它是个单链链表(singly linked list)。递归生成所有小节，每次递减构造器的第一个参数，到零为止。为给每节 `Snake` 赋予唯一的标记，在递归调用构造器时，`char` 类型的第二个参数会递增。

`increment()` 方法递归增加每个标记，便于你看到的变化，而 `toString()` 方法递归打印每个标记。从程序输出可以看到，`Object.clone()` 只复制了第一节 `Snake`，因此这是浅层拷贝。如果你想整条 `Snake`（每节 `Snake`）都被复制，既是深层拷贝，为此必须在重载的 `clone()` 方法中执行额外的操作。

通常，你会在可克隆类的每个子类中调用 `super.clone()`，以保证基类所有的操作（包括 `Object.clone()`）都被执行。然后，对对象中的每个引用，都明确地调用 `clone()`。否则，那些引用会被别名化，仍指向原本的对象。这与调用构造器的方式类似：基类的构造器先执行，然后是其直接继承者，依此类推，直到最末端子类的构造器。可惜 `clone()` 不是构造器，这不会自动发生。你必须自己处理这个过程。

克隆一个组合对象

在对组合对象做深层拷贝时，你会遇到一个问题。你必须假设：成员对象的 `clone()` 方法会在其引用上依次执行深层拷贝，并依此类推。这相当于一个承诺。它实际上表示，要想让深层拷贝起作用，你就必须控制所有类的代码，或者至少对深层拷贝涉及的类要足够了解，才能知道它们是否正确执行了各自的深层拷贝。

下面的例子演示了在对组合对象做深层拷贝时，你必须要做的事情：

```
//: appendixA:DeepCopy.java
// Cloning a composed object.
// {Depends: junit.jar}
import junit.framework.*;

class DepthReading implements Cloneable {
    private double depth;
    public DepthReading(double depth) { this.depth =
depth; }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch(CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return o;
    }
    public double getDepth() { return depth; }
    public void setDepth(double depth){ this.depth = depth; }
    public String toString() { return String.valueOf(depth); }
}

class TemperatureReading implements Cloneable {
    private long time;
    private double temperature;
    public TemperatureReading(double temperature) {
        time = System.currentTimeMillis();
        this.temperature = temperature;
    }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch(CloneNotSupportedException e) {
```

```

        e.printStackTrace();
    }
    return o;
}
public double getTemperature() { return temperature; }
public void setTemperature(double temperature) {
    this.temperature = temperature;
}
public String toString() {
    return String.valueOf(temperature);
}
}

```

```

class OceanReading implements Cloneable {
    private DepthReading depth;
    private TemperatureReading temperature;
    public OceanReading(double tdata, double ddata) {
        temperature = new TemperatureReading(tdata);
        depth = new DepthReading(ddata);
    }
    public Object clone() {
        OceanReading o = null;
        try {
            o = (OceanReading)super.clone();
        } catch(CloneNotSupportedException e) {
            e.printStackTrace();
        }
        // Must clone references:
        o.depth = (DepthReading)o.depth.clone();
        o.temperature =
            (TemperatureReading)o.temperature.clone();
        return o; // Upcasts back to Object
    }
    public TemperatureReading getTemperatureReading() {
        return temperature;
    }
    public void setTemperatureReading(TemperatureReading
tr){
        temperature = tr;
    }
    public DepthReading getDepthReading() { return depth; }
    public void setDepthReading(DepthReading dr) {
        this.depth = dr;
    }
}

```

```

    public String toString() {
        return "temperature: " + temperature +
            ", depth: " + depth;
    }
}

public class DeepCopy extends TestCase {
    public DeepCopy(String name) { super(name); }
    public void testClone() {
        OceanReading reading = new OceanReading(33.9,
100.5);
        // Now clone it:
        OceanReading clone = (OceanReading)reading.clone();
        TemperatureReading tr =
clone.getTemperatureReading();
        tr.setTemperature(tr.getTemperature() + 1);
        clone.setTemperatureReading(tr);
        DepthReading dr = clone.getDepthReading();
        dr.setDepth(dr.getDepth() + 1);
        clone.setDepthReading(dr);
        assertEquals(reading.toString(),
            "temperature: 33.9, depth: 100.5");
        assertEquals(clone.toString(),
            "temperature: 34.9, depth: 101.5");
    }
    public static void main(String[] args) {
        junit.textui.TestRunner.run(DeepCopy.class);
    }
} ///: ~

```

DepthReading 和 TemperatureReading 十分相似，它们都只包含基本类型。因此，它们的 clone() 方法也很简单：调用 super.clone() 然后返回结果。注意，这两个类的 clone() 代码相同。

OceanReading 由 DepthReading 和 TemperatureReading 对象组成。所以，如果要 做深层拷贝，它的 clone() 必须克隆 OceanReading 内部的所有引用。为此，super.clone() 的结果必须转型为 OceanReading 对象（因此你能够访问 depth 和 temperature 的引用）。

深层拷贝 ArrayList

我们再看看本附录先前的 Cloning.java 示例。这次 Int2 类是可克隆的，所以 ArrayList 可以被深层拷贝：

```

//: appendixa: AddingClone.java
// You must go through a few gyrations
// to add cloning to your own class.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Int2 implements Cloneable {
    private int i;
    public Int2(int ii) { i = ii; }
    public void increment() { i++; }
    public String toString() { return Integer.toString(i); }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch(CloneNotSupportedException e) {
            System.err.println("Int2 can't clone");
        }
        return o;
    }
}

```

```

// Inheritance doesn't remove cloneability:
class Int3 extends Int2 {
    private int j; // Automatically duplicated
    public Int3(int i) { super(i); }
}

```

```

public class AddingClone {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        Int2 x = new Int2(10);
        Int2 x2 = (Int2)x.clone();
        x2.increment();
        System.out.println("x = " + x + ", x2 = " + x2);
        // Anything inherited is also cloneable:
        Int3 x3 = new Int3(7);
        x3 = (Int3)x3.clone();
        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new Int2(i));
        System.out.println("v: " + v);
        ArrayList v2 = (ArrayList)v.clone();
        // Now clone each element:
    }
}

```

```

        for(int i = 0; i < v.size(); i++)
            v2.set(i, ((Int2)v2.get(i)).clone());
        // Increment all v2's elements:
        for(Iterator e = v2.iterator(); e.hasNext(); )
            ((Int2)e.next()).increment();
        System.out.println("v2: " + v2);
        // See if it changed v's elements:
        System.out.println("v: " + v);
        monitor.expect(new String[] {
            "x = 10, x2 = 11",
            "v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]",
            "v2: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]",
            "v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]"
        });
    }
} ///: ~

```

Int3 继承自 Int2，并添加了新的基本类型成员：int j。也许你认为需要重载 clone() 方法，以确保 j 也被复制，但事情并非如此。当 Int2 的 clone() 因 Int3 的 clone() 而被调用时，它又调用了 Object.clone()，后者会判断它操作的是 Int3，并且复制 Int3 对象的所有位（bit）。只要你没有向子类中添加需要克隆的引用，那么无论 clone() 定义于继承层次中多深的位置，只需调用 Object.clone() 一次，就能完成所有必要的复制。

可以看到，对 ArrayList 深层拷贝而言，以下操作是必须的：克隆了 ArrayList 之后，必须遍历 ArrayList 中的每个对象，逐一克隆。对 HashMap 做深层拷贝时，也必须做类似的操作。

此例余下部分用以显示克隆的效果：一旦对象被克隆出来，你就能够在修改它的时候，不对原始对象造成影响。

通过序列化（**serialization**）进行深层拷贝

当你在思考 Java 的对象序列化操作时（在第十二章中介绍），可以观察到，如果将对象序列化之后再将其反序列化（deserialized），那么其效果相当于克隆对象。

那么为何不用序列化操作实现深层拷贝呢？下例比较了两种方法的耗时：

```

///: appendixa:Compete.java
import java.io.*;

class Thing1 implements Serializable {}
class Thing2 implements Serializable {
    Thing1 o1 = new Thing1();
}

```

```

class Thing3 implements Cloneable {
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch(CloneNotSupportedException e) {
            System.err.println("Thing3 can't clone");
        }
        return o;
    }
}

```

```

class Thing4 implements Cloneable {
    private Thing3 o3 = new Thing3();
    public Object clone() {
        Thing4 o = null;
        try {
            o = (Thing4)super.clone();
        } catch(CloneNotSupportedException e) {
            System.err.println("Thing4 can't clone");
        }
        // Clone the field, too:
        o.o3 = (Thing3)o3.clone();
        return o;
    }
}

```

```

public class Compete {
    public static final int SIZE = 25000;
    public static void main(String[] args) throws Exception {
        Thing2[] a = new Thing2[SIZE];
        for(int i = 0; i < a.length; i++)
            a[i] = new Thing2();
        Thing4[] b = new Thing4[SIZE];
        for(int i = 0; i < b.length; i++)
            b[i] = new Thing4();
        long t1 = System.currentTimeMillis();
        ByteArrayOutputStream buf= new
        ByteArrayOutputStream();
        ObjectOutputStream o = new
        ObjectOutputStream(buf);
        for(int i = 0; i < a.length; i++)
            o.writeObject(a[i]);
    }
}

```



```

// Now get copies:
ObjectInputStream in = new ObjectInputStream(
    new ByteArrayInputStream(buf.toByteArray()));
Thing2[] c = new Thing2[SIZE];
for(int i = 0; i < c.length; i++)
    c[i] = (Thing2)in.readObject();
long t2 = System.currentTimeMillis();
System.out.println("Duplication via serialization: " +
    (t2 - t1) + " Milliseconds");
// Now try cloning:
t1 = System.currentTimeMillis();
Thing4[] d = new Thing4[SIZE];
for(int i = 0; i < d.length; i++)
    d[i] = (Thing4)b[i].clone();
t2 = System.currentTimeMillis();
System.out.println("Duplication via cloning: " +
    (t2 - t1) + " Milliseconds");
}
} ///: ~

```

Thing2 和 Thing4 都包含成员对象，因此可以做深层拷贝。有趣的是，编写 Serializable 类很容易，但要复制它们则花费更多的工作。克隆类在编写时要多做些工作，但实际复制对象时则相当简单。结果也很有趣，以下是三次运行的输出：

```

Duplication via serialization: 547 Milliseconds
Duplication via cloning: 110 Milliseconds

```

```

Duplication via serialization: 547 Milliseconds
Duplication via cloning: 109 Milliseconds

```

```

Duplication via serialization: 547 Milliseconds
Duplication via cloning: 125 Milliseconds

```

在早期版本的 JDK 中，序列化需要的时间远大于克隆（大约慢 15 倍），而且序列化的耗时波动很大。最近版本的 JDK 加快了序列化操作，其耗时显然也更稳定了。在这里，它比克隆大约慢四倍，作为克隆操作的替代方案，这个耗时已经进入了合理的范围。

向继承体系的更下层增加克隆能力

如果你创建了一个类，其基类缺省为 Object，那么它缺省是不具备克隆能力的（下一节会看到）。只要你不明确地添加克隆能力，它就不会具备。但是你可以向任意层次的子类添加克隆能力，从那层以下的子类，也就都具备了克隆能力，就像这样：

```

///: appendixa:HorrorFlick.java

```

```
// You can insert Cloneability at any level of inheritance.
package appendixa;
import java.util.*;

class Person {}
class Hero extends Person {}
class Scientist extends Person implements Cloneable {
    public Object clone() {
        try {
            return super.clone();
        } catch(CloneNotSupportedException e) {
            // This should never happen: It's Cloneable already!
            throw new RuntimeException(e);
        }
    }
}
class MadScientist extends Scientist {}

public class HorrorFlick {
    public static void main(String[] args) {
        Person p = new Person();
        Hero h = new Hero();
        Scientist s = new Scientist();
        MadScientist m = new MadScientist();
        //! p = (Person)p.clone(); // Compile error
        //! h = (Hero)h.clone(); // Compile error
        s = (Scientist)s.clone();
        m = (MadScientist)m.clone();
    }
} ///: ~
```

在类继承体系中添加克隆能力之前，编译器会阻止你的克隆操作。当 `Scientist` 类添加了克隆能力后，`Scientist` 及其所有子类就都具备了克隆能力。

为何采用此奇怪的设计？

是否这一切看起来像是一种奇怪的安排？是的，它确实奇怪。你可能想知道为何如此设计？这种设计背后有何意图？

起初，Java 被设计成为一种控制硬件设备的语言，根本没考虑到 Internet。现在，Java 作为通用性编程语言，程序员需要具备克隆对象的能力。因此，`clone()` 被添加到根类 `Object` 中，原本声明为 `public` 方法，这样你就能复制任意对象。这似乎是最方便的解决方案，但是之后呢，它会有什么危害吗？

是的，当 Java 被视为终极的 Internet 编程语言时，情况就变了。安全问题突显了出来，当然，这都是使用对象所带来的问题，因为你必定不愿意任何人都能克隆你的机密对象。所以你现在看到的设计，是在最初简单而直接的设计上，做了许多修补之后的版本：Object 中的 clone() 被声明为 protected。你必须重载它、实现 Cloneable 接口、并做异常处理。

值得注意的是，只有真正需要调用 Object 的 clone() 方法时，你才必须实现 Cloneable 接口，因为在运行期会检查你的类是否实现了 Cloneable 接口。不过，为了使具备克隆能力的对象保持一致性（毕竟 Cloneable 是空的），即使不调用 Object 的 clone() 方法，你仍然应该实现此接口。

控制克隆能力

为了移除克隆能力，你也许会建议将 clone() 方法声明为 private。但是这行不通，因为对于基类的方法，无法在子类中削弱其访问能力。然而，我们必须有能力控制某个对象是否可以被克隆。对此你可能会有以下态度：

1. 不关心。你并不做任何克隆操作，即使你的类不可克隆，但是只要愿意，就能向其子类添加克隆能力。这只有在缺省的 Object.clone() 能够合理地处理类中所有属性时才起作用。
2. 支持 clone()。按照标准的惯例：实现 Cloneable 接口、重载 clone() 方法。在重载的 clone() 中，调用 super.clone()，并捕获所有异常（所以你重载的 clone() 不会抛出异常）。
3. 有条件地支持克隆。如果你的类（例如容器类）包含其他对象的引用，它们不一定是可克隆的，但你的 clone() 方法应该试着克隆它们，如果抛出异常，只需将异常传给程序员。例如，考虑一种特殊的 ArrayList，它需要克隆自己包含的所有对象。编写这样的 ArrayList 时，你并不知道客户端程序员会向你的 ArrayList 存入何种类型的对象，因此你也不知道它们能否被克隆。
4. 不实现 Cloneable 接口，但是以 protected 方式重载 clone() 方法，为所有属性创建正确的复制行为。于是该类的任何子类，都可以重载 clone() 并调用 super.clone() 产生正确的复制行为。注意，你的 clone() 可以（并且应该）调用 super.clone()，即使 super.clone() 预期的是个 Cloneable 对象（否则会抛出异常）。没人会直接对你的类的对象调用 clone()，只能通过其子类才行，而要想让它正常工作，其子类必须实现 Cloneable 接口。
5. 不实现 Cloneable 接口，重载 clone() 使之抛出异常，以阻止克隆操作。只有此类的所有子类，都在各自的 clone() 中调用 super.clone()，这种阻止克隆的方法才起作用。否则，程序员还是有可能绕开它。
6. 将你的类声明为 final 以阻止克隆。如果它的任何父类（祖先类）都没有重载 clone()，那么此方法就行不通了。如果父类重载了 clone()，那么让你的类再次重载 clone()，并抛出 CloneNotSupportedException。将类声明为 final，是唯一有保证的防止克隆的方法。此外，当处理机密对象，或需要控制对象的数量时，应该将所有构造器都设置为 private，然后提供一个（或多个）创建对象的专用方法。这些方法可以限制创建对象的数量和条件。（对此有一个特别的例子：singleton 模式。可以从 www.BruceEckel.com 上的《Thinking in Patterns (with Java)》中找到。）

下面的例子演示了各种方法，以实现克隆能力，然后“关闭”继承体系下层子类的克隆能力：

```
//: appendixA:CheckCloneable.java
// Checking to see if a reference can be cloned.
import com.bruceeckel.simpletest.*;

// Can't clone this because it doesn't override clone():
class Ordinary {}

// Overrides clone, but doesn't implement Cloneable:
class WrongClone extends Ordinary {
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone(); // Throws exception
    }
}

// Does all the right things for cloning:
class IsCloneable extends Ordinary implements Cloneable {
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

// Turn off cloning by throwing the exception:
class NoMore extends IsCloneable {
    public Object clone() throws CloneNotSupportedException
    {
        throw new CloneNotSupportedException();
    }
}

class TryMore extends NoMore {
    public Object clone() throws CloneNotSupportedException
    {
        // Calls NoMore.clone(), throws exception:
        return super.clone();
    }
}

class BackOn extends NoMore {
    private BackOn duplicate(BackOn b) {
        // Somehow make a copy of b and return that copy.
    }
}
```

```

        // This is a dummy copy, just to make the point:
        return new BackOn();
    }
    public Object clone() {
        // Doesn't call NoMore.clone():
        return duplicate(this);
    }
}

// You can't inherit from this, so you can't override
// the clone method as you can in BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
    private static Test monitor = new Test();
    public static Ordinary tryToClone(Ordinary ord) {
        String id = ord.getClass().getName();
        System.out.println("Attempting " + id);
        Ordinary x = null;
        if(ord instanceof Cloneable) {
            try {
                x = (Ordinary)((IsCloneable)ord).clone();
                System.out.println("Cloned " + id);
            } catch(CloneNotSupportedException e) {
                System.err.println("Could not clone " + id);
            }
        } else {
            System.out.println("Doesn't implement Cloneable");
        }
        return x;
    }
}

public static void main(String[] args) {
    // Upcasting:
    Ordinary[] ord = {
        new IsCloneable(),
        new WrongClone(),
        new NoMore(),
        new TryMore(),
        new BackOn(),
        new ReallyNoMore(),
    };
    Ordinary x = new Ordinary();
    // This won't compile; clone() is protected in Object:
    //! x = (Ordinary)x.clone();
}

```

```

// Checks first to see if a class implements Cloneable:
for(int i = 0; i < ord.length; i++)
    tryToClone(ord[i]);
monitor.expect(new String[] {
    "Attempting IsCloneable",
    "Cloned IsCloneable",
    "Attempting WrongClone",
    "Doesn't implement Cloneable",
    "Attempting NoMore",
    "Could not clone NoMore",
    "Attempting TryMore",
    "Could not clone TryMore",
    "Attempting BackOn",
    "Cloned BackOn",
    "Attempting ReallyNoMore",
    "Could not clone ReallyNoMore"
});
}
} ///: ~

```

第一个类，**Ordinary**，代表我们在本书中常见的类型：它不是“关闭”克隆能力，而是不支持也不阻止克隆。但是，如果你有一个 **Ordinary** 子类的对象，其引用向上转型为 **Ordinary** 后，你无法分辨它是否可克隆。

WrongClone 类演示了实现克隆机制的错误方式。它以 **public** 方式重载了 **Object.clone()**，但是没有实现 **Cloneable**，因此调用 **super.clone()** 时（最终会调用 **Object.clone()**），会抛出 **CloneNotSupportedException**，所以无法克隆。

IsCloneable 执行了所有正确的操作：重载 **clone()** 方法、实现 **Cloneable** 接口。然而，它的 **clone()** 方法，以及上例中的其他方法都没有捕获 **CloneNotSupportedException**，而是将它传给方法的调用者，后者必须用 **try-catch** 区块包住 **clone()**。在你自己的 **clone()** 方法中，通常会在 **clone()** 内捕获 **CloneNotSupportedException**，而不是将它传出。如你所见，本例是为演示才将异常传递出来。

NoMore 类尝试“关闭”克隆能力，采用了 Java 设计者建议的方式：在子类的 **clone()** 中抛出 **CloneNotSupportedException**。当 **TryMore** 类的 **clone()** 调用 **super.clone()** 时，会抛出异常，阻止克隆。

但是，在重载的 **clone()** 方法中，如果程序员不按“正确”方法调用 **super.clone()** 又怎么样呢？在 **BackOn** 中可以看到这是如何发生的。**BackOn** 使用独立的 **duplication()** 复制当前对象，在 **clone()** 中它取代了 **super.clone()**。这不会抛出异常，而且新的类也是可克隆的。因此，无法依赖抛出异常来防止克隆能力。**ReallyNoMore** 示范了唯一不会有问题的解决方案。类声明为 **final**，就不可以被继承了。这意味着，如果在 **final** 的类中，**clone()** 方法抛出异常，由于它不会被子类修改，所以肯定能阻止克隆。（不能从继承体系任意层

的类直接显示明确地调用 `Object.clone()`，只能调用 `super.clone()` 访问其直接父类。）因此，如果你的对象需要考虑安全因素，可以将类声明为 `final`。

`CheckCloneable` 类的第一个方法 `tryToClone()` 以 `Ordinary` 对象为参数，使用 `instanceof` 检查是否可以被克隆。如果可以，将对象转型为 `IsCloneable`，调用 `clone()`，再将结果类型转回给 `Ordinary`，其间会捕获任何异常。注意，这里使用了运行期类型识别机制（RTTI，参见第十一章）打印类名，以便于你看到程序进展。

在 `main()` 中，创建了不同类型的 `Ordinary` 对象，向上转型为 `Ordinary` 后存贮在数组中。这之后的两行代码，创建了一个平凡的 `Ordinary` 对象，并试图克隆它。然而，这行代码不能通过编译，因为 `clone()` 在 `Object` 中是 `protected` 方法。余下的代码遍历数组，尝试克隆每个对象，并报告每次克隆成功与否。

现在做个小结，如果你希望一个类可以被克隆：

1. 实现 `Cloneable` 接口。
2. 重载 `clone()`。
3. 在你的 `clone()` 中调用 `super.clone()`。
4. 在你的 `clone()` 中捕获异常。

做到这些即可获得令人满意的效果。

拷贝构造器

克隆机制的建立似乎是很复杂的过程，也许应该有别的解决方案。如前所述，使用序列化是一种方法。你可能还会想到另一种方法（特别是，如果你是 C++ 程序员的话），做一个特殊的构造器，它的工作就是复制对象。在 C++ 中这称为拷贝构造器（*copy constructor*）。乍看起来，这似乎是显而易见的解决方案，可实际上这行不通。见下例：

```
//: appendixA:CopyConstructor.java
// A constructor for copying an object of the same
// type, as an attempt to create a local copy.
import com.bruceeckel.simpletest.*;
import java.lang.reflect.*;

class FruitQualities {
    private int weight;
    private int color;
    private int firmness;
    private int ripeness;
    private int smell;
    // etc.
    public FruitQualities() { // Default constructor
        // Do something meaningful...
```

```

    }
    // Other constructors:
    // ...
    // Copy constructor:
    public FruitQualities(FruitQualities f) {
        weight = f.weight;
        color = f.color;
        firmness = f.firmness;
        ripeness = f.ripeness;
        smell = f.smell;
        // etc.
    }
}

class Seed {
    // Members...
    public Seed() { /* Default constructor */ }
    public Seed(Seed s) { /* Copy constructor */ }
}

class Fruit {
    private FruitQualities fq;
    private int seeds;
    private Seed[] s;
    public Fruit(FruitQualities q, int seedCount) {
        fq = q;
        seeds = seedCount;
        s = new Seed[seeds];
        for(int i = 0; i < seeds; i++)
            s[i] = new Seed();
    }
    // Other constructors:
    // ...
    // Copy constructor:
    public Fruit(Fruit f) {
        fq = new FruitQualities(f.fq);
        seeds = f.seeds;
        s = new Seed[seeds];
        // Call all Seed copy-constructors:
        for(int i = 0; i < seeds; i++)
            s[i] = new Seed(f.s[i]);
        // Other copy-construction activities...
    }
    // To allow derived constructors (or other

