计算机科学丛书

Java 编程思想

第 二 版

Bruce Eckel　著

智慧东方工作室　译

目　　录

写在前面的话

我的兄弟 Todd 目前正在进行从硬件到编程领域的工作转变。我曾提醒他下一次大革命的重点将是遗传工程。

我们的微生物技术将能制造食品、燃油和塑料；它们都是清洁的，不会造成污染，而且能使人类进一步透视物理世界的奥秘。我认为相比之下电脑的进步会显得微不足道。

但随后，我又意识到自己正在犯一些科幻作家常犯的错误：在技术中迷失了（这种事情在科幻小说里常有发生）！如果是一名有经验的作家，就知道绝不可就事论事，必须以人为中心。遗传对我们的生命有非常大的影响，但无法肯定它会抹淡计算机革命——或至少信息革命——的影响。信息涉及到人与人之间的沟通：的确，汽车和轮子的发明都非常重要，但它们最终亦如此而已。真正重要的还是我们与世界的联系，而其中最关键的便是通信！

这本书或许能说明一些问题。许多人认为我有点儿大胆或者稍微有些狂妄，居然把所有家当都摆到了 Web 上。“这样还有谁会买它呢？”他们问。假如我是一个十分守旧的人，那么绝不这么干。但我确实不想再沿原来的老路写一本计算机参考书了。我不知道最终会发生什么事情，但的确相信这是我对这本书作出的最明智的决定。

至少有一件事是可以肯定的，人们开始向我发送纠错反馈。这是一个绝对令人震惊的体验，因为读者能看到书中每一个角落，并揪出那些藏匿得很深的技术及语法错误。这样一来，和其他以传统方式发行的书不同，我就能及时改正已知的所有类别的错误，而不是让它们最终印成铅字，堂而皇之地出现在各位的面前。俗话说，“当局者迷，旁观者清”。人们对书中的错误是非常敏感的，往往毫不客气地指出：“我想这样说是错误的，我的看法是……”。在我仔细研究后，往往发现自己确有不当之处，而那些都是当初写作时根本没有意识到的（检查多少遍也不行）。我意识到这是群体力量一个可喜的反映，它使这本书最终显得与众不同！

但我随之又听到了另一个声音：“好吧，你在那儿放的电子版的确很有创意，但我想要的是从真正的出版社那里印刷出来的一个版本！”事实上，我作出了许多努力，让它用普通打印机机就能得到很好的阅读效果，但仍然不象真正印刷的书那样正规。许多人不想在屏幕上看完整本书，也不喜欢拿着一叠纸阅读。无论打印格式有多么好，这些人喜欢是仍然是真正的“书”（激光打印机的墨盒也太贵了一点）。现在看来，计算机的革命仍未使出版界完全走出传统的模式。但是，有一个学生向我推荐了未来出版的一种模式：书籍将首先在互联网上出版，然后只有在绝对必要的前提下，才会印刷到纸张上。目前，为数众多的书籍销售都不十分理想，许多出版社都在亏本。但如采用这种方式出版，就显得灵活得多，也更容易保证赢利。

这本书也从另一个角度也给了我深刻的启迪。我刚开始的时候以为 Java“只是另一种程序设计语言”。这个想法在许多情况下都是成立的。但随着时间的推移，我对它的学习也愈加深入，开始意识到其基本宗旨与我见过的其他所有语言都有所区别。

程序设计与对复杂性的操控有很大的关系：针对一个打算解决的问题，它的复杂程度取决用于解决它的机器的复杂程度。正是由于这一复杂性的存在，我们的程序设计项目屡屡失败。对于我以前接触过的所有编程语言，它们都没能跳过这一框框，由此便决定了它们的主要设计目标是克服程序开发与维护中的复杂性。当然，许多语言在设计时就已考虑到了复杂性的问题。但从另一个角度看，实际设计时肯定又会有另一些问题浮现出来，你必须同时把它们考虑到这个“复杂性”的问题里。不可避免地，其他那些问题最后会变成最令程序员头痛的。例如，C++必须同 C 保持向后兼容（使 C 程序员能尽快地适应新环境），同时又要保

证编程效率。C++在这两个方面都设计得很好，为其赢得了不少的声誉。但它们同时也暴露出了一些额外的复杂性，从而阻碍了某些项目的成功实现（当然，你可以责备程序员和管理层，但假如一种语言能通过捕获你的错误而提供帮助，它为什么不那样做呢？）。作为另一个例子，Visual Basic（VB）同当初的 BASIC 有着紧密联系。而 BASIC 并没有打算设计成一种能全面解决问题的语言，所以堆加到 VB 身上的所有扩展都造成了令人头痛和难于管理和维护的语法。另一方面，C++、VB 和其他如 Smalltalk 之类的语言均在复杂性的问题上下了一番功夫。由此得到的结果便是，它们在解决特定类型的问题时是非常成功的。

在理解到 Java 最终的目标是减轻程序员的负担时，我才真正感受到了震憾，尽管它的潜台词好象是说：“除了缩短时间和减小产生健壮代码的难度以外，我们不关心其他任何事情。”在目前这个初级阶段，达到那个目标的后果便是代码不能特别快地运行（尽管有许多保证都说 Java 终究有一天会运行得多么快），但它确实将开发时间缩短到令人惊讶的地步——几乎只有创建一个等效 C++程序一半甚至更短的时间。节省下来的时间可用来产生更大的效益，但 Java 并不仅止于此。它甚至更上一层楼，将重要性越来越明显的一切复杂任务都封装在内，比如网络程序和多线程处理等等。Java 的各种语言特性和库在任何时候都能使那些任务轻而易举完成。而且最后，它解决了一些真正有些难度的复杂问题：跨平台程序、动态代码改换以及安全保护等等。换在从前，其中任何一个都能使你头大如斗。所以不管我们见到了什么性能问题，Java 的保证仍然是非常有效的：它使程序员显著提高了程序设计的效率！

在我看来，编程效率提升后影响最大的就是 Web。网络程序设计以前非常困难，而 Java 使这个问题迎刃而解（而且 Java 也在不断地进步，使解决这类问题变得越来越容易）。网络程序的设计要求我们相互间更有效率地沟通，而且至少要比电话通信来得便宜（仅仅电子邮件就为许多公司带来了好处）。随着我们网上通信越来越频繁，令人震惊的事情会慢慢发生，而且它们令人吃惊的程度绝不亚于当初工业革命给人带来的震憾。

在各个方面：创建程序；按计划编制程序；构造用户界面，使程序能与用户沟通；在不同类型的机器上运行程序；以及方便地编写程序，使其能通过因特网通信——Java 都提高了人与人之间的“通信带宽”。而且我认为通信革命的结果可能并不单单是数量庞大的比特到处传来传去那么简单。我们得认清真正的革命到底发生在哪里——人和人之间的交流变得更方便了。个体与个体之间，个体与组之间，组与组之间，甚至星球与星球之间。有人预言，下一次大革命的发生就是由于有足够多的人和足够多的相互联系而造成的，而这种革命是以整个世界为基础发生的。Java 可能是、也可能不是促成那次革命的直接因素，但我在这里至少感觉自己在做一些有意义的工作——尝试教会大家一种重要的语言！

第二版前言

针对本书第一版，许多人给我提供了许多出色的意见。当然，非常高兴能有这样的结果，我对大家的热心支持，表示感谢。但是，我也听到了少许抱怨的声音。而且出于某方面的原因，其中一项抱怨甚至是“这本书的个头太大了。”之所以会有如此抱怨，假如这些人唯一的理由便是“页码太多”，那么我可能真的马上就要“晕倒”。这使我想起了一则著名的笑话——奥地利皇帝曾向莫扎特抱怨说：你的曲子里音符太多！（注意我并不打算以任何方式自喻莫扎特）。除此之外，我还能想到的一个理由是，这些人尚未体验到 Java 语言本身的复杂性，而且并没有通读完全书。例如，我最喜爱的一本参考书是由 Cay Horstmann 和 Gary Cornell 合著的《Core Java》（由 Prentice-Hall 出版）。那本书的“个头”甚至还要大，以至于不得不分成上下两卷。不过，在本书第二版的编纂过程中，我确实也在减小体积方面下了一番功夫，删掉了一些已经过时或者不再重要的章节。我这样做并不觉得十分“可惜”，因为第一版的全文仍然保留在我的网站和第二版的配套光盘中，仍然采用可供免费下载的格式（网站地址是 www.BruceEckel.com）。如果你想参考第一版的内容，不妨大胆下载，这对作者来说，也是一个不小的“安慰”。例如，你会发现原书的最后一章“项目”已从新版里取消了；有两个项目已被合并到其他章节里，另一些项目则不再适合“新形势”了。另外，“设计范式”一章由于内容太多，现已转移到专门的一本书里（也可从网站免费下载）。因此，至少从表面上看，新书会变得“苗条”得多。

但不幸的是，实情并非如此！

最大的问题是 Java 语言本身也在不断进步。最值得一提的便是它的 API 扩展集，曾许诺向开发人员提供大量标准接口，涉及你能想到的几乎所有方面。这样说罢，假如最终出现了一个名为 JToaster 的 API（作者开的一个玩笑，一种假想的 API，专门用于“烤面包”，取其无所不包之意——译注），我可能一点儿都不会觉得奇怪！不过，尽管这些问题不容忽视，但对所有类进行详细探讨已超出了本书的范围，那恐怕该是其他作者的事情！

在这些 API 中，最重要的无疑是在服务器一端运行的 Java（主要包括 Servlets 和 Java 服务器页，即 JSP），它们是解决万维网（WWW）问题的一个出色方案。请想想，现在存在着形形色色的 Web 浏览器平台，它们相互间的兼容性极差。仅采用客户端编程的手段，恐怕并不能真正解决我们的问题。另一个大问题是，如何方便地创建一个应用程序，令其实现与数据库的沟通、电子商务、数据保密等等？这便涉及到“企业 Java 豆”（EJB）的运用。所有这些主题都被归纳到原书的“网络编程”一章里；在新书中，则正式更名为“分布式计算”。随着时代的进步，“分布式计算”已成为所有人关心的一个主题。在这一章中，我还为大家引入了一个新概念，那就是“Jini”（发音作“genie”，但并非什么缩写形式，而是一个专门的称呼）。它是一种非常时髦的技术，可从根本上转变我们的编程和思考问题方式。也就是说，它能让我们换从另一个角度，来看待不同程序间的沟通。而且理所当然地，本书从头到尾都改用了 Swing GUI 库。不过同样地，假如你需要老的 Java 1.0/1.1 内容，请从 www.BruceEckel.com 下载本书第一版（配套光盘上也有，后面还会详细说明）。

全书除针对 Java 2 的语言特点作了大量修订和增补之外，专门讲述集合的那一章（第 9 章）也将重点完全转到了 Java 2 的集合上。另外，那一章的深度也有所增加，论述了与集合有关的一些重要问题，特别是散列函数的工作原理（使你知道该如何正确创建它）。本书还有另一些值得注意的删补和变动，其中包括完全重写了第 1 章，以及删除了我认为不再适合在印刷版中出现的一些附录及其他内容。总之，在第二版中，我删除了那些不再需要或者不

再适合 Java 2 的内容（尽管依然保留在第一版），同时增加了大量新内容，尽可能改进了我能想到的每一方面，以适应新形势的需要。随着语言本身的不断进步（尽管步子已迈得不象以前那么大），本书往后无疑还会有新的修订版本出台。

对那些仍然觉得本书“个头”过大的人，我在这里表示抱歉。尽管你可能不相信，但我确实已作出了最大努力，来尽量减少它的体积。不过，抛开“个头”不谈，新书仍有足够精采之处，保证能让你满意。至少，本书仍然有自己的免费电子版（从网站下载，配套光盘上也有）。因此，假如你有一部笔记本电脑，便完全不必携带沉重的书本到处走动。再进一步，假如你有一部 PDA，甚至还能下载到这本书的 Palm Pilot 版本，进一步为它“减肥”。这里说一句题外话，有个人告诉我说，他每天晚上最舒适的事情便是躺在床上，用自己的 Palm 阅读这本书，同时打开屏幕背光，免得干扰自己的老婆（我真心希望他这样会很快入睡）。如果你不喜欢在计算机屏幕上阅读，我也知道有些人每次都打印出本书的一章，看完后再接着打印下一章。

Java 2

新书修订时，Sun 公司 Java 开发包（JDK）的 1.3 版本已临近正式发布日期，而 JDK 1.4 的一些技术细节也已公布。尽管所有这些版本号全以“1”开头，但根据大家约定俗成的规矩，自 JDK 1.2 开始的所有版本都通称“Java 2”！也就是说，自 Java 1.2 开始，“老 Java”已发生了重大变化，它以全新的面貌，出现在世人面前！

事实上，本书第一版便是围绕“老 Java”展开的。读完那本书，你便知道老 Java 存在着诸多缺点，人们对此颇多抱怨。但 Java 2 不同，语言本身发生了根本性变化，成为一种更为“现代”的语言。设计人员为此进行了大量辛勤工作，作出了大量改进和补充，使其成为一种更为合理、令人更舒服的语言！

本书的第二版便是完全针对 Java 2 编写的。在这本书中，我大刀阔斧删除了不再适合 Java 2 的大量内容，一切都围绕新语言展开。之所以丝毫不觉得可惜，是由于删除的内容依然全文保留在本书第一版的电子版中，网站和配套光盘上都可找到（如果仍然要用 Java 2 之前的一个版本进行编程，那些内容便是你必需的）。另外，既然任何人都可从 java.sun.com 免费下载 JDK 的升级版本（Java 2），所以我在这里也要鼓励所有开发者都升级自己的 Java 版本，其间不会涉及到任何升级费用。而在这本书的帮助下，一切都会来得轻松加愉快！

不过，关于版本号，这里也要提醒大家注意一点，JDK 1.3 有一些改进是我非常喜欢的，但到了 Linux 平台，JDK 1.3 的版本号变成了 JDK 1.2.2。Linux（见 www.Linux.org）和 Java 绝对是非常班配的一对，它运行 Java 有着足够快的速度，是目前最重要的服务器平台之一——快速、可靠、健壮、安全、易于维护而且免费！是计算史上一次真正的革命（以前从未有一种平台同时具有全部这些特点）。而采用 Servlet 的形式，Java 已被证明是一种出色的服务器端编程环境。和传统 CGI 编程相比，Servelet 技术是一个重大的进步（详情可见本书“分布式计算”一章）。

因此，尽管我更乐意全部采用最新的语言特性，但前提是一切都在 Linux 环境下编译，所以假如你安装有 Linux，那么解开源码文件，并在 Linux 下编译（当然要用最新的 JDK），那么一切都会正确无误。当然，在代码中出现了 JDK 1.3 特有的一些功能时，本书都加上了必要的注释，提醒你注意。

配套光盘

随同本书，你还会得到一片配套光盘，就卡在本书的后面。过去，我并没采用随书送光盘的做法，因为觉得容量如此巨大的光盘，却只装少得可怜的几 KB 源代码，好象有点儿“大

材小用”。为此，我更乐意让大家从网站下载源码。不过，这一次有所不同，你会发现该光盘采用了非常独特的设计。

当然，本书用到的所有源码都可在光盘上找到。但除此以外，整本书也包括到了其中，并采用了几种文件格式。其中，我最喜欢的一种格式是 HTML，因为它调用速度够快，而且有完整的索引链接——点按索引或目录中的一项，便能直接跳至相应位置。

光盘共计 300 多兆内容，其中“块头”最大的是一堂多媒体课程，名为“Thinking in C:C++ 和 Java 基础”。最开始，我打算委托 Chuck Allison 做一张多媒体光盘，把它作为一个独立的产品销售。但到最后一刻，我改变了主意，决定随《Thinking in C++》和《Thinking in Java》这两本书的第二版，把它免费送给读者。原因很简单，我发现许多人在开始学习这两种语言的时候，都没有足够的 C 编程经验！

有的人甚至这样想：“我很聪明呀，不必先学 C，再来学 C++或 Java！”但在实际授课过程中，我却发现这些人尽管确实很“聪明”，但最后都慢慢落后于那些有 C 语言基础的同学，原因不言而喻。

因此，鉴于 C 基本语法的重要性，光盘提供了这方面的辅导，请你根据自己的实际情况，决定是否该“补一下课”，先熟悉一下 C 语言的基础知识。

另外，正是由于提供了 C 的基础课程，本书的读者群似乎还可以扩大。尽管第 3 章（控制程序流程）也讲到了 Java 和 C 的一些共同点，但光盘课程显得更加全面、更易上手，因为它假定自己的授课对象只掌握了极少的 C 知识，对学生的要求比本书的要求低得多！

在这儿，我唯一的希望就是通过光盘和这本书，把更多的人吸引到 Java 编程中来，壮大 Java 开发人员群体，共同促进这个优秀语言的发展！

引　言

同人类使用的任何语言一样，Java 也为我们提供了一种手段，来表达自己的思想。

如运用得当，和其他方式相比，随着问题变得愈大和愈复杂，这种表达方式的方便性和灵活性会愈加显露无遗。

但要注意的是，不可将 Java 简单想象成一系列特性的集合；孤立地看，有些特性是没有任何意义的。只有将重点放在“设计”上、而非放在简单的“编码”上，才可真正体会出 Java 的强大！为了以这种方式理解 Java，首先必须掌握它和编程的一些基本概念。本书向大家列举了一系列编程问题，解释了它们为何会成为问题，并讲述了如何用 Java 来解决这些问题。为此，本书进行了特别编排，每一章都讲述了如何用语言来解决一种特定类型的问题。采用这种方式，我希望引导您一步一步进入 Java 的世界，最终把它变成您所习惯的、最自然的一种“语言”。

贯穿本书，我试图在您的大脑里建立一个模型——或者说一个“知识结构”。这样可加深对语言的理解。若遇到难解之处，应学会把它填入这个模型对应的地方，然后自行演绎出答案。事实上，学习任何语言时，假如脑海里已有一个现成的知识结构，往往会起到事半功倍的效果。

1. 前提

本书假定读者多少有些编程经验。应知道程序是一系列语句的集合，知道子程序／函数／宏是什么，知道象“If”这样的控制语句，也知道象“while”这样的循环结构，等等。注意这些东西在大量语言里都是类似的。假如您学过一种宏语言，或者用过 Perl 之类的工具，那么它们的基本概念其实并无什么区别。总之，只要能习惯基本编程概念，便可顺利阅读本书。当然，C/C++程序员在阅读时会占到多一些的便宜。但即便不熟悉 C，也一样不要把自

己排除在外（尽管往后的学习可能要付出更多的努力）。我会首先解释面向对象编程的概念，以及 Java 的基本控制机制，所以不必担心自己会打不好基础。况且，本书第一次练习的便是基本的流程控制语句。

在本书中，尽管经常都会提及 C 和 C++语言的一些特性，但请注意，本书的重点是 Java，而非 C 和 C++，尽管后者是前者的重要基础。

这样做的目的很简单，是想帮助所有程序员都能正确对待那两种语言。毕竟，Java 是从它们那里派生出来的。我将试着尽可能简化这些引述和参考文字，并会在恰当的时候，向那些没有 C/C++背景的程序员解释他们通常不太熟悉的内容。

2. Java 的学习

在我第一本书《Using C++》面市的几乎同一时间（Osborne/McGraw-Hill 于 1989 年出版），我也开始了 C++的授课生涯。程序语言的教授早已是我的职业；不过自 1989 年以来，我在世界各地连续见到了许多昏昏欲睡、满脸茫然或者困惑不解的面容。等和这些学生熟悉之后，便从他们的作业中发现了一些特别重要的问题。最好笑的是，即便那些上课时面带会心微笑或者频频点头的学生，在对许多问题的认识上，也往往存在观念上的混淆。

在过去几年的“软件开发会议”中，都由我主持 C++分组研讨会（现在变成了 Java 研讨会）。通过这些研讨会，我终于发现了症结所在：有的演讲人试图在很短的时间内向听众灌输过多的内容。所以到最后，尽管听众的水平都还可以，而且提供的材料也很充足，但仍然损失了一部分听众。

不过，我是那些采取传统授课方式的人之一，想使每个学生都跟得上课程进度！

为此，有段时间，我编制了大量教学简报。经过不断试验和修订（或称“迭代”-Iteration，这是 Java 程序设计非常有用的一项技术），最后成功摸索出了一套独特的授课方法，在其中集成了我根据教学经验总结出来的全部东西——我在很长一段时间里都在使用和完善这套方法。

为了让学生成功掌握一门语言，需要经历一系列分散的、易于消化的小步骤、小阶段。而且在每个阶段的学习完成后，都有一些适当的练习。我目前已在 Java 公开研讨会上公布了这一课程，大家可到 http://www.BruceEckel.com 了解详情（对研讨会的介绍也以 CD-ROM 的形式提供，具体信息可在同样的 Web 站点找到）。

从每次研讨会收到的反馈都帮助我修改及重新制订学习材料的重点，直到我最后认定它已成为一个完善的教学载体为止。但本书并非仅仅是一本教科书——我尝试在其中装入尽可能多的信息，并按主题进行了有序的分类。无论如何，这本书的主要宗旨是为那些独立学习的人士服务，他们正准备深入一门新的程序设计语言，并没有太大的可能参加此类专业研讨会或学习班。

3. 目标

象我的前一本书《Thinking in C++》一样，这本书也针对一种“语言”的学习，进行了良好的结构与组织。特别要说明的是，我的目的是建立一套有序的机制，以帮助我在自己的研讨会上更好地进行语言教学。在我思考书中的某一章时，实际上是在想如何教好一堂课。我的目标是得到一系列规模适中的教学模块，可在合理的时间内教完。随后是一些精心挑选的练习，也能当即在课堂上完成。

在这本书中，我想达成的目标如下：

(1) 每一次都将教学内容向前推进一小步，便于读者在继续后面的学习前消化以前的内容。

(2) 采用的例子尽可能简短。当然，这样做有时会妨碍我解决“现实世界”的问题。但

我同时也发现对那些新手来说，如果他们能理解每一个细节，那么一般会产生更大的学习兴趣。而假如他们一开始就被要解决的问题的深度和广度所震惊，那么一般都不会收到很好的学习效果。另外在实际教学过程中，对能够摘录的代码数量是有严重限制的。当然，这样做的一个后果是，无疑会有些人批评我采用了“不真实的例子”。不过，只要能起到良好的效果，我宁愿接受这一指责。

(3) 要揭示的特性按照我精心挑选的顺序依次出场，而且尽可能符合读者的思想历程。当然，我不可能永远都做到这一点；在那些情况下，会给出一段简要的声明，指出这个问题。

(4) 只把我认为有助于理解语言的东西介绍给读者，而不是把我知道的一切东西都抖出来，这并非藏私。我认为信息的重要程度是存在一个合理的层次的。有些情况是 95％的程序员都永远不必了解的。如强行学习，只会干扰他们的正常思维，从而加深语言在他们面前表现出来的难度。以 C 语言为例，假如你能记住运算符优先次序表（我从来记不住），那么就可以写出更“聪明”的代码。但再深入想一层，那也会使代码的读者／维护者感到困扰。所以忘了那些次序吧，在拿不准的时候加上括号即可。

(5) 每一节都有明确的学习重点，所以教学时间（以及练习的间隔时间）非常短。这样做不仅能保持读者思想的活跃，也能使问题更容易理解，对自己的学习产生更大的信心。

(6) 提供一个坚实的基础，使读者能充分理解问题，以便更容易转向一些更加困难的课程和书籍。

4. 联机文档

由 Sun 微系统公司提供的 Java 语言和库（可免费下载）配套提供了电子版的用户帮助手册，可用任何一种 Web 浏览器阅读。此外，由其他厂商开发的几乎所有类似产品都有一套类似的辅助文档系统。而目前出版的与 Java 有关的几乎所有书籍似乎都在重复这份文档。所以你要么已经拥有了它，要么也能立即下载。因此，若非特别必要，否则本书不会重复那份文档的内容。因为一般地说，用 Web 浏览器查找与类有关的资料比在书中查找方便得多（通过网络发行的东西更新也快）。只有在需要对文档进行补充，以便你能理解一个特定的例子时，本书才会提供有关类的一些附加说明。

5. 章节

本书在设计时认真考虑了人们学习 Java 语言的方式。在我授课时，学生们的反映有效地帮助了我认识哪些部分是比较困难的，需特别加以留意。我也曾经一次讲述了太多的问题，但得到的教训是：假如包括了大量新特性，就需要对它们全部作出解释，而这特别容易加深学生们的混淆。因此，我进行了大量努力，使这本书一次尽可能地少涉及一些问题。

所以，我在书中的目标是让每一章都讲述一种语言特性，或者只讲述少数几个相互关联的特性。这样一来，读者在转向下一主题时，就能更容易地消化前面学到的知识。

下面列出对本书各章的一个简要说明，它们与我实际进行的课堂教学是对应的。

(1) 第 1 章：对象入门

这一章是对面向对象的程序设计（OOP）的一个综述，其中包括对“什么是对象”之类的基本问题的回答，并讲述了接口与实现、抽象与封装、消息与函数、继承与合成以及非常重要的多态的概念。这一章会向大家提出一些对象创建的基本问题，比如构造函数、对象存在于何处、创建好后把它们置于什么地方以及魔术般的垃圾收集器（能够清除不再需要的对象）。要介绍的另一些问题还包括通过违例实现的错误控制机制、反应灵敏的用户界面的多线程处理以及连网和因特网等等。大家也会从中了解到是什么使得 Java 如此特别，它为什么取得了这么大的成功，以及与面向对象的分析与设计有关的问题。

(2) 第 2 章：一切都是对象

从本章开始，大家可以着手写出自己的第一个 Java 程序。为此，必须对一些基本概念作出解释，其中包括对象“引用”的概念；怎样创建一个对象；对原始数据类型和数组的一个介绍；作用域以及垃圾收集器清除对象的方式；如何将 Java 中的所有东西都归为一种新数据类型（类），以及如何创建自己的类；函数、参数以及返回值；名字的可见度以及使用来自其他库的组件；static 关键字；注释和嵌入文档等等。

(3) 第 3 章：控制程序流程

本章开始介绍起源于 C 和 C++，由 Java 继承的所有运算符。除此以外，还要学习运算符一些不易使人注意的问题，以及涉及强制转型、升迁以及优先次序的问题。随后要讲述的是基本的流程控制以及选择运算，这些是几乎所有程序设计语言都具有的特性：用 if-else 实现选择；用 for 和 while 实现循环；用 break 和 continue 以及 Java 的标签式 break 和 contiune（它们被认为是 Java 中“不见的 gogo”）退出循环；以及用 switch 实现另一种形式的选择。尽管这些与 C 和 C++中见到的有一定的共通性，但多少存在一些区别。除此以外，所有示例都是完整的 Java 示例，能使大家很快地熟悉 Java 的外观。

(4) 第 4 章：初始化和清除

本章开始介绍构造函数，它的作用是担保初始化的正确实现。对构造函数的定义要涉及函数重载的概念（因为可能同时有几个构造函数）。随后要讨论的是清除过程，它并非肯定如想象的那么简单。用完一个对象后，通常可以不必管它，垃圾收集器会自动介入，释放由它占据的内存。这里详细探讨了垃圾收集器以及它的一些特点。在这一章的最后，我们将更贴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。

(5) 第 5 章：隐藏实现

本章要探讨将代码封装到一起的方式，以及在库的其他部分隐藏时，为什么仍有一部分处于暴露状态。首先要讨论的是 package 和 import 关键字，它们的作用是进行文件级的封装（打包）操作，并允许我们构建由类构成的库（类库）。此时也会谈到目录路径和文件名的问题。本章剩下的部分将讨论 public，private 以及 protected 三个关键字、“友好”访问的概念以及各种场合下不同访问控制级的意义。

(6) 第 6 章：复用类

继承的概念是几乎所有 OOP 语言中都占有重要的地位。它是对现有类加以利用，并为其添加新功能的一种有效途径（同时可以修改它，这是第 7 章的主题）。通过继承来复用原有的代码时，一般需要保持“基类”不变，只是将这儿或那儿的东西串联起来，以达到预期的效果。然而，继承并不是在现有类基础上制造新类的唯一手段。通过“合成”，亦可将一个对象嵌入新类。在这一章中，大家将学习在 Java 中重复使用代码的这两种方法，以及具体如何运用。

(7) 第 7 章：多态

如果没有别人的指点，完全靠自学，可能要花 9 个月的时间，才能真正发现和理解“多态”（也有叫“多形性”的）的问题。这一特性实际是 OOP 一个重要的基础。通过一些小的、简单的例子，读者可知道如何通过继承来创建一系列类型，并通过它们共有的基类，对那个系列中的对象进行操作。通过 Java 的多态概念，同一系列中的所有对象都具有了共通性。这意味着我们编写的代码不需要再依赖特定的类型信息。这使程序更易扩展，包容力也更强。由此，程序的构建和代码的维护可以变得更方便，付出的代价也会更低。

(8) 第 8 章：接口和内部类

Java 提供了第三种方式来建立复用关系，这是通过“接口”来实现的。要注意的是，这里说的“接口”是对象接口一种纯粹的抽象，但它并非仅仅一个抽象类那么简单。这是由于，通过创建一个能向上强制转型为多个基类型的类，C++的“多重继承”可在 Java 里轻松实现！

从表面看，内部类就像一种简单的代码隐藏机制：可把一个类放在另一个类里！但随着学习的深入，大家会慢慢理解到，内部类的功能远不止此。最起码的一点，它能探查自己周围存在的其他类，并可与它们通信。利用内部类，可以写出更为简洁的代码。尽管属于一种全新的概念，但只要熟悉并掌握了它，就会发现用内部类编程，其实是一件相当“爽”的事情！

(9) 第 9 章：对象的容纳

对一个非常简单的程序来说，它可能只拥有一系列数量固定的对象，而且对象的“生存时间”或者“存在时间”是已知的。但通常，我们的程序会在不定的时间创建新对象，只有在程序运行时方可知道到它们的详情。此外，除非进入运行时间，否则无法知道所需对象的准确数量，甚至无法得知它们的确切类型。为解决这个常见的程序设计问题，我们需要拥有这样一种能力：可在任何时间、任何地点创建任意数量的对象。本章将深入探讨由 Java 2 为我们提供的容器库。利用这些“容器”，可在使用对象的时候，有效地包容（封装）它们。这些容器既包括一些简单的数组，也包括一些非常复杂的数据结构等，比如 ArrayList 和 HashMap 等等。

(10) 第 10 章：违例差错控制

Java 最基本的设计宗旨之一便是：假如程序设计有误，便不会真正运行起来！编译器会尽可能提前捕捉到这些错误。但在某些情况下，除非进入运行时间，否则一些问题是发现不了的。这些问题要么属于编程错误，要么则是一些自然的出错状况，它们“出错的前提”是：只有在作为程序正常运行的一部分时，才会成立。Java 为此提供了“违例控制”机制，用于控制程序运行时产生的一切问题。这一章将解释 try、catch、throw、throws 以及 finally 等关键字在 Java 中的工作原理。并讲述什么时候应当果断地“掷”出违例，以及在捕获到违例后该采取什么操作。此外，大家还会学习 Java 的一些标准违例，如何构建自己的违例，违例发生在构造函数中怎么办，以及违例控制器如何定位等等。

(11) 第 11 章：Java I/O 系统

理论上，任何程序都可分割为三大功能块：输入、处理和输出。换句话说，I/O（输入／输出）是所有程序中最关键的部分。在这一章中，大家将学习 Java 为此提供的各种类，如何用它们读写文件、内存块以及控制台等。“老”I/O 和“新”I/O 的区别将在此得到重点强调。此外，本节还要探讨如何获取一个对象、对其进行“流式”加工（使其能保存到磁盘或通过网络传送）以及重新构建它等等。所有这些操作都是通过 Java 的“对象序列化”来完成的。另外，本章还会讲述在 Java ARchive(JAR)文件格式中采用的 Java 压缩库。

(12) 第 12 章：运行时间类型鉴定

若只有指向基类的一个引用，Java 的“运行时间类型鉴定”（RTTI）使我们能获知一个对象的准确类型是什么。一般情况下，我们需要有意忽略一个对象的准确类型，通过 Java 的动态绑定机制（多态），为那一类型设置正确的行为。但在少数情况下，对于只有一个基础引用的对象，我们仍有必要提前知道它的准确类型。获知准确类型后，便可更有效地执行一些特殊操作。本章解释了 RTTI 的用途、如何使用，甚至解释了在适当的时候，如何正确地放弃它。另外，Java 的“反射”特性也会在这里得到介绍。

(13) 第 13 章：创建窗口和小程序

Java 配套提供了名为“Swing”的一套 GUI（图形用户界面）函数库。它实际包括了一系列特殊的类，除了能为你的 Java 程序赋予一个漂亮的图形界面之外，还有效解决了不同平台间的移植问题。用它做出来的视窗程序既可以是一个需要依赖浏览器才能运行的“小程序”（Applet），也可以是一个能独立运行的、完整的应用程序。

本章将引导大家进入 Swing 的世界，同时简介如何创制简单的 World Wide Web 小程序（依付 Web 浏览器运行）。另外，本章将首次向大家介绍一个重要概念：JavaBeans（Java