```



```

// methods) to put in different qualities:
protected void addQualities(FruitQualities q) {
    fq = q;
}
protected FruitQualities getQualities() {
    return fq;
}
}

```

```

class Tomato extends Fruit {
    public Tomato() {
        super(new FruitQualities(), 100);
    }
    public Tomato(Tomato t) { // Copy-constructor
        super(t); // Upcast for base copy-constructor
        // Other copy-construction activities...
    }
}

```

```

class ZebraQualities extends FruitQualities {
    private int stripedness;
    public ZebraQualities() { // Default constructor
        super();
        // do something meaningful...
    }
    public ZebraQualities(ZebraQualities z) {
        super(z);
        stripedness = z.stripedness;
    }
}

```

```

class GreenZebra extends Tomato {
    public GreenZebra() {
        addQualities(new ZebraQualities());
    }
    public GreenZebra(GreenZebra g) {
        super(g); // Calls Tomato(Tomato)
        // Restore the right qualities:
        addQualities(new ZebraQualities());
    }
    public void evaluate() {
        ZebraQualities zq = (ZebraQualities)getQualities();
        // Do something with the qualities
        // ...
    }
}

```

```
}  
}
```

```
public class CopyConstructor {  
    private static Test monitor = new Test();  
    public static void ripen(Tomato t) {  
        // Use the "copy constructor":  
        t = new Tomato(t);  
        System.out.println("In ripen, t is a " +  
            t.getClass().getName());  
    }  
    public static void slice(Fruit f) {  
        f = new Fruit(f); // Hmmmm... will this work?  
        System.out.println("In slice, f is a " +  
            f.getClass().getName());  
    }  
    public static void ripen2(Tomato t) {  
        try {  
            Class c = t.getClass();  
            // Use the "copy constructor":  
            Constructor ct = c.getConstructor(new Class[] { c });  
            Object obj = ct.newInstance(new Object[] { t });  
            System.out.println("In ripen2, t is a " +  
                obj.getClass().getName());  
        }  
        catch (Exception e) { System.out.println(e); }  
    }  
    public static void slice2(Fruit f) {  
        try {  
            Class c = f.getClass();  
            Constructor ct = c.getConstructor(new Class[] { c });  
            Object obj = ct.newInstance(new Object[] { f });  
            System.out.println("In slice2, f is a " +  
                obj.getClass().getName());  
        }  
        catch (Exception e) { System.out.println(e); }  
    }  
    public static void main(String[] args) {  
        Tomato tomato = new Tomato();  
        ripen(tomato); // OK  
        slice(tomato); // OOPS!  
        ripen2(tomato); // OK  
        slice2(tomato); // OK  
        GreenZebra g = new GreenZebra();  
    }  
}
```

```

        ripen(g); // OOPS!
        slice(g); // OOPS!
        ripen2(g); // OK
        slice2(g); // OK
        g.evaluate();
        monitor.expect(new String[] {
            "In ripen, t is a Tomato",
            "In slice, f is a Fruit",
            "In ripen2, t is a Tomato",
            "In slice2, f is a Tomato",
            "In ripen, t is a Tomato",
            "In slice, f is a Fruit",
            "In ripen2, t is a GreenZebra",
            "In slice2, f is a GreenZebra"
        });
    }
} ///: ~

```

乍看之下有点奇怪，水果（fruit）当然有品质（qualities），为什么不将那些品质作为 Fruit 类的属性成员直接放到 Fruit 类中呢？有两个可能的原因。

第一个原因是，你希望更容易地插入或修改品质。注意，Fruit 有一个 `protected addQualities()` 方法，允许子类调用。（你也许会想到，符合逻辑的做法是写一个 `protected` 的 Fruit 构造器，它以 `FruitQualities` 为参数。但是构造器不能继承，它在第二层或更底层的子类中就不可用了。）通过将水果品质做成独立的类 `FruitQualities`，然后使用组合，获得了更好的灵活性，可以在特殊的 Fruit 对象的生命周期中间改变其品质。

`FruitQualities` 独立成为对象的第二个原因是，你可以通过继承和多态机制添加新的品质，或是改变其行为。请注意 `GreenZebra`（我种过的一种番茄，很漂亮），它的构造器调用 `addQualities()`，并且传入一个 `ZebraQualities` 对象，`ZebraQualities` 继承自 `FruitQualities`，因此可以在基类中转型为 `FruitQualities`。当然，`GreenZebra` 使用 `FruitQualities` 时，必须向下转型为正确类型（如 `evaluate()` 中所见），不过它知道正确的类型是 `ZebraQualities`。

还有 `Seed` 类，`Fruit`（定义为携带有自己的种子）⁴ 包含一个 `Seed` 数组。

最后，注意每个类都有拷贝构造器，它们必须负责调用基类和成员对象的拷贝构造器，以达到深层拷贝的效果。`CopyConstructor` 类测试拷贝构造器。`ripen()` 方法接收 `Tomato` 参数，对其执行拷贝构造器，以生成对象副本。

```
t = new Tomato(t);
```

而 `slice()` 以更通用的 `Fruit` 对象为参数，对它进行复制。

⁴ 除了可怜的鳄梨（avocado），被改良得只是“肥大”罢了。

```
f = new Fruit(f);
```

在 `main()` 中测试了各种不同的 `Fruit`。从程序输出可以看到问题所在。在 `slice()` 内，对 `Tomato` 做拷贝构造之后，其结果不再是 `Tomato` 对象，而只是 `Fruit`。它丢失了所有 `Tomato` 特有的信息。而测试 `GreenZebra` 时，`ripen()` 和 `slice()` 将其分别变成 `Tomato` 和 `Fruit`。因此，很不幸，在 `Java` 中使用拷贝构造器创建对象的局部拷贝是不可行的。

为什么在 C++ 中可行？

拷贝构造器是 `C++` 的基础功能，因为它自动生成对象的局部拷贝。而前例证明这在 `Java` 中不可行。为什么？在 `Java` 中，我们操控的都是引用；而在 `C++` 中，不但有类似引用的东西，甚至可以直接传递对象。`C++` 中拷贝构造器的目的是：通过传值的方式传递对象时，复制此对象。所以此机制在 `C++` 中运作的很好。但你应该记住，它在 `Java` 中行不通，所以不要使用。

只读类

虽然在适当的情况下，`clone()` 生成的局部拷贝可以满足我们的需求，但这也是个典型，它强制程序员（`clone()` 方法的作者）必须负责避免别名的负面效应。当你开发的类库具有通用目的、被广泛使用，使你不能假设你的类总是在恰当的位置被克隆时，又会怎么样吗？或者更可能的情况是，如果你为了效率而允许出现别名——为了避免不必要的复制对象，但你不希望因为别名而产生负面影响，这是又会怎样呢？

一种解决方法是创建“恒常对象（*immutable objects*）”，它属于只读类。在你的类中，不要定义会修改对象内部状态的方法。对于这样的类，出现别名也不会有影响，因为你只能读取对象的内部状态，即使很多代码都读取同一个对象，也没有问题。

作为恒常对象的简单示例，可以参考 `Java` 标准类库中所有基本类型的“包装”类。你可能已经发现了，如果你想在容器中存储 `int`，例如 `ArrayList`（只接受 `Object` 引用），可以先将 `int` 用标准类库的 `Integer` 类包装：

```
//: appendixA: ImmutableInteger.java
// The Integer class cannot be changed.
import java.util.*;

public class ImmutableInteger {
    public static void main(String[] args) {
        List v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new Integer(i));
        // But how do you change the int inside the Integer?
    }
} ///: ~
```

`Integer` 类（其他“包装”类也同样）已很简单的方式实现了“恒常性”：它没有让你去修改对象内容的方法。

如果你确实需要一个对象，它包含基本类型成员，并且此成员可以修改。你就必须创建自己的类。幸运的是，这很简单。下面的类采用了 `JavaBean` 的命名惯例：

```
//: appendixA: MutableInteger.java
// A changeable wrapper class.
import com.bruceeckel.simpletest.*;
import java.util.*;

class IntValue {
    private int n;
    public IntValue(int x) { n = x; }
    public int getValue() { return n; }
    public void setValue(int n) { this.n = n; }
    public void increment() { n++; }
    public String toString() { return Integer.toString(n); }
}

public class MutableInteger {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        List v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new IntValue(i));
        System.out.println(v);
        for(int i = 0; i < v.size(); i++)
            ((IntValue)v.get(i)).increment();
        System.out.println(v);
        monitor.expect(new String[] {
            "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]",
            "[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]"
        });
    }
} ///: ~
```

如果不需要保持私有性，缺省初始化为零就可以满足要求（那就不需要构造器了），并且你不关心打印问题（那就不需要 `toString()` 了），那么 `IntValue` 还可以更简化：

```
class IntValue { int n; }
```

取出元素并对其进行类型转换操作，虽然显得有点笨拙，但那是 `ArrayList` 的功能，而不是 `IntValue` 的。

创建只读类

你可以创建自己的只读类。见下例：

```
//: appendixA:Immutable1.java
// Objects that cannot be modified are immune to aliasing.
import com.bruceeckel.simpletest.*;

public class Immutable1 {
    private static Test monitor = new Test();
    private int data;
    public Immutable1(int initVal) {
        data = initVal;
    }
    public int read() { return data; }
    public boolean nonzero() { return data != 0; }
    public Immutable1 multiply(int multiplier) {
        return new Immutable1(data * multiplier);
    }
    public static void f(Immutable1 i1) {
        Immutable1 quad = i1.multiply(4);
        System.out.println("i1 = " + i1.read());
        System.out.println("quad = " + quad.read());
    }
    public static void main(String[] args) {
        Immutable1 x = new Immutable1(47);
        System.out.println("x = " + x.read());
        f(x);
        System.out.println("x = " + x.read());
        monitor.expect(new String[] {
            "x = 47",
            "i1 = 47",
            "quad = 188",
            "x = 47"
        });
    }
} ///: ~
```

所有数据都是 `private`，而且你看到了，没有要去修改这些数据的 `public` 方法。实际上，虽然 `multiply()` 方法修改了对象，但它创建了一个新的 `Immutable1` 对象，并没有修改原始对象。

f()方法以 Immutable1 对象为参数，对其执行了各种操作，而 main()中的输出说明，f()没有修改 x。因此，x 对象可以有很多别名，也不会造成伤害，因为 Immutable1 类的设计保证了对象不会被修改。

恒常性（immutability）的缺点

创建恒常的类，初看起来似乎是一种优雅解决方案。然而，无论何时当你需要一个被修改过的此类的对象的时候，必须要承受创建新对象的开销，也会更频繁地引发垃圾回收。对某些类而言，这不成问题，但对另一些类（例如 String 类），其代价可能昂贵得让人不得不禁止这么做。

解决之道是创建一个可以被修改的伴随类（companion class）。当你需要做大量修改动作时，可以转为使用可修改的伴随类，修改操作完毕后，再转回恒常类。

前面的例子在修改之后能够展示这种方法：

```
//: appendixA:Immutable2.java
// A companion class to modify immutable objects.
import com.bruceeckel.simpletest.*;

class Mutable {
    private int data;
    public Mutable(int initVal) { data = initVal; }
    public Mutable add(int x) {
        data += x;
        return this;
    }
    public Mutable multiply(int x) {
        data *= x;
        return this;
    }
    public Immutable2 makeImmutable2() {
        return new Immutable2(data);
    }
}

public class Immutable2 {
    private static Test monitor = new Test();
    private int data;
    public Immutable2(int initVal) { data = initVal; }
    public int read() { return data; }
    public boolean nonzero() { return data != 0; }
    public Immutable2 add(int x) {
        return new Immutable2(data + x);
    }
}
```

```

    }
    public Immutable2 multiply(int x) {
        return new Immutable2(data * x);
    }
    public Mutable makeMutable() {
        return new Mutable(data);
    }
    public static Immutable2 modify1(Immutable2 y) {
        Immutable2 val = y.add(12);
        val = val.multiply(3);
        val = val.add(11);
        val = val.multiply(2);
        return val;
    }
    // This produces the same result:
    public static Immutable2 modify2(Immutable2 y) {
        Mutable m = y.makeMutable();
        m.add(12).multiply(3).add(11).multiply(2);
        return m.makeImmutable2();
    }
    public static void main(String[] args) {
        Immutable2 i2 = new Immutable2(47);
        Immutable2 r1 = modify1(i2);
        Immutable2 r2 = modify2(i2);
        System.out.println("i2 = " + i2.read());
        System.out.println("r1 = " + r1.read());
        System.out.println("r2 = " + r2.read());
        monitor.expect(new String[] {
            "i2 = 47",
            "r1 = 376",
            "r2 = 376"
        });
    }
} ///:~

```

Immutable2 内的一些方法，与之前相同，每当需要做修改时，就生成一个新对象，以保证原对象的恒常性。例如 `add()` 和 `multiply()` 方法。Immutable2 的伴随类是 Mutable，它也有 `add()` 和 `multiply()` 方法，但它们是直接修改 Mutable 对象，而不是生成新对象。此外，Mutable 有一个方法，它使用自己的数据生成一个 Immutable2 对象，反之亦然。

两个 static 方法 `modify1()` 和 `modify2()`，演示了两种不同的方法，得到同样的结果。在 `modify1()` 中，所有的工作都在 Immutable2 类中完成。可以看到，在这个过程中创建了四个新的 Immutable2 对象。（每次对 `val` 重新赋值，前一个对象就成为了垃圾。）

在 `modify2()` 中可以看到，第一个动作是接收 `Immutable2 y`，并且由 `y` 生成 `Mutable` 对象。（这与先前调用 `clone()` 相似，但是创建了不同类型的对象。）然后使用 `Mutable` 对象，无需创建新对象即可执行大量的修改操作。最后，转回 `Immutable2` 对象。这个过程产生了两个新对象（`Mutalbe` 对象，以及作为结果的 `Immutable2` 对象），而不是四个。

当下列情况发生时，此方法十分有用：

1. 你需要恒常的对象，而且
2. 你经常需要做大量的修改，或者
3. 创建新的恒常对象代价昂贵。

恒常的 `String`

考虑下面的代码：

```
//: appendixa:Stringer.java
import com.bruceeckel.simpletest.*;

public class Stringer {
    private static Test monitor = new Test();
    public static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = new String("howdy");
        System.out.println(q); // howdy
        String qq = upcase(q);
        System.out.println(qq); // HOWDY
        System.out.println(q); // howdy
        monitor.expect(new String[] {
            "howdy",
            "HOWDY",
            "howdy"
        });
    }
} ///: ~
```

当 `q` 传递给 `upcase()` 时，其实是传入 `q` 的引用的副本。引用指向的对象仍然在它原来的位置上。传递引用时，即复制了引用。

从 `upcase()` 的定义可以看到，传入的引用名为 `s`，只在 `upcase()` 运行时它才存在。一旦 `upcase()` 执行完毕，局部引用 `s` 也就消失了。将原始字符串转为大写字符后，`upcase()` 返回此结果。当然，它其实是返回指向结果的引用。不过，它返回的引用是指向一个新对象，原先的 `q` 被放在一边。这是怎么发生的呢？

隐式的常量

如果你这么写：

```
String s = "asdf";  
String x = Stringer.upcase(s);
```

你真的想用 `upcase()` 方法修改参数吗？通常，你不会这样。因为对于使用参数的方法而言，参数通常只是提供信息，很少需要修改。这是很重要的保证，它使得代码易于编写和理解。

在 C++ 中，此“保证”的可用性很重要，以至于 C++ 添加了 `const` 这个特殊的关键字，让程序员确保一个引用（C++ 中的指针或引用）不可以被用来修改源对象。不过 C++ 程序员必须非常细心，记住使用 `const` 的每一处。这很容易令人混淆且容易忘记。

重载 '+' 与 StringBuffer

`String` 类的对象被设计为恒常的，并使用了前面介绍的伴随类技术。如果查看 JDK 文档中的 `String` 类（稍后会对此进行总结），你会发现，类中每个设计修改 `String` 的方法，在修改的过程中，确实生成并返回了一批新的 `String` 对象。最初的 `String` 并没有受到影响。C++ 的 `const` 提供由编译器支持的对象恒常性，Java 没有这样的功能。想要获得恒常对象，你必须自己动手，就像 `String` 那样。

由于 `String` 对象是恒常的，对某个 `String` 可以随意取多个别名。因为它是只读的，任何引用也不可能修改该对象，也就不会影响其他引用。所以，只读对象很好地解决了别名问题。

这似乎可以解决所有问题。每当需要修改对象时，就创建一堆修改过的新版对象，如同 `String` 那样。然而，对某些操作而言，这太没有效率了。`String` 的重载操作符 '+' 就是个重要的例子。重载是指，对特定的类，'+' 被赋予额外的含义。（为 `String` 重载的 '+' 和 '+='，是 Java 唯一重载的操作符，而且 Java 不允许程序员重载其他操作符。）⁵

使用 `String` 对象时，'+' 用来连接 `String` 对象：

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

你可以想象它是如何工作的。`String "abc"` 可能有一个 `append()` 方法，它生成了一个连接了 "abc" 和 `foo` 的新的 `String` 对象。新的 `String` 连接 "def" 之后，生成另一个新的 `String`，依此类推。

⁵ C++ 允许程序员任意重载操作符。因为这通常是很复杂的过程（参见《*Thinking in C++, 2nd edition*》第十章，Prentice Hall 出版社，2000 年），Java 设计者认为这是“糟糕的”功能，不应该包括在 Java 中。它其实并没有那么糟糕，具有讽刺意味的是，比起 C++ 来，在 Java 中使用操作符重载容易多了。从 Python 中可以观察到这一点（参考 www.Python.org），它具有垃圾回收器和简单易行的操作符重载机制。

这当然可以运行，但它需要大量的中间 `String` 对象，才能生成最终的新 `String`，而那些中间结果需要作垃圾回收。我怀疑 `Java` 的设计者起先就是这么做的（这是软件设计中的一个教训，你无法知道所有事情，直到形成代码，并运作起来）。我猜他们发现了这种做法有着难以忍受的低效率。

解决之道是可变的伴随类，类似前面演示的例子。`String` 的伴随类称作 `StringBuffer`，在计算某些表达式，特别是 `String` 对象使用重载过的 '+' 和 '+=' 时，编译器会自动创建 `StringBuffer`。下面的例子说明其中发生了什么：

```
//: appendixA:ImmutableStrings.java
// Demonstrating StringBuffer.
import com.bruceeckel.simpletest.*;

public class ImmutableStrings {
    private static Test monitor = new Test();
    public static void main(String[] args) {
        String foo = "foo";
        String s = "abc" + foo + "def" + Integer.toString(47);
        System.out.println(s);
        // The "equivalent" using StringBuffer:
        StringBuffer sb =
            new StringBuffer("abc"); // Creates String!
        sb.append(foo);
        sb.append("def"); // Creates String!
        sb.append(Integer.toString(47));
        System.out.println(sb);
        monitor.expect(new String[] {
            "abcfooodef47",
            "abcfooodef47"
        });
    }
} ///: ~
```

在生成 `String s` 的过程中，编译器使用 `sb` 执行了大致与下面的工作等价的代码：创建一个 `StringBuffer`，使用 `append()` 向此 `StringBuffer` 对象直接添加新的字符串（而不是每次制作一个新的副本）。虽然这种方法更高效，但是对于引号括起的字符串，例如 `"abc"` 和 `"def"`，它并不起作用，编译器会将其转为 `String` 对象。所以，尽管 `StringBuffer` 提供了更好的效率，可能仍会产生超出你预期数量的对象。

String 和 StringBuffer 类

下面总结了 `String` 与 `StringBuffer` 都可用的方法，你可以感受到与它们交互的方式。表中并未包含所有方法，只包含了与本讨论相关的重要方法。重载的方法被置于单独一列。

首先是 String 类:

方法	参数, 重载	用途
Constructor	重载: 缺省、空参数、 String 、 StringBuffer 、 char 数组、 byte 数组.	创建 String 对象。
length()		String 的字符数。
charAt()	int 索引	在 String 中指定位置的字符。
getChars() , getBytes()	复制源的起点与终点、复制的目标数组、目标数组的索引。	复制 char 或 byte 到外部数组。
toCharArray()		生成 char[] , 包含 String 的所有字符。
equals() , equals-IgnoreCase()	做比较的 String	两个 String 的等价测试。
compareTo()	做比较的 String	根据词典顺序比较 String 与参数, 结果为负值、零、或正值。大小写有别。
regionMatches()	当前 String 的偏移位置、另一个 String 、及其偏移、要比较的长度。重载增加了“忽略大小写”。	返回 boolean , 代表指定区域是否相匹配。
startsWith()	测试起始 String 。重载增加了参数的偏移。	返回 boolean , 代表是否此 String 以参数为起始。
endsWith()	测试后缀 String 。	返回 boolean , 代表是否此 String 以参数为后缀。
indexOf() , lastIndexOf()	重载: char 、 char 和起始索引、 String 、 String 和起始索引。	如果当前 String 中不包含参数, 返回 -1, 否则返回参数在 String 中的位置索引。 LastIndexOf() 由末端反向搜索。
substring()	重载: 起始索引、起始索引和终止索引	返回新的 String 对象, 包含特定的字符集。
concat()	要连接的 String	返回新 String 对象, 在原 String 后追加参数字符。
replace()	搜索的旧字符, 用来替代的新字符	返回指定字符被替换后的新 String 对象。如果没有发生替换, 返回原 String 。
toLowerCase()		返回所有字母大小写修改后

toUpperCase()		的新 String 对象。如果没有改动则返回原 String 。
trim()		去除两端的空白字符后，返回新 String 。如果没有改变，则返回原 String 。
valueOf()	重载: Object 、 char[] 、 char[] 和偏移和长度、 boolean 、 char 、 int 、 long 、 float 、 double 。	返回一个 String ，内含参数表示的字符。
intern()		为每一个唯一的字符序列生成一个且仅生成一个 String 引用。

可以看到,当必须修改字符串的内容时,**String** 的每个方法都会谨慎地返回一个新的 **String** 对象。还要注意, 如果内容不需要修改, 方法会返回指向源 **String** 的引用。这节省了存储空间与开销。

以下是 **StringBuffer** 类:

方法	参数, 重载	用途
Constructor	重载: 空参数、要创建的缓冲区长度、 String 的来源。	创建新的 StringBuffer 对象。
toString()		由此 StringBuffer 生成 String 。
length()		StringBuffer 中的字符个数。
capacity()		返回当前分配的空间大小。
ensureCapacity()	代表所需容量的整数	要求 StringBuffer 至少有所需的空间。
setLength()	代表缓冲区中字符串长度的整数	截短或扩展原本的字符串。如果扩展, 以 null 填充新增的空间。
charAt()	代表所需元素位置的整数。	返回 char 在缓冲区中的位置。
setCharAt()	代表所需元素位置的整数, 和新的 char 值	修改某位置上的值。
getChars()	复制源的起点与终点、复制的目的端数组、目的端数组的索引。	复制 char 到外围数组。没有 String 中的 getBytes() 。
append()	重载: Object 、 String 、	参数转为字符串, 然后追加

	char[]、char[] 和偏移和长度、 boolean、char、int、long、float、double .	到当前缓冲区的末端,如果必要,缓冲区会扩大。
insert()	被重载过,第一个参数为插入起始点的偏移值: Object、String、char[]、boolean、char、int、long、float、double .	第二个参数转为字符串,插入到当前缓冲区的指定位置。如果必要,缓冲区会扩大。
reverse()		逆转缓冲区内字符的次序。