豆）。大家切不可小看了这种“Java 豆”（也有干脆叫“咖啡豆”的）。创建 RAID 程序构建工具的时候，它是一个不可或缺的基本工具（RAID 是“快速应用程序开发”的意思）。

(14) 第 14 章：多线程

Java 有一套内建的机制，提供了对多个并发子任务的支持，我们把这些子任务称作“线程”，它们可在同一个程序内同时运行。除非你的机器安装了多个处理器，否则多个子任务想同时运行，这就是唯一能够采用的形式。尽管还有别的许多重要用途，但在打算创建一个“反应灵敏”的用户界面时，多线程的运用显得尤为重要。举个例子来说，在采用了多线程技术后，尽管当时还有别的任务在执行，但用户仍可毫无阻碍地按下一个按钮，或键入一些文字。本章将对 Java 的多线程处理机制进行探讨，并介绍相关语法。

(15) 第 15 章　分布式计算

编写一个要在网络上使用的程序时，你会发现有一大堆 Java 库摆在面前。没有恰当的指引，单凭自己的力量，很难从中摆弄出一个正确的头绪。本章将为大家奠定局域网和 Internet 通信基础，同时介绍一些相关的 Java 类，用它们可简化此类功能的实现。同时，这里还出现了两个重要的概念：Servlets 和 JSP，两者均用于服务器端的程序支持；另外，大家还得注意 JDBC 和 RMI。其中，JDBC 是“Java 数据库连接”的简称，而 RMI 是“远程方法调用”的简称。最后，本章还会指导大家接触 JNI、JavaSpaces 和 Enterprise JavaBeans（EJB，即“企业咖啡豆”）这几项新技术。

(16) 附录 A：传递和返回对象

在 Java 中，若想同一个对象沟通，唯一的办法便是通过“引用”。因此，如何将一个对象传递到函数里面，以及如何从函数里返回一个对象，便成为我们必须掌握的知识。在这个附录中，我们解释了在函数中进出时，管理对象所需的一些知识。另外，也侧重讲述了 String 类，它通过一种不同的方法来解决问题。

(17) 附录 B：Java 固有接口（JNI）

许多人爱用 Java 的一个理由便是它“便于移植”。但是，假如一个 Java 程序“不幸”地真的具有了“完全”的移植能力，那么它必然也会存在一些重大缺陷：速度慢，而且不能访问一些只由某种平台提供的特殊服务。

因此，假如事先知道自己的程序会在哪种平台上运行，就应尽量避免这些缺陷的产生，方法便是找出一些特定的操作，把它们变成“固有方法”（Nativev Mehtods），从而加快操作速度。固有方法本质上是用另一种程序语言写的函数（目前只支持用 C/C++写的）。本附录将为大家奠定这方面的基础，让你能写出简单的程序，和非 Java 代码“打交道”。

(18) 附录 C：Java 编程指南

本附录提供了大量建议，帮助大家进行低级程序设计和代码编写。

(19) 附录 D：推荐读物

列出我觉得特别有用的一系列 Java 参考书。

6. 练习

要想巩固对新知识的掌握，我发现“趁热打铁”的简单练习非常管用。所以读者在每一章结束时都能找到一系列练习题。

大多数练习都很简单，在合理的时间内即可完成。如将本书作为教材，可考虑在课堂内完成。老师要注意观察，确定所有学生都已消化了讲授的内容。有些练习要难一些，是为那些有兴趣继续深入的读者准备的。大多数练习都可在较短时间内做完，有效地检测和加深刚才学到的知识。有些习题较具挑战性，但都不会太麻烦。事实上，练习中碰到的问题在实际应用中也会经常碰到（或者说，这些问题会主动找上门来）。

答案在一份电子文档里，名为《The Thinking in Java Annotated Solution Guide》，可在

www.BruceEckel.com 上找到。

7. 多媒体光盘

本书有两张配套光盘。第一张随书免费赠送，名为“Thinking in C”，帮你准备一些必要的 C 基础知识，讲述一些基本的 C 语法，以便更顺利地理解和学习 Java。

第二张光盘则以本书内容为基础，作为一个单独的产品提供，其中包含完整的“Hands-On Java”培训课程。这个课程历时五天，共计 15 个小时，与本书数百个知识点同步。事实上，我在讲课时，用的教材便是这本书，所以两者是绝佳的搭档。

这张光盘载有五天的全部授课实录，如果你没有条件亲自来上课，便何考虑购买此光盘，再利用手中的书，完成一整套对你大有裨益的 Java 学习过程。

第二张光盘只能通过 www.BruceEckel.com，直接在线订购。

8. 源码

本书所有源码都作为保留版权的“免费软件”提供，可以独立软件包的形式获得，亦可从 http://www.BruceEckel.com 处下载。为保证大家获得的都是最新版本，我将用这个正式站点发行代码以及本书电子版。亦可在其他站点找到电子书和源码的镜像版（有些站点已在 http://www.BruceEckel.com 处列出）。但无论如何，都应首先检查正式站点，确定镜像站点提供的确是最新版本。您可在课堂和其他教育场所免费发布这些代码。

之所以仍然“保留版权”，主要目的是保证源码得到正确引用，并防止在未经许可的情况下，在商业用途的印刷材料中发布我的代码。通常，只要源码获得了正确引用，在大多数媒体中使用本书的示例都没什么问题。

在每个源码文件中，都能发现下述版权声明文字：

```
//:! :CopyRight.txt
Copyright ©2000 Bruce Eckel
Source code file from the 2nd edition of the book
"Thinking in Java." All rights reserved EXCEPT as
allowed by the following statements:
You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in Java" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
```

```
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
Bruce Eckel or the publisher be liable for any
lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///:~
```

可在自己的开发项目中使用这些代码，也可在课堂上引用（包括你的展示材料）。但要保证在每个源码文件中，上述版权声明文字都得到了全文保留。

a. 编码样式

在本书正文中(指印刷版——译注)，标识符（函数、变量和类名）以粗体印刷。大多数关键字也采用粗体——除了一些频繁用到的关键字（若全部采用粗体，会使页面拥挤难看，比如那些“类”的名字）。

对于本书的示范程序，我采用了一种特定的编码样式。该样式和 Sun 公司在其网站上采用的完全一致（参见 java.sun.com/docs/codeconv/index.html），而且似已获得大多数 Java 开发环境的支持。如果你还看过我早期出版的其他作品，便会注意到 Sun 的编码样式实际上有点儿“追随”我的意思——这当然令我高兴，尽管我自己并未在这方面向 Sun 主动施加过任何“影响”。由于编码样式目前仍然是个敏感问题，极易招来数小时的辩论不休，所以我在这儿想要说明的是，我并不打算通过这些示例来建立一种样式“标准”。之所以采用这些样式，完全出于我自己的考虑。由于 Java 是一种形式非常自由的编程语言，所以读者完全可根据自己的感觉，选用自己认为合适的编码样式。

本书所有程序代码都是由字处理软件自动包括在正文中的，它们直接取自编译好的文件。因此，书内印刷的代码都应能正常工作，其中不存在任何输入或打字错误，不会造成编译错误。会造成编译错误的代码已用注释//!标出。所以很容易发现，也很容易用自动方式进

行测试。由读者发现并向作者报告的错误首先会在发行的源码中改正，然后在本书的更新版中校订（所有更新都会在 Web 站点 http://www.BruceEckel.com 处公布）。

9. Java 版本

判断一个程序的行为是否正确时，我通常以 Sun 公司的 Java 开发平台为准。

迄今为止，Sun 已发布了 Java 的三个重要版本：1.0，1.1 及 2（尽管 Sun 发布的 JDK 仍然采用 1.2、1.3、1.4 等版本号，但从 1.2 开始，都通称为 Java 2）。Java 2 的发布，使 Java 进入了一个全盛期。特别要指出的是，它的用户界面工具终于得到了极大的加强。本书讨论的重点和采用的测试平台都是 Java 2。不过，我有时还是用到了 Java 2 的一些早期特性，以便它能在 Linux 下编译执行（通过目前我能拿到的 Linux JDK）。

如果你想学习一些本书没有提及的早期语言特性，建议你从 www.BruceEckel.com 处免费下载本书第一版。另外，在本书的配套光盘上，也能找到它的电子版。

在阅读本书的过程中，你会发现当我需要提及一个早期版本时，不会再使用它的子版本号。在这本书中，我提到的全部 Java 版本仅有三种：Java 1.0、Java 1.1 和 Java 2，不会出现象 Java 1.1.2 或者 Java 1.2.1 这样的提法！

10. 课程和培训

我的公司提供了一个五日制的公共培训课程，以本书的内容为基础。每章的内容都代表着一堂课，并附有相应的课后练习，以便巩固学到的知识。一些现场授课语音和辅助用的幻灯片可在本书的配套光盘上找到，最大限度地方便各位读者。欲了解更多的情况，请访问：

http://www.BruceEckel.com

公司也提供了咨询服务，指导客户完成整个开发过程——特别是在您的单位首次接触 Java 开发的时候。

11. 错误

无论作者花多大精力来避免，错误总是从意想不到的地方冒出来。如果您认为自己发现了一个错误，请在源文件（可在 http://www.BruceEckel.com 和配套光盘中找到）里指出有可能是错误的地方，填好我在网上提供的表格，提交给我。如有必要，附上源文件，并指出你推荐的纠正方法。在此一并感谢所有热心读者的帮助！

12. 封面设计

《Thinking in Java》一书的封面创作灵感来源于 American Arts & CraftsMovement（美洲艺术&手工艺品运动）。这一运动起始于世纪之交，1900 到 1920 年达到了顶峰。它起源于英格兰，具有一定的历史背景。当时正是机器革命的风暴席卷整个大陆的时候，而且受到维多利亚地区强烈装饰风格的巨大影响。Arts&Crafts 强调的是原始风格，回归自然则是整个运动的核心。那时对手工制作推崇备至，手工艺人特别得到尊重。正因为如此，人们远远避开现代工具的使用。这场运动对整个艺术界造成了深远的影响，直至今天仍受到人们的怀念。特别是我们面临又一次世纪之交，强烈的怀旧情绪难免涌上心来。计算机发展至今，已走过了很长的一段路。我们更迫切地感到：软件设计中最重要的是设计者本身，而不是流水化的代码编制。如设计者本身的素质和修养不高，那么最多只是“生产”代码的工具而已。

我以同样的眼光来看待 Java：作为一种将程序员从操作系统繁琐机制中解放出来的尝试，它的目的是使人们成为真正的“软件艺术家”。

无论作者还是本书的封面设计者（自孩提时代就是我的朋友）都从这一场运动中获得了灵感。所以接下来的事情就非常简单了，要么自己设计，要么直接采用来自那个时期的作品。

此外，封面向大家展示的是一个标本箱，生物学家可能会用它展示自己的昆虫标本。我们认为这些昆虫都是“对象”，全部置于更大的“收集箱”对象里，再统一置入“封面”这个对象里。它向我们揭示了面向对象编程技术最基本的“集合”概念。当然，作为一名程序员，大家对于“昆虫”或“虫”是非常敏感的（“虫”在英语里是 Bug，后指程序错误——译注）。这里的“虫”、已被抓获，在一只广口瓶中杀死，最后禁闭于一个小小的展览盒里——暗示 Java 有能力寻找、显示和消除程序里的“虫”（这是 Java 最具特色的特性之一）。

13. 致谢

首先，感谢那些和我一起制定培训班学习计划、提供参考和开展教学活动的所有朋友，他们是：Andrea Probaglio、Dave Bartlett（同时也是本书第 15 章的主要负责人）、Bill Benners 和 Larry O'Brien。在我执着于研究 JAVA 最佳教学模式的过程中，他们给予我极大的支持和帮助。还有瑞士的ROlf Andre Klaedtke；在我首次在欧洲开设培训班时热情接待过我的Martin Vlcek、Martin Byer、vlada & Pavel Lahoda、Martin the Bear、布拉格的 Hanka 和意大利的 Marco Cantu。

感谢 Doyle Street Cohousing Community（道尔街社区）对我两年以来的宽容，本书第一版的编写就是在此完成的。其次，感谢 Kevin 和 Sonda Donovan 把自己在科罗拉多州 Crested Butte 的房子租给我，使我整个夏天能安心写作。同时，还要感谢 Crested Butte 市的友邻和 Rocky Mountain Biological Laboratory（落基山生物实验室）的工作人员，他们总是面带微笑，宽容我的“胡作非为”。

接下来，该对“摩尔文艺作品代理公司”的 Claudette Moore 致以诚挚的谢意，如果没有她的信心与毅力，恐怕我至今无法梦想成真。

Jeff Pepper 同样功不可抹，我的头两部书就是他的帮助下出版的，当时他就职于 Osborne/McGraw-Hill 出版社。真是无巧不成书，本书出版时，Jeff 恰好在 Prentice-Hall 出版社工作，理所当然，他再次为本书的出版扫清了所有可能的障碍，使我的出书经历非常之愉快。在此深深地对 Jeff 说一声——你对我真的很重要。

除此以外，还要特别感谢 Gen Kiyooka：在我接触网络的若干年来，他一直很慷慨，让我使用他公司的 WEB 服务器。这对我的帮助简直无法用金钱来衡量。

其他的还有：Cay Horstmann（《Core Java》作者之一）、D'Arcy Smith（Symantec 公司）和 Paul Tyma（《Java Primer Plus》作者之一），是他们帮助我进一步理清了 JAVA 的概念和精髓。

另外，那些在“Java 软件开发研讨会”上热情洋溢的同志们，培训班里认真的学生们，他们提出的见解使本书的素材愈发成熟。

至于 JAVA 素材被制作为教学光盘一事，要特别感谢 Larry 和 Tina O'Brien，（有关详情，请访问 http://www.BruceEckel.com）。

此时，还忘不了那些针对本书第一版，提出不同见解的热情读者，他们是：Kevin Raulerson（发现了多处重大错误）、Bob Resendes（提出许多疏漏之处）、John Pinto、Joe Dante、Joe Sharp（提出疑问）、David Combs（就书中的语法和表达提出正确意见）、Robert Stephenson 博士、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，还有许多热心读者。

本书第一版的宣传和电子版在欧洲的发行，多亏了 Prof. Ir. Marc Meurrens，没有他的参与，本书当时无法取得成功。

在编写本书的过程中，一些机智灵巧的技术人员曾走进我的生活，后来还和我成了无话不谈的好朋友，他们既平凡又与众不同，因为他们喜欢练瑜珈和其他类似的精神锻炼，和他

们接近后，我发觉自己经常思如泉涌。这些曾帮助我获得精神净化和技术帮助的技术人员是：Kraig Brockschmidt、Gen Kiyooka 和 Andrea provaglio，感谢他们让我了解了意大利的 Java 和程序设计状况，如今，他们已作为 MindView 小组的合伙人，在美国工作。

对我而言，在充分了解 Delphi 的基础上再接触 Java，当然容易得多，因为在概念和语言设计原理上，两者几乎是相通的。在我编写本书过程中，一些研究 Delphi 的朋友们为我提供了许多帮助，使我能洞察一些不易为人注意的编程环境。他们是 Marco Cantu（又是意大利人——会说拉丁语的人在学习 Java 方面难道有天赋吗？）、Neil Rubenking（一个喜欢瑜珈／素食／打坐的人，同时还酷爱计算机）和 Zack Urlocker（伴我环游世界的“死党”）。

本书的面世，与 Richard Hale Shaw 的远见和支持（当然还有 Kim）是分不开的。Richard 和我花数月的时间将素材揉合在一起，共同探讨促使 JAVA 爱好者尽快入门、精通的最佳方案。另外，还要感谢 KoAnn Vikoren、Eric Eaurot、DeborahSommers、Julie Shaw、Nicole Freeman、Cindy Blair、Barbara Hanscome、Regina Ridley、Alex Dunne 和 MFI 的其他成员。特别感谢 Tara Arrowood，不停地给予我丰富的灵感。

本书的设计、封面设计以及封面照片是我的朋友 Daniel Will-Harris 的佳作。他是一名杰出的作家兼设计家（http://www.WillHarris.com）。早在初中时就已显露出其过人的数学天赋，当时还没有计算机和桌面排版技术，他就敢于挑战高年级学生。虽然如此，本书的初稿仍然由我制作，所以如果有录入错误，那就别冤枉好人，应该算在我头上。本书的写作和初稿生成均采用 Microsoft Word 97 for Windows。正文字体采用的是 Georgia，标题采用的是 Verdana，封面字体是 ITC Rennie Marckintosh。

感谢为我提供编译程序的一些著名公司：Borland、Blackdown group (for Linux)，当然还有 Sun。

特别感谢我的老师和所有的学生（同时也是我的老师）。其中最有趣的一位写作老师是 Gabrielle Rico（《Writing the Natural Way》一书的作者，该书是 Putnam 在 1983 年出版的）。我将永远珍惜自己在爱萨伦度过的日子。

感谢曾支持过我的朋友们：Andrew Binstock、Steve Sinofsky、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、其他的意大利朋友（Laura Fallai、Corrado、Ilsa 和 Cristina Giustozzi）、Chris 和 Laura Strand、Almquists、Brad Jerbic、Marilyng 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、Kevin Donovan 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 以及他们全家。最后，当然还要感谢我的父母亲。

a. Internet 赞助人

感谢那些帮我重写程序示例，使之能采用 Swing 库的朋友们，他们是：Jon Shvarts、Thomas Kirsch、Rahim Adatia、Rajesh Jain、Ravi Manthena、Banu Rajamani、Jens Brandt、Nitin Shivaram、Malcolm Davis 和其他帮助过我的朋友们。他们的帮助使本书增色不少。

第 1 章　对象入门

计算机革命起源于机器。因此，程序语言的起源似乎就是为了看起来要“像”那种机器。

但是，计算机同机器相比，毕竟还是存在很大区别的，因为它们属于“思想放大工具”（正如斯蒂芬 • 乔布斯常说的那样：它是“思想的坐骑”），是一种不同种类的表达媒体。因此，这种工具越来越不象一种机器，而是逐渐倾向于成为我们思想的一部分。同时，它也越来越象其他传统的表达方式，比如书写、绘画、雕刻、动画和电影等等。在计算机向一种全方位的思想表达载体的转变过程中，“面向对象编程”（OOP）正是一股重要的促进力量。

本章将向大家介绍 OOP 的基本概念，包括对各种开发方法的一个综述。这一章乃至整本书，都假定你有使用一种过程化编程语言的经验（尽管并不一定要是 C）。如果你认为自己还要作更多的准备，特别是先掌握好 C 的基本语法，再来学习这本书，那么建议你先看看随书提供的《Thinking in C:Foundations for C++ and Java》培训光盘；亦可从 www.BruceEckel.com 处观看。

本章提供的是保证您往后顺利学习的背景知识及补充材料。如果心头没有底，许多人在进入面向对象的编程世界时，往往会产生心有力而力不足的感觉。为此，本章会介绍许多基本的概念，便于大家对 OOP 有一个大概的理解。不过，也有些人并不喜欢先掌握这些基本的东西，而是迫不及待地想快速学习一些“技巧”。这里要建议大家的是：假如基本的东西没掌握好，往后的学习便会变得非常困难。等遇到问题时再来“补课”，你会产生很大的“挫折感”，这对保持学习的兴趣极为不利。因此，如果你是 OOP 的新手，最好先耐心完成本章的学习，再跳至其他章节。当然，跳过它直接扑向 Java 也未尝不可，你仍可用 Java 写出一些初级程序。然而，这一堂课终究是要“补”的，否则无法真正理解对象的重要性，以及如何利用对象完成程序设计！

1.1　抽象的进步

所有程序语言的最终目的都是提供一种“抽象”方法。一种较有争议的说法是：解决问题的复杂程度直接取决于抽象的种类及质量。这儿的“种类”是指准备对什么进行“抽象”？汇编语言是对基础机器的少量抽象。后来的许多“命令式”语言（如 FORTRAN、BASIC 和 C）是对汇编语言的一种抽象。与汇编语言相比，这些语言已有了长足的进步，但它们的抽象原理依然要求我们着重考虑计算机的结构，而非考虑问题本身的结构。在机器模型（位于“方案空间”）与实际解决的问题模型（位于“问题空间”）之间，程序员必须建立起一种联系。这个过程要求人们付出较大的精力，而且由于它脱离了编程语言本身的范围，造成程序代码很难编写，而且要花较大的代价进行维护。由此造成的副作用便是一门完善的“编程方法”学科。

为机器建模的另一个方法是为要解决的问题制作模型。对一些早期语言来说，如 LISP 和 APL，它们的做法是“从不同的角度观察世界”——“所有问题都归纳为列表”或“所有问题都归纳为算法”。PROLOG 则将所有问题都归纳为决策链。对于这些语言，我们认为它们一部分是面向基于“强制”的编程，另一部分则是专为处理图形符号设计的。每种方法都有自己特殊的用途，适合解决某一类的问题。但只要超出了它们力所能及的范围，就会显

得非常笨拙。

面向对象的程序设计则在此基础上跨出了一大步，程序员可利用一些工具表达问题空间内的各种元素。由于这种表达非常普遍，所以[①]不必受限于特定类型的问题。我们将问题空间中的元素以及它们在方案空间的表示物称作“对象”（Object）。当然，还有一些在问题空间没有对应体的其他对象。通过添加新的对象类型，程序可进行灵活的调整，以便与特定的问题配合。所以在阅读方案的描述代码时，会读到对问题进行表达的话语。与我们以前见过的相比，这无疑是一种更加灵活、更加强大的语言抽象方法。总之，OOP 允许我们根据问题来描述问题，而不是根据方案。然而，仍有一个联系途径回到计算机。每个对象都类似一台小计算机；它们有自己的状态，而且可要求它们进行特定的操作。与现实世界的“对象”或者“物体”相比，编程“对象”与它们也存在共通的地方：它们都有自己的特征和行为。

有些语言设计者指出，面向对象的程序设计本身并不足以方便地解决全部编程问题，并提倡将各种不同的方法集中到一块儿，最终研究出一种“多态程序设计语言”[①]。

Alan Kay 为我们总结出了 Smalltalk 的五大基本特征，这是第一例取得成功的面向对象程序设计语言，也是 Java 的基础语言。通过这些特征，我们可理解“纯粹”的面向对象程序设计方法是什么样的：

(1) 所有东西都是对象。可将“对象”想象为一种新型变量；它保存着数据，但可对那个对象“发出请求”，要求它对自己执行一些操作。理论上讲，可从要解决的问题身上提取出所有概念性组件，再在程序中把它表示成一个对象。

(2) 每个程序都是一大堆对象的组合；通过消息的传递，一个对象可告诉另一个对象该做什么。为了向对象发出请求，需向那个对象“发送一条消息”。更具体地讲，可将消息想象为一个调用请求，它调用的是从属于目标对象的一个子例程或函数。

(3) 每个对象都有自己的存储空间，可容纳其他对象。换句话说，通过封装一个现有的对象，还可生成一个新对象。因此，即使一个非常复杂的程序，在我们面前也显得异常简单：一切都是对象！

(4) 每个对象都有一种类型。根据语法，每个对象都是某个“类”的一个“实例”。其中，“类”（Class）是“类型”（Type）的同义词。一个类最重要的特征就是“能将什么消息发给它？”。

(5) 同一类所有对象都能接收相同的消息。这实际是别有含义的一种说法，大家不久便能理解。由于类型为“圆”（Circle）的一个对象也属于类型为“形状”（Shape）的一个对象，所以一个圆完全能接收形状消息。这意味着可让程序代码统一指挥“形状”，令其自动控制所有符合“形状”描述的对象，其中自然包括“圆”。这一特性称为对象的“可替换性”，是 OOP 最重要的概念之一。

1.2 每个对象都有一个接口

亚里士多德或许是认真研究“类型”概念的第一人，他曾谈及“鱼类和鸟类”的问题。在世界首例面向对象语言 Simula-67 中，第一次用到了这样的一个概念：

“所有对象——尽管各有特色——都属于某一系列对象的一部分，这些对象具有通用的特征及行为”。

在 Simula-67 这种初级 OOP 语言中，首次用到了 class 关键字，它为程序引入了一个全新的类型。

[①] 参见由 Timothy Budd 编著的《Multiparadigm Programming in Leda》一书，Addison-Wesley 出版社 1995 年出版。

Simula 是一个很好的例子。正如这个名字所暗示的，它的作用是“模拟”（Simulate）象“银行出纳员”这样的经典问题。在这个例子里，我们有一系列出纳员、客户、帐号以及交易等。每类成员（元素）都具有一些通用的特征：每个帐号都有一定的余额；每名出纳都能接收客户的存款；等等。与此同时，每个成员都有自己的状态；每个帐号都有不同的余额；每名出纳都有一个名字。所以在计算机程序中，能用独一无二的实体分别表示出纳员、客户、帐号以及交易。这个实体便是“对象”，而且每个对象都隶属一个特定的“类”，那个类具有自己的通用特征与行为。

因此，在面向对象的程序设计中，尽管我们真正要做的是新建各种各样的数据“类型”（Type），但几乎所有面向对象的程序设计语言都采用了“class”关键字。当您看到“type”这个字的时候，请同时想到“class”；反之亦然。[②]。

既然一个“类”描述的是一系列对象，所有对象均具有相同的特征（数据元素）和行为（功能），那么这个“类”实际就是一个“数据类型”。这是由于，举个例子来说，一个浮点数不也有一系列特征和行为吗？但两者的差别在于，程序员定义一个“类”是为了描述某个具体的问题，而不是被强制使用一个现有的数据类型，在机器中表示出一个存储单元。

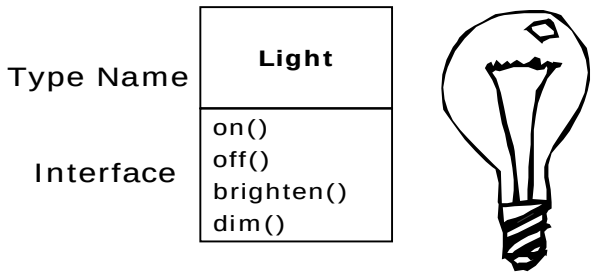
根据自己的需要，我们可加入新的数据类型，从而对程序语言进行扩展。程序设计系统本身欢迎任何新类的加入，并会象对待其他固有类型一样，对所有新类都“一视同仁”。

要注意的是，面向对象的程序设计方法并非仅仅具有“模拟”或“仿真”功能。无论你是否同意，事实上任何程序都是对我们打算设计的那个系统的一个“模拟”。不过，只有 OOP 技术，才能将一系列复杂的问题轻易归纳成一个简单的方案。

建好一个类后，可根据情况生成从属于那一“类”的许多对象。随后，可将那些对象作为要解决问题中存在的元素进行处理。事实上，当我们进行面向对象的程序设计时，面临的最大一项挑战性就是：如何在“问题空间”（问题实际存在的地方）的元素与“方案空间”（对实际问题进行建模的地方，如计算机）的元素之间建立理想的“一对一”对应或映射关系。

如何利用对象完成真正有用的工作呢？必须有一种办法能向对象发出请求，令它做出一些有用的事情，比如完成一次交易、在屏幕上画一些东西或者打开一个开关等等。每个对象仅能接受特定的请求。我们向对象发出的请求是通过它的“接口”（Interface）定义的，对象的“类型”或“类”则规定了它的接口形式。“类型”与“接口”的等价或对应关系是面向对象程序设计的基础。

下面让我们以电灯泡为例：




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

[②] 有的人作了进一步区分，他们强调“类型”决定了接口，而“类”是那个接口的一种特殊实现方式。

接口规定了对一个特定对象来说，我们可发出“什么”请求。但是，其他某个地方还必须有恰当的代码，来满足我们发出的请求。这些代码，再加上隐藏起来的数据，便构成了我们对一个问题的“实现”（Implementation）。从程序化编程的角度出发，这一点似乎并不难理解。每个“类型”都必须有一个对应的“函数”，用它满足每一个可能的“请求”。而当我们对一个对象发出了特定的请求之后，相应的函数便会被调用。我们通常将这一过程概括为“向对象发了一条消息”（发了一个请求）；同时，对象根据自己收到的消息，知道自己该做什么（执行代码）。

在这个例子中，类型／类的名称是 Light，可向 Light 对象发出的请求包括打开（on）、关闭（off）、变得更亮（brighten）或者变得更暗（dim）。通过简单地声明一个名字（lt），我们为 Light 对象创建了一个“引用”。然后用 new 关键字新建类型为 Light 的一个对象。再用等号将其赋给引用。为了向对象发送一条消息，我们列出引用名（lt），再用一个句点符号（.）把它同消息名称（on）连接起来。从中可以看出，使用一些预先定义好的类时，我们在程序里采用的代码是非常简单和直观的。

上述图例采用的是 UML 格式，即“统一建模语言”（Unified Modeling Language）。具体地讲，就是将每个“类”表示成一个方框，类型名显示在方框顶部，打算描述的所有数据成员都放在方框中部，而“成员函数”（从属于该对象的函数，负责接收我们发给那个对象的消息）都放在方框的底部。通常，只有类名和公共成员函数才用 UML 示意图表示，所以中间部分并不显示出来。假如关心的只是类名，那么底部的那部分也不用显示出来。

1.3 实现的隐藏

为方便后面的讨论，让我们先对这一领域的从业人员作一下分类。从根本上说，大致有两方面的人员涉足面向对象的编程：“类创建者”（创建新数据类型的人）以及“客户程序员”（在自己的应用程序中采用现成数据类型的人[③]。对客户程序员来讲，最主要的目标就是收集一个充斥着各种类的编程“工具箱”，以便快速开发（或者改编）出符合自己要求的应用程序。而对类的创建者来说，他们的目标则是从头构建一个类，只向客户程序员开放有必要开放的东西（接口），其他所有实现细节都隐藏起来。为什么要这样做？因为隐藏起来之后，客户程序员就不能接触和改变那些细节，所以原创者不用担心自己的作品会受到非法修改，也能保证它们不会对其他人造成干扰。隐藏起来的部分通常也是一个对象最“脆弱”的部分，一个粗心大意的客户程序员，可以会在不经意间，轻易破坏掉整个对象。所以，将实现隐藏起来之后，还有助于减少程序错误（Bug）的出现。“实现的隐藏”重要性不言而喻！

我们身处一个充满“关系”的社会，形形色色的人、各种各样的事物，相互间都难免发生这样或那样的“关系”。对参与这种关系的每一方而言，最重要的一点就是明确划分出它们的职责范围，不可越界！在你创建一个库时，便相当于同客户程序员建立了一种关系。对方也是程序员，但他们的目标是组合出一个特定的应用（程序），或者以你的库为基础，构建一个更大的库。

如果任何人都能使用一个类的所有成员，那么客户程序员可对那个类做任何事情，没有办法强制他们遵守任何约束。即便非常不愿客户程序员直接操作类内包含的一些成员，但只要没有作出访问控制，就没办法阻止这一情况的发生——所有东西都会暴露无遗。

有两方面的原因促使我们控制对成员的访问。第一个原因是防止程序员接触他们不该接

[③] 感谢我的朋友 Scott Meyers，是他帮我起了这个名字。

触的东西——通常是内部数据类型的设计思想。若只是为了解决特定的问题，用户只需操作接口即可，毋需明白这些信息。我们向用户提供的实际是一种服务，因为他们很容易就可看出哪些对自己非常重要，以及哪些可忽略不计。

进行访问控制的第二个原因是允许库设计人员修改内部结构，不用担心它会对客户程序员造成什么影响。例如，我们最开始可能设计了一个形式简单的类，以便简化开发。以后又决定进行改写，使其更快地运行。若接口与实现方法早已隔离开，并分别受到保护，就可放心做到这一点，只要求用户重新链接一下即可。

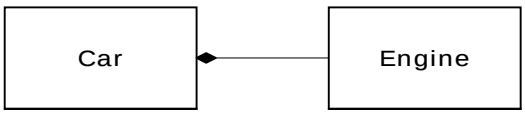
Java 采用三个显式（也就是说，必须明确指定）关键字来设置类边界：public、private 和 protected。这些关键字的使用和含义都是相当直观的，它们决定了谁能使用后续的定义内容。“public”（公共）意味着后续的定义任何人均可使用。而在另一方面，“private”（私有）意味着除您自己、类型的创建者以及那个类型的内部函数成员，其他任何人都不能访问后续的定义信息。private 在您与客户程序员之间竖起了一堵墙。若有人试图访问私有成员，就会得到一个编译期错误。“protected”（受保护的）与“private”相似，只是一个继承的类可访问受保护的成员，但不能访问私有成员。继承的问题不久就要谈到。

Java 还有一种“默认”访问权限。如上述几个关键字都没用，便默认为它。有些时候，我们也把它叫作“friendly”（友好的）访问，因为类可以访问同一封装内其他类的友好成员，但这个封装外部相同的友好成员却表现为“private”（私有）性质。