最常用的方法是 `append()`, 当计算包含 '+' 和 '+=' 操作符的 `String` 表达式时, 编译器会使用它。`Insert()` 方法有类似的形式, 这两个方法都会在缓冲区中进行大量操作, 而不需创建新对象。

String 是特殊的

到目前为止, 你已经看到了, `String` 类不同于 Java 中一般的类。`String` 有很多特殊之处, 它不仅仅只是一个 Java 内置的类, 它已经成为 Java 的基础。而且事实上, 双引号括起的字符串都被编译器转换为 `String` 对象, 还有专门重载的 '+' 和 '+=' 操作符。在本附录中, 你还能看到其他特殊之处: 使用伴随类 `StringBuffer` 精心建构的恒常性, 以及编译器中的一些额外的魔幻式的功能。

总结

在 Java 中, 所有对象标识符都是引用, 而且所有对象都在堆 (heap) 上创建, 只有当对象不再使用的情况下, 垃圾回收器才工作。所有这些都改变了对对象的操作方式, 特别是传递与返回对象。例如, 在 C 和 C++ 中, 如果在方法中要初始化一块存储空间, 可能需要用户向方法传入那块存储空间的地址。否则, 你必须为谁负责回收那块存储空间而操心。如此, 接口和对此方法的理解将变得更复杂了。但是在 Java 中, 你永远也无需担心承担此责任, 也不用操心需要某个对象时它是否存在。因为 Java 会帮你分担。你可以在需要对象时 (立刻) 创建它, 而无需因为要为对象负责而操心传递的技巧。你只需简单地传递引用。有时, 这种简化微不足道, 有时则令人难以置信。

所有这些神奇, 带来了两个缺点:

1. 由于额外的内存管理, 你总会付出效率上的代价 (虽然代价可能很小), 而且在程序耗时上总会有细微的不确定性 (因为当内存不足时, 垃圾回收器会强行介入)。对大多数应用程序而言, 是好处大于缺点, 而且还有专门提高程序运行速度的 `hotspot` 技术, 使之不再是个大问题。
2. 别名效应: 有时你会意外地赋予同一个对象两个引用, 只有当两个引用以为指向了不同的对象时, 这才会成为问题。这是需要你多注意的地方, 必要时, 使用 `clone()` 或者复制对象以防止其他引用对出乎意料的改变感到惊讶。另一

种情况是，你为了效率而支持别名，创建恒常的对象，其操作可返回同类型的（或不同类型）新对象，但永远不会修改源对象，所以此对象的任何别名都不会改变对象。

有些人认为Java的克隆机制是一份修修补补的设计，永远都不应使用，所以他们实现自己版本的克隆⁶，而且永不调用`Object.clone()`方法，如此，就消除了实现`Cloneable`以及捕获`CloneNotSupportedException`异常的需求。这肯定是合理的方法，而且因为标准Java库很少支持`clone()`，所以它显然是安全的。

练习

所选习题的答案都可以在《The Thinking in Java Annotated Solution Guide》这本书的电子文档中找到，你可以花少量的钱从www.BruceEckel.com处获取此文档。

1. 论证别名的第二个层次。写一个方法，它以对象的引用为参数，但是不修改此对象。不过，此方法调用第二个方法，向其传递这个引用，而第二个方法会修改该对象。
2. 创建 `MyString` 类，它包含一个 `String` 对象，在构造器中使用参数对此 `String` 初始化。增加 `toString` 方法和 `concatenate()` 方法，后者向内部的字符串追加一个 `String` 对象。在 `MyString` 中实现 `clone()`。创建两个 `static` 方法，各自接收 `MyString x` 引用为参数，然后调用 `xconcatenate("test")`，但是在第二个方法中先调用 `clone()` 方法。测试这两个方法，并说明效果的差异。
3. 创建 `Battery` 类，它包含一个用 `int` 表示的 `battery` 的数量（作为唯一的标识信息）。令其可克隆，添加 `toString()` 方法。然后创建 `Toy` 类，它包含一个 `Battery` 数组，其 `toString()` 方法打印所有 `battery`。为 `Toy` 编写 `clone()` 方法，它自动克隆所有的 `Battery` 对象。测试克隆 `Toy` 并打印结果。
4. 修改 `CheckCloneable.java`，令所有 `clone()` 方法都捕获 `CloneNotSupportedException` 异常，而不是传递给调用者。
5. 使用可变的伴随类技术，写一个恒常的类，它包含 `int`、`double` 和 `char` 数组。
6. 修改 `Compete.java`，向 `Thing2` 和 `Thing4` 添加更多的成员对象，然后观察，是否能确定复杂度带来的时间变化，它是否只是简单的线性关系，或者它是否看起来更复杂。
7. 由 `Snake.java` 开始，创建一个深层拷贝版本的 `Snake` 类。
8. 令 `CloningCollection` 类实现 `Collection` 接口，通过 `private ArrayList` 提供容器功能。重载 `clone()` 方法，使 `CloningCollection` 执行“有条件的深层拷贝”。它尝试对容器内的所有元素都执行 `clone()`，但是如果做不到，就丢弃被别名化的引用。

⁶ 这是Doug Lea给我的建议，他帮助我解决了此问题，他说他直接为每个类编写一个`duplicate()`函数。

附录 B: Java 编程指南

这份附录包含了一些建议，在你进行详细设计和编写代码的时候可以提供帮助。

就这些建议本身而言，它们只是一些指南而不是规定。你要把它们视作灵感的来源。记住，在某些情况下你要进行妥协，甚至打破这些指南的规定。

设计

- 1. 优雅终将得到回报。**从短期利益来看，要想对问题提出优雅的解决方案，似乎需要投入更多的时间，不过一旦它能够工作，就能够很容易地适应新的环境，而不是要花上数以小时计，甚至以天计或以月计的辛苦代价时，这时你将得到回报（尽管没人能够准确衡量这些回报）。你得到的程序不仅易于编写和调试，而且还易于理解和维护，这就是其价值所在。明白这一点需要经验，因为你在精心设计程序的时候生产率不会很高。要抵抗住浮躁心态的诱惑；欲速则不达。
- 2. 先能运行，再求快速。**即使你确信某段代码非常重要，它将成为整个系统的瓶颈，也必须遵守这一点。别着急。先尽可能简化设计，让系统运转起来。如果性能不够理想，再求助于性能分析工具。你几乎总会发现，你“以为的”那些瓶颈，其实都不是真正的问题所在。要把时间用在刀刃上。
- 3. 谨记“分而治之”原则。**如果待解决的问题过于复杂，先设想一下程序的基本操作，并且假定已经有一小段“神奇代码”能够处理最困难的部分。这段“神奇代码”可以看成是一个对象，你可以编写程序使用它，然后再回过头来研究这个对象，把它最困难的部分包装成其它对象，依此类推。
- 4. 区分类的编写者和使用者（客户端程序员）。**作为“客户”，类的使用者不需要也不希望知道类的底层工作方式。类的编写者必须是程序设计方面的专家，这样编写出来的类才有可能在新手使用的情况下，仍然能够稳定运行。请把类想象成对其它类的“服务提供者”。程序库只有在内部实现对用户来说是透明的情况下，才会易于使用。
- 5. 编写类的时候，类的名称要非常清晰，使得注释成为多此一举。**你的目标应该是提供给客户端程序员简单明了的接口。为此，在恰当的时候可以考虑方法重载，以得到直观且易于使用的接口。
- 6. 你的分析和设计所产生的系统中的类、它们的公共接口，以及类之间（尤其是与基类之间）的联系，必须达到最少。**如果你在设计中产生了过多的类，请回顾一下，这些代码在程序的整个生命周期中能产生效益吗？如果并非如此，你就要付出维护的代价。对于不能提高生产率的任何东西，开发团队的成员不会自觉地进行维护；这也是许多设计方法无能为力的地方。
- 7. 尽量让所有东西自动化。**首先编写测试代码（在你编写类之前），并把它和要测试的类

放在一起。你可以使用某种构建工具，来自动运行测试。你也许会用到Ant，它是Java构建工具的事实标准。这样，只要执行测试程序，所有改动就可以自动获得验证，有了错误也可以立刻发现。因为你信赖测试框架所具有的安全性，所以当你发现新的需求时，会大胆地进行全面修改。请记住，程序语言最大的改进，来自类型检查、异常处理等机制所赋予的内置测试行为。但这些功能只能协助你到达某种程度，其它工作还需要你自己完成。要开发一个健壮的系统，你得自己编写测试用例来验证类或程序的性质。

8. **在编写类之前先编写测试代码，以验证这个类是否设计完备。**如果你写不出测试代码，就说明其实你还不清楚类的功能。此外，在编写测试代码的过程中，通常还能够发现类需要具有的额外特性或限制。而这些特性和限制并不总是能够通过分析和设计得到。测试代码也可作为使用类的范例。
9. **所有软件设计中的问题，都可以通过“从概念上引入额外的概念上的间接层次”得到简化。**这是软件工程领域的基本原则¹，也是抽象的依据。而抽象正是面向对象程序设计的主要性质。在面向对象编程中，我们也可以这么说：“如果代码过于复杂，那么就引入更多的对象。”
10. **引入的间接层次要有意义（与准则 9 相应）。**这里所指的意义可以像“将常用代码放入一个方法内”这么简单。如果你加入了无意义的间接层次（通过抽象或封装等等），那就会和没有引入间接层一样糟糕。
11. **尽可能使类原子化。**每个类要具有简单明了的用途，用它来向别的类提供服务。如果类或系统设计得过于复杂，请将它分割成几个较简单的类。一个最明显的判断依据就是类的大小：如果类很大，那它很可能负担太重，就应该被分割。

建议重新设计类的线索有：

- 1) 复杂的 **switch** 语句：请考虑使用多态。
 - 2) 有许多方法，各自处理类型极为不同的操作：请考虑划分成不同的类。
 - 3) 有许多成员变量，用来表示类型极为不同的属性：请考虑划分成不同的类。
 - 4) 其它建议请参考《**Refactoring: Improving the Design of Existing Code**》，Martin Fowler 著，(Addison-Wesley 1999)。
12. **当心冗长的参数列表。**参数列表过长将使得方法调用变得难以编写、阅读和维护。你应该试着将方法放到更合适的类中，并/或使用对象作为参数。
 13. **不要一再重复。**如果某段代码不断出现于许多派生类方法中，应将该段代码置于基类的某个方法中，然后在派生类方法中进行调用。这样不仅可以减少代码数量，也易于修改。有时候，找出这种通用代码还可以为接口增加实用的功能。在不牵涉继承的情况下，也可能会遇到这种情况：如果类中的几个方法使用了重复的代码，请把这些代码移到某个

¹ Andrew Koenig为我解释了这一原则。

方法里，然后在别的方法中进行调用。

14. **小心switch语句或嵌套的if-else语句。**这通常预示着“以编程方式判断类型”的代码，也就是说究竟会执行哪一段程序代码，将依据某种类型信息来判断（开始的时候，可能不清楚确切的类型）。通常可以使用继承和多态机制来替代此类代码；多态方法在调用时会自动进行类型检查，这样更可靠，扩展起来也更容易。
15. **从设计观点来看，要找出变动的因素，并使它和不变的因素分离。**也就是说，找出系统中可能会改变的元素，将它们封装于类中，这样就不会被迫重新设计系统。你可以在《Thinking in Patterns (with Java)》（从www.BruceEckel.com下载）学习到大量的此类技术。
16. **不要依靠子类化来扩展基础功能。**如果类接口中的某个元素非常重要，那么它应该被放进基类，而不是在继承时添加。如果你依靠派生来添加方法，也许你应该重新考虑整个设计。
17. **更少意味着更多。**从类的最小接口开始，尽量在能够解决问题的前提下让它保持简单明了。先别急着考虑类被使用的所有方式。一旦它被实际使用，你自然会明白该如何扩展接口。不过，一旦类被使用后，你就不能在不影响用户代码的情况下缩减接口。不过加入更多方法倒没什么问题，这不会对用户代码造成影响，它们只需重新编译即可。即使使用新方法取代了旧方法的功能，也请你保留原有接口（如果你愿意的话，可以在底层实现中将功能进行合并）。如果你要通过“加入更多参数”来扩充原有接口，可以用新参数写一个重载的方法；这样，就不会影响对原有方法的调用。
18. **大声朗读你的类，确保它们符合逻辑。**使得基类和派生类之间保持“是一个”（is-a）的关系，让类和成员对象之间保持“有一个”（has-a）的关系。
19. **在判断应该使用继承还是组合的时候，考虑一下是否需要向上转型成基础类型。**如果不需要，请优先考虑组合（也就是使用成员对象）。这样可以消除对多个基类的需求。如果你采用继承，用户会假定它们可以被向上转型。
20. **采用字段来表示数值的变化，使用方法重载来表示行为的变化。**也就是说，如果你发现某个类中含有一些状态变量，而类的方法会根据这些状态变量表现出不同的行为，那么或许你就应该重新设计，在子类和方法的重载中表达这种行为上的差异。
21. **小心重载。**方法不应该把参数值作为执行代码的条件。在这种情况下，你应该编写两个或多个重载方法作为替代。
22. **使用异常体系** — 最好是从Java 标准异常体系中派生出特定的异常类。这样，处理异常的用户便可以在捕获基本异常之后，编写处理程序来捕获指定的异常。即使你派生了新的异常类，以前的客户端代码仍然能通过基础类型来捕获这个异常。
23. **有时候仅仅使用聚合就能完成工作。**比如飞机上的“旅客舒适系统”，它包括若干分离的部件：座椅、空调、视频设备等等，你需要在飞机对象上产生许多这样的部件。需要

把它们声明为私有成员，然后构建一个全新的接口吗？不，在这种情况下，这些部件也属于公共接口的一部份，所以你应该加入的是公有成员对象。这些对象具有各自的实现，所以仍然是安全的。注意，仅仅使用聚合并不是常用的解决方案，但有时候的确能解决问题。

24. **从客户端程序员和程序维护者的角度进行思考。**你设计的类要尽可能地易于使用。在设计类的时候，你应该预先考虑可能的变化，并使这些变化以后可以轻易完成。
25. **当心“巨型对象综合症”。**习惯于过程式程序设计的程序员，在刚接触面向对象程序设计领域的时候，往往会遇到这样的问题。因为他们最终还是习惯于编写过程式程序，并将它们放进一个或几个巨型对象中。注意，除了应用程序框架之外，对象应该代表程序中的概念，而不是程序本身。
26. **如果你只能采用某种别扭的方式才能实现某个功能，请将这个部份局限在某个类内部。**
27. **如果你只能采用某种不可移植的方式才能实现某个功能，请将其抽象成服务，并局限在某个类内部。**这样一个附加的间接层次，就可以防止不可移植的部份扩散到程序的其它部分。这个惯用法的一个具体应用就是Bridge设计模式。
28. **对象不应仅仅用来持有数据。**它还应该具有精心定义的行为。在某些情况下使用“数据对象”是恰当的，但只有在通用容器不适用时，才会刻意使用数据对象来包装、传输大批数据项。
29. **在原有类的基础上编写新类时，首先考虑组合。**只在必要情况下才使用继承。如果在可以使用组合的地方仍然选择了继承，就会为设计引入不必要的复杂度。
30. **使用继承和方法重载来表达行为上的差异，使用字段来表示状态的变化。**一个要不得的极端例子，就是派生出不同的类来表示不同颜色，而不是用一个color字段来表示颜色。
31. **当心“变异性”（variance）。**语意不同的两个对象可能会拥有相同的动作（或者职责）。这里自然而然会产生一种诱惑：仅仅为了从继承中获得某些好处，就让其中一个类成为另一个类的子类。这就是所谓“变异性”，即在没有任何正当理由的情况下，人为地制造出一个其实并不存在的超类/子类关系。一个较好的解决办法是写一个共用的基类，把两个类都作为它的派生类，这样它们将共享同一个接口；这种方法会占用更多空间，但你仍旧可以从继承中得到好处，也许还能在设计上获得重要发现。
32. **注意继承期间的限制。**最清晰的设计，将在派生类里添加新的功能。如果在继承过程中去掉旧功能，而没有添加新功能，这样的设计就值得怀疑。不过，也有例外情况，如果你在旧类库，那么在子类中对原有类进行限制，就比重组整个类继承层次更有效率。这时，在原有类基础上的新类就符合它应有的地位。
33. **使用设计模式来消除那些“纯粹的功能性代码”。**也就是说，如果你的类只应该产生一

个对象，请不要马上到程序的开头处，写下“只能产生一个对象”这样的注释。你应把它包装成单件（**singleton**）。如果主程序中有繁多且混乱的“用来创建对象”的代码，请使用类似“工厂方法”的创建型模式，以封装创建动作。消除那些“纯粹的功能性代码”不仅可以让你的程序更容易理解和维护，也可以使你的代码更强壮，以防止那些善意的维护者无心造成的破坏。

34. 当心因“过分分析而导致无从下手”的情况。请记住，在你获得所有信息之前，必须让项目持续向前推进。而且理解未知部份最好也是最快的方式，就是尝试向前推进一步而不是在完全理解之后才开始工作。除非已经找到了解决方案，否则你无法知道如何解决。你在某个类或一组类中造成的错误，并不会伤害整个系统的完整性。这是**Java**内置的防火墙，请让它们工作起来。
35. 当你认为自己已经获得了一个优秀的分析、设计或实现时，作一次全面评审。邀请团队之外的某个人（不一定非要是个顾问），也可以是公司里其它团队的成员。请他从旁观者的角度评审你的工作，这样能够在尚可轻易地作出修改的阶段就发现问题，相比因此而付出的时间和金钱代价，这样的收益更高。

实现

1. 一般来说，请遵守**Sun**的程序编写习惯。相关规范可以从这里取得：java.sun.com/docs/codeconv/index.html（本书尽可能遵守了这些习惯）。众多**Java**程序员看到的程序代码，都遵循这些习惯。如果你固执地使用自己过去的编写风格，会给读你程序的人带来一些困难。不论你决定采用何种编写习惯，要确保在整个项目中保持一致。你可以在<http://jalopy.sourceforge.net>上找到一个用来重排**Java**代码的免费工具，在<http://jcsc.sourceforge.net>找到一个免费的样式检查器。
2. 无论使用何种编写风格，如果你的团队（或整个公司，那就更好了）能够加以标准化，那么的确会带来显著效果。当有人的编写风格与标准不符时，每个人都可能会认为不妥，这就迫使他去进行改进。标准化的价值在于，分析程序代码时将更省脑力，因而你可以专注于代码的实质意义。
3. 遵守标准的大小写规范。类名称的第一个字母应为大写。数据成员、方法、对象（引用）的第一个字母应为小写。标识符的字应该连在一起，所有非首字的第一个字母都应该大写。例如：

ThisIsAClassName

thisIsAMethodOrFieldName

如果你在 **static final** 基本类型的定义处指定了常量进行初始化，那么该标识符应全为大写（字之间用下划线连接），它代表一个编译期常量。

“包”是个特例，其名称均为小写，即使对中间的字也是如此。域名后缀（**com, org, net,**

edu 等等) 也应为小写。(这是Java 1.1 和Java 2 之间的不同之处。)

4. 不要为私有字段名称加上你自己的“修饰”符号。这通常以“前置下划线和字符”的形式出现。匈牙利命名法(Hungarian notation)是其中最糟糕的例子。在这种命名法中,你得加入额外字符来表示数据的类型、用法、位置等等。这就使你好像是在使用汇编语言,编译器不能提供任何额外帮助一样。这样的标记容易让人混淆,又难以阅读,也不易推广和维护。要是你觉得必须给名称加上修饰以防止混淆,你的代码也许已经过于复杂而令人迷惑,需要简化的其实是你的代码。
5. 编写通用性的类时,请遵守标准形式(canonical form)。包括定义 equals()、hashCode()、toString()、clone() (实现 Cloneable 接口,或者选择其它对象复制策略,比如序列化),并实现 Comparable 和 Serializable 接口。
6. 对于那些“获得或改变私有字段值”的方法,请使用JavaBean的“get”、“set”、“is”等命名习惯。即使你当时并不认为自己在编写JavaBean。这样不仅可以把你的类当成Bean来使用,而且这也是这类方法的标准命名方式,它能够使读者更容易理解。
7. 对于你编写的每一个类,请使用JUnit为它编写测试用例(参见www.junit.org网站,以及第 15 章的范例)。在项目中使用类的时候,也不必移除类的测试代码,这样如果代码有所变动,你可以重新执行测试。测试代码也可以作为类的使用范例。
8. 有时你需要通过继承,才能访问基类的保护成员。这可能会令你觉得需要多个基类。如果不需要向上转型,可以先派生一个新类以对保护成员进行访问,然后在需要访问那个保护成员的所有类中,将新派生的类作为成员对象,而不要直接使用继承。
9. 应避免纯粹为了提高执行速度而采用final方法。只有在程序能够运行,但速度不够快,并且性能分析工具显示对此方法的调用将成为瓶颈的时候,才应该考虑把方法标记为final。
10. 如果两个类因某种功能性原因而产生了关联(比如容器和迭代器),那么请试着让其中一个类成为另一个的内部类。这不仅强调了二者间的关联,而且把类嵌套到其它类中,就可以在同一个包中重复使用类的名称。Java容器库在每个容器内都定义了一个内部类Iterator,这就为容器提供了统一的接口。使用内部类的另一个原因是,把它作为私有实现的一部份。这时,使用内部类是为了隐藏实现,而不是为了体现类之间的关联,也不是因为上面提到的命名空间污染问题。
11. 在任何时候,你都要警惕那些相互之间高度耦合的类,请评估一下使用内部类为程序编写和维护带来的好处。使用内部类不能消除类之间的耦合,而是使耦合关系更明显,使用起来也更方便。
12. 不要跳入过早优化的陷阱。过早优化其实并不明智。尤其是在系统构建初期,先别为是否使用本地方法、是否将某些方法声明为final、以及是否调整代码效率等问题而烦恼。你的主要目的应该是先证明设计的正确性。即使在设计本身也有效率需求的情况下,也要遵循“先能运行,再求快速”的准则。