1.4 方案的重复使用

创建并测试好一个类后，它应当（在理想情况下）马上变成一个有用的代码单元。但并不象许多人猜想的那样，这种重复使用的能力并非那么容易实现；它要求作者有较高的设计水平，以及丰富的经验，否则很难获得一个真正成功的作品。但一旦有了这样的一个设计，你的作品往往就能方便地重复使用。许多人都认为，代码或设计方案的重复使用是面向对象程序设计具有的最重要的优点之一。

要想重复使用一个类，最简单的办法便是直接使用那个类的对象。但同时也能将那个类的一个对象置入一个新类。我们把这叫作“创建一个成员对象”。新类可由任意数量及类型的其他对象构成。无论如何，只要新类达到自己的设计要求即可。这其实就是“创作”（Composition）的概念——在现有类的基础上“创作”出一个新类——另一种更常用的说法是：“集合”出一个新类；此外，我们有时也将“创作”称为“包含”关系，比如：“每辆车都包含一个发动机”。



（在上述 UML 图示中，用一个实心的小菱形表示“创作”或“包含”关系，它指出当前有一辆车，车里包含了某样东西。不过，我以后会采用一种更简单的方式来表达：只用一条直线，没有菱形，指出两者间存在一种“关联”[④]）

对象的创作具有极大的灵活性。新类的“成员对象”通常设为“私有”（Private），使用

[④] 通过示意图，大多数细节都可一目了然。从现在开始，大家不必再纠缠于“集合”或“创作”等等概念中，只要明白存在什么样的关系便成。

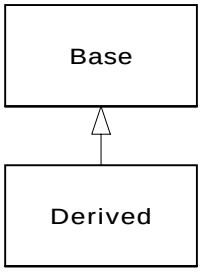
这个类的客户程序员不能访问它们。这样一来，我们可在不干扰客户代码的前提下，从容地修改那些成员。也可在程序的“运行时”对成员作出修改，这进一步增大了灵活性。后面要讲到的“继承”并不具备这种灵活性，因为编译器必须对通过继承创建的类加以限制。

由于继承实在太重要，所以在面向对象的程序设计中，它经常都被重点强调。也正是由于这个原因，作为新加入这一领域的程序员，或许早便先入为主地认为“继承应当随处可见”。不过，沿这一思路产生的设计将是非常笨拙的，会大大增加程序的复杂程度。相反，新建类的时候，首先应考虑“创作”出新对象；这样显得更加简单和灵活。利用对象的创作，我们的设计可保持清爽。以后当你需要用到继承的时候，就会清楚地认识到这一点。

1.5 继承：复用接口

就其本身来说，对象可变成一个对我们来说相当方便的工具。利用它，可根据“概念”，将数据和功能封装到一起。这样便可正确表达出一个“问题空间”，不用刻意遵照机器的表达方式。在程序设计语言中，这些“概念”通过 class 这个关键字，表示成一系列基本单元。

我们费尽心思做出一种数据类型后，假如不得不又新建一种类型，令其实现大致相同的功能，那会是一件非常令人灰心的事情。但若能利用现成的数据类型，对其进行“克隆”，再根据情况进行添加和修改，那就理想多了。“继承”（Inheritance）正是针对这一目标而设计的。但继承并不完全等价于克隆。在继承过程中，若原始类（正式名称叫作基类、超类或父类）发生了变化，修改过的“克隆”类（正式名称叫作继承类或者子类）也会反映出这种变化。



（在上述 UML 图示中，箭头从派生类指向基类。不过正如大家以后会看到的，也完全可能同时存在多个派生类。）

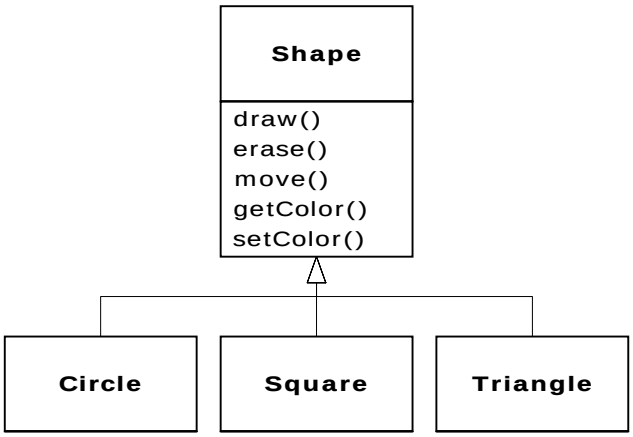
一个“类型”还能做其他许多事情，并非仅仅对一系列对象施加限制那么简单。此外，它还保持着与其他类型的关系。尽管两个类型可能具有通用的一系列特征及行为，但其中一个类型还可能具有比另一个类型多的特征，而且还可能对更多的消息加以处理（或者用不同的方式，来处理相同的消息）。对于类型之间的“共通性”，继承是如何表达的呢？它是用“基类型”和“派生类型”来表达的！

其中，一个“基类型”中包含了所有共通的特征及行为，从它派生出去的其他所有类型都无一例外地具有这些特性及行为。在自己打算构建的程序系统中，一个“基类型”表达出了与其中一系列对象有关的核心思想！从基类型上，我们又能派生出其他类型，以不同的形式，来真正实现这些核心思想。

必须用一个例子来更清楚地阐明上述理论。在此，请大家想象一台垃圾回收机器。其中，我们的基类型是“垃圾”，每块垃圾都有自己的重量、价值等等（特征）；可对其采取的处理方式则包括粉碎、熔化或者分解等等。从这一点出发，我们可派生出更加具体的垃圾类型，它们具有一些额外的特征（一个废瓶子具有一种颜色）或行为（一块铝可能粉化，一块钢可

能具有磁性）。除此以外，有些行为也可能不同（废纸的价值取决于它的类型和状态）。而通过继承，我们可建立起一个类型层次，准确表达出如何根据不同的类型，来针对性地解决问题。

我们不妨来看看另一个“形状”的例子。这里的“形状”指的是象圆、正方形这样的几何形状，可能在计算机辅助设计或游戏模拟中用到。其中，我们的基类型便是“形状”，每种形状都具有大小、颜色、位置等特征（或者“属性”）。针对每个形状，我们能采取的操作（行动）包括描绘、删除、移动、上色等等。从基类出发，我们可派生（继承）出一系列更具体的形状类型，包括圆、正方形、三角形等等。每种派生出来的类型都具有自己独有的一系列特征及行为（比如，某些形状可以翻转）。有些行为还可能以不同的方式进行，比如在计算一种形状的面积的时候，针对不同的形状，要采用不同的计算公式。最终，我们建立起一个类型层次，同时反映出各种形状的相同及不同之处。



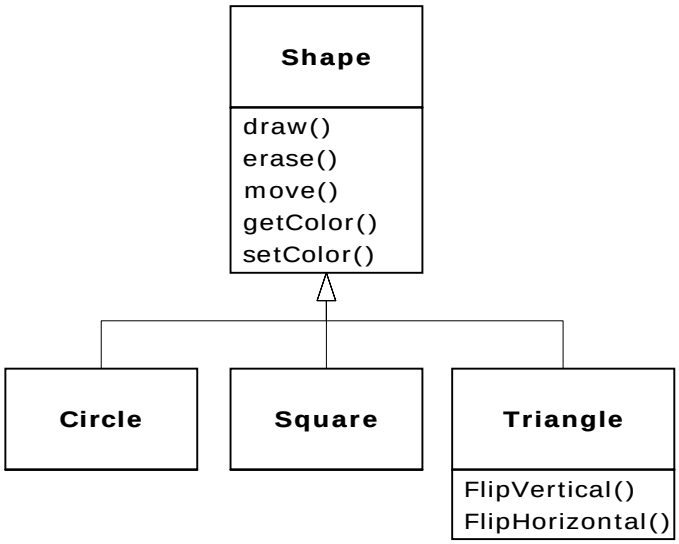
在问题空间内直接提出方案对我们来说极其有好处，因为这样便不再需要一系列中间模型，从对问题的描述“艰难”地切换至对方案的描述！使用对象，类型的层次结构便是我们的主模型，所以从对现实世界的系统描述，可直接转向对程序代码系统的描述。事实上，在运用面向对象的程序设计时，人们产生的困惑之一便是“一切过于简单”。最开始，正是由于这种过于“简单”的思路，所以面对一个复杂的问题时，人们往往不知如何是好。因此，必须改变自己的传统思维，使之迎合“面向对象”世界的思路。

从现有的一个类型中继承时，便相当于创建了一个新类型。这个新类型不仅包含了现有类型的所有成员（尽管 private 成员被隐藏起来，且不能访问），但更重要的是，它复制了基类的接口。也就是说，可向基类的对象发送的所有消息亦可原样发给派生类的对象。根据我们能够发送的消息，可以知道一个类的类型。因此，派生类具有与基类相同的类型！在前例中，就是“圆是一种几何形状”。为真正理解面向对象程序设计的含义，首先必须认识到通过继承实现的“类型等价”概念。

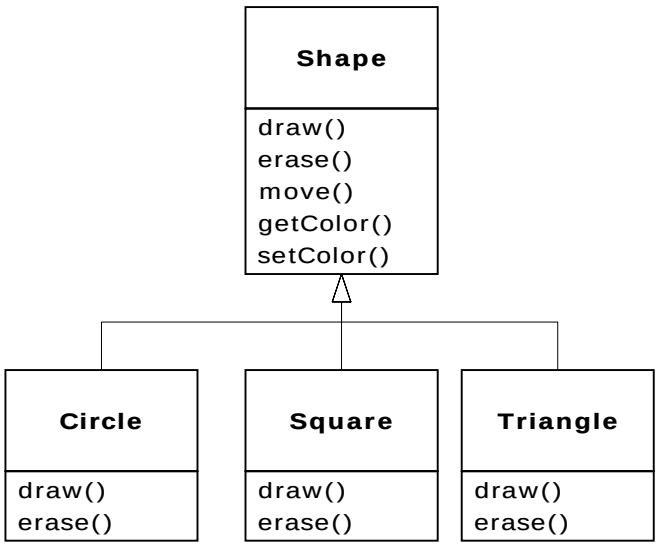
由于基类和派生类都有相同的接口，所以伴随那个接口，必须有一系列具体的、用来实现它的代码。也就是说，对象接收到一条特定的消息后，必须能够执行某些相应的代码。若只是简单地继承一个类，而不进行其他任何加工，那么来自基类接口的方法就会照搬到派生类。这意味着派生类的对象不仅有相同的类型，也有同样的行为，这一后果通常是我们不愿见到的。我们希望它们之间有所区别！

有两种做法可将新的派生类与原来的基类区分开。第一种做法十分简单：为派生类添加新函数（功能）。这些新函数并非基类接口的一部分。进行这种处理时，一般都是意识到基类不能满足我们的要求，所以需要添加更多的函数。这是一种最简单、最基本的继承用法，大多数时候都可完美地解决我们的问题。然而，事先还是要仔细调查自己的基类是否真的需

要这些额外的函数。进行面向对象的编程时，这种设计方案的“发现”与“迭代”经常都会发生。



尽管“继承”有时会暗示（特别是在 Java 中，它用来指示继承关系的关键字就是“extends”，即“扩展”的意思）我们打算为接口增添新的函数，但也并非一定如此。为了将新类区分开，第二种、也是更重要的一种做法是改变原来基类函数的行为。我们把这一处理过程称为对那个函数的“变身”或者“覆盖”(O*verriding*)。



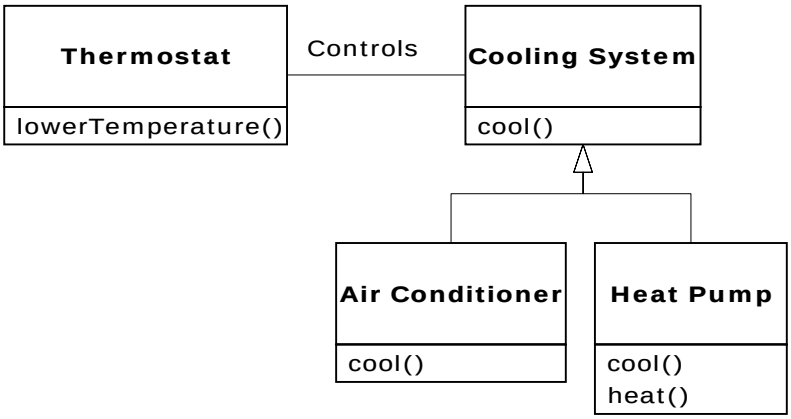
要想使一个函数“变身”，只需为派生类中的函数创建一个新定义即可。我们的目标是：“尽管函数接口没变，但它对我的新类型要做一些不同的事情！”

1.5.1 等价与类似关系

针对继承，可能会产生这样一个争论：继承之后，只能使基类的函数“变身”吗（同时

不新增那些不在基类里的成员函数）？若答案是肯定的，则派生类型就是与基类完全相同的类型，因为两者拥有完全相同的接口。这样造成的结果就是：我们完全能将派生类的一个对象换成基类的一个对象！可将其想象成一种“纯替换”。在某种意义上，这是进行继承的一种理想方式。此时，我们通常认为基类和派生类之间存在一种“等价”关系——因为我们可以理直气壮地说：“圆就是一种几何形状”。为了对继承进行测试，一个办法就是看看自己是否能把它们套入这种“等价”关系中，看看是否有意义。

但在许多时候，我们必须为派生类型加入新的接口元素。这样不仅扩展了接口，同时也创建了一种新类型。这种新类型仍可替换成基类型，但这种替换并不是完美的，因为不可从基类里访问新函数。我们将其称作“类似”[5]关系；新类型拥有旧类型的接口，但也包含了其他函数，所以不能说它们是完全等价的。举个例子来说，让我们考虑一下制冷机的情况。假定我们的房间连好了用于制冷的各种控制器；也就是说，我们已拥有必要的“接口”来控制制冷。现在假设机器出了故障，我们把它换成一台新型冷热两用空调，冬天和夏天均可使用。冷热空调“类似”制冷机，但能做更多的事情。但由于我们的房间只安装了控制制冷的系统，所以它们只限于同新机器的制冷部分打交道。新机器的接口已得到了扩展，但现有的系统并不知道除原来接口之外的任何东西，它无法与新增的制热模块打交道。



当然，在大家看到这样的设计后，会清楚认识到“制冷系统”这个基类不再那么“基础”，或者说不再显得“通用”。为此，应将其重命名为“温控系统”，令其同时包括制热模块——在这种情况下，“替换”的原理是行得通的。然而，上述示意图只是代表代码设计及现实世界中可能发生的一个例子。

认识了等价与类似的区别后，再进行替换时就会有把握得多。尽管大多数时候“纯替换”已经足够，但你会发现在某些情况下，仍有明显的理由需要在派生类的基础上增添新功能。通过前面对这两种情况的讨论，相信大家已心中有数该如何做。

1.6 利用多态实现对象的互换使用

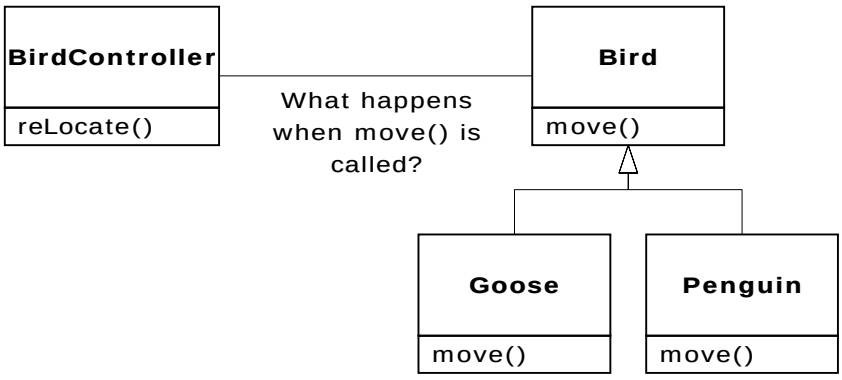
进行类型的继承时，通常不想把一个对象当作它所属的那种特定类型来对待，而是希望当作它的基类型对待。这样一来，在我们写程序的时候，就不必非要依赖某种特定的类型。

[5] 这是我自造的术语。

在前述的“几何形状”例子中，用来处理常规形状的函数用不着关心自己处理的对象到底是圆、正方形、三角形还是其他甚至根本未曾定义过的形状。这些函数只需将一条消息发给形状对象了事，不必操心对象收到消息后，会如何进行后期加工。

这样做的好处在于，即便以后有新类型加入，也不会影响到这些代码。而事实上，为了使一个面向对象的程序适应新形势的需要，以后往往还要加入大量新类型。如果每加入一个新类型，都要对原来的代码进行修改，岂非使语言本身的优势丧失殆尽？例如，在前面的例子中，我们可派生出一个新的子类型，名为“五边形”，然后只需增添和五边形处理有关的代码即可，不用对常规的形状处理代码进行修改。通过新类型的派生对程序进行扩展是一种非常重要的能力，因为它在显著改进设计的同时，还可有效地降低软件的维护费用。

不过，这样做也有一个问题，因为我们是在试图将从属于派生类型的对象当作它们的通用基类型对待（圆当作形状看，自行车当作车辆看，鸬鹚当作鸟看，等等）。假如有函数指示一种通用的形状在屏幕上画出来，指示一辆车开走，或指示鸟类飞行，那么在编译期，编译器无法准确地知道自己到底该执行哪部分代码。这就是我想要表明的要点：消息送出后，程序员不想知道最终执行的是哪部分代码；画图函数可同时应用于圆、正方形或者三角形，而根据具体类型，对象会自己执行正确的代码。假如确实不必知道最终执行的是哪部分代码，那么在我们新增一个子类型的时候，在不必更改函数调用的前提下，它执行的代码也可能发生变化。因此，编译器无法准确地知道最终执行的是什么代码。那么，面对这一情况，它会怎么做呢？举个例子来说，在下面的图示中，BirdController 对象只能对常规的 Bird（鸟类）对象进行操作，而且用不着知道这些“鸟”准确属于什么类型。从 BirdController 的角度出发，这样做当然显得非常方便，因为用不着再写其他代码来判断一只鸟的准确类型是什么，也不用知道那只鸟的行为。那么，在忽略一只鸟的具体类型的前提下，一旦调用了 move() 函数，又会出现什么情况呢？此时，会根据各种鸟习性的不同，作出准确的移动方式吗（天鹅会奔跑、飞行或游泳；企鹅则只会奔跑或游泳）？



答案在于，面向对象的编程语言采用了一种与众不同、但非常有效的设计手法：作为编译器，它本身并不能真正发出一个函数调用。那些非 OOP 语言的编译器则不同，由它们产生的函数调用称作：“早期绑定”。大家对这个术语也许会觉得有点儿陌生；即使以前听说过，很少有人用心想过它的真正意思。它的真正意思是：首先由编译器向一个特定的函数名发出一个调用，再由链接器分析这个调用，将其指向最终要执行的代码的绝对位置处。而在 OOP 的世界中，除非到了运行时，否则程序根本不能决定代码的准确位置。为此，一条消息发给一个通用对象后，还必须采取其他方案。

为解决这个问题，面向对象的语言采用了“后期绑定”的概念。一条消息发给一个对象后，实际调用的代码要等到运行时间才会确定。编译器能做的事情只是保证你调用的函数真

的存在，并会对参数和返回值进行类型检查（如果有误，就叫“类型错误”）。总之，它唯一不知道的就是最终执行的代码是什么。

为真正实现这种后期绑定，Java 从绝对函数调用中借用了一点儿代码。这段代码利用对象中保存的信息，可计算出函数体所在的地址（具体过程还会在第 7 章详述）。这样一来，根据那段代码的内容，每个对象的行为都可能有所不同。将一条消息发给一个对象后，对象实际可正确判断出自己该采取的行动。

在某些语言中（特别是 C++），必须明确指出自己希望一个函数具有后期绑定能力。不过在默认情况下，在这些语言中，成员函数并不会进行动态绑定。这当然就会带来一定的问题。Java 对此进行了改进，成员函数的绑定是默认进行的，我们不必每次都要记住增添额外的关键字，便可正确实现这一功能。

仍然是前面那个“几何形状”的例子。所有类（均基于同一个通用接口）都在本章早些时候以图解的方式表示出来。为了让大家理解“多态”的概念，在此需要写一小段代码，用它忽略类型的特定细节，并令其只和基类打交道。这段代码其实是从与特定类型有关的信息中剥离并加工出来的，所以不仅写起来简单，理解起来也容易。另外，假如通过继承的方式，加入了 Shape 的一个新类型（比如一个“六边形”），我们写的代码仍然可以使用。换言之，我们的程序（即使非常小）是具有“扩展能力”的。

假如用 Java 来写一个“方法”或“函数”（马上就要学习具体如何写），就象下面这样：

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

由于设计该函数的宗旨是能与任何“形状”（Shape）沟通，所以它和打算描绘或删除的任何具体的几何形状类型都是无关的。如果在程序的其他某个地方，我们使用了 doStuff()函数：

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

那么，对 doStuff()的调用无论如何都能正确工作，不管对象的准确类型是什么。

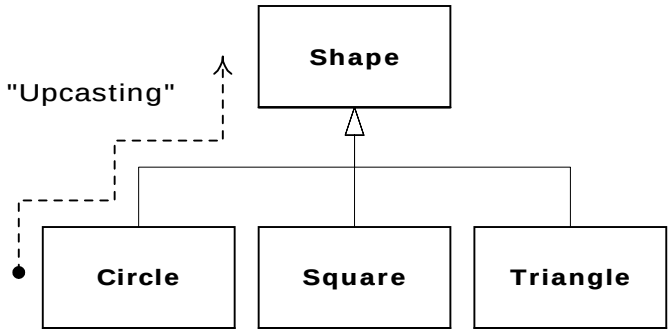
显然，这是非常有用的一个编程技巧。请大家看看下个函数调用：

```
doStuff(c);
```

此时，我们将一个 Circle（圆）传递给一个本来期待 Shape（形状）的函数。由于圆是一种几何形状，所以 doStuff()能正确地进行处理。也就是说，凡是 doStuff()能发给一个 Shape 的消息，Circle 也能接收。所以这样做是安全的，在逻辑上也是说得通的。

我们将这种把派生类型当作它的基类型处理的过程叫作“Upcasting”（向上强制转型）。其中，“cast”（强制转型）是指根据一个现成的模型创建；而“Up”（向上）表明继承的方向是从“上面”来的——即基类位于顶部，而派生类在下方展开。因此，“强制转型为基类”

就相当于在继承体系中“向上”移动，这便是“向上强制转型”这个名字的来历。



在面向对象的程序里，一般都要在某个地方用到向上强制转型。这是避免去调查准确类型的一个好办法。请看看 doStuff()里的代码：

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

注意它并未这样表达：“如果你是一个 Circle，就这样做；如果你是一个 Square，就那样做；等等”。若那样编写代码，就需检查一个 Shape 所有可能的类型，如圆、矩形等等。这显然是非常麻烦的，而且每次添加了一种新的 Shape 类型后，都要相应地进行修改。在这儿，我们只需说：“你是一种几何形状，我知道你能将自己删掉，即 erase()；也能自己描绘，即 draw()。请去采取那些行动，并自行控制所有的细节吧！”

对于 doStuff()中的代码来说，它们往往能采取正确的操作，这一点给人留下了深刻印象。假如为了一个 Circle（圆）调用 draw()，那么和为一个 Square（正方形）或 Line（直线）调用同样的函数相比，两者所执行的代码是不同的。但将 draw()消息发给一个匿名的 Shape 时（不知它到底属于何种“形状”），根据实际的 Shape 类型，就会发生正确的行为。这一点当然令我们非常震惊，因为就象前面指出的那样，当 Java 编译器为 doStuff()编译代码时，它并不能知道最终操作的准确类型是什么。所以最开始的时候，我们不得不怀疑它最后会调用基类 Shape 的 erase()和 draw()版本，而不是调用具体的 Circle、Square 或 Line 的版本。正是由于“多态”技术的采用，我们的疑虑被彻底打消了，最终的操作会正确无误地进行。编译器和运行时间系统会为我们控制所有细节；作为程序员，我们只需要知道打消心中的疑虑，而且更重要的是，知道如何利用这一特性设计自己的程序。

总之，将一条消息发给一个对象后，对象会采取正确的操作（即使其中牵扯到“向上强制转型”的运用）。

1.6.1 抽象基类和接口

设计程序时，我们经常都希望基类只为自己的派生类提供一个接口。也就是说，我们不想其他任何人实际创建基类的一个对象，只希望向上强制转型成它，以便使用它们的接口。为达到这个目的，需要把那个类变成“抽象”的——使用 abstract 关键字。若有人试图创建抽象类的一个对象，编译器就会阻止他们。这种工具可有效地强制实行一种特殊的设计。

亦可用 abstract 关键字描述一个尚未实现的方法——作为一个“根”使用，指出：“这是适用于从这个类继承的所有类型的一个接口函数，但目前尚没有对它进行任何形式的实现。”抽象方法也许只能在一个抽象类里创建。继承了一个类后，那个方法就必须实现，否

则继承的类也会变成“抽象”类。通过创建一个抽象方法，我们可以将一个方法置入接口中，不必再为那个方法提供可能毫无意义的主体代码。

interface（接口）关键字将抽象类的概念更延伸了一步，它完全禁止了任何函数定义。“接口”是一种相当有效和常用的工具，因为它能将接口同具体的实现代码完美分隔开。另外，如果愿意，亦可将多个接口都合并到一起，只是不能从多个普通类或抽象类中继承。

1.7 对象的创建和存在时间

从技术角度讲，OOP 的全部含义就是抽象数据类型、继承以及多态等。不过在某些情况下，我们还不得不注意另一些问题。本节将就这些问题进行探讨。

最重要的问题之一是对象的创建及破坏方式。对象需要的数据位于哪儿，如何控制对象的“存在时间”呢？针对这个问题，解决的方案是各异其趣的。C++认为程序的执行效率是最重要的一个问题，所以它允许程序员作出选择。为获得最快的运行速度，存储以及存在时间可在编写程序时决定，只需将对象放置在堆栈（有时也叫作自动或定域变量）或者静态存储区域即可。这样便为存储空间的分配和释放提供了一个优先级。某些情况下，这种优先级的控制是非常有价值的。然而，我们同时也牺牲了灵活性，因为在编写程序时，必须知道对象的准确的数量、存在时间、以及类型。如果要解决的是一个较常规的问题，如计算机辅助设计、仓储管理或者空中交通控制，这一方法就显得太局限了。

第二个方法是在一个内存池中动态创建对象，该内存池亦叫“堆”或者“内存堆”。若采用这种方式，除非真正运行，否则根本不知道到底需要多少个对象，也不知道它们的存在时间有多长，以及准确的类型是什么。这些参数都在程序正式运行时才决定的。若需一个新对象，只需在需要它的时候在内存堆里简单地创建它即可。由于存储空间的管理是运行期间动态进行的，所以在内存堆里分配存储空间的时间比在堆栈里创建的时间长得多（在堆栈里创建存储空间一般只需要一个简单的指令，将堆栈指针向下或向下移动即可）。由于动态创建方法使对象本来就倾向于复杂，所以查找存储空间以及释放它所需的额外开销不会为对象的创建造成明显的影响。除此以外，更大的灵活性对于常规编程问题的解决是至关重要的。

Java 采用的是第二个方法，这也是它采用的唯一方法[⑥]。每次想创建一个对象时，都要用 new 这个关键字来构建那个对象的一个动态实例。

但还要考虑另一个问题，亦即对象的“存在时间”或者“生存时间”（Lifetime）。若在堆栈或者静态存储空间里创建一个对象，编译器就能正确知道对象的存在时间有多长，到时会自动“破坏”或“清除”它。然而，假如在一个“堆”里创建，编译器就没法子知道它的准确生存时间。在象 C++这样的语言里，必须在程序中硬性规定何时破坏一个对象。但假如操作不当，就可能造成“内存漏洞”的出现（出现一片谁都用不了的内存区域），这是 C++程序最令人头痛的问题之一。为此，Java 采用了一种名为“垃圾收集器”的机制，可自动探测出一个不再使用的对象，并自动将其清除。显然，我们用这种“垃圾收集器”会方便许多，因为它免去了我们必须进行跟踪和必须写一系列相关代码的苦恼。更重要的是，用垃圾收集器可有效防止出现内存漏洞（许多 C++项目在它面前都会显得“相形见拙”）。

本节剩下的部分将讨论操纵对象时要考虑的另一些因素。

---

⑥ “原始类型”（后面还会详细介绍）是一个例外。

1.7.1 集合与迭代器

针对一个特定问题的解决，如果事先不知道需要多少个对象，或者它们的持续时间有多长，那么也不知道如何保存那些对象。既然如此，怎样才能知道那些对象要求多少空间呢？事实上根本无法办到，除非进入运行时。

在面向对象的设计中，大多数问题的解决办法似乎都有些轻率——只是简单地创建另一种类型的对象。为解决这一问题，新型对象可容纳指向其他对象的引用。当然，也可以用数组来做同样的事情，那是大多数语言都具备的一种功能。但还有其他类型的对象，同样能解决我们的问题。这种新对象通常叫作“容器”（亦叫作一个“集合”，但在 Java 库中，“集合”有其特殊的含义。为免混淆，本书采用了“容器”这个称呼）。在需要的时候，容器会自动“扩容”，以便与我们打算在其中放置的东西配合。所以，我们事先不必知道最终要在一个容器里装下多少对象。只需创建一个容器，以后的事情让它自己负责好了。

幸运的是，设计优良的 OOP 语言都配套提供了一系列容器。在 C++中，它们是以“标准模板库”（STL）的形式提供的。Object Pascal 用自己的“可视组件库”（VCL）提供容器。Smalltalk 提供了一套非常完整的容器。而 Java 也用自己的标准库提供了容器。在某些库中，一个常规容器便可满足人们的大多数要求；而在另一些库中（特别是 C++的库），则面向不同的需求提供了不同类型的容器。例如，我们可用一个“向量”（即 Vector，在 Java 中则叫作 ArrayList－数组列表），统一对所有元素的访问方式；另外，再用一个链接列表，用于统一所有元素的插入方式。这样一来，我们就可根据自己的需要，选择恰当的类型。在容器库中，另外还包括了组合（Sets）、队列（Queues）、散列表（Hashtables）、树（Trees）、堆栈（Stacks）等等。

所有容器都提供了某种形式的读写机制，便于我们放入对象，或者拿出对象。通常会有一些专用的函数，让我们将元素添加到容器里，或从中取出那些元素。通常，在取出元素时，我们会遇到更多的问题，这是由于单选函数的功能有限。假如想对容器中的一系列元素进行处理或比较，而不是仅仅针对一个元素，又该如何是好呢？

办法就是使用一个“迭代器”（Iterator），它属于一种对象，负责选择容器内的元素，并把它们提供给迭代器的用户。作为一个类，它也具有一定级别的抽象特性。利用这种抽象，可将容器的细节与用于访问那个容器的代码隔离开。通过迭代器的作用，容器被抽象成一个简单的序列。迭代器允许我们遍历那个序列，同时毋需关心其底层结构——换言之，不管它是一个 ArrayList（数组列表）、LinkedList（链接列表）、堆栈（Stack），还是一个别的什么东西。

这样一来，我们就可以灵活地改变底层数据结构，不会对程序里的代码造成干扰。Java 最开始（在 1.0 和 1.1 版中）提供的是一个标准迭代器，名为 Enumeration（枚举），为它的所有容器类提供服务。Java 2 则新增了一个更完整的容器库，其中包含了一个名字就叫“Iterator”的迭代器，它可以做的事情比老式的 Enumeration 多得多。

从设计角度出发，我们需要的是一个全功能的序列。通过对它的操纵，应该能解决自己的问题。如果一种类型的序列即可满足我们的所有要求，那么完全没有必要再换用不同的类型。但是，有两方面的理由促使我们需要选择容器。首先，容器提供了不同的接口类型以及外部行为。堆栈的接口与行为与队列的不同，而队列的接口与行为又与一个组合（Set）或者列表（List）不同。可根据需要作出不同选择，灵活地改进自己的设计。

其次，不同的集合在进行特定的操作时往往有不同的工作效率。最好的例子便是数组列表（ArrayList 和链接列表（LinkedList）的区别。它们都属于简单的序列，可以拥有完全一致的接口和外部行为。但在执行一些特定的任务时，对系统造成的开销却是完全不同的。随身访问数组列表内的元素时，进行的是一种“常时”操作——换言之，无论选择的元素是什

么，用时都是一样的。但在一个链接列表中，若想四处移动，随机挑选一个元素，就需付出“惨重”的代价。而且假设某个元素位于列表较远的地方，找到它所需的时间也会长许多。但在另一方面，如果想在序列中部插入一个元素，用链接列表就比用数组列表划算得多。这些以及其他操作都有不同的执行效率，具体取决于序列的基础结构是什么。在设计阶段，我们可试着先用一个链接列表。最后调整性能的时候，再视情况换成数组列表。由于抽象是通过迭代器进行的，所以两种列表可任意选择和切换，不会对代码造成太大的影响。