13. 尽可能缩小对象的作用域，这样对象的可见范围和生存期也都会尽可能地小。这就降低了出现“在错误的语境中使用对象，导致难以发现的错误”的机会。假设你有个容器，以及一段遍历该容器的代码。如果你复制该代码，并将它用于遍历新容器，你就很可能会以旧容器的大小做为新容器的访问上界。然而，在遍历旧容器的时候如果超出了容器范围，编译期就会捕获错误。
14. 使用**Java**标准库提供的容器。精通它们的用法，将极大地提高你的工作效率。优先选择**ArrayList**来处理顺序结构，选择**HashSet**来处理集合、选择**HashMap**来处理关联数组，选择**LinkedList**来处理栈（而不用**Stack**，即便你可以创建适配器来得到栈的接口）和队列（如书中所见，也可以通过适配器得到）。当使用前三个的时候，你应该把它们分别向上转型为**List**，**Set**和**Map**，这样就可以在必要的时候以其它方式实现。
15. 对一个健壮的程序而言，每一个部件都必须健壮。在用**Java**编写类的时候，应充分运用所有能够提高程序健壮性的设施：访问控制、异常、类型检查和同步控制等等。这样，你在构建系统时就能够安全地进行进一步的抽象。
16. 宁可在编译期发生错误，也不要执行期发生错误。试着尽可能在最靠近问题发生点的地方处理问题。对于任何异常，都要在具有足够信息解决问题的最近的处理程序处进行捕获。在当前阶段应尽可能地对异常做点什么；如果实在无法解决问题，应该重新抛出异常。
17. 当心冗长的方法定义。方法应该是简洁的功能单元，它要描述并实现类接口中的某个离散的部分。过长且复杂的方法不仅难以维护，而且代价高昂。或许它的负担太重了。如果出现了这样的方法，应该把它分割成几个方法。这种方法还提醒你或许要编写新类。短小精悍的方法同样能够在类中被复用。（有时候方法必须很大才行，但它们应当只做一件事情）
18. 尽量使用“**private**”关键字。一旦你把库的特征（包括方法、类、字段）标记为 **public**，你就再也不能去掉它们。要是你非要这么做的话，就会破坏别人的已有代码，使它们必须被重写或重新设计。如果你只公开必要部份，那么其余部分就可以自由改变，而不用担心影响别人。设计总是会演化，所以这种自由度非常重要。在这种方式下，实现的变动对派生类造成的影响最小。在处理多线程问题的时候，保持私有性尤其重要，因为只有私有字段可以受到保护，而不用担心被未受同步控制的使用所破坏。
- 只在包内访问的类，也应该具有**private**字段，但通常最好使用具有包内访问权限的方法对它们进行访问，而不是将这些字段声明为**public**。
19. 大量使用注释，并使用**javadoc**的“文档注释语法”来生成程序的文档。不过，注释应该体现代码的真正涵义；如果只是把代码已经明确表示的内容简单重复一遍，会令人厌烦。请注意，通常**Java**类和方法的名称都很详细，这也在某种程度上降低了对注释的需求。
20. 避免使用“魔术数字”，就是那种固定在代码里的数字。要想对它们进行修改，那真是一场恶梦，因为你永远无法知道“100”究竟是代表“数组大小”还是“完全不同的其

它东西”。你应该使用带有描述性名称的常量，并在程序中使用这个常量名称。这将使程序更易于理解和维护。

21. **在编写构造器时，请考虑异常。**在最佳情况下，构造器不会有任何抛出异常的动作。稍次的情况下，只允许类组合强壮的类，或者从强壮的类继承，这样异常被抛出的时候，并不需要清理。在其它情况下，你必须在**finally**子句里清理组合过的类。如果某个构造器一定会失败，恰当的动作就是抛出异常，这样调用者就不至于盲目地认为对象已被正确创建而继续执行。
22. **在构造器中只作必要的动作：将对象设定为正确状态。**要积极避免在构造器内调用其它方法（**final**方法除外），因为这些方法可能会被其他人所重载，这就可能会在构造期间得到意外的结果（详细情形请参考第 7 章）。小型而简单的构造器抛出异常或引发问题的可能性要小得多。
23. **客户端程序员用完对象之后，如果你的类需要任何清理动作，请将此动作放到某个精心定义的方法中，**比如命名为**dispose()**，这样能清楚说明其用途。此外，在类里放置一个布尔型标志，用来表明**dispose()**是否已经被调用过，这样**finalize()**可以检查其“终结条件”（请参考第 4 章）。
24. **finalize()**方法的职责只应该是，为了调试的目的而检查对象的“终结条件”（请参考第 4 章）。在特殊情况下，可能会需要释放一些不能被垃圾回收器回收的内存。因为垃圾回收器可能不会被调用来处理你的对象，所以无法使用**finalize()**执行必要的清理。基于这个原因，你得自己编写**dispose()**方法。在类的**finalize()**方法中，请检查并确认对象已被清理，如果对象尚未被清理，这表明是一个编程错误，请抛出派生自**RuntimeException**的异常。在使用这种模型前，请先确认**finalize()**在你的系统上可以正常工作（你可能需要调用**System.gc()**来确认此行为）。
25. **如果对象在某个特定范围内必须被清理（而不是作为垃圾被回收），请使用以下方法：**先初始化对象，如果成功的话，立刻进入一个带有**finally**子句的**try**块，在**finally**子句中进行清理动作。
26. **在继承的时候如果重载了 finalize() 方法，要记得调用 super.finalize()。**（但如果你直接从**Object**类继承，就不需调用。）你应该在被重载的**finalize()**方法中最后（而不是首先）去调用**super.finalize()**，这样能保证如果你还需要使用基类中的某个组件的话，它们还是合法的。
27. **当你编写固定大小的对象容器时，它们要能够被传送给一个数组，**尤其是从某个方法返回此容器时。通过这种方式，你可以获得数组的“编译期类型检查”的好处，而且接收数组可能不需要“先将数组中的对象加以类型转换”就能够加以使用。请注意，容器库的基类**Java.util.Collection**具有两个**toArray()**方法，它们都能够达到这个目的。
28. **优先选择接口而不是抽象类。**如果你知道某些东西将成为基类，你应当优先把它们设计成接口；只有在必须放进方法定义或成员变量时，才把它改为抽象类。接口只和客户希望的动作有关，而类则倾向于实现细节。

29. 为了避免一个十分令人泄气的经验，请检查**classpath**，确保所有未放进包里的类的名称互不相同。否则编译器会先找到另一个名称相同的类，它会认为这样没有意义，并报告错误信息。如果你怀疑**classpath**出了问题，试着从**classpath**中的每个起点搜寻同名的.class文件。你最好还是将所有类都放到相应的包里。
30. 注意无心的重载错误。如果你在重载基类方法时没有正确拼写方法的名称，其后果是增加了一个新方法，而没有重载原有的方法。然而这样做完全合法，因此编译器或运行期系统不会向你报告错误信息；只是你的代码就无法正确工作了。
31. 当心过早优化。先能运行，再求快速。只有在你必须（也就是说只有在证明某段代码确实存在性能瓶颈）时才这么做。除非你已经使用性能分析工具找出了瓶颈所在，否则你只是在浪费时间。性能调整的隐性成本在于它会令你的代码变得难以阅读和维护。
32. 记住，代码被阅读的时间多于它被编写的时间。清晰的设计能使程序易于理解。注释、详细说明、测试用例和使用范例的作用是无价的。它们能帮助你和你的后继者。如果在没有其它信息的情况下，要想从Java文档中找出一些有用的信息时，你所遭遇的挫败就足以让你相信这一点。

附录 C：补充材料

本书还备有一些补充材料，包括随书光碟中的课程资料、讲座，以及通过 **MindView** 网站提供的服务。

这份附录是对这些补充材料的说明，你可以据此判断它们是否会对你有所帮助。

“Java 基础”多媒体讲座

书后所附光碟内提供了一些基础资料，你可以用它来为学习本书或参加“**Thinking in Java**”讲座作准备。这张容量超过 400 兆的光碟内包含了一门完整的多媒体课程：“Java 基础”。其中的“**Thinking in C**”讲座，介绍了 C 语言的语法、操作符和函数，这些都是 Java 语法的基础。此外，它还包含了我编写和赠送的“**Hand-on Java**”多媒体讲座的第二版前七讲的内容。尽管以前完整的“**Hand-on Java**”光碟是单独出售的（第三版的“**Hand-on Java**”光碟也是如此），但我决定加入第二版中前七讲的内容。因为在本书第三版中，这几讲的内容并无太大变化，它（以及 **Thinking in C**）不仅能够向你提供本书和“**Thinking in Java**”讲座的基础知识，并且还可以让你感受到“**Hand-on Java**”光碟第三版的质量和價值。

本书“简介”这一节对光碟内容有详细说明。

Thinking in Java 技术讲座

我的公司，“**MindView**”有限公司，提供 5 天、专家级、面向公众的室内培训讲座，其内容基于本书的资料。它以前的名称是“**Hand-on Java**”讲座，这是我们主要的介绍性课程，能够为学习更高级的课程打下基础。每次课程的内容是从各章精选出来的，课程之后是带指导的练习时间，这样学生就能够得到单独指导。你可以从 www.MindView.net 得到有关时间、地点、介绍及其它详细信息。

“Hand-on Java”多媒体讲座第三版

“**Hand-on Java**”多媒体讲座第三版，包含了“**Thinking in Java**”讲座里的扩展内容，同时它也基于本书的资料。它至少可以在不用舟车劳顿并能够减少花费的情况下为你提供生动的课程。根据书中的每个章节，它附带了音频讲座和相应的幻灯片。我编写了这个讲座（最近 **Andrea Provaglio** 也参加了进来，他负责这些讲座的所有有声版本），光碟里的内容也由我讲授。“**Hand-on Java**”多媒体讲座第三版可以在 www.MindView.net 购买。

“对象与系统设计”技术讲座

本讲座从很受欢迎的“对象与模式”讲座衍生而来，**Bill Venners**和我在过去几年一直在开那个讲座。但它的内容越来越多，所以我们将它分成两部分：这一部分和将在本附录后面说明的“**Thinking in Patterns**”讲座。

良好的面向对象设计中，一个重要部分就是精心设计的对象。本讲座（持续一周）的主要部分是对象设计专题讨论，它的重点在于，为了得到设计良好的对象，所需遵循的准则和习惯用法。它们将被仔细讲解和分析，并由学生进行讨论。这是专题讨论的主要部分，其目的是促进参与者之间对设计的讨论，使每个人都能学习到对方的经验和观点。对象设计专题讨论将带给你整套行之有效的准则和具体的习惯用法，你可以在将来的对象设计中使用它们。

此讲座的其它部分将着重于开发和构建系统的过程，其主要内容是所谓的“敏捷方法”或者“轻量级方法学”，尤其是极限编程（**XP**）。我们将对这些方法学作出总体介绍，以及类似“索引卡片”的小工具，它是在《**Planning Extreme Programming**》（**Beck and Fowler, 2002**）一书中介绍的计划编制技术，还有用于对象设计的**CRC**卡片、结对编程、迭代计划、单元测试、自动构建、源代码控制，以及其它类似主题。这个课程还包括了一个采用**XP**方法的项目，将在一周内开发完成。

请访问 www.MindView.net 得到有关时间、地点、介绍及其它详细信息。

Thinking in Enterprise Java

本书从《**Thinking in Java**》中部分讲述高级主题的章节派生而来。它并不是《**Thinking in Java**》的第二卷，而是着眼于企业级程序设计中的高级主题。这本书现在可以从 www.BruceEckel.com 免费下载。由于是一本单独的书，因此它的篇幅可以随着内容的需要而扩展。与《**Thinking in Java**》一样，它的目标是向读者提供一本易于理解，涵盖企业级编程技术的介绍。并为读者学习更深入的主题做准备。

以下列出的是书中讨论的部分主题：

- 企业级程序设计介绍
- 使用 **Socket** 和 **Channel** 进行网络编程
- 远程方法调用（**RMI**）
- 连接到数据库
- 命名与目录服务
- **Servlet**
- **Java** 服务器页面（**JSP**）
- 标签、**JSP** 片段和表示语言
- 自动产生用户界面
- 企业级 **Java** 构件（**EJB**）
- 可扩展标记语言（**XML**）
- **Web** 服务

- 自动测试

你可以从www.BruceEckel.com网站上了解《Thinking in Enterprise Java》的进展情况。

J2EE 技术讲座

本讲座将使用基于 **Java**, 支持 **Web** 的分布式应用程序, 向你介绍现实世界的实际开发过程。它涵盖了 **J2EE** 及其关键技术: 企业级 **Java** 构件 (**EJB**), **Servlet**, **Java** 服务器页面 (**JSP**), 以及基本的架构模式, 用来把这些技术集成起来以形成可维护的应用程序。

学完本课程, 你将对 **J2EE** 架构有一个全面的理解: 它用来解决什么问题, 如何选择恰当的工具, 以及怎样得到你的解决方案。

请访问 www.MindView.net 得到有关时间、地点、介绍及其它详细信息。

Thinking in Patterns (with Java)

面向对象设计领域的重大进步之一, 就是“设计模式”运动的兴起, 它记载于《设计模式》, **Gamma, Helm, Johnson & Vlissides** 著 (**Addison-Wesley 1995**)。这本书介绍了 **23** 种不同的解决方案, 它们都专门针对某些特定类型的问题, 并以 **C++** 语言表述。《设计模式》已经成为经典之作, 它是面向对象程序员之间进行交流的共同词汇, 这几乎就成了一种规定。

《*Thinking in Patterns*》介绍了设计模式的基本概念, 并附有 **Java** 范例。这本书并不希望成为《设计模式》的简单重复, 而是希望带来 **Java** 角度的新观点。它并不仅限于传统的 **23** 种模式, 还包括了其它一些恰当的问题解决方案。

这本书脱胎于《*Thinking in Java*》第一版的最后一章, 随着内容的不断发展, 把它单独成书就显得合情合理。在编写这份附录的时候, 它还处于写作之中, 但其中的素材已经无数次在“对象和模式”讲座的实践中使用过(这个讲座现在已经被划分为“对象与系统设计”讲座和“*Thinking in Patterns*”讲座)。

Thinking in Patterns 技术讲座

本讲座从“对象与模式”讲座衍生而来, **Bill Venners** 和我在过去几年一直在开那个讲座。但它的内容越来越多, 所以我们将它分成两部分: 这一个和本附录前面做过说明的“对象与系统设计”讲座。

本讲座严格遵循《*Thinking in Patterns*》一书里的资料和表述, 所以要了解本讲座的内容, 最好是从 www.MindView.net 下载这本书。

许多表述都是设计演化过程的实例, 先从初始解决方案开始, 然后通过演化过程, 得到更恰当的设计。其中的最后一个案例(垃圾回收模拟)已经随着时间而演化, 你可以把这个演化过程作为一个原型, 这样你自己的设计在开始的时候就对这类特定问题有了足够的思路, 然后对这一类问题演化出灵活的方案。

- 极大增强设计的灵活度。
- 内置的可扩展性和可重用性。
- 使用模式语言进行设计之间的交流。

每次课程之后将有一些模式练习待你解决，它将引导你编写代码来应用特定的模式，以得到编程问题的解决方案。

请访问 **www.MindView.net** 得到有关时间、地点、介绍及其它详细信息。

设计咨询与评审

我的公司还以提供咨询、辅导、设计评审和实现评审的方式，在项目的整个开发周期为你提供帮助，它对你的首个Java项目尤具价值。请访问www.MindView.net以获得详细信息。

附录 D：资源

软件

从java.sun.com获得的**JDK**（Java开发工具包）。即使你选择了第三方开发环境，万一遇到了可能是编译器出错的情况，手头有一套**JDK**总是不错的。可以把**JDK**作为检验标准，因为如果**JDK**有错误，那么这个错误广为人知的机会也应该比较高。

从java.sun.com获得HTML格式的**JDK文档**。在我所见过的介绍标准Java库的参考书中，不是内容过时，就是有所遗漏。尽管Sun的这份HTML文档有不少小错误，而且有时过于简陋，不过它至少列出了所有的类和方法。人们对使用在线资源而不是印刷书籍开始可能会有些不习惯，不过克服这一点相当值得。先浏览一下HTML文档，至少你可以得到大概的印象。如果你做不到这一点，就去找一本印刷书籍吧。

书籍

Thinking in Java, 2nd Edition。本书所附光碟里就有此书，这是一个包含了完整索引并以颜色突出语法的HTML版本。你也可以从www.BruceEckel.com 免费下载。它还包含了一些没有收录到第三版的内容；详细情况请参考书中的目录。

Thinking in Java, 1st Edition。本书所附光碟里就有此书，这是一个包含了完整索引并以颜色突出语法的HTML版本。你也可以从www.BruceEckel.com 免费下载。它还包含了一些比较旧的、不适合被放进第二版的内容。

Just Java 2, 5th edition by Peter van der Linden (Prentice Hall, 2002)。这本书不仅实用而且非常生动。作者的方法常常与我不谋而合，坚持通过问题来发现完整的细节，所以它常常能给你一些在别处找不到的答案。

Core Java 2, Volume I—Fundamentals (Prentice-Hall, 1999)和Volume II—Advanced Features (2000), by Horstmann & Cornell。巨大并且全面。每当我需要寻找某些答案时，就会想到它们。当你读完《Thinking in Java》，需要更进一步时，我推荐这两本书。

The Java Class Libraries: An Annotated Reference, by Patrick Chan and Rosanna Lee (Addison-Wesley, 1997)。尽管内容有些过时，但这是你应该拥有的JDK参考书：详细的说明令其使用起来非常方便。《Thinking in Java》的一位技术评审曾经这么说：“如果我只能有一本Java书，那就是它了（哦，当然，除了你那本之外）”。我并不像这位评审那样兴奋。这本书很庞大且昂贵，其中提供的示例品质并不能令我满意。不过你要是遇到某个疑难问题，这本书能提供比其它可供选择的书籍更深入（也更厚）的解答。

Java Network Programming, 2nd Edition, by Elliott Rusty Harold (O'Reilly, 2000)。直到阅读了这本书，我才开始理解Java的网络机制。我发现了作者的网站Café au Lait，它

为Java开发提供了令人兴奋、有主见、最新潮的观点，而且没有任何厂商色彩。作者会定期更新网站内容，以跟上Java快速变化的各种新闻。请参考www.cafeaulait.org。

Design Patterns, by Gamma, Helm, Johnson and Vlissides (Addison-Wesley, 1995)。在程序设计领域发起设计模式运动的开山之作。

Practical Algorithms for Programmers, by Binstock & Rex (Addison-Wesley, 1995)。书中算法以C实现，也很容易转换成Java。每个算法都有详细说明。

分析与设计

Extreme Programming Explained, by Kent Beck (Addison-Wesley, 2000)。我爱这本书。是的，我倾向于采用激进的方法来解决问题，但我总觉得应该会有与众不同、更好的软件开发过程，我认为XP已经很接近这个标准了。另一本对我有同样震撼的书是

《PeopleWare》(后面介绍)，它主要探讨环境和团队文化中的协作。《Extreme Programming Explained》探讨的是程序设计，它要推翻为人所知的绝大多数方法，甚至是最新的“研究发现”。他们甚至非常激进，声称任何有关项目的全景描述只要没有花费你太多的时间，而且你愿意将它们丢掉，那么它们就是好的选择。（你会注意到这本书的封面上没有“UML认证标志”）。我会以某家公司是否采用XP来决定是否为他们工作。这本书短小精悍，章节很短，读起来很轻松，而且能够激励思考。你可以开始想象自己工作在这样的环境中，它会带给你全新的视野。

UML Distilled, 2nd Edition, by Martin Fowler (Addison-Wesley, 2000)。初次接触UML时，你大概会有畏难情绪。因为里面充满了各种图和细节。根据Fowler的说法，其实大部份内容都非必要，所以他直接讨论本质内容。对大多数项目来说，你只要把少数几种图作为工具就够了。Fowler关注的是拿出一份好的设计，而不是要得到这一份好设计所需要的全部制品。这是一本优秀、短小精悍、易于阅读的书籍；如果你需要理解UML，这本书是首选。

The Unified Software Development Process, by Ivar Jacobsen, Grady Booch, and James Rumbaugh (Addison-Wesley, 1999)。我原本做好了不喜欢这本书的打算。此书似乎具有烦人的大学教科书才有的所有特征。但是我惊喜地发现，全书不仅脉络清晰，而且令人愉快。尽管书中有几个概念似乎作者也不甚明了。其中最好的一点是，整个过程非常具有实用价值。这不仅是XP，也是UML的强大力量之一。即使你无法接受XP，但在大多数人已经搭上“UML就是好”的顺风车（且不论他们实际经验如何）的情况下，你也许会接受本书。我认为此书应当是推广UML的旗舰。当你读完Fowler的《UML Distilled》，还准备深入学习的话，可以选择这本书。

在选择任何方法之前，先听听立场中立人士的看法，会很有帮助。人们往往在尚未真正了解自己的需要，或尚未知道某种方法能为你做什么之前，就轻率地作出选择。“别人正在使用”，听起来似乎很有道理。不过，人们常有一种奇怪心理：如果他们想要相信某种方法真能解决问题，他们就会去尝试（这种实验态度很好），但如果不能解决问题，他们便可能加倍努力并开始大声宣称，他们发现了很伟大的东西（这种拒绝承认的态度不好）。这里的假设是，

如果有一些人和你在同一艘船上，你就不会感到孤单，哪怕那艘船正驶向未知的地方（甚至正在下沉）。

我并不是在说所有方法学都没有前途，而是提醒你应该用某种理念来武装自己，这种理念能够帮助你坚持实验模式（“这种方法不可行，让我们试试其它方法”），并摆脱否认模式（“不，这其实不是问题。一切都是那么美好，我们不需要改变”）。我认为在你选择某种方法之前，应该先阅读下列几本书，它们会带给你这种理念。

Software Creativity, by Robert Glass (Prentice Hall, 1995)。在完整地方法论角度进行讨论的书籍中，这是我见过最好的一本。本书集合了Glass 所撰写或获得（P.J. Plauger是其中一位作者）的许多小品文和论文，这些文章反映出他多年来对这个课题的思考和研究。这些文章十分有趣，而且长度适中；既非漫无目的，也不会让你感到无聊。作者也不是毫无根据，其中引用了数以百计的其它论文和研究报告。所有程序员和管理者在陷入方法论的泥沼前，都应该好好阅读这本书。

Software Runaways: Monumental Software Disasters, by Robert Glass (Prentice Hall, 1997)。这本书最出色的地方是，它直接把我们带到以前从未讨论过的软件开发的最前沿：有多少项目不仅失败了，而且是一败涂地。我发现大多数人仍然认为“这不可能发生在我身上”，或是“这不会重演”，这种侥幸心理会使我们处于劣势。要把“任何事都可能出错”牢记在心，这样才能以更好的心态使事情向正确的方向发展。

Peopware, 2nd Edition, by Tom Demarco and Timothy Lister (Dorset House, 1999)。这是必读书目。它不仅有趣，而且会动摇你的世界观，摧毁你不切实际的假设。虽然书中的背景是软件开发，但其讨论的内容适用于一般项目和团队。其重点放在人及人的需求，而不是技术和技术的需求上。作者所讨论的是如何建立一个让人们能够快乐工作并且有高生产率的环境，而不是讨论这些人应该遵守哪些规则才能成为称职的机器零件。我认为正是后一种态度，造成了程序员在采用某种方法的时候先拍手叫好，然后并不作出任何改变。

Secrets of Consulting: A Guide to Giving & Getting Advice Successfully, by Gerald M. Weinberg (Dorset House, 1985)。一本很棒的书，我最喜欢的书籍之一。如果你准备当一名顾问，或者想与顾问合作愉快，请选择本书。书中的章节很短，里面有很多故事和轶闻来引导你如何付出最小的代价来看清问题的实质。也可以参考《**More Secrets of Consulting**》，2002 年出版，或其它Weinberg的作品。

Complexity, by M. Mitchell Waldrop (Simon & Schuster, 1992)。这本书记录了一群来自不同领域的科学家，聚集于新墨西哥州圣达菲（Santa Fe），一起讨论他们各自学科领域无法解决的现实问题（经济学里的股市问题、生物学里的生命起源问题、社会学里的人类行为问题等等）。凭借跨越物理、经济、化学、数学、计算机科学、社会学、以及其它学科的方式，针对这些问题发展出一套学科交叉的解决方案。更重要的是，思考这类极复杂问题的另一种方式正在成形：抛弃“数学决定论”和“以方程式预测所有行为”的错误认知，迈向“先观察、找出模式、试着以任何可能的手段仿真”的方式。比如，书中记录了遗传算法的面世。我相信，这种思考方式对我们研究和管理日益复杂的软件项目十分有用。

Python

Learning Python, by Mark Lutz and David Ascher (O'Reilly, 1999)。一本针对程序员的入门读物，也是我最喜欢的程序语言，和Java配合效果更好。本书还包括对Jython的介绍。使用Jython，你可以将Java和Python整合进同一个程序（Jython解释器能产生Java字节码；所以不用加入任何特殊操作就可以达到目的）。这个语言的相关组织承诺将为我们带来最大的可能性。

我的作品

以下以出版顺序排列。并非所有书籍都能在市场上找到。

Computer Interfacing with Pascal & C, (Self-published via the Eisis imprint, 1988. 只能通过www.BruceEckel.com取得)。这是在CP/M为主流而DOS正在崛起的时代，一本带有电子学背景的简介书。我使用高级语言通过计算机并行端口进行控制，来驱动各种电子设备。本书内容改写自我最初（也是最好的）在Micro Cornucopia杂志上发表的专栏文章（根据Software Development Magazine杂志资深编辑Larry O'Brien的说法，这是曾经出版过的最好的计算机杂志，他们甚至计划建造一个花盆里的机器人！）。但是，唉，在互联网出现很久之前Micro Cornucopia杂志就已经消失了。编写这本书给我带来了极好的出版经验。

Using C++, (Osborne/McGraw-Hill, 1989)。我的第一本C++书籍。本书已经绝版，被其第二版所取代，并改名为《C++ Inside & Out》。

C++ Inside & Out, (Osborne/McGraw-Hill, 1993)。如上所述，本书实际上是《Using C++》的第二版。本书内容已经相当准确，但在1992年前后我以《Thinking in C++》取而代之。你可以在www.BruceEckel.com 中找到更多本书信息，也可以下载源代码。

Thinking in C++, 1st Edition, (Prentice Hall, 1995)。

Thinking in C++, 2nd Edition, Volume 1, (Prentice Hall, 2000)。可以从www.BruceEckel.com下载。

Thinking in C++, 2nd Edition, Volume 2, 与Chuck Allison合著 (Prentice Hall, 2003)。可以从www.BruceEckel.com下载。

Thinking in C#, By Larry O'Brien and Bruce Eckel. 从Larry的《Thinking in Java into C#》改写而来，我也提供了一些帮助(Prentice Hall, 2003)。

Black Belt C++: the Master's Collection, Bruce Eckel, editor (M&T Books, 1994)。本书已绝版。书中收录了许多C++杰出人物在“Software Development Conference”会议的演讲和文章，我是这个会议的主席。本书封面使我决定对自己以后所有书籍的封面设计进行控制。

Thinking in Java, 1st Edition, (Prentice Hall, 1998)。本书第一版，赢得了 Software Development Magazine生产力奖、Java Developer's Journal编辑推荐奖、JavaWorld读者推荐最佳书籍奖。在本书后面的光碟里有收录，也可于www.BruceEckel.com下载。

Thinking in Java, 2nd Edition, (Prentice Hall, 2000)。这一版赢得了JavaWorld编辑推荐最佳书籍奖。在本书后面的光碟里有收录，也可于www.BruceEckel.com下载。

索引

请注意，某些名字会以大写字母的形式重复出现。本书内容遵循了 **Java** 风格，大写名字代表 Java 类，而小写名字代表一般性的概念。

!