最后，请记住容器只是一个用来存放对象的“储藏所”。如果那个储藏所能满足我们的所有需要，就完全没必要关心它具体是如何实现的（这是大多数类型对象的一个基本概念）。假如在一个专门的编程环境中工作，它由于其他原因而具有内在的一系列开销，那么数组列表和链接列表之间的性能差异就可能无关紧要。我们可能只需要一种类型的序列。我们甚至能想象一下一个“完美”的容器抽象，它能根据自己的使用方式，自动改变基层的实现。

1.7.2 单根结构

在面向对象的程序设计中，自 C++问世起便显得尤为突出的一个问题是：所有类最终是否都应从单独一个基类继承？在 Java 中（与其他几乎所有 OOP 语言一样），对这个问题的答案都是肯定的，而且这个终级基类的名字很简单，就是一个“Object”。这种“单根结构”具有许多方面的优点。

单根结构中的所有对象都有一个通用接口，所以它们最终都属于相同的类型。另一种方案（就象 C++那样）是我们不能保证所有东西都属于相同的基类型。从向后兼容的角度看，这一方案可与 C 模型更好地配合，而且可以认为它的限制更少一些。但假如我们想进行纯粹的面向对象编程，那么必须构建自己的结构，以期获得与内建到其他 OOP 语言里的同样的便利。此时，便需要添加我们要用到的各种新类库，还要使用另一些不兼容的接口。理所当然地，这也需要付出额外的精力使新接口与自己的设计方案配合（可能还需要多重继承）。为得到 C++额外的“灵活性”，付出这样的代价值得吗？当然，如果真的需要——如果早已是 C 专家，如果对 C 有难舍的情结——那么就真的很值得。但假如你是一名新手，首次接触这类设计，象 Java 那样的替换方案也许会更省事一些。

单根结构中的所有对象（比如所有 Java 对象）都可保证拥有一些特定的功能。在自己的系统中，我们知道对每个对象都能进行一些基本操作。采用一个单根结构，再加上所有对象都在内存堆中创建，便可极大简化参数的传递（这在 C++里可是一个非常让人头痛的的概念）。

利用单根结构，我们可以更方便地实现一个垃圾收集器（Java 甚至自带了这个东西）。与此有关的必要支持可安装在基类中，而垃圾收集器又可将适当的消息发给系统内的任何对象。假如没有这种单根结构，系统通过一个“引用”来操纵对象，那么实现垃圾收集器的途径会有很大的不同，而且会面临许多障碍。

由于运行时类型信息肯定存在于所有对象中，所以永远不会遇到判断不出一个对象的类型的情况。这对系统级的操作来说显得特别重要（比如在违例控制的时候）；而且也能在程序设计中获得更大的灵活性。

1.7.3 容器库与容器的使用

由于容器是我们常常都要用到的一种工具，所以一个容器库是十分必要的，它应该可以方便地重复使用。这样一来，我们就可以方便地取用各种容器，将其插入自己的程序。Java 便提供了这样的一个库，它应能满足我们大多数时候的需要。

向下强制转型与模板／通用性

为了使这些容器能够复用，Java 提供了一种通用类型，以前曾把它叫作“Object”。单根结构意味着：“所有东西归根结底都是一个对象！”因此，容纳了 Object 的一个容器实际可容纳任何东西。这就使得容器的复用变得十分方便。

要想使用这样的一个容器，只需添加指向它的对象引用即可，以后便可通过这个引用，来重复使用对象。但由于容器只能容纳 Object，所以在我们向容器里添加对象引用时，它会向上强制转型成 Object，这样便丢失了它的身份或者标识信息。再次用它的时候，得到的是一个 Object 引用，而非指向我们早先置入的那个类型的引用。所以怎样才能恢复它的本来面貌，令其再次出现我们早先置入容器内的一系列有用的接口呢？

在这里，我们再次用到了强制转型（Cast）的概念。但这一次不是在分级结构中向上强制转型成一种更“通用”的类型，而是向下强制转型成一种更“特殊”的类型。这种强制转型方法叫作“向下强制转型”（Downcasting）。举个例子来说，我们知道在向上强制转型的时候，Circle（圆）属于 Shape（形状）的一种类型，所以向上强制转型是安全的，因为圆肯定是一种形状。但是，在面临一个 Object 的时候，我们并不知道它到底是 Circle 还是 Shape，所以很难保证向下强制转型是安全的，除非确切地知道自己要操作的是什么（它到底是不是圆？否则就会闹出“形状是一种圆”这样的笑话）。

不过，这样做也不是完全危险的，因为假如向下强制转型成为错误的东西，会得到我们称为“违例”（Exception）的一种运行时错误。我们稍后即会对此详加解释。从一个容器中提取对象引用时，必须通过某种方式，准确地记住它们是什么，以保证向下强制转型的正确进行。

向下强制转型和运行时间检查都要求花费额外的时间来运行程序，而且程序员必须付出额外的精力。既然如此，我们能不能创建出一个“智能”集合，让它自己检查容纳的类型呢？这样一来，便可避免进行向下强制转型，也不用担心会出现错误。答案是肯定的，我们可采用“参数化类型”的概念，它们是编译器能自动定制的类，可与特定的类型配合。例如，通过使用一个参数化容器，编译器可对那个容器进行定制，使其只接受 Shape，而且只提取 Shape。

参数化类型是 C++一个重要的组成部分，部分原因是由于 C++没有采用单根结构。在 C++中，用于实现参数化类型的关键字是“template”（模板）。Java 目前尚未提供参数化类型，因为它采用了单根结构，似乎可以实现类似的功能（尽管显得有些“笨”）。不过，目前有一份提议正在审议之中，其中便提出了要在 Java 中增加参数化类型，采用和 C++模板大致相同的语法。

1.7.4 清除时的困拢：谁来打扫卫生？

每个对象都需要资源才能“生存”，其中最引人注目的资源便是内存。如果不再需要使用一个对象，必须将其清除，以释放出这些资源，以便让给其他对象使用。假如一个程序非常简单，那么对象的清除并不显得是个大问题：创建一个对象，在需要的时候用它，然后将其清除或者“破坏”掉就可以了。不过令人遗憾的是，我们平常遇到的问题往往都要比这复杂得多。

举个例子来说，假设我们要设计一套系统，用它管理一个机场的空中交通（同样的模型也可能适于管理一个仓库的货柜、或者一套影带出租系统、或者宠物店的宠物房）。这初看似乎十分简单：构造一个集合用来容纳飞机，然后创建一架新飞机，将其置入集合。对进入空中交通管制区的所有飞机都如此处理。至于清除，在一架飞机离开这个区域的时候把它简单地删去即可。

但实际情况可没有这么简单，我们可能还需要另一套系统来记录与飞机有关的数据。当然，和主控制功能不同的是，这些数据的重要性一开始可能并不会显露出来。例如，这种记

录反映的可能是飞离机场的所有小飞机的飞行计划。由此，我们得到了由小飞机构成的第二个容器。一旦创建了一个飞机对象，如果它是一架小飞机，那么也必须把它置入这个容器。然后在系统空闲期，再对这个容器中的对象进行一些后台处理。

上述问题还可以进一步复杂化：如何才能知道什么时间删除对象呢？用完对象后，系统的其他某些部分可能仍然要发挥作用。同样的问题也会在其他大量场合出现。而且在一些特殊的程序设计系统中（如 C++），在用完一个对象之后，必须明确地将其删除，这就使得问题变得越来越复杂。

在 Java 中，“垃圾收集器”在设计时已考虑到了内存的释放问题（尽管这并不包括清除一个对象涉及到的其他方面）。垃圾收集器“知道”一个对象在什么时候不再使用，然后会自动释放那个对象占据的内存空间。采用这种方式，另外加上所有对象都从同一个根类 Object 继承的事实，而且由于我们只能在内存堆中以单向方式创建对象，所以 Java 的编程要比 C++的编程简单得多。现在，我们只需在 Java 中作出少量选择，即可克服原先存在于 C++中的大量障碍。

垃圾收集器对效率及灵活性的影响

既然优点多多，在 C++里为何没有得到充分发挥呢？好吧，事实上，为了这种编程的方便性，我们还是付出了一定代价的。其中最大的代价便是在运行时系统开销。正如早先提到的那样，在 C++中，我们可在堆栈中创建对象。在这种情况下，对象会得以自动清除（但也丧失了在运行时间随心所欲创建对象的灵活性）。在堆栈中创建对象是为对象分配存储空间最有效的一种方式，也是释放那些空间最有效的一种方式。在内存堆（Heap）中创建对象可能要付出昂贵得多的代价。假如总是从同一个基类继承，并使所有函数调用都具有“多态”特征，那么也不可避免地需要付出一定的代价。但垃圾收集器是一种特殊的问题，因为我们永远不能确定它什么时候启动或者要花多长的时间。这意味着在 Java 程序执行期间，永远存在着一种不确定的因素。所以在某些特殊场合下，我们必须避免用它——比如在一个程序的执行必须保持稳定、连贯的时候（通常把它们叫作“实时程序”，尽管并不是所有实时编程问题都有这方面的要求）。

C++语言的设计者曾向全世界所有 C 程序员发出请求（最后的结果非常成功）：不要在语言里加入可能对 C++的速度或使用造成影响的任何特性，要不然的话干脆用 C 好了。这个目的达到了，但代价就是 C++的编程不可避免地复杂起来。Java 比 C++简单，但付出的代价是执行效率以及一定程度的灵活性。但对大多数程序设计问题来说，Java 无疑都应当是我们的首选。

1.8 违例控制：解决错误

从最古老的程序语言开始，“出了错怎么办？”一直都是设计者们需要解决的一大难题。由于很难设计出一套完美的错误控制方案，许多语言干脆简单地忽略这个问题，将其转嫁给库设计人员，让他们疲于测试产品的各种运行环境。显然，这样做的效果十分不理想，因为这样的测试永远不可能“全面”的。对大多数错误控制方案来说，最主要的一个问题是它们严重依赖程序员的警觉性，而不是依赖语言本身的一系列强制性手段。假如程序员不够警惕——若比较匆忙，这几乎是肯定会发生的事情——程序所依赖的错误控制方案便会失效。

“违例控制”将错误控制方案内置到程序语言当中，有时甚至内建到操作系统内。这里的“违例”（Exception）属于一个特殊的对象，它会从产生错误的地方“扔”或“掷”出来。随后，这个违例会被设计用于控制特定类型错误的“违例控制器”捕获。在情况变得不对劲的时候，可能有几个违例控制器并行捕获对应的违例对象。由于采用的是独立的执行路径，

所以不会干扰我们的常规执行代码。这样便使代码的编写变得更加简单，因为不必经常性强制检查代码。除此以外，“掷”出的一个违例不同于从函数返回的错误值，也不同于由函数设置的一个标志。那些错误值或标志的作用是指示一个错误状态，是可以忽略的。但违例不能被忽略，所以肯定能在某个地方得到处置。最后，利用违例能够可靠地从一个糟糕的环境中恢复。此时一般不需要退出，我们可以采取某些处理，恢复程序的正常执行。显然，这样编制出来的程序显得更加可靠。

Java 的违例控制机制与大多数程序设计语言都有所不同。因为在 Java 中，违例控制模块是从一开始就封装好的，所以你必须使用它!如果没有自己写一些代码来正确地控制违例，就会得到一条编译期出错提示。这样便可保证程序的连贯性，使错误控制变得更加容易。

这里要提醒大家注意的是，“违例控制”并非面向对象程序语言的一种“专属品”，尽管在面向对象的程序设计语言中，违例通常是用一个对象表示的，但早在 OOP 语言问世之前，违例控制的概念便已出现了。

1.9 多 线 程

在计算机编程中，一个基本的概念便是同时能对多个任务加以控制。许多程序设计问题都要求程序能暂时停下手头的工作，改为处理其他一些问题，再返回主进程。我们可通过多种途径达到这个目的。最开始的时候，那些拥有机器低级知识的程序员编写出一些“中断服务例程”，主进程的暂停是通过硬件级的中断实现的。尽管这是一个有用的办法，但编出的程序很难移植。要把程序转到一种新出产的机器上运行，不仅速度变慢，代价也颇为高昂（倒不如完全重写整个程序了）。

有些时候，对一些实时性要求很强的任务，中断是很有必要的。但还存在其他许多问题，它们只要求将问题划分进入独立运行的程序片断中，使整个程序能更迅速地响应用户的请求。在一个程序中，这些独立运行的片断叫作“线程”（Thread），利用它编程的概念就叫作“多线程处理”。多线程处理一个常见的例子就是用户界面。采用了线程之后，用户可在任何时间按下一个按钮，然后程序会立即作出响应，而不是让用户等程序完成了当前任务以后才开始响应，那会让用户非常“恼火”。

最开始，线程只是用于分配单个处理器的处理时间的一种工具。但假如操作系统本身支持多个处理器，那么每个线程都可分配给一个不同的处理器，真正进入“并行运算”状态。从程序设计语言的角度看，多线程操作最有价值的特性之一就是程序员不必关心到底使用了多少个处理器。程序在逻辑意义上被分割为数个线程；假如机器本身安装了多个处理器，那么程序会运行得更快，毋需作出任何特殊调校。

根据前面的论述，大家可能感觉线程处理非常简单。但必须注意一个问题：共享资源！如果有多个线程同时运行，而且它们试图访问相同的资源，问题便产生了。举个例子来说，两个进程不能将信息同时发送给一台打印机。为解决这个问题，对那些可共享的资源来说（比如打印机），它们在使用期间必须进入锁定状态。换句话说，一个线程可将资源锁定，在完成了它的任务后，再解开（释放）这个锁，使其他线程可接着使用同样的资源。

Java 的多线程机制已内建到语言中，这使一个可能会非常复杂的问题变得简单起来。对多线程处理的支持是在“对象”这一级加以支持的。所以，一个执行线程可表达为一个对象。Java 也提供了有限的资源锁定方案。它能锁定任何对象占用的内存（内存实际是多种共享资源中的一种），所以同一时间只能有一个线程使用特定的内存空间。为达到这个目的，需要用到 synchronized（已同步）关键字。其他类型的资源必须由程序员明确锁定，这通常要求程序员创建一个对象，用它代表一把锁，所有线程在访问那个资源时都必须先检查这把锁是否“锁”着。

1.10 持 久 性

创建一个对象后，只要我们需要，它便会一直存在下去。但在程序结束时，对象的“生存期”也会宣告结束。尽管这一现象表面上非常合理，但深入追究就会发现，假如在程序停止运行以后，对象也能继续存在，并能保留它的全部信息，那么在某些情况下，将是一件非常有价值的事情！下次启动程序时，对象仍在那里，里面保留的信息仍是程序上一次运行时的那些信息。当然，也可将信息写入一个文件或者数据库，实现相同的效果。但既然我们的宗旨是“将所有东西都变成对象”，所以假如能将对象声明成“永远存在”，并令其为我们照看其他所有细节，无疑会是一个非常有用的功能。

Java 提供了对“有限永远存在”的支持，这意味着我们可将对象简单地保存到磁盘上，以后任何时间都可取回。之所以称它为“有限”的，是由于我们仍需明确发出调用，进行对象的保存和取回工作。另外，JavaSpaces（第 15 章还会详细讨论）支持某种形式的“对象永久保存”。在未来的 Java 版本中，对“持久性”的支持有望更加全面。

1.11 Java 和 Internet

既然 Java 只不过是一种程序语言，大家可能会奇怪它为什么值得如此重视，为什么还有这么多的人认为它是计算机程序设计的一个里程碑呢？如果你是一位传统的编程人员，对于这个问题可能会感到不好理解。在这里不妨告诉你：Java 除了能解决传统的程序设计问题以外，还能解决 World Wide Web（万维网）上的编程问题！

1.11.1 什么是 Web？

Web 这个词刚开始显得有些泛泛，似乎“冲浪”、“网上存在”以及“主页”等等都和它拉上了一些关系。甚至还有一种“Internet 综合症”的说法，对许多人狂热的上网行为提出了质疑。我们在这里有必要作一些深入的探讨，但在这之前，必须理解客户机／服务器系统的概念，这是充斥着许多令人迷惑的问题的又一个知识领域。

客户机／服务器计算

客户机／服务器系统的基本思想是我们能在一个统一的地方集中存放信息资源。一般将数据集中保存在某个数据库中，根据其他人或者机器的请求将信息投递给对方。客户机／服务器概述的一个关键在于信息是“集中存放”的。所以我们能方便地更改信息，然后将修改过的信息发放给信息的消费者。将各种元素集中到一起，信息仓库、用于投递信息的软件以及信息及软件所在的那台机器，它们联合起来便叫作“服务器”（Server）。而对那些驻留在远程机器上的软件，它们需要与服务器通信，取回信息，进行适当的处理，然后在远程机器上显示出来，这些就叫作“客户机”（Client）。