!.119

!=.117;operator.1032

&

&.122

&&.119

&=.122

.

.NET.68

@

@author.101

@deprecated.102

@docRoot.100

@inheritDoc.100

@link.100

@param.101

@return.101

@see.100

@since.101

@throws.102

@version.101

[

[]:indexing operator[].202

^

^.122

^=.122

|

|.122

||.119

|=.122

+

+.115;operator+for String .1061

<

<.117

<<.123

<<=.123

<=.117

=
==.117:operator.1032

>
>.117
>=.117
>>.123
>>=.123

A

abstract :class .293;inheriting from an abstract class.294;abstract keyword.294;vs.interface.321 抽象：类；继承一个抽象类；关键字 abstract；与 interface 相比
Abstract Window Toolkit (AWT).765 抽象窗口工具箱（AWT）
Abstract Button.805 AbstractButton
Abstraction.32 提取
Abstract Sequential List .551 AbstractSequentialList
AbstractSet.503 AbstractSet
access:class.232;control.215,236;innerclasses&accessrights.342;packageaccessand
friendly .225;specifiers . 39,215,224;within a directory ,via the default package .227
访问：类；控制；内隐类和访问权限；包的访问和 friendly；修饰词；在一个目录中，通过缺省的包
action command .833 动作命令
Action Event.833,895 ActionEvent
Action Listener.785 ActionListener
actor ,in use cases .1003 角色，在用例之中的
adapters : listener adapters.800 适配器：侦听器适配器
add(),ArrayList.492 add(),ArrayList
addActionListener().892,899 addActionListener()
addChangeListener.838 addChangeListener
addition.111 增加物
addListener.793 addListener
addXXXListener().794 addXXXListener()
Adler32.645 阿德勒（姓氏）
aggregate array initialization.202 数组聚合初始化
aggregation.40 聚合
aliasing.110;and String.1061;during a method call .1022 别名；和字符串；方法调用时
align.773 对齐方式
allocate().617 allocate()
allocateDirect().617 allocateDirect()
alphabetic vs.lexicographic sorting.477 字母表排序与字典排序相比较
analysis:and design,object-oriented.997;paralysis.998;requirements analysis.1001 分析：和设计，面向对象；麻痹；需求分析
AND:bitwise.129;logical(&&).119 AND：位；逻辑（&&）
anonymous inner class.335,586,782;and table-driven code.551 匿名内隐类；和表驱动的代码
Ant,automated build process using .994 antcall.948 Ant，使用 antcall 的自动构建过程
applet.768;advantages for client /server systems.769;align.773;archive tag , for HTML and JAR

files .857;classpath.775;codebase.773;combined applets and applications.776;name.773;
 applet; 客户端/服务器 系统的优点; 对齐方式; HTML 和 JAR 文件的标记存档文件; 类路
 径; codebase; 结合 applet 和应用; 名称;
 packaging applets in a JAR file to optimize loading .857;parameter.773;placing inside a web
 page.771;restrictions.768;signed.858 通过对一个 JAR 文件中的 applets 分包来优化装载; 参数;
 放入一个网页中; 约束; 标记过的
 appletviewer.773 applet 浏览器
 application:application builder .882;application framework.360;application framework , and
 applets.770;comebined applets and applications.776;windowed applications.775 应用: 应用构
 建者; 应用框架; 应用框架和 applets; 结合 applets 和应用; 窗口化的应用
 archive tag , for HTML and JAR files.857 HTML 和 JAR 文件的标记存档文件
 argument:constructor.167;final .268,587;passing a reference into a method.1022;variable argument
 lists(unknown quantity and type arguments).206
 参数: 构造器; final; 传一个引用到一个方法; 可变参数列表 (未知数量和类型的参数)
 array .453;associative array .520; associative array ,Map.483;bounds checking .204;comparing
 arrays.472; 数组; 关联数组; 关联数组, 映射; 边界校验; 比较数组
 copying an array.472; 复制一个数组
 dynamic aggregate initialization syntax.457; 动态聚合初始化语法
 element comparisons.473; 元素的比较
 first-class objects.455; 一级对象
 initialization.202;length.204,455;multidimensional.208;of objects.455;of primitives.455;return an
 array.459 初始化; 长度; 多维的; 对象的; 基本型别; 返回一个数组
 ArrayList.497,505,509,549,554,926;add().492;get().492.497;size().492;type-conscious
 ArrayList.495 ArrayList; add();get();size();对型别敏感的 ArrayList
 Arrays class, container utility.460 Arrays 类型, 容器效用
 Arrays.asList().569 Arrays.asList()
 Arrays.binarySearch().478 Arrays.binarySearch()
 Arrays.fill().469 Arrays.fill()
 asCharBuffer().619 asCharBuffer()
 assertion,JDK1.4.930 断言 (assertion), JDK1.4
 assigning objects.108 指定对象
 assignment.108 指定
 associative array.480,520;Maps.483 关联数组; 映射
 atomic operation.734 原子操作
 auto-decrement operator.115 自减操作符
 auto-increment operator.115 自增操作符
 automated:build process using Ant.944;unit testing.911 自动: 构建过程, 使用 Ant; 单元测试
 automatic type conversion.243 自动类型转换
 available().606 available()

B

bag.481 包
 base 16.131 基本
 base 8.131 基本
 base class.229,245,283; 基类

abstract base class .293; 抽象基类
 base-class interface .288;constructor .229;constructors and exceptions .250;initialization .248 基
 类接口; 构造器; 构造器和异常; 初始化
 base types .42 基本型别
 BASIC:language.73;Microsoft Visual BASIC.882 BASIC: 语言; ;微软 Visual BASIC.
 Basic concepts of object-oriented programming(OOP).31 面向对象编程的基本概念
 BasicArrowButton.806 BasicArrowButton
 BeanInfo:custom BeanInfo.903 BeanInfo: 定制的 BeanInfo
 Beans:and Borland's Delphi.882; Beans:和 Borland's Delphi
 and Microsoft's Visual BASIC.882;application builder.882;bound properties.903;component .883;
 和 微软的 Visual BASIC; 应用构建者; 边界属性; 构件
 constrained properties .903;custom BeanInfo.903;custom property editor.903;custom property
 sheet.903;events.883;EventSetDescriptors.890;FeatureDescriptors().890;getMethodDescriptors().
 890;getName().889;
 受限属性; 定制的 BeanInfo; 定制属性编辑器; 定制属性表单; 事件; EventSetDescriptors;
 FeatureDescriptors(); getMethodDescriptors();getName();
 getPropertyDescriptors().889; getPropertyDescriptors()
 getPropertyType().889; getPropertyType()
 getReadMethod().889; getReadMethod()
 getWriteMethod().889;indexed property .903; getWriteMethod(); 经过索引的属性;
 Introspector.887;JAR files for packaging.901; 内省器; 用于分包的 JAR 文件
 manifest file.901;Method.890;MethodDescriptors.890;naming convention .884; 声明文件; 方法;
 方法描述; 命名规则
 properties.883;PropertyChangeEvent.903;propertyDescriptors.889; 属 性 ;
 PropertyChangeEvent; propertyDescriptors
 propertyVetoException.903; propertyVetoException
 reflection .883,886;Serializable.894;visual programming.882 反射; Serializable; 可视化编程
 Beck,Kent.1092 Beck, Kent
 Binary:numbers.131;numbers,printing .126;operators.122 二进制: 数字; 数字, 打印; 操作
 符
 binarySearch().478 binarySearch()
 binding:dynamic binding .284;dynamic,late,or run-time binding .279; 绑定: 动态绑定; 动态,
 后期或执行期绑定
 early.48;late.48;late binding .284;method call binding .284;run-time binding.284 前期; 后期;
 后期绑定; 方法调用绑定; 执行期绑定
 BitSet.573 BitSet
 Bitwise:AND.129;AND operator(&).
 122;copy.1032;EXCLUSIVE OR XOR(^).122;NOT~.122;operators.122;
 OR.129;ORoperator(|).122 位: AND; AND 操作符; 复制; 异或 (^); 非~; 操作符; 或;
 OR 操作符 (|)
 blank final .267 留白的 final
 blocking:and available().607;and threads.744;onI/O.750 阻塞: 和 available(); 和线程; 关于 I/O
 Booch,Grady.1093 Booch,Grady
 Book:errors,reporting.25;updates of the book.24 书: 错误报告; 书的更新

Boolean.143;algebra.122;and casting .130
 operators that won't work with Boolean .117;vs.C and C++.120 Boolean; 代数; 和对 Boolean
 无效的转型操作符; 与 C 和 C++相比较
 BorderLayout.786 BorderLayout
 Borland .904;Delphi.882 Borland; Delphi
 bound properties .903 边界属性
 bounds checking ,array .204 边界校验, 数组
 BoxLayout.789 BorderLayout
 break keyword .151 break 关键字
 breakpoints,debugging .978 断点, 除错
 browser,class browser.232 浏览器, 类浏览器
 buffer,nio.615 缓冲, nio
 BufferedInputStream .596 BufferedInputStream
 BufferedOutputStream.598 BufferedOutputStream
 BufferedReader.404,600,605 BufferedReader
 BufferedWriter.600,607 BufferedWriter
 build process,using Ant .944 构建过程, 使用 Ant
 build.xml.946 构建 xml
 buildfile.946 构建文件
 builds,daily,Extreme Programming.953 每天构造, 极限编程
 business objects/logic.872 业务对象/逻辑
 button:creating your own.801;radio button.819;swing.779,805 按钮: 制作你自己的; radio 按钮;
 swing;
 ButtonGroup.806,819 ButtonGroup
 ByteArrayInputStream.592 ByteArrayInputStream
 ByteArrayOutputStream.593 ByteArrayOutputStream
 ByteBuffer.615 ByteBuffer

C

C#:programming language .68;Thinking in C# .68 C#编程语言; C#编程思想
 C++ .117;copy constructor .1049;exception handling .411; C++; 拷贝构造器; 异常处理
 Standard Container ;Library aka STL.481; 标准容器; Library aka STL.481
 Templates.497;vector classes ,vs .array and Array;ist .454 模版; 向量类型与数组合 ArrayList
 相比较
 callback .473,475,585,781;and inner classes.357 回调; 和内隐类
 capacity ,of a HashMap or HashSet .539 HashMap 或 HashSet 的能力
 capitalization of packagenames .90 大写的包名称
 case statement .158 状态陈述
 cast .50,177,425;and containers .491;and primitive types .144;from float or double to integral,
 truncation.161;operators .130 转型; 和容器; 和基本型别; 从 float 或 double 到 intergral, 截
 断; 操作符;
 catch:catching an exception .374;catching any exception .382;keyword .375 catch: 捕捉一个异
 常; 捕捉任何异常; 关键字
 certificate ,applet signing .861 证书, applet 标记
 chained exceptions in JDK 1.4.416 在 JDK1.4 中的连锁的异常

change ,vector of .362	变化, 向量
channel ,nio .615	通道, nio
CharArrayReader.599	CharArrayReader
CharArrayWriter.599	CharArrayWriter
CharBuffer.619	CharBuffer
CharSequence.678	CharSequence
Charset .620	Charset
check box .817	checkbox
check instruction ,design by contract .935	检查指令, 契约式设计
checked exceptions .410;converting to unchecked exceptions .416	检查过的异常; 转变为未检查的异常
CheckedInputStream .643	CheckedInputStream
CheckOutputStream .643	CheckOutputStream
Checksum class .645	Checksum 类型
Class .35.231;abstract class .293;access .232;anonymous inner class .335,586,782;	
base class .229,245,283;browser .232;class hierarchies and exception handling .408;	
class literal .428,436;creators .38;defining the interface .1014;derived class .283;equivalence ,and	
instanceof/isInstance() .440;final classes .271;inheritance diagrams .263;	类型; 抽象类; 访问;
匿名内隐类; 基类; 浏览器; 类的层次和异常处理; 类的语法; 创建者; 定义接口; 派生类;	
相等, 和 instanceof/isInstance(); final 类型; 继承图	
inheriting from an abstract class .294;inheriting from inner classes .349;	继承一个抽象类; 继承一个内隐类
initialization & class loading .273;initialization of fields .192;	初始化和类的装载; 域的初始化
initializing member at point of definition .193;initializing the base class .248;inner class .331;inner	
class nesting within any arbitrary scope .337;inner classes .875;inner classes & access	
rights .342;inner classes and overriding .350;inner classes and super .349;	
在定义的时候对成员进行初始化; 初始化基类; 内隐类; 任意范围内的内隐类的嵌套; 内隐	
类和访问权限; 内隐类和覆写; 内隐类和 super	
inner classes and Swing .794;inner classes and upcasting .333;	内隐类和 Swing; 内隐类和向上转型
inner classes in methods & scopes .335 ;inner classes , identifiers and .class files .354;instance of	
.33;intializing the derived class .248;keyword .41;	在方法和范围内的内隐类; 内隐类, 标识符和 class 文件; instance of
loading .275;member initialization .243;multiply-nested .348;order of initialization .195;	装载; 成员初始化; 多重嵌套; 初始化的次序
private inner classes .362;public class ,and compilation units .217;	private 内隐类; public 类型和编译单元
read-only classes .1054;referring to the outer class object in an inner class .347;	只读类; 在一个内隐类中引用外围类
static inner classes .344;style of creating classes .232;subobject .248	static 内隐类; 创建 classes 的风格; 子对象
Class .808;Class object .426,668,733;	Class; Class 对象
forName().428,798;getClass().383;	forName(); getClass()
getConstructors().447;getInterfaces().443;	getConstructors(); getInterfaces()

getMethods().447;getName().444;	getMethods();getName()
getSuperclass().443;isInstance. .438;	getSuperclass();isInstance
isInterface().444;newInstance().443;	isInterface();newInstance()
object .199;printInfo().444;RTTI using the Class object .441	对象; printInfo(); 使用 Class 对象的 RTTI
ClassCastException .312,430	ClassCastException
classpath .219,775;ant .948	类路径; ant
class-responsibility-collaboration (CRC) cards .1005	类职责协作 (CRC) 卡
cleanup :and garbage collector .252;	清理: 和垃圾收集
guaranteeing with finalize().186;	使用 finalize()来保证
performing .185;with finally .396	执行; 使用 finally
clear() , nio .617	clear(), nio
client programmer .38 ; vs .library creator .215	客户程序员与类库创建者相比较
clipboard , system .854	剪贴板, 系统
clone() .1028;and composition .1035;and inheritance .1042;	
Object .clone ().1032;removing/turning off cloneability .1044;super .clone().1032,1048;	clone();
和组合; 和继承; Object.clone(); 清除/关掉可克隆性; super.clone()	
supporting cloning in derived classes .1044	在派生类中支持克隆
Cloneable interface .1029	Cloneable 接口
CloneNotSupportedException .1032	CloneNotSupportedException
close() .606	close()
closure , and inner classes .357	闭包, 和内隐类
code :coding standards .24,1071;organization .225;re-use.241;revision control system.950;source	
code .22	代码: 编码标准; 组织; 复用; 版本控制系统; 源代码
codebase .773	codebase
coding style .103	编码风格
collection .481,560;class .453	集群; 类
Collections.enumeration().571	Collections.enumeration()
Collections .fill().484	Collections .fill()
Collections .reverseOrder().476	Collections .reverseOrder()
collision:during hashing .536;name .222	冲突: 在 hashing 过程中; 名称
com .bruceeckel .swing . 778	com .bruceeckel .swing
combo box .820	组合框
comma operator .128,150	逗号操作符
comments ,and embedded documentation .97	注释, 和嵌入式的文档
commit , CVS.953	提交, CVS
Commitment ,Theory of Escalating .760	提交, 逐步升高理论
common interface .293	公共接口
common pitfalls when using operators . 129	使用操作符时共有的缺点
Comparable .473,518	Comparable
Comparator .475,518	Comparator
compareTo(),in java.lang.Comparable.473	compareTo(), 在 java.lang.Comparable 中
comparing arrays .472	比较数组
compilation unit .217	编译单元

compile-time constant .264 编译期常量
compiling a java program .95 编译一个 java 程序
component ,and javaBeans .883 组件, 和 javaBeans
composition .40,241;and cloning .1035; 组合; 和克隆
and design .306;and dynamic behavior change .307;choosing composition vs.inheritance .258; 和
设计; 和动态行为变化; 在组合和继承之间进行选择
combining composition & inheritance .250;vs.inheritance .264 结合组合和继承; 与继承相比较
compression ,library .643 压缩, 库
concept ,high .1000 概念, 高
concurrency ,and Swing .875 并发, 和 Swing
ConcurrentModification Exception .566 ConcurrentModificationException
conditional operator .127 条件操作符
conference , Software Development Conference .10 会议, 软件开发会议
console :input .606;sending exceptions to .415;Swing display framework in com.bruceeckel.swing
777 控制台: 输入; 发送异常到; 在 com.bruceeckel.swing 中的 Swing 显示框架
const ,in C++ .1061 C++中的 const
constant :compile-time constant .264; constant:编译期常量
constant folding .264;groups of constant values .324;implicit constants ,and String .1061 常量
包入; 常量值组; 隐式常量和字符串
constrained properties .903 受约束的属性
constructor .165;and anonymous inner classes .335;and exception handling .404;
and exceptions .403;and finally .404;and overloading .169;
and polymorphism .297;arguments .167;base-class constructor .299; 构造器; 和匿名内隐类;
和异常处理; 和异常; 和 finally; 和重载; 和多态; 参数; 基类构造器
base-class constructors and exceptions .250; 基类构造器和异常
behavior of polymorphic methods inside constructors .303; 构造器中多态方法的行为
C++ copy constructor 1049;calling base-class constructors with arguments .249; C++拷贝构造
器; 调用基类的带参数的构造器
calling from other constructors 181; Constructor class for reflection .445; 从其他的构造器调
用; 用于反射的 Constructor 类
default .178;initialization during inheritance and composition .250; 缺省; 继承和组合中的初始
化
name .166;no-arg constructors .169;order of constructor calls with inheritance .298;
return value .168;static construction clause .199; 名称; 无参数的构造器; 继承中的构造器的
调用次序; 返回值; 静态构造子句
synthesized default constructor access.449 合成的缺省构造器的访问
consulting & mentoring provided by MindView , Inc..1085 MindView 提供的咨询和培训
container :class.453,480;of primitives .458;utilities for container classes .485 容器: 类; 基本型
别的; 容器类的作用
continue keyword .151 continue 关键字
contract ,design by (DBC),and assertions .934 基于契约的设计, 断言
control:access .39,236;framework ,and inner classes .360 控制: 访问; 框架, 和内隐类
conversion :automatic .243;narrowing conversion .130,177;
widening conversion .130 转换: 自动; 窄化转换; 宽化转换

copy:copying an array .470; deep copy .1028;shallow copy .1027 复制: 复制一个数组; 深层复制; 浅层复制
copyright notice :source code .22 版权声明: 源代码
costs ,startup .1018 费用, 启动
coupling .376 耦合
coverage testing .986 覆盖测试
CRC,class-responsibility-collaboration cards .1005 CRC; 类职责协作卡
CRC32 .645 CRC
critical section , and synchronized block .738 临界区, 和同步锁

D

daemon threads .710 后台线程
daily builds .953 每天的构建
data:final .264;primitive data types and use with operators .134;static initialization .196 数据:
final; 基本数据类型及其与操作符的使用; 静态初始化
data type:equivalence to class .35 数据类型: 与类型等价
DataFlavor .857 DataFlavor
DatagramChannel .640 DatagramChannel
DataInput .602 DataInput
DataInputStream 596,600,606,608 DataInputStream
DataOutput .602 DataOutput
DataOutputStream .598,601,608 DataOutputStream
DBC,design by contract , and assertions .934 DBC, 基于契约的设计, 和断言
dead,Thread .744 死亡, 线程
deadlock:conditions 756;multithreading .752 死锁: 条件; 多线程
debugger,JDK .978 除错器, JDK
decode(),character set .622 decode(),字符集
decorator design pattern .594 装饰设计模式
decoupling ,via polymorphism .49,279 解耦合, 通过多态性
decrement operator .115 自减操作符
deep copy .1028,1034;using serialization to perform deep copying .1040 深层复制; 使用序列化
进行深层复制
default constructor .178;access the same as the class .449;synthesizing a default constructor .249
缺省构造器; 与那个类相同的可访问性; 合成一个缺省构造器
default keyword ,in a switch statement .158 在 switch 语句中的 default 关键字
default package .227 缺省包
DefaultMutableTreeNode .849 DefaultMutableTreeNode
defaultReadObject().664 defaultReadObject()
DefaultTreeModel .849 DefaultTreeModel
defaultWriteObject().664 defaultWriteObject()
DeflaterOutputStream .643 DeflaterOutputStream
Delphi ,from Borland .882 Delphi ,来自 Borland
Demarco ,Tom .1094 Demarco ,Tom
dependencies, ant .948 依赖, ant
dequeue .481 两头开口的队列

derived: derived class .283;derived class ,initializing .248;type .42 派生：派生类；派生类，初始化； 型别

design .309;adding more methods to a design .237;analysis and design ,object-oriented .997;and composition .306;and inheritance .306;and mistake .237;five stages of object design .1007; 设计；在设计中加入更多的方法；分析和设计，面向对象；和组合；和继承；和错误；对象设计的 5 个阶段

library design .215;patterns .1012,1017 库的设计；模式

design by contract (DBC) ,and assertions .934 基于契约的设计，和断言

design patterns .5,235;decorator .594;singleton .235;template method .360,639,742

destructor .183 ,185,396;java doesn't have one .252 设计模式；装饰；单件；模版函数；析构函数；java 没有

development ,incremental .261 开发，增量式的

diagram: class inheritance diagrams .263;inheritance .50;use case .1002 图：类继承图；继承；用例

dialog box .838 对话框

dialog ,file .843 文件对话框

dialog ,tabbed .824 页签式对话框

dictionary .520 词典

digital signing .768 数字签名

dining philosophers , threading 753 哲学家就餐，多线程

directory: and packages .224;creating directories and paths .588;lister .584 目录：和包；创建目录和路径；制表者

display framework , for Swing .777 Swing 的显示框架

dispose() .839 dispose()

division .111 除法

doclets 991 小文档

documentation .12;comments & embedded documentation .97 文档；注释和嵌入式的文档

double: and threading .734;literal value marker (d or D) 132 double: 和多线程；用于确定值的字符（d 或 D）

do-while .148 do-while

downcast .263,310,430;type-safe downcast in run-time identification .430 向下转型；在执行期识别的型别安全的向下转型

drawing lines in Swing .835 在 Swing 中画直线

drop-down list .820 下拉列表

dynamic: behavior change with composition .307;binding .279,284 动态：用组合来改变行为；绑定

dynamic aggregate initialization syntax for arrays .457 数组的动态聚合初始化语法

E

early binding .48,284 前期绑定

East ,BorderLayout .786 East ,BorderLayout

editor ,creating one using the Swing JTextPane .816 编辑器，使用 Swing JTextPane 创建一个

efficiency :and arrays .453;and final .272;and threads .701 效率：和数组；和 final；和多线程

elegance ,in programming .1013 优雅，在程序设计中

else keyword .145 else 关键字

encapsulation .231 封装
encode(),character set .623 encode(), 字符集
endian:big endian .628;little endian .628 数端: 大数端; 小数端
enum ,groups of constant values in C & C++ .324 枚举, 在 C 和 C++中的一组常量值
Enumeration .570 Enumeration
equals() .118,518;and hashed data equals(); 和经过 hash 的数据
structures .533;conditions for defining property .531;overriding for HashMap .531 结构; 定义属性的条件; 对 HashMap 的覆写
equivalence := .117;object equivalence .117 相等: =; 对象相等
error :handling with exceptions .371; 错误: 与异常一起进行处理
recovery .491;reporting .411;reporting errors in book .25;reporting ,with the logging API in JDK 1.4 .954;standard errors stream .377 恢复; 报告; 报告书中的错误; 使用 JDK1.4 种的日志 API 进行报告; 标准错误流
Escalating Commitment ,Theory of .760 逐级提交, 理论
Event :event-driven programming .780;event-driven system .360; 事件: 事件驱动编程; 事件驱动系统
events and listeners .794;JavaBeans .883;listener .793; 事件和侦听器; JavaBeans; 侦听器
listener ,order of execution .872;model , Swing .793;multicast .871;侦听器, 执行次序; 模型, Swing; 群播;
multicast ,and JavaBeans .896;responding to a Swing event .780; 群播, 和 JavaBeans; 对一个 Swing 事件做出相应
Swing event model .870;unicast .871 Swing 事件模型; 单播
EventSetDescriptors .890 EventSetDescriptors
Evolution ,in program development .1010 演进, 在程序开发中
Exception :and base-class constructors .250;
and constructors .403;and inheritance .400,408;and the console .415; 异常: 和基类的构造器; 和构造器; 和继承; 和控制台
catching an exception .374;catching any exception .382; 捕捉一个异常; 捕捉任何异常
changing the point of origin of the exception .386; 改变异常的来源
checked .410;class hierarchies ;408;constructors .404; 检验过的; 类的层次; 构造器
converting checked to unchecked .416;creating your own .376;design issues .407; 将检查过的变为未检查的; 创建你自己的; 设计文档
Error class .392;Exception lass .392;exception handler .375;exception handling .371; Error 类; Exception 类; 异常处理器; 异常处理
exception matching .408;FileNotFoundException .406; 异常匹配; FileNotFoundException
fillInStackTrace() .384;finally .395; fillInStackTrace();finally
guarded region .374;handler .372; 警戒区; 处理器
handling .253;JDK 1.4 chained exceptions 416; 处理; JDK1.4 连锁的异常
losing an exception ,pitfall .399;NullPointerException .392;
丢失一个异常, 缺点; NullPointerException
printStackTrace() .384; printStackTrace()
restrictions .400;re-throwing an exception .384;RuntimeException .393; 限制; 重抛出一个异常; RuntimeException
specification .381,412;termination vs. resumption 376;Throwable .382;throwing an

exception .373;try .396;try block .375;typical uses of exceptions .418 说明书; 终止和恢复;
 Throwable; 抛出一个异常; try; try 模块; 异常的典型运用
 exceptional condition .372 异常条件
 exponential notation .132 幂符号
 extending a class during inheritance .44 在继承中对一个类型进行扩充
 extends .229,247,309;and interface .324;keyword .245 扩展; 和接口; 关键字
 extensible program .288 可扩展的程序
 extension ,pure inheritance vs.extension .308 扩展, 纯继承与扩展的比较
 extension ,sign .123 符号扩展
 extension ,zero .123 零扩展
 extensions ,ant949 扩展, ant
 Externalizable .656;alternative approach to using .662 Externalizable; 另外一个可以使用的方法
 Extreme Programming :and daily builds .953;XP .1013,1092 极限编程; 和每日构建; XP

F

fail fast containers . 566 当机立断出局的容器
 false . 119 false
 FeatureDescriptor . 903 FeatureDescriptor
 Field, for reflection . 445 Field, 为反射
 fields, initializing fields in interfaces . 327 域, 在接口中初始化域
 FIFO . 514 先进先出
 FIFO queue . 938 先进先出队列
 file: characteristics of files . 588; dialogs .
 843; File class . 584, 592, 601; File.list()
 . 584; incomplete output files, errors
 and flushing . 607; JAR file .217;
 locking . 640; memory-mapped files .
 636
 文件: 文件的特性; 对话框; File 类; File.list(); 不完整的输出文件; 错误和刷新; JAR 文
 件; 锁死; 内存映射文件
 File Transfer Protocol(FTP) . 774 文件传输协议 (FTP)
 FileChannel . 615 FileChannel
 FileDescriptor . 592 FileDescriptor
 FileReader . 605 FileReader
 FileInputStream . 592 FileInputStream
 FileLock . 640 FileLock
 FilenameFilter . 584 FilenameFilter
 FileNotFoundException . 406 FileNotFoundException
 FileOutputStream . 593 FileOutputStream
 FileReader . 404, 599 FileReader
 FileReader . 599, 607 FileReader
 fillInStackTrace () . 384 fillInStackTrace ()
 FilterInputStream . 592 FilterInputStream
 FilterOutputStream . 593 FilterOutputStream

FilterReader . 600 FilterReader
 FilterWriter . 600 FilterWriter
 final . 315; and efficiency . 272; and private
 . 269; and static . 264; argument . 268,
 587; blank finals . 267; classes . 271;
 data . 264; keyword . 264; method .
 284; methods . 269, 306; static
 primitives . 266; with object references .
 265
 Final; 和效率; 和 private; 和 static; 参数; 留白的 final; 类型; 数据; 关键字; 方法; 方
 法; static 基本型别; 和对对象引用
 finalize() . 183, 407; and inheritance .
 300; calling directly . 186 finalize(); 和继承; 直接调用
 finally . 253, 256; and constructors . 404;
 keyword . 395; pitfall . 399 finally; 和构造器; 关键字; 缺点
 finding.class files during loading . 219 在装载的时候寻找类文件
 flavor, clipboard . 854 风味; 剪贴板
 fip(), nio . 617 fip(),nio
 float: floating point true and false . 120 float: 浮点, true 和 false
 float, literal value maker(F) . 132 float: 用于确定数值的字符 (F)
 FlowLayout . 787 FlowLayout
 flushing output files . 607 刷新输出文件
 focus traversal . 767 焦点横移
 folding, constant . 264 包入, 常量
 for keyword . 148 for 关键字
 forName() . 428, 798 forName()
 FORTRAN . 132 FORTRAN
 forward referencing . 194 向前引用
 Fowler, Martin . 414, 998, 1011, 1093 Fowler, Martin
 Framework: application framework and
 applets . 707; control framework and
 inner classes . 360
 框架: 应用框架和 applets; 控制框架和内隐类
 FTP, File Transfer Protocol(FTP) . 774 FTP, 文件传输协议 (FTP)
 function: member function . 37; overriding
 . 44 函数: 成员函数; 覆写

G

garbage collection . 183, 186; and cleanup .
 252; forcing finalization . 256; how the
 collector works . 188; order of object
 reclamation . 256; reachable objects .
 545
 垃圾收集; 和清理; 强迫初始化; 收集器是如何工作的; 对象收回的次序; 可碰触的对象

generator . 484; object, to fill arrays and containers . 461	
生成器; 对象, 用于填充数组和容器	
generics:Java generics . 412	泛型: Java 泛型
get(): ArrayList . 492, 497; HashMap . 524	get(): ArrayList; HashMap
getBeanInfo(). 887	getBeanInfo()
getBytes(). 606	getBytes()
getChannel(). 616	getChannel()
getClass(). 383, 441	getClass()
getConstructor(). 807	getConstructor()
getConstructors(). 447	getConstructors()
getContentPane(). 771	getContentPane()
getContents(). 857	getContents()
getEventSetDescriptors(). 890	getEventSetDescriptors()
getInterfaces(). 443	getInterfaces()
getMethodDescriptors(). 890	getMethodDescriptors()
getMethods(). 447	getMethods()
getModel(). 849	getModel()
getName(). 444' 889	getName()
getPriority(). 710	getPriority()
getPropertyDescriptors(). 889	getPropertyDescriptors()
getPropertyType(). 889	getPropertyType()
getReadMethod(). 889	getReadMethod()
getSelectedValues(). 821	getSelectedValues()
getState(). 832	getState()
getSuperClass(). 443	getSuperClass()
getTransferData(). 857	getTransferData()
getTransferDataFlavors(). 857	getTransferDataFlavors()
getWriteMethod(). 889	getWriteMethod()
Glass, Robert . 1094	Glass, Robert .
glue in BoxLayout . 789	BoxLayout 中的胶水
GNU make . 945g	GNU 构造
goto, lack of in Java . 152	goto, 在 Java 中缺乏
graphical user interface(GUI) . 360, 765	图形化用户接口 (GUI)
graphics . 842; Graphics class . 836	图形; Graphics 类
greater than (>) . 117	大于 (>)
greater than or equal to (>=) . 117	大于等于 (>=)
greedy quantifiers . 677	贪婪标识符
GridBagLayout . 788	GridBagLayout
GridLayout . 788, 877	GridLayout
group, thread . 760	组, 线程
groups, regular expression . 681	组, 正则表达式
guarded region, in exception handling .	警戒区, 在异常处理中

GUI: graphical user interface . 360, 765; GUI: 图形化用户接口
 GUI builders . 766 GUI 构建者
guidelines: coding standards . 1071; object 方针: 编码标准; 对象开发
 development . 1008
GZIPInputStream . 643 GZIPInputStream
GZIPOutputStream . 643 GZIPOutputStream

H

handler, exception . 375 处理者, 异常
has-a . 40; relationship, composition . 259 有一个; 关系, 组合
hash function . 536 hash 函数
hashCode() . 515, 521, 529, 536; and
 hashed data structures . 533; issues
 when writing . 540; overriding for
 HashMap . 531; recipe for generating
 decent . 541
 hashCode(); 和经过 hash 的数据结构; 写作过程中的文章; 对于 HashMap 的覆写; 方法用于生成一个正规的
hashing . 533; and hash codes 529;
 external chaining . 536; perfect hashing
 function . 536
 hashing; 和 hash 码; 外部连锁; 完美的 hashing 函数
HashMap . 520, 549, 824 HashMap
HashSet . 515, 554 HashSet
Hashtable . 560, 571 Hashtable
hashNext(), Iterator . 498 hashNext(), Iterator
Hexadecimal . 131 十六进制
hiding, implementation . 38, 231 隐藏, 实现
high concept . 1000 高概念
HPROF, JDK profiler . 987 HPROF, JDK 刻画者
HTML on Swing components . 845 在 Swing 组件上的 HTML
HTMLConverter . 863 HTML 转化者

I

I/O: and threads, blocking . 745;
 available() . 606; between threads .
 750; blocking on I/O . 750; blocking,
 and available() . 607;
I/O: 和线程, 锁死; available(); 线程之间; 在 I/O 上的锁死; 锁死和 available ()
 BufferedInputStream . 596; BufferedInputStream
 BufferedOutputStream . 598; BufferedOutputStream
 BufferedReader . 404, 600, 605; BufferedReader
 BufferedWriter . 600, 607; BufferedWriter
 ByteArrayInputStream . 592; ByteArrayInputStream

ByteArrayOutputStream . 593; ByteArrayOutputStream
 characteristics of files . 588; 文件的特性
 charArrayReader . 599; charArrayReader
 CharArrayWriter . 599; CharArrayWriter
 CheckedInputStream . 643; CheckedInputStream
 CheckedOutputStream . 643; close() .
 606; compression library . 643; console
 input . 606; controlling the process of
 serialization . 656; DataInput . 602;
 CheckedOutputStream; close(); 压缩库; 控制台输入; 控制序列化的过程; DataInput
 DataInputStream . 596, 600, 606, 608;
 DataOutput . 602; DataOutputStream .
 598. 601. 608; DeflaterOutputStream .
 643; directory liste . 584; directory,
 creating directories and paths . 588;
 DataInputStream; DataOutput; DataOutputStream; DeflaterOutputStream; 目录列表; 目
 录, 创建目录和路径
 Externalizable . 656; File , 592, 601; File
 class . 584; File.list() . 584; Externalizable; File; File 类; File.list()
 FileDescriptor . 592; FileReader .
 605; FileInputStream . 592; FileDescriptor; FileReader; FileInputStream
 FileNameFilter . 584; FileOutputStream
 . 593; FileReader . 404, 599; FileWriter .
 599, 607; FilterInputStream . 592
 FileNameFilter; FileOutputStream; FileReader; FileWriter; FilterInputStream
 FilterOutputStream . 593; FilterReader .
 600; FilterWriter . 600; from standard
 input . 612; GZIPIputStream . 643;
 GZIPOutputStream . 643;
 FilterOutputStream; FilterReader; FilterWriter; 来自于标准输入; GZIPIputStream;
 GZIPOutputStream
 InflaterInputStream . 643; input . 590; InflaterInputStream; 输入
 InputStream . 590; InputStreamReader
 . 599; internationalization . 599; library
 . 583; lightweight persistence . 650;
 InputStream; InputStreamReader; 国际化; 库; 轻量级持久性
 LineNumberInputStream . 596;
 LineNumberreader . 600; mark() .
 603; mkdirs() . 590; network I/O . 615;
 new nio . 615; ObjectOutputStream .
 651; output . 590; OutputStream . 590'
 593; OutputStreamWriter . 599; pipe .
 591; piped streams . 610;
 LineNumberInputStream; LineNumberreader; mark(); mkdirs(); 网络 I/O; new nio;

ObjectOutputStream; 输出; OutputStream; OutputStreamWriter; 管道; 管道化的流
 PipedInputStream . 592; PipedInputStream
 PipedOutputStream . 592, 593; PipedOutputStream
 PipedReader . 599; PipedWriter . 600,
 607, 608; PushbackInputStream . 596;
 PipedReader; PipedWriter; PushbackInputStream
 Pushbackreader . 600; Pushbackreader
 RandomAccessFile . 601, 602, 608; RandomAccessFile
 read() . 590; readDouble() . 609; read();readDouble();
 Reader . 590, 598, 599; readExternal() .
 656; readLine() . 407, 600, 607, 613;
 readObject() . 651; redirecting standard
 I/O . 613; renameTO() . 590; reset() .
 603; seek() . 602, 609;
 Reader; readExternal(); readLine(); readObject(); 重定向到标准 I/O; renameTo(); reset();
 seek()
 SequenceInputStream . 592 , 601; SequenceInputStream
 Serializable . 656; setErr(PrintStream) .
 614; setIn(InputStream) . 614;
 setOut(PrintStream) . 614;
 Serializable; setErr(PrintStream); setIn(InputStream); setOut(PrintStream)
 StreamTokenizer . 606; StringBuffer .
 592; StringBufferInputStream . 592;
 Stringreader . 599, 606; StringWriter .
 599; System.ERR . 612; System.in . 606,
 612; System.out . 612; transient . 660;
 StreamTokenizer ; StringBuffer ; StringBufferInputStream ; Stringreader ; StringWriter ;
 System.ERR; System.in; System.out; transient
 typical I/O configurations . 603; 典型的 I/O 配置
 Unicode . 599; write() . 590; Unicode; write()
 writeBytes() . 608; writeChars() . 608;
 writeDouble() . 609; writeExternal() .
 656; writeObject() . 651; Writer . 590,
 598, 599; ZipEntry . 647;
 writeBytes(); writeChars(); writeDouble(); writeExternal(); writeObject(); Writer; ZipEntry
 ZipInputStream . 643; ZipInputStream
 ZipOutputStream . 643; ZipOutputStream
 Icon . 808; Icon
 if-else statement. 127, 145; if-else 陈述句
 IllegalMonitorStateException . 746; IllegalMonitorStateException
 ImageIcon . 808; ImageIcon
 immutable objects . 1055; 不可变的对象
 implementation . 36; and interface . 258,
 315; and interface, separating . 39; and

interface, separation . 231; hiding . 38,
231, 333; separation of interface and
implementation . 793
实现; 和接口; 和接口, 分离; 和接口, 分离; 隐藏; 分离接口和实现
implements keyword . 316 implements 关键字
import keyword . 216 import 关键字
increment operator . 115 自增操作符
incremental development . 261 增量式的开发
indexed property . 903 索引过的属性
indexing operator[] . 202 索引操作符[]
indexOf(), String . 448 indexOf(), 字符串
InflaterInputStream . 643 InflaterInputStream
inheritance . 41, 229, 241, 245, 279; and
cloning . 1042; and final . 272; and
finalize() . 300; and synchronized .
900; choosing composition vs.
inheritance . 258; class inheritance
diagrams . 263; combining composition
& inheritance . 250; designing with
inheritance . 306; diagram . 50;
extending a class during . 44; extending
interfaces with inheritance . 323; from
an abstract class . 294; from inner
class . 349; inheritance and method
overloading vs. overriding . 256;
initialization with inheritance . 274;
multiple inheritance in C++ and Java .
319; pure inheritance vs. extension .
308; specialization . 259; vs.
composition . 264
继承; 和克隆; 和 final; 和 finalize(); 和同步; 在组合和继承之间进行选择; 类继承图; 结
合组合和继承; 使用继承进行设计; 图; 在……之中扩展类; 使用继承扩展接口; 从一个抽
象类; 从一个内隐类; 继承和方法重载与覆写相比较; 继承中的初始化; C++和 Java 中的
多重继承; 纯继承与扩展相比; 特殊化; 与组合相比
initial capacity, of a hashMap or HashSet .
539
初始化 HashMap 或 HashSet 的容量
initialization: and class loading . 273; array
initialization . 202; base class . 248;
class member . 243; constructor
initialization during inheritance and
composition . 250; initializing class
member at point of definition . 193;
initializing with the constructor . 165;

instance initialization . 201, 340; lazy .
243; member initializers . 299; non-
static instance initialization . 201; of
class fields . 192; of method variables .
192; order of initialization . 195, 305;
static . 275; with inheritance . 274

初始化：和类的装载；数组的初始化；基类；类成员；在继承和组合中的构造器初始化；在
定义时对类成员进行初始化；使用构造器进行初始化；实例初始化；“懒惰”；成员初始化；
non-static 实例初始化；类的域；方法变量；初始化次序；static；在继承中

inline method calls . 269 内联函数调用

inner class . 331, 875; access rights . 342;
and super . 349; and overriding . 350;
and control frameworks . 360; and
Swing . 794; and upcasting . 333;
anonymous inner class . 586, 782; and
table-driven code . 551; callback . 357;
closure . 357; hidden reference to the
object of the enclosing class . 344;
identifiers and .class files . 354; in
methods & scopes . 335; inheriting from
inner classes . 349; nesting within any
arbitrary scope . 337; private inner
classes . 362; referring to the outer class
object . 347; static inner classes 343

内隐类；访问权限；和 super；和覆写；和控制框架；和 Swing ；和向上转型；匿名内隐类；
和表驱动代码；回调；闭包；隐藏的外围类的引用；标识符和.class 文件；在方法和范围之
内的；从内隐类继承；在任何范围内嵌套；private 内隐类；引用外围类的对象；static 内隐
类

input, console . 606 输入，控制台

InputStream . 590; InputStream

InputStreamReader . 599 InputStreamReader

insertNodeInto() . 849 InputStreamReader

instance: instance initialization . 340; non-
static instance initialization . 201; of a
class . 33

实例：实例初始化；non-static 实例初始化；一个类的

instanceof: dynamic instanceof . 438;

keyword . 430

instanceof: 动态的 instanceof; 关键字

Integer: parseInt() . 843; wrapper class .
205

Integer: parseInt(); 外覆类

interface: and implementation , separation
of . 39, 231, 793; and inheritance . 323;

base-class interface . 288; Cloneable
interface used as a flag . 1029; common
interface . 293; defining the class . 1014;
for an object . 34; initializing fields in
interfaces . 327; keyword . 315; name
collisions when combining interface .
322; nesting interfaces within classes
and other interfaces . 328; private, as
nested interfaces . 330; Runnable . 716;
upcasting to an interface . 319; user .
1004; vs. abstract . 321; vs.
implementation . 258

接口： 和实现，分离； 和继承； 基类接口； 将 Cloneable 接口作为 flag 使用； 公共接口； 定义类型； 为一个对象； 初始化接口中的域； 关键字； 组合接口时的名称冲突； 将接口与其他类型和接口嵌套； private， 作为嵌套的接口； Runnable； 向上转型到一个接口； 用户； 与 abstract 相比较； 与实现相比较

interfaces: graphical user interface(GUI) .

360, 756; responsive user . 722 接口： 图形化用户接口（GUI）； 反应灵敏的用户
internationalization, in I/O library . 599 国际化， 在 I/O 库中

Internet Service Provider(ISP) . 774 Internet 服务提供商（ISP）

interrupt() . 759; threading . 707, 713 interrupt(); 多线程

intranet . 769; and applets . 769 内部网； 和 applets

Introspector . 887 内省器

invariant, design by contract . 936 不变量， 基于契约的设计

invokeAndWait() , SwingUtilities . 880 invokeAndWait(), SwingUtilities

invokeLater() , SwingUtilities . 880 invokeLater(), SwingUtilities

is-a . 308; relationship, inheritance . 259;

and upcasting . 262; vs. is-like-a

relationship . 45

“是一个”； 关系， 继承； 和向上转型； 与 “好像一个” 关系相比

isDaemon() . 711 isDaemon()

isDataFlavorSupported() . 857 isDataFlavorSupported()

isInstance . 438 isInstance

isInterface() . 444 isInterface()

is-like-a . 309 “好像一个”

ISP(Internet Service Provider) . 774 ISP（Internet 服务提供商）

iteration, in program development . 1010 迭代， 在程序开发中

iterator . 497, 505, 549; hasNext() . 498;

next() . 498

迭代器； hasNext(); next()

J

J2EE.5 J2EE

Jacobsen, Ivar. 1093 Jacobsen, Ivar

JApplet. 785; menus. 827 JApplet; 菜单

JAR.901;archive tag,for HTML and JAR

Files.857;file.217;jar files and
classpath.221;packaging applets to
optimize loading.857;utility.648

JAR; 标记库, HTML 和 JAR 文件的; 文件; jar 文件和类路径; 将 applets 分包以优化装载;
作用

Java: and pointers.1021;and set-top
boxes.122;AWT.765;compiling and
running a program.95;containers
library.481;Java Foundation Classes
(JFC/Swing).765;Java Plugin.858;
Java Virtual Machine(JVM).426;Java
Web Start.864;operators.107;public
Java seminars.11;strategies for
transition to.1016;versions.24;why it
succeeds.72

Java: 和指针; 和置顶框; AWT; 编译和运行一个程序; 容器库;

Java 基础类 (JFC/Swing); Java 插件; Java 虚拟机 (JVM); Java Web Start;
操作符; Java 公开讨论会; 转变的策略; 版本; 它为什么成功

JavaBeans:see Beans.882 JavaBeans: 见 Beans

javac.96 javac

javadoc.97 javadoc

JavaMail API.969 JavaMail API

JButton.808;Swing.779 JButton; Swing

JCheckBox.808,817 JCheckBox

JCheckBoxMenuItem.828,832 JCheckBoxMenuItem

JComboBox.820 JComboBox

JComponent.810,835 JComponent

JDB,Java Debugger.978 JDB,Java 除错器

JDialog.838;menus.827 JDialog; 菜单

JDK 1.1:I/O streams.598 JDK1.1: I/O 流

JDK 1.4.7;assertion mechanism.930;
chained exceptions.416;Java Web
Start.863;LinkedHashMap.527;
LinkedHashSet.515;logging API.954;
new I/O.615;preferences API.673;
regular expressions.675

JDK 1.4: 断言机制; 连锁异常; Java Web Start;

LinkedHashMap; LinkedHashMap; 日志 API; 新 I/O;
优先选择的 API; 正则表达式

JDK,downloading and installing.95 JDK, 下载和安装

JFC,Java Foundation Classes(Swing).765 JFC, Java 基础类 (Swing)

JFileChooser.843 JFileChooser

JFrame.778,785;menus.827 JFrame; 菜单

JIT,just-in-time compilers.191	JIT, 即时编译器
JLabel.771,812	JLabel
JList.821	JList
JMenu.827,832	JMenu
JMenuBar.827,833	JMenuBar
JMenuItem.808,827,832,833,835	JMenuItem
JNLP,Java Network Launch Protocol.864	JNLP, Java 网络发射协议
join(),threading.713	join(), 多线程
JOptionPane.825	JOptionPane
Joy,Bill.116	Joy,Bill
JPanel.785,806,835,878	JPanel
JPopupMenu.834	JPopupMenu
JProgressBar.847	JProgressBar
JRadioButton.808,819	JRadioButton
JScrollPane.785,814,823,849	JScrollPane
JSlider.847	JSlider
JTabbedPane.824	JTabbedPane
JTable,Swing.850	JTable, Swing
JTextArea.783,854	JTextArea
JTextField.781,810	JTextField
JTextPane.816	JTextPane
JToggleButton.806	JToggleButton
JTree.847	JTree
JUnit unit testing framework.925	JUnit 单元测试框架
JVM(Java Virtual Machine).426	JVM (Java 虚拟机)

K

Key,applet signing ·861 关键词: applet 签名

Keyboard:navigation,and Swing ·767; Shortcuts 833 键盘: 导航, Swing 快捷方式

keySet() ·560

Koenig,Andrew ·1073

L

Label ·153 标签

Labeled:break ·153;continue·153 带标签的: break; continue

Late binding ·48,279,284 后绑定

Layout,controlling layout with layout managers ·785 布局, 用布局管理器控制布局

Lazy initialization ·243 后初始化

Left_shift operator(<<)·123 左移运算符

Length:array member·204;for arrays·455 Length: 数组长度

Less than(<)·117 小于

Less than or equal to (<=)·117 小于等于

Lexicographic vs. client programmer·215;design·215;use·216 词典 vs 客户程序员; 设计; 使用

LIFO·513 后进先出

Lightweight:persistence-650;Swing components-767 轻量：持久；
LineNumberInputStream-596
LineNumberReader-600
Linked list-481
LinkedHashMap-483,520,527,560
linkedHashSet-515,557
LinkedList-509514,554
Linux-7
List:boxes-821;drop-down list-820 列表：框；下拉列表
List-454,480,481,509,821;sorting and searching-560 列表：分类与查找
Listener:adapters-800;and events-794;classes-875;interfaces-799 侦听适配器；事件；类；接口
Lister,Timothy-1094
ListIterator-509
Literal:class literal-428,436;double-132;float-132;long-132;value-131 标记：类标记
Little endian-628 小数端
Load factor,of a HashMap or HashSet-539 载入因数，
Loading:.class files-219;initialization&class loading-273;loding a class-275 载入：类文件；初始
化与类载入；载入一个类
Lock(),file locking-640
Lock,for multithreading-732
Locking,file-640 阻塞，文件
Logarithms,natural-132 对数；自然对数
Logging API in JDK 1.4-954 JDK 1.4-954 中记录 API
Logical:AND-129;operator and short-circuiting-120;operator-119;OR-129 逻辑：与；运算符与
短路；运算符：或
Long,and threading-734 long 线程
Long,literal value marker(L) -132 long 数值标记
Look&feel,pluggable-852
lvalue-108

M

Mail,Simple Mail Transfer Protocol (SMTP)-971 邮件传输协议
Main()-246
Maintenance,program-1010 维持；程序
Make utility,for program building-944 使有效；程序建构
makefile-944 创建文件
management obstacles-1018 管理障碍
manifest file,for JAR files-648,901 验证文件
Map-454,480,481,520,557 映射
Map-Entry-533 映射入口
MappedByteBuffer-636
Mark()-603
Matcher,regular experssion-680 匹配器，常规表达式
Math.random()-524;values produced by-161

Mathematical operators·111 数学运算符

Max()·561

Member:initializers·299;member function·37;object·40 成员初始化;成员函数;对象

Memory exhaustion,solution via references·545 内存耗尽,通过引数解决;

Memory-mapped files·636 内存映射文件

Mentoring and training·1019,1020 指导与训练

Menu:JDialog,JApplet,JFrame·827;JPopupMenu·834 目录

Message box,in Swing ·825 消息框

Message,sending·35 消息,发送

Messenger object·959 消息对象

Method:adding more methods to a design·237 加入更多方法

- Aliasing during method calls·110,1022 调用中的重名
- Behavior of polymorphic methods inside constructors·303 构造器中多态方法的行为
- Distinguishing overloaded methods·171 区分重载方法
- final·269,284,306
- initialization of method variables·192 方法变量初始化
- inline method calls·269 内联方法调用
- inner classes in methods&scopes·335 方法与范围的内隐类
- lookup tool·796 查询工具
- method call binding·284 方法调用绑定
- overloading ·168 重载
- overriding private·292 覆写私有
- passing a reference into a method ·1022 为方法传递引用
- polymorphic method call·279 多态方法调用
- private ·306;protected methods·260 private; protected 方法
- recursive·501 递归
- static·182,284
- synchronized method and blocking·745 同步方法与阻塞

Method·890;for reflection·445

MethodDescriptors·890

Methodology,analysis and design·997

Meyers,Scott·38

Microsoft·904;Visual BASIC·882

Min()·561

Mission statement ·1000

Mistakes,and design·237

Mkdirs()590

Mnemonics(keyboard shortcuts) ·833

modulus·111 系数

monitor,for multithreading·732 监视器;多线程

mono·68

mozilla·66

multicast·895;event,and JavaBeans·896;multicast events·871 群播;事件,与 JavaBeans;群播事件

multidimensional arrays·208 多维数组
multiparadigm programming·33 多范例编程
multiple inheritance,in C++ and Java·319 多充继承, 在 C++与 Java 中
multiplication·111 乘法
multiply-nested class·348 多重嵌套类
multitasking·699 多任务
multithreading·699;and containers·566;blocking·744;deciding what methods to
synchronize·900;drawbacks·761;runnable·875;when to use it 761
多线程; 容器; 阻塞; 决定使用何种方法同步; 缺陷; 可运行; 何时使用
multi_tiered systems·872 多层系统
mutex·731 互斥体
mutual exclusion,threading·731 互斥, 线程

N

name·773;clash·216;collision·222;collisions when combining interfaces·322;creating unique
package names·219;spaces·216
名字; 冲突; 冲突; 连接接口的冲突; 创建唯一的包名; 空间
narrowing conversion·130,144,177 窄化转换
natural logarithms·132 自然对数
nesting interfaces·328 嵌套接口
network I/O ·615 网络输入输出
new I/O ·615
new operator ·183;and primitives,array ·204 新操作数; 基本型别数组
newInstance()·808;reflection ·443 newInstance(); 反射
next(),iterator ·498 next(); 叠代器
nio ·615;buffer ·615;channel ·615 nio; 缓冲; 通道
no-arg constructors ·169 无参数构造器
north borderlayout ·786
not equivalent(!=)·117
Not,logical(!)·119
Notify()·745
Notifyall() ·745
Notifylisteners()899
Null·82
NullPointerException·392 空指针异常
numbers,binary·131 数字, 二进制

O

object•33;aliasing•110;arrays are first-class objects•455;assigning objects by copying
references•108;business object/logic•872;Class
object•668,733;creation•166;equivalence•117;equivalence vs.reference
equivalence•117,1032;final•265;five stages of object design•1007;guidelines for object
development•1008;immutable objects•1055;interface to•34; lock, for multithreading•732;
member•40;object-oriented programming•424;process of creation•199;serialization•650;web of

objects•651,1028
Object•454;Class
object•426;clone()•1028,1032;equals()•118;getClass()•441;hashCode()•521;standard root class,
default inheritance from•245;wait() and notify()•746;
对象; 别名; 数组是一级对象; 复制引用部署对象; 业务对象/逻辑; 类对象; 创建, 等价,
等价 vs 引用等价; final; 对象设计的五个阶段; 对象设计的方针; 持久对象; 接口; 锁,
多线程; 成员; 面向对象编程; 创建过程; 序列化; 对象的网, 对象;
类对象; clone(); equals(); getClass(); hashCode(); 标准根类, 缺省继承, wait()与 notify()
object-oriented: analysis and design•997;basic concepts of object-oriented programming(OOP)•31;
面向对象: 分析与设计; 面向对象编程的基本概念
ObjectOutputStream•651;
obstacles,management•1018; 障碍, 管理
Octal•131; 八进制
ones complement operator•122; 一元运算符
OOP•231;analysis and design•997;basic characteristics•33;basic concepts of object-oriented
programming•31;protocol•316;Simula-67 programming language•34;substitutability•33;
面向对象; 分析与设计; 基本特性; 面向对象编程的基本概念; 协议; Simula-67 编程语言;
替换性
operation,atomic•734; 操作, 自动
operator•107;+ and += overloading for String•246;+, for String•1061;== and !=
•1032;binary•122;bitwise•122;casting•130;comma operator•128,150;common
pitfalls•129;indexing operator[]•202;logical•119;logical operators and
short-circuiting•120;ones-complement•122;operator overloading for
String•1061;overloading•128;precedence•108;precedence
mnemonic•133;relational•117;shift•123;ternary•127;unary•115,122;
运算符; 用于 String 的+与+=; String 的+, ==与!=; 二进制; 位宽; 转型; 逗号运算符; 常
见缺点; 索引运算符[]; 逻辑的; 逻辑运算符与短路; 一元; String 运算符重载; 优先, 优
先记忆; 关系的; 切换; 三重的; 一元的
optimizing, and profiling•985; 优化与刻画;
optional methods, in the Java 2 containers•567; 可选方法, 在 Java 2 容器里
OR•129;(||)119; 或
order:of constructor calls with inheritance•298;of initialization•195,273,305; 继承中构造器调用
的顺序; 初始化
organization,code•225; 组织, 代码
OutputStream•590,593;
OutputStreamWriter•599;
overflow, and primitive types•143; 溢出, 基本型别
overloading;and constructors•169;distinguishing overloaded methods•171;lack of name hiding
during inheritance•256;method overloading•168;on return values•177;operator + and +=
overloading for String•246;operator overloading•128;operator overloading for String•1061;
overloading vs.overriding•256;
重载: 构造器; 区分重载方法; 在继承中没有名字隐藏; 方法重载; 返回值; 用于 String 运
算符+与+=重载; 运算符重载; String 的运算符重载; 重载与覆写
overriding;and inner classes•350;function•44;overloading vs.overriding•256;private methods•292;
覆写: 内隐类; 函数; 重载与覆写; 私有方法

P

Package•216;access, and friendly•225;and directory structure•224;creating unique package names•219;default package•227;names, capitalization•90;package access, and protected•260;visibility,access•333;

包; 访问, friendly; 目录结构; 创建唯一的包名; 缺省包; 名称, 大写; 包访问, protected; 可见性, 访问

paintComponent()•835,842;

painting on a JPanel in Swing•835; 在 Swing 中的 JPanel 上绘图

pair programming•1015; 结对编码

paralysis,analysis•998; 麻痹, 分析

parameter,applet•773; 参数, applet

parameterized type•497; 参数化类型

parseInt()•843;pass:pass by value•1026;passing a reference into a method•1022; 传递: 值传递; 传递引用到方法

pattern, regular expression•675; 模式, 常规表达式

patterns,design•235,1012,1017; 模式, 设计

perfect hashing function•536; 完美的哈希函数

performance:and final•272;issues•1019; 性能与 final; 报告

persistence•665;lightweight persistence•650; 持续; 轻量持续

PhantomReference•545;philosophers,dining and threading•753;

pipe•591; 管道

PipedStreams•610;

PipedInputStream•592;

PipedOutputStream•592,593;

PipedReader•599,750;

PipedWriter•599,750;

planning,software development•1000; 计划, 软件开发

Plauger, P.J.•1094;pluggable look & feel•852;

Plugin,Java•858

;pointer:and Java•1021;Java exclusion of pointers•357; 指针与 Java; Java 中没有指针

polymorphism•47,279,312,424,450;and constructors•297;behavior of polymorphic methods inside constructors•303; 多态; 构造器; 在构造器中多态方法的行为

portability in C,C++ and Java•133; C,C++ 与 Java 可携带性

position, absolute, when laying out Swing components•789; 位置, 绝对, 当布局 Swing 组件

possessive quantifiers•677; 所有格标识符

postcondition,design by contract•936; 后置条件, 基于契约的设计

precedence,mnemonic operator precedence•133; 优先; 记忆运算符优先

precondition,design by contract•936; 前提条件

preferences,JDK1.4API•673; 参考

prerequisites,for this book•31; 这本书的准备知识

primitive:comparison•118;containers of primitives•458;data types, and use with operators•134;dealing with the immutability of primitive wrapper classes•1055;final•264;final static primitives•266;initialization of class fields•192;types•80;wrappers•525;

基本型别比较; 基本型别容器; 数据类型, 使用运算符; 处理基本型别外覆类的持久性; **final**;
final static 基本型别; 初始化类的域; 类型; 外覆
printlnInfo()•444;
printing arrays•462; 打印数组
println()•499;
printStackTrace()•383,384;
PrintStream•598;
PrintWriter•600,607,608;
Priority, thread•708; 优先级, 线程
private•39,215,224,228,260,732;illusion of overriding private methods•269;inner
classes•362;interfaces,when nested•330;method overriding•292;methods•306;
problem space•32,262;
private; 设想覆写 **private** 方法; 内隐类; 借口, 嵌套; 覆写方法; 方法
process, and threading•699; 进程, 多线程
producer-consumer, threading•747; 生产者消耗; 多线程
profiling; and optimizing•985;JVM interface•986; 刻画与优化; JVM 接口
programmer, client•38; 编程人员; 客户
programming; basic concepts of object-oriented programming (OOP)•31;coding
standards•1071;event-driven programming•780;Extreme Programming(XP)•1013,1092;in the
large•73;maintenance•1010;multiparadigm•33; object-oriented•424;pair•1015;
编程: 面向对象编程的基本概念; 编码规范; 事件驱动编程; 极限编程, 在大型维护中; 多
个范例; 面向对象; 对
progress bar•846; 进度条
promotion, to int•133,143; (基本型别) 转换, 到实型
property•883;bound properties•903;custom property editor•903;custom property
sheet•903;indexed property•903; 属性; 边界属性; 定制属性编辑器; 定制属性表; 索引属
性
PropertyChangeEvent•903;
PropertyDescriptor•889;
PropertyVetoException•903;
protected•39,215,224,229,260;and package access•260;is also package access•231;use in
clone()•1029;
protected; 包访问; 包访问; 在 **clone()** 中使用
protocol•316; 协议
prototyping, rapid•1012; 原型化, 快速的
public•39,215,224,226;and interface•316;class, and compilation units•217; ; 借口, 类, 编译单
元
pure: inheritance, vs. extension•308;substitution•45,308; 纯继承 vs 扩充继承; 取代
PushbackInputStream•596;
PushBackReader•600;
put(), HashMap•525;
Python•75•1007;

Q

quantifier:greedy•677;possessive•677;regular
expression•677;reluctant•677;queue•481,514;FIFO•938;
标识符：贪婪；所有格；常规表达式；冗余；队列；先进先出

R

RAD(Rapid Application Development)•444; 快速应用开发
radio button•819; radio 按钮
random number generator, values produced by •161; 随机数产生器，数值从下面函数产生
random()•524;
RandomAccessFile•601,602,608,617;
rapid prototyping•1012; 快速原型化
reachable objects and garbage collection•545; 可及对象与垃圾回收
read()•590;NIO•617;
readDouble()•609;
Reader•590,598,599;
readExternal()•656;
reading from standard input•612; 读取标准输入
readLine()•407,600,607,613;
readObject()•651;with Serializable•662;
recursion,unintended via toString()•500; 重现，通过 toString()非故意的
redirecting standard I/O•613; 重定向标准输入输出
refactoring•1011;
reference:assigning objects by copying references•108;final•265;finding exact type of a base
reference•426;null•82;reference equivalence vs.object equivalence•117,1032;
引用：通过复制引用部署对象；final；查找基引用的精确类型；空；参考等价 vs 对象等价
reference counting garbage collection•188; 引用计算垃圾回收
Reference,from java.lang.ref•545;
referencing forward•194;reflection•444,445,796,886;and Beans•883;and weak
typing•414;difference between RTTI and reflection•446;example•806;

regex•675;
regular expressions, JDK1.4•675; 规则表达式
relational operators•117; 关系运算符
reluctant quantifiers•677; 冗余标识符
removeActionListener()•794;
removeXXXListener().794;
renameTo()•590;
reporting errors in book•25; 在书中报告错误
request, in OOP•35; 需求
requirements analysis•1001 需求分析
reset()•603;
responsive user interfaces•722; 响应用户接口
resume()•745;and deadlocks•758; resume(); 死锁
resumption, termination vs.resumption,exception handling•376; 恢复，中止 vs 恢复，异常处理

re-throwing an exception•384; 重抛出一个异常
return:an array •459;constructor return value•168; overloading on return value•177; 返回: 一个数组; 构造器返回值; 返回值的重载
reusability•40; 可重用性
reuse•1008;code reuse•241;existing class libraries•1018;reusable code•882; 重用; 代码重用; 存在的类库; 可重用代码
revision control system,source code •950; 修订控制系统, 源代码
rewind()•622;
right-shift operator(>>)•123; 右移位运算符
rollover•810;
RTTI:and cloning•1033;cast•425;Class object•426,441,808;ClassCastException•430;Constructor class for reflection•445;downcast•430;Field•445;getConstructor()•807;instanceof keyword•430;isInstance•438;Method•445;newInstance()•808;reflection•444;reflection,difference between•446;run-time type identification•311;type-safe downcast•430;
RTTI: 克隆; 转型; 类对象; 构造器类; 向下转型; 域; getConstructor(); instanceof 关键词; isInstance; 方法; newInstance();
Rumbaugh,James•1093;
Runnable•875;interface•716;Thread•744; 可运行; 接口; 线程
running a Java program•95; 运行一个 Java 程序
run-time binding•284;polymorphism•279; 运行时绑定; 多态
run-timetype identification:(RTTI)•311;misuse•450;shape example•423;when to use it•450; 运行时型别判定: 误用; shape 实例; 何时使用;
RuntimeException•393,416,454,914;rvalue•108; 运行时异常

S

Safety, and applet restrictions•768; 安全, applet 限制
scenario•1002; 脚本
scheduling•1004; 调度
scope:inner class nesting within any arbitrary scope•337;inner classes in methods scopes•335;use case•1010; 范围: 嵌套任意范围的内隐类; 在方法范围内的内隐类; 使用实例
scrolling in wing•785; 在侧面滚动
searching:an array•478;sorting and searching Lists•560; 查询数组; 分类与查询列表
section,critical section and synchronized block•738; 区, 临界区
seek()•602,609;
semaphore•729; 信号量
seminars:public Java seminars•11;training,provided by MindView,Inc.•1085;
sending a message•35; 发送消息
separating business logic from UI logic•872; 从 UI 逻辑中分离商业逻辑
separation of interface and implementation•39,231,793; 接口与实现分离
SequenceInputStream•592,601;
Serializable•650,656,661,671,894;readObject()•662;writeObject()•662;
serialization:and object storage•665;and transient•660;controlling the process of serialization•656;defaultReadObject()•664;defaultWriteObject()•664;to perform deep copying•1040;Versioning•665; 序列化接口: 对象恢复; 瞬时; 控制序列化过程;

defaultReadObject(); defaultWriteObject(); 进行深拷贝; Versioning
ServerSocket•640;
Set•454,480,481,515; 集
setActionCommand()•833; setActionCommand()
setBorder()•812; setBorder()
setContents()•856; setContents()
setErr(PrintStream)•614; setErr(PrintStream)
setIcon()•809; setIcon()
setIn(InputStream)•614; setIn(InputStream)
setLayout()•785; setLayout()
setMnemonic()•833; setMnemonic()
setOut(PrintStream)•614; setOut(PrintStream)
setPriority()•710; setPriority()•710
setToolTipText()•810; setToolTipText()
setup(),JUnit•926; setup(),JUnit
shallow copy•1027,1034; 浅拷贝
shape:example•42,285;example, and run-time type identification•423; shape: 实例; 实例, 执
行期类型识别
shift operators•123; 切换运算符
short-circuit, and logical operations•120; 短路, 逻辑操作
shortcut,keyboard•833; shortcut 关键词
show()•840; show()
shuffle()•561; shuffle()
side effect•107,116,178,1024; 边界效应
sign extension•123; 标记拓展
signed applets•858; 被标记的 applets
signed two's complement•127; 有符号二补数
Simula-67 programming language•34,231; 编程语言
sine wave•835; 正弦波
singleton design pattern•235; singleton 设计模式
size(),ArrayList•492;
size,of a HashMap or HashSet•539;
sizeof(),lack of in Java•133; sizeof() Java 中没有
sleep()•745;threading•706;
slider•846; 滑动条
Smalltalk•33,183;
SMTP,Simple Mail Transfer Protocol•971; 简单邮件传输协议
SocketChannel•640;
SoftReference•545;
Software Development Conference•10;
software development methodology•998; 软件开发方法
solution space•32; 解决空间
sorting•473;and searching Lists•560; 分类; 查找列表
source code•22;copyright notice•22;revision control system•950; 源代码; 版权; 修订控制系统

South, BorderLayout•786;
space:name spaces•216;problem space•32;solution space•32; 空间: 名字空间; 问题空间; 解
决空间
specialization•259; 特殊化
specification:exception specification•381,412;system specification•1001; 规范: 异常规范, 系
统规范
specifier:access specifiers•39,215,224; 修饰词, 访问修饰词
Stack•513,572;
standard input, reading from•612; 标准输入, 读取
standards, coding•24,1071; 标准, 编码
startup costs•1018; 启动代价
stateChanged()•838; stateChanged()
statement,mission•1000; 陈述, 任务
static•315;and final•264;block•199;construction clause•199;data initialization•196;final static
primitives•266;initialization•275;initialization clause•428;iner
classes•344;keyword•182;method•182,284;strong type checking•410,909;synchronized static•733;
static; final; 块; 创建语句; 数据初始化; final static 数据原型; 初始化; 初始化语句; 内
隐类; iner 关键词; 方法; 强类型检验; 同步的 static
STL,C++•481;
stop():and deadlocks•758;deprecation in Java 2•758;
stop(): 死锁; 在 Java 中不提倡
stream, I/O•590; 流, 输入输出
StreamTokenizer•600;
String:automatic type conversion•495;class methods•1060;concatenation with operator+
•128;immutability•1060;indexOf()•448;lexicographic vs.alphabetic
sorting•477;methods•1063;operator+•128,495;operator + and +=
overloading•246;toString()•242,493;
String; 自动类型转换; 类方法; 用+串接; 永恒性; indexOf(); 词典 vs 字母排序; 方法; +
运算符; +=运算符;
StringBuffer•592;methods•1066;
StringBufferInputStream•592;
StringReader•599,606;
StringSelection•856;
StringTokenizer•693;
StringWriter•599;
strong static type checking•410,909; 对强静态类型的检验
structs,in BorderLayout•789;
style:coding style•103;of creating classes•232; 风格; 编码风格; 创建类
subobject•248,258; 子对象
substitutability, in OOP•33; 可替换性
substitution principle•45; 替换策略
subtraction•111; 减法
super•249;and inner classes•349; 超类; 内隐类
super keyword•247;

super.clone()•1029,1032,1048; super.clone()
superclass•247; 超类
suspend()•745;and deadlocks•758; suspend()与死锁
Swing•765;and concurrency•875;component examples•804;component,using HTML
with•845;event model•793,870;
Swing, 同步; 组件实例; 组件, 使用 HTML; 事件模型
switch keyword•157;
synchronized•61,732;and inheritance•900;and wait() & notify()•745;block, and critical
section•739;containers•566;deciding what methods to synchronize•900;method, and
blocking•745;static•733
同步的, 继承; wait(), notify(), 阻塞; 关键域; 容器; 决定同步的方法同步; 方法, 与阻
塞; static
system clipboard•854 系统剪切板
system specification•1001 系统规范
system.arraycopy()•470 system.arraycopy()·
system.err•377,612;redirecting•918 system.err; 重定向
system.in•606,612 system.in
system.out•612;redirecting•918 system.out; 重定向
system.out.println()•499 system.out.println()
systemNodeForPackage(),preferences API•674 systemNodeForPackage(), preferences API

T

Tabbed dialog•824 表格会话
Table,Swing JTable•850 表, Swing JTable
Table-driven code, and anonymous inner classes•551 表驱动代码, 匿名内隐类
Tasks,ant•948 任务, ant
teardown,jUnit•926 拆卸, jUnit·
template method, design pattern•360,639,742 模板方法, 设计模式
template, in c++•497 模板
templates, Java generics(equivalent) ·412 模板, Java 范型 (等价)
termination condition, and finalize()•186 终止条件, finalize()
termination vs. resumption, exception handling•376 终止与恢复, 异常处理
ternary operator•127 三元操作数
testing: automated•1014; automated unit testing•911; Extreme Programming(XP) ·1013; simple
framework•913; techniques•346; unit testing•247
测试, 自动的; 自动单元测试; 极限编程; 简单框架; 技术; 单元测试
theory of escalating Commitment•760
thinking in C#•68
thinking in enterprise java•6
this keyword•179 this 关键字
thread 699; and runnable•875; blocked•744; daemon
threads•710; dead•744; deadlock•752; deciding what methods to
synchronize•900; drawbacks•761; getPriority()•710; group•760; I/O
between•750; interrupt()•759; isDaemon()•711; long and double non-atomicity•734; new

Thread-744;notify()-745;notifyAll()-745;performance-989;priority-708;resume()-745;resume(),and
deadlocks-758;run()-701;Runnable-744;Runnable interface-716;setPriority()-710;sharing limited
resources-723;sleep()-745;start()-703;states-744;stop(),deprecation in Java 2-758;stop(),and
deadlocks-758;suspend()-745;suspend(),and deadlocks-758;synchronized method and
blocking-745;threads and efficiency-701;wait()-745;when to use threads-761

线程;可运行的线程;被阻止的;守护线程;死亡;死锁;决定同步何种方法;缺陷;getPriority();
组; I/O; interrupt(); isDaemon(); long 与 double non-atomicity; 新线程; notify(); notifyAll();
性能; 优先级; resume(); resume(), 死锁; run(); 可运行的; 可运行接口; setPriority(); 共
享有限资源; sleep(); start(); 状态; stop(); Java 中不提倡的; stop(), 与死锁; suspend();
suspend()与死锁; 同步的方法与阻塞; 线程与效率; wait(); 何时使用线程

throw keyword-374 抛出关键词

Throwable-386;base class for Exception-382 可抛出; 用于异常的基类

throwing and exception-373 抛出与异常

timer-726 定时器

toArray()-560 toArray()

tool tips-810 工具提示

TooManyListenersException-871,895 过多监听器异常

ToString()-242,493,499,549 ToString()

training-1017;and mentoring-1019,1020;seminars provided by mindView,Inc. -1085 训练; 指导;
mindView,Inc 提供的讨论

Transferable-857 可传输的

transferForm()-618 transferForm()

transferTo()-618 transferTo()

transient keyword-660 瞬时关键词

translation unit-217 翻译单元

tree-847 树

treeMap-520,560 树映射

treeSet-515,554 树集合

true-119 true

try-256,396;try block in exceptions-375 try; 在异常中的 try 模块

tryLock(),file locking-640 tryLock(), 文件锁定

two's complement,signed-127 二补数, 有符号的

type:base-42;data type equivalence to class-35;derived-42;finding exact type of a base
reference-426;parameterized type-497 类型; 基类; 等同于类的数据类型; 导出; 找出原本
引用的精确类型; 参数化类型

U

UML-1006;indicating composition-40;Unified Modeling Language-37,1093 统一建模语言; 表
示组合

Unary.minus(-)-115;operator-122;operator-115;PLUS(+)-115 一元: 减号, 运算符; 运算符,
加号

unchecked exceptions,converting from checked-416 未检验异常, 从已检验转换

unicast-895;unicast events-871

Unicode-599 统一字符编码标准

Unified Modeling Language(UML)-37,1093 统一建模语言
Unit testing-247;automated-911 单元测试; 自动
Unmodifiable,making a Collection or Map
unmodifiable-564 不可修改; 使集群与映射不可修改
unsupported methods,in the Java 2
containers-576 不支持的方法, 在 Java 2 容器中
UnsupportedOperationException-567 不支持的操作异常
upcasting-50,262,279,424;and interface-319;inner classes and upcasting-333 向上转型, 接口;
内隐类与向上转型
update, CVS-952 更新, CVS
updates of the book-24 书的更新
use case-1001;iteration-1010;scope-1010 应用实例; 反复; 范围
userinterface-1004;responsive, 用户接口; 响应, 使用多线程
with threading-700,722
userNoderForPackage(),preferences API-674 userNoderForPackage(), 优化 API

V

Value,preventing change at run time-264 数值: 避免在执行期改变
Variable:defining a variable-150; 变量: 定义一个变量
 initialization of method variables-192; 方法变量初始化
 variables argument lists (unknown quantity and type of arguments) -206; 变量参数列表 (未知数量和类型的参数)
Vector-554,570,572
vector of change-362,1011 修改向量
versioning,serialization-665 版本控制, 序列化
versions of Java-24 Java 版本
visibility, package visibility-333 可视化, 可视化包
Visual BASIC, Microsoft-882
visual programming-882; environments-766 可视化编程, 环境
volatile-734 瞬时

W

wait()-745
Waldrop, M.Mitchell-1095
WeakHashMap-547
weakly typed language-48 弱类型语言
WeakReference-545
Web: placing an applet inside a Web page-771; safety, and applet restrictions-768 Web: 将一个
applet 置于一个 Web 页面内部; 安全和 applet 限制
web of objects-651,1028 对象网
Web Start, Java-864
West, BorderLayout-786
while-147
white-box testing-943 白盒测试

widening conversion·130 拓宽转换
wild-card·998 通配符
windowClosing()·839
windows applications·775 视窗化应用
wrapper, dealing with the immutability of primitive wrapper classes·1055 包装器, 处理原始类型包装器类的恒定性
write()·590;nio·618
writeBytes()·608
writeChars()·608
writeDouble()·609
writeExternal()·656
writeObject()·651;with Serializable·662
Writer·590,598,599

X

XOR(Exclusive-OR)·122
XP(Extreme Programming)·1013 极限编程

Y

yield(), threading·704,731

Z

zero extention·123
ZipEntry·647
ZipInputStream·643
ZipOutputStream·643

www.mindview.net

exceptional learning experiences

Thinking in Java Hands-On Seminar

Learn the programming language
of the World Wide Web

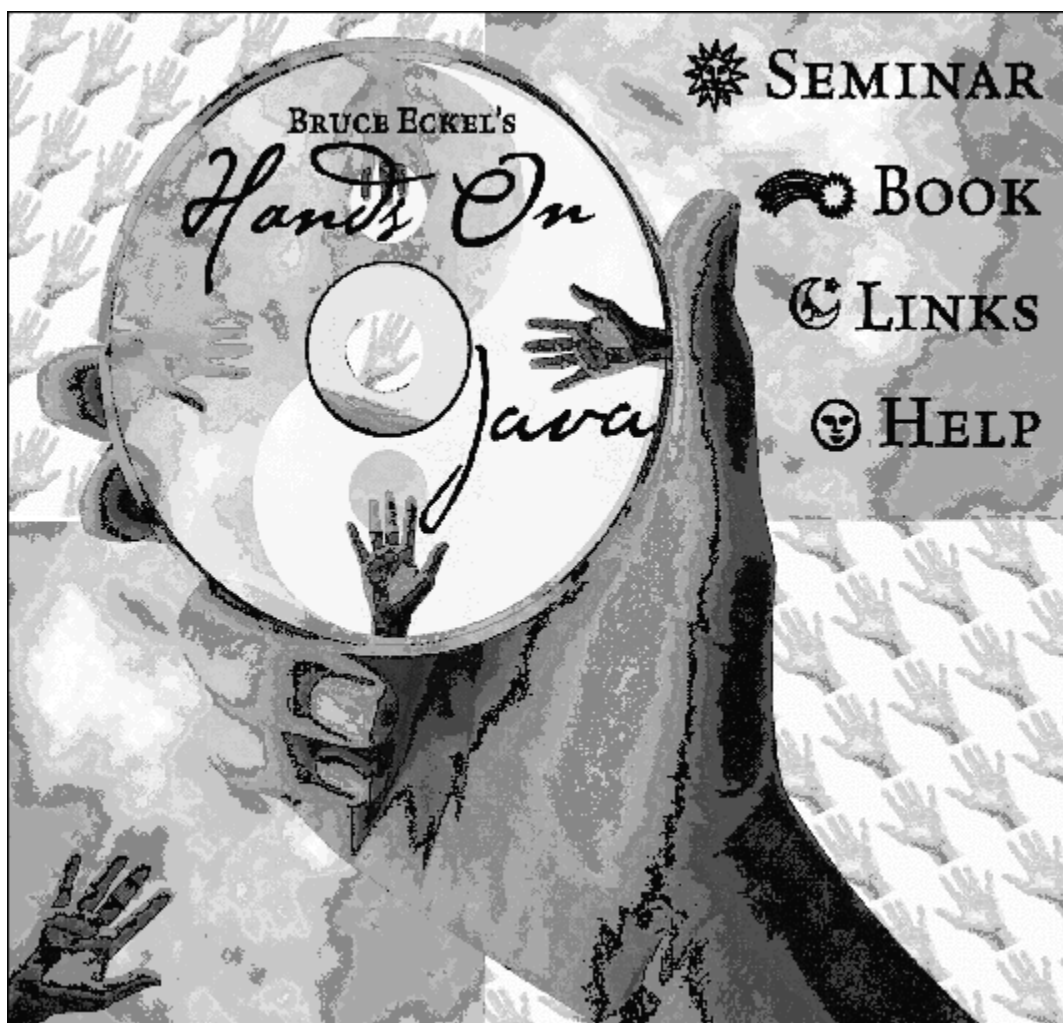
This step-by-step seminar covers each
essential subject in a lecture followed by
coached programming exercises.

This course is the gateway to more advanced MindView seminars.
You're ready if you've been through the *Foundations for Java CD*
included with the book, *Thinking in Java*, 3rd edition.

查看 www.MindView.net 网站

以便得到更多细节以及下一次 **Java 编程思想研讨会** 召开的日期和场所，该研讨会：

- ◆ 基于本书
- ◆ 由最好的 MindView 小组成员讲授
- ◆ 请在研讨会期间亲自留意
- ◆ 包括课堂上的程序设计练习
- ◆ 也提供中/高级研讨会
- ◆ 数以百人已经享受了这次研讨会——请见网站上他们的感谢信



Bruce Eckel's Hands-On Java Seminar

Multimedia CD: 3rd Edition follows this book

It's like coming to the seminar!

Available at *www.BruceEckel.com*

- ◆ Java 研讨会的手稿可以从多媒体光盘上获取！
- ◆ 所有的讲稿都有幻灯片和同步音频声音进行讲解。你仅需播放来观看和倾听讲稿。
- ◆ 由 Bruce Eckel 开创和讲授。
- ◆ 基于本书中的材料。
- ◆ 示例讲稿可从网站 *www.BruceEckel.com* 获取。

MindView 公司的授权协议书，该公司是 Bruce Eckel 和 Chuck Allison 制作的 Java 多媒体光盘的基地。

这张光盘由 “Java 编程思想（第 3 版）” 这本书配套提供。

READ THIS AGREEMENT BEFORE USING THIS "Foundations for Java" (Hereafter called "CD"). BY USING THE CD YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, IMMEDIATELY RETURN THE UNUSED CD FOR A FULL REFUND OF MONIES PAID, IF ANY.

©2003 MindView, Inc. All rights reserved. Printed in the U.S.

软件需求

这张光盘的目的是提供本书的一些内容，而不是提供查看这些内容所需的相关软件。这张光盘的内容是 **HTML** 格式，可以通过网络浏览器（已经用微软的 **Internet** 浏览器 6 和 **Mozilla** 测试过，对于 **Mozilla** 见网站 www.Mozilla.org）以及 **MP3** 播放器（例如可以从 www.Real.com 免费下载的 **RealPlayer**）来查看。不过，你要自己负责为自己的系统正确地安装合适的软件。

本光盘包含的文本、图像以及其他的媒介和对它们的编辑授权你遵守 **MindView** 公司制定的这项协议中的各种条款和限制条件，该公司在 **La Mesa** 市 **Valle** 街 5343 号的 91941 认证中心有一个营业所。你使用包含在这张光盘上的其他程序和材料的权利是由与光盘上那些程序和材料（也就是“其他协议”）一起分配的发布协议所支配的。如果这个协议和其他协议相冲撞，这个协议应该处于支配地位。通过使用这张光盘，你允诺遵守这个协议的条款和限制条件。**MindView** 公司，除了它所包含的属于第三方供应商材料的内容范围，拥有本书内容和所有的知识产权的授权资格。除了那些在这个协议中已经明确授权给你的权利，本书内容的其他所有权限属于 **MindView** 公司，并且这些供应商的各自利益也显而易见。

1. 有限权限

MindView 公司授予一种有限的、非独家的、不可转让的，可以在单台专用计算机（不包括网络服务器）上使用的权限。如果你不能遵守这个协议或其他任何协议的任何条款，此协议和你的权限在此之下应该自动终止。在终止之时，你允诺毁掉这张光盘和这张光盘的所有备份，无论法律是允许的还是不允许的，它是你的权利也由你来掌控。

2. 额外限制

- a. 你不应该（并且不应该允许其他人或单位）直接地或间接地，通过电子的或其他方式，复制（除了法律允许的存档目的）、出版、发布、出租、租赁、出售、发布从属证书、分配或以其他方式传递该光盘的内容或其中任何部分。
- b. 你不应该（并且不应该允许其他人或单位）为了商业利益使用本光盘或其中的任何部分，或者通过合并、修改来创建无创意的作品或者翻译这些内容。
- c. 你不应该（并且不应该允许其他人或单位）无视 **MindView** 或它的供应商的版权、商标或来自于光盘任何部分或者任何相关材料的所有权通告或说明。

3. 许可

- a. 除了本光盘内容表明的，你应当把本软件仅仅当作一本书来看待。不过，你可以将它备份到计算机中使用，并且是出于备份这个软件以防止你的投资损失这个唯一目，将它备份存档的。顺便提一下，**MindView** 公司声称“仅仅当作一本书来看待”意思是指，这个软件可以让很多人使用并且可以自由地从一台计算机转移到另一台计算机，只要不会

在一个地方或一台计算机上正在使用而此时也在另一个地方使用就行。正如一本书不能在同一时间不同地点供不同的人阅读，软件也不能在同一时间在不同地点供不同的人使用。

- b. 只要你所有材料的内容都是由 **MindView** 公司所提供的，你就可以按照现场演讲、现场研讨会或者现场运行的方式，展示或示范未被修改过的内容。
- c. 使用这张光盘的其他许可证和授权必须从 **MindView** 公司的网站 <http://www.MindView.net> 上直接获取。（在此网站上也可以购买这张光盘的大量备份。）

4. 担保的责任规范声明

内容和光盘仍按照原来的样子来提供的，而没有任何类型的保证，或者明确的或者含蓄的，包含的、不限定的、任何厂商的授权和用于某一特殊目的的商业行为。关于光盘和内容的全部风险有你承担。**MindView** 公司和它的供应商不承担任何像光盘中的瑕疵，内容的准确性或者在光盘和内容中的疏忽这样的责任。**MindView** 公司和它的供应商不担保、承担，或者不对使用，使用的效果，以及根据正确性、准确性、可靠性、通用性得知的产品效果发表看法，那些内容或者会满足你的需求，或者光盘的操作是连续的或者无误的，或者书中或光盘中的任何纰漏都是正确的。**MindView** 公司和它的供应商不承担由于光盘的使用或对本书的翻译而引起的任何损失、损坏或花费。一些声明不允许的禁令，或不允许隐含保证的限制规定，或不允许对偶然的或作为结果而发生的损失的责任限制，因此上面所有的限制规定或禁令不适用于你。

MindView 公司或它的供应商为你所有的损伤、损失以及行为因素（无论是以合约、民事纠纷或者其他方式）担负的全部责任不可能超过你支付这张光盘的费用。

MindView 公司和 **Prentice Hall** 公司明确拒绝用于某种特殊用途的商业目的的隐含保证。**MindView** 公司和 **Prentice Hall** 公司没有给出任何口头或书面信息或通知，他们的经销商、发行商、代理商或员工应该任命一个被担保人。你可能具有其他权利，这会随国家的不同而有所不同。

IN NO EVENT SHALL MINDVIEW, INC., BRUCE ECKEL, CHUCK ALLISON, PRENTICE HALL, NOR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION OR DELIVERY OF THE PRODUCT BE LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS CD AND ITS CONTENT, OR ARISING OUT OF THE INABILITY TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW, INC., OR ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. MINDVIEW, INC. SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING SERVICES FROM MINDVIEW, INC., AND MINDVIEW, INC. HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

这张光盘是这本“**Java** 编程思想（第 3 版）”的补充材料。**Prentice Hall** 公司的唯一职责是

在随书配套提供的光盘发生故障的情况下，提供另一张替代光盘。这张替代光盘担保从购买日期的 60 天内有效。MindView 公司不为这张光盘承担其他任何职责。

没有为这张光盘提供任何技术支持

下面是这些公司在美国的商标，在其他国家也应该是保护商标：Sun 和 Sun Logo, Sun Microsystems, Java, 所有基于 Java 的名称、理念和 Java Coffee Cup 都是 Sun Microsystem 的注册商标；RealPlayer 和 RealOne Player 是 RealNetwork 公司的注册商标；Internet Explorer、the Windows Media Player、DOS、Windows 95、Windows 98、Windows NT、Windows 2000 和 Windows XP 是 Microsoft 的注册商标。

Java 基地

Multimedia Seminar-on-CD ROM

©2003 MindView, Inc. All rights reserved

WARNING: BEFORE OPENING THE DISC PACKAGE, CAREFULLY READ THE TERMS AND CONDITIONS OF THE LICENSE AGREEMENT & WARANTEE LIMITATION ON THE PREVIOUS PAGES.

本书的光盘包是一个由同步幻灯片和语音讲座组成的多媒体研讨会。这个研讨会的目的是向你介绍进入 Java 所必需的基础原则。这张光盘还包括本书中源代码的一个超链接，本书 HTML 版本的一个超链接以及本书第 1 和第 2 HTML 版本形式的超链接。

这张光盘可以在具有音响系统的大多计算机上运行，包括 Windows, Linux, 和 Mac OS/9 and OS/10。不过，你必须：

1. 如果你没有安装一个 web 浏览器，请在你的机器上安装一个。测试过的浏览器包括 **Mozilla (www.Mozilla.org)** 和 **Microsoft Internet Explorer** 的最新版本。
2. 在你的机器上安装一个 MP3 音频播放器。这些幻灯片设计成是使用来自于 **www.Real.com** 的免费 **RealPlayer**，或者是其他的 MP3 播放器（较新的机器在安装前就配套有 MP3 播放器）。
3. 此时这里你应该能够播放光盘上的讲座。使用最近版本的 **Internet Explorer** 或者 **Mozilla** 网络浏览器，打开会在光盘上看到的 **Install.html** 文件。它会向你介绍这张光盘，并介绍怎样更进一步使用光盘。注意：在 **Window** 机器上或一些 **Linux** 机器上，当你将光盘插入播放器时，这张光盘会自动运行