这样看来，客户机／服务器的基本概念并不复杂。这里要注意的一个主要问题是单个服务器需要同时向多个客户提供服务。在这一机制中，通常少不了一套数据库管理系统，使设计人员能将数据布局封装到表格中，以获得最优的使用。除此以外，系统经常允许客户将新信息插入一个服务器。这意味着必须确保客户的新数据不会与其他客户的新数据冲突，或者说需要保证那些数据在加入数据库的时候不会丢失（用数据库的术语来说，这叫作“事务处理”）。客户软件发生了改变之后，它们必须在客户机器上构建、调试以及安装。所有这些会使问题变得比我们一般想象的复杂得多。另外，对多种类型的计算机和操作系统的支持也是一个大问题。最后，性能的问题显得尤为重要：可能会有数百个客户同时向服务器发出请求。

所以任何微小的延误都是不能忽视的。为尽可能缓解潜伏的问题，程序员需要谨慎地分散任务的处理负担。一般可以考虑让客户机负担部分处理任务，但有时亦可分派给服务器所在地的其他机器，那些机器亦叫作“中间件”（中间件也用于改进对系统的维护）。

所以在具体实现的时候，其他人发布信息这样一个简单的概念可能变得异常复杂。有时甚至会使人产生完全无从着手的感觉。客户机／服务器的概念在这时就可以大显身手了。事实上，大约有一半的程序设计活动都可以采用客户机／服务器的结构。这种系统可负责从处理订单及信用卡交易，一直到发布各类数据的方方面面的任务——股票市场、科学研究、政府运作等等。在过去，我们一般为单独的问题采取单独的解决方案；每次都要设计一套新方案。这些方案无论创建还是使用都比较困难，用户每次都要学习和适应新界面。客户机／服务器问题需要从根本上加以变革！

Web 是一个巨大的服务器

Web 实际就是一套规模巨大的客户机／服务器系统。但它的情况要复杂一些，因为所有服务器和客户都同时存在于单个网络上面。但我们没必要了解更进一步的细节，因为唯一要关心的就是一次建立同一个服务器的连接，并同它打交道（即使可能要在全世界的范围内搜索正确的服务器）。

最开始的时候，这是一个简单的单向操作过程。我们向一个服务器发出请求，它向我们回传一个文件，由于本机的浏览器软件（亦即“客户”或“客户程序”）负责解释和格式化，并在我们面前的屏幕上正确地显示出来。但人们不久就不满足于只从一个服务器传递网页。他们希望获得完全的客户机／服务器能力，使客户（程序）也能反馈一些信息到服务器。比如希望对服务器上的数据库进行检索，向服务器添加新信息，或者下一份订单等等（这也提供了比以前的系统更高的安全要求）。在 Web 的发展过程中，我们可以很清晰地看出这些令人心喜的变化。

Web 浏览器的发展终于迈出了重要的一步：某个信息可在任何类型的计算机上显示出来，毋需任何改动。然而，浏览器仍然显得很原始，在用户迅速增多的要求面前显得有些力不从心。它们的交互能力不够强，而且对服务器和因特网都造成了一定程度的干扰。这是由于每次采取一些要求编程的操作时，必须将信息反馈回服务器，在服务器那一端进行处理。所以完全可能需要等待数秒乃至数分钟的时间才会发现自己刚才拼错了一个单词。由于浏览器只是一个纯粹的查看程序，所以连最简单的计算任务都不能进行（当然在另一方面，它也显得非常安全，因为不能在本机上面执行任何程序，避开了程序错误或者病毒的骚扰）。

为解决这个问题，人们采取了许多不同的方法。最开始的时候，人们对图形标准进行了改进，使浏览器能显示更好的动画和视频。为解决剩下的问题，唯一的办法就是在客户端（浏览器）内运行程序。这就叫作“客户端编程”，它是对传统“服务器编程”的一个非常重要的拓展。

1.11.2 客户端编程

Web 最初采用的“服务器一浏览器”方案可提供交互式内容，但这种交互能力完全由服务器提供，为服务器和因特网带来了不小的负担。服务器一般为客户浏览器产生静态网页，由后者简单地解释并显示出来。基本 HTML 语言提供了简单的数据收集机制：文字输入框、复选框、单选钮、列表以及下拉列表等，另外还有一个按钮，只能由程序规定重新设置表单中的数据，以便回传给服务器。用户提交的信息通过所有 Web 服务器均能支持的“通用网关接口”（CGI）回传到服务器。包含在提交数据中的文字指示 CGI 该如何操作。最常见的行动是运行位于服务器的一个程序。那个程序一般保存在一个名为“cgi-bin”的目录中（按下 Web 页内的一个按钮时，请注意一下浏览器顶部的地址窗，经常都能发现“cgi-bin”的

字样）。大多数语言都可用来编制这些程序，但其中最常见的是 Perl。这是由于 Perl 是专为文字的处理及解释而设计的，所以能在任何服务器上安装和使用，无论采用的处理器或操作系统是什么。

今天的许多 Web 站点都严格地建立在 CGI 的基础上，事实上几乎所有事情都可用 CGI 做到。唯一的问题就是响应时间。CGI 程序的响应取决于需要传送多少数据，以及服务器和因特网两方面的负担有多重（而且 CGI 程序的启动比较慢）。Web 的早期设计者并未预料到当初绰绰有余的带宽很快就变得不够用，这正是大量应用充斥网上造成的结果。例如，此时任何形式的动态图形显示都几乎不能连贯地显示，因为此时必须创建一个 GIF 文件，再将图形的每种变化从服务器传递给客户。而且大家应该对输入表单上的数据校验有着深刻的体会。原来的方法是我们按下网页上的提交按钮（Submit）；数据回传给服务器；服务器启动一个 CGI 程序，检查用户输入是否有错；格式化一个 HTML 页，通知可能遇到的错误，并将这个页回传给我们；随后必须回到原先那个表单页，再输入一遍。这种方法不仅速度非常慢，也显得非常繁琐。

解决的办法就是客户端的程序设计。运行 Web 浏览器的大多数机器都拥有足够强的能力，可进行其他大量工作。与此同时，原始的静态 HTML 方法仍然可以采用，它会一直等到服务器送回下一个页。客户端编程意味着 Web 浏览器可获得更充分的利用，并可有效改善 Web 服务器的交互（互动）能力。

对客户端编程的讨论与常规编程问题的讨论并没有太大的区别。采用的参数肯定是相同的，只是运行的平台不同：Web 浏览器就象一个有限的操作系统。无论如何，我们仍然需要编程，仍然会在客户端编程中遇到大量问题，同时也有很多解决的方案。在本节剩下的部分里，我们将对这些问题进行一番概括，并介绍在客户端编程中采取的对策。

插件

学习客户端编程的时候，要攻克的第一个难关便是插件设计。利用插件（Plug-Ins），程序员可以方便地为浏览器添加新功能，用户只需下载一些代码，把它们“插入”浏览器的适当位置即可。这些代码的作用是告诉浏览器“从现在开始，你可以进行这些新活动了”（仅需下载这些插入一次）。有些快速和功能强大的行为是通过插件添加到浏览器的。但插件的编写并不是一件简单的任务。在我们构建一个特定的站点时，可能并不希望涉及这方面的工作。对客户端程序设计来说，插件的价值在于它允许专业程序员设计出一种新的语言，并将那种语言添加到浏览器，同时不必经过浏览器原创者的许可。由此可以看出，插件实际是浏览器的一个“后门”，允许创建新的客户端程序设计语言（尽管并非所有语言都是作为插件实现的）。

脚本编制语言

正是由于插件的问世，造成了全世界各种各样的脚本语言出现爆炸性发展。通过这种脚本语言，可将用于自己客户端程序的源码直接插入 HTML 页，而对那种语言进行解释的插件会在 HTML 页显示的时候自动激活。脚本语言一般都倾向于尽量简化，易于理解。而且由于它们是从属于 HTML 页的一些简单正文，所以只需向服务器发出对那个页的一次请求，即可非常快地载入。缺点是我们的代码全部暴露在人们面前。另一方面，由于通常不用脚本编制语言做过份复杂的事情，所以这个问题暂且可以放在一边。

脚本语言真正面向的是特定类型问题的解决，其中主要涉及如何创建更丰富、更具有互动能力的图形用户界面（GUI）。然而，脚本语言也许能解决客户端编程中 80％的问题。你碰到的问题可能完全就在那 80％里面。而且由于脚本编制语言的宗旨是尽可能地简化与快速，所以在考虑其他更复杂的方案之前（如 Java 及 ActiveX），首先应想一下脚本语言是否

可行。

目前讨论得最多的脚本编制语言包括 JavaScript（它与 Java 没有任何关系；之所以叫那个名字，完全是一种市场策略）、VBScript（同 Visual Basic 很相似）以及 Tcl/Tk（来源于流行的跨平台 GUI 构造语言）。当然还有其他许多语言，也有许多正在开发中。

JavaScript 也许是目前最常用的，它得到的支持也最全面。无论 Netscape Navigator，Microsoft Internet Explorer，还是 Opera 等 Web 浏览器，目前都提供了对 JavaScript 的支持。除此以外，市面上讲解 JavaScript 的书籍也要比讲解其他语言的书多得多。有些工具还能利用 JavaScript 自动产生网页。当然，如果你已有 Visual Basic 或者 Tcl/Tk 的深厚功底，那么用它们当然要简单得多，起码可以避免学习新语言的烦恼（仅仅解决 Web 方面的问题就已够让人头痛了）。

Java

如果说一种脚本语言能解决 80％的客户端程序设计问题，那么剩下的 20％又该怎么办呢？它们属于一些高难度的问题吗？目前最流行的方案就是 Java。它不仅是一种功能强大、高度安全、可以跨平台使用以及国际通用的程序设计语言，也是一种具有旺盛生命力的语言。对 Java 的扩展是不断进行的，提供的语言特性和库能够很好地解决传统语言不能解决的问题，比如多线程操作、数据库访问、连网程序设计以及分布式计算等等。Java 通过“小程序”（Applet）巧妙地解决了客户端编程问题。

小程序（或“程序片”）是一种非常小的程序，只能在 Web 浏览器中运行。作为 Web 页的一部分，小程序代码会自动下载回来（这和网页中的图片差不多）。激活小程序后，它会执行一个程序。小程序的一个优点体现在：通过它，一旦用户需要客户软件，软件就可从服务器自动地“拉”回来。它们能自动取得客户软件的最新版本，不会出错，也没有重新安装的麻烦。由于 Java 的设计原理，程序员只需要创建程序的一个版本，那个程序能在几乎所有计算机以及安装了 Java 解释器的浏览器中运行。由于 Java 是一种全功能的编程语言，所以在向服务器发出一个请求之前，我们能先在客户端做完尽可能多的工作。例如，再也不必通过因特网传送一个请求表单，再由服务器确定其中是否存在一个拼写或者其他参数错误。大多数数据校验工作均可在客户端完成，没有必要坐在计算机前面焦急地等待服务器的响应。这样一来，不仅速度和响应的灵敏度得到了极大的提高，对网络和服务器造成的负担也可以明显减轻，这对保障因特网的畅通是至关重要的。

与脚本程序相比，Java 小程序的另一个优点是它采用编译好的形式，所以在客户端是看不到源码的。当然在另一方面，反编译 Java 小程序也并非是件难事，而且代码的隐藏一般并不是个重要的问题。大家要注意另两个重要的问题。正如本书以前会讲到的那样，编译好的 Java 小程序可能包含了许多模块，所以要多次访问（连接）服务器才能全部下载回来（在 Java 1.1 和更高的版本中，这个问题已得到了有效的改善——利用 Java 压缩档，即 JAR 文件——它允许设计者将所有必要的模块都封装到一起，供用户统一下载）。在另一方面，脚本程序是作为 Web 页正文的一部分集成到 Web 页内的。这种程序一般都非常小，可有效减少对服务器的点击数。另一个因素是学习方面的问题。不管你平时听别人是怎么说的，但 Java 实际并不是一种十分容易便可学会的语言。如果你以前是一名 Visual Basic 程序员，那么转向 VBScript 会是一种最快捷的方案。由于 VBScript 可以解决大多数典型的客户机 / 服务器问题，所以一旦上手，就很难下定决心再去学习 Java。如果对脚本编制语言比较熟，那么在转向 Java 之前，建议先熟悉一下 JavaScript 或者 VBScript，因为它们可能已能满足你的要求，不必经历学习 Java 的艰苦过程。

ActiveX

在某种程度上，Java 的一个有力竞争对手应该是微软的 ActiveX，尽管它采用的是完全不同的一套实现机制。ActiveX 最早是一种纯 Windows 的方案。经过一家独立的专业协会的努力，ActiveX 现在已具备了跨平台使用的能力。实际上，ActiveX 的意思是“假如你的程序同它的工作环境正常连接，它就能进入 Web 页，并在支持 ActiveX 的浏览器中运行”（IE 固化了对 ActiveX 的支持，而 Netscape 需要一个插件）。所以，ActiveX 并没有限制我们使用一种特定的语言。比如，假设我们已经是一名有经验的 Windows 程序员，能熟练地使用象 C++、Visual Basic 或者 BorlandDelphi 那样的语言，就能几乎不加任何学习地创建出 ActiveX 组件。事实上，ActiveX 是在我们的 Web 页中使用“历史遗留”代码的最佳途径。

安全

自动下载和通过因特网运行程序听起来就象是一个病毒制造者的梦想。在客户端的编程中，ActiveX 带来了最让人头痛的安全问题。点击一个 Web 站点的时候，可能会随同 HTML 网页传回任何数量的东西：GIF 文件、脚本代码、编译好的 Java 代码以及 ActiveX 组件。有些是无害的；GIF 文件不会对我们造成任何危害，而脚本编制语言通常在自己可做的事情上有着很大的限制。Java 也设计成在一个安全“沙箱”里在它的小程序中运行，这样可防止操作位于沙箱以外的磁盘或者内存区域。

ActiveX 是所有这些里面最让人担心的。用 ActiveX 编写程序就象编制 Windows 应用程序——可以做自己想做的任何事情。下载回一个 ActiveX 组件后，它完全可能对我们磁盘上的文件造成破坏。当然，对那些下载回来并不限于在 Web 浏览器内部运行的程序，它们同样也可能破坏我们的系统。从 BBS 下载回来的病毒一直是个大问题，但因特网的速度使得这个问题变得更加复杂。

目前解决的办法是“数字签名”，代码会得到权威机构的验证，显示出它的作者是谁。这一机制的基础是认为病毒之所以会传播，是由于它的编制者匿名的缘故。所以假如去掉了匿名的因素，所有设计者都不得不为它们的行为负责。这似乎是一个很好的主意，因为它使程序显得更加正规。但我对它能消除恶意因素持怀疑态度，因为假如一个程序便含有 Bug，那么同样会造成问题。

Java 通过“沙箱”来防止这些问题的发生。Java 解释器内嵌于我们本地的 Web 浏览器中，在小程序装载时会检查所有有嫌疑的指令。特别地，小程序根本没有权力将文件写进磁盘，或者删除文件（这是病毒最喜欢做的事情之一）。我们通常认为小程序是安全的。而且由于安全对于营建一套可靠的客户机 / 服务器系统至关重要，所以会给病毒留下漏洞的所有错误都能很快得到修复（浏览器软件实际需要强行遵守这些安全规则；而有些浏览器则允许我们选择不同的安全级别，防止对系统不同程度的访问）。

大家或许会怀疑这种限制是否会妨碍我们将文件写到本地磁盘。比如，我们有时需要构建一个本地数据库，或将数据保存下来，以便日后离线使用。最早的版本似乎每个人都能在线做任何敏感的事情，但这很快就变得非常不现实（尽管低价“互联网工具”有一天可能会满足大多数用户的需要）。解决的方案是“签了名的小程序”，它用公共密钥加密算法验证小程序确实来自它所声称的地方。当然在通过验证后，签了名的一个小程序仍然可以开始清除你的磁盘。但从理论上说，既然现在能够找到创建人“算帐”，他们一般不会干这种蠢事。Java 1.1 为数字签名提供了一个框架，在必要时，可让一个小程序“走”到沙箱的外面来。

数字签名遗漏了一个重要的问题，那就是人们在因特网上“移动”的速度。如下载回一个错误百出的程序，而它又很不幸地真的干了某些蠢事，需要多久的时间才能发觉这一点呢？这也许是几天，也可能几周之后。发现了之后，又如何追踪当初肇事的程序呢（以及如何确定它的责任到底有多大）？

因特网和内联网

Web 是解决客户机 / 服务器问题的一种常用方案，所以最好能用相同的技术解决此类问题的一些“子集”，特别是公司内部的传统客户机 / 服务器问题。对于传统的客户机 / 服务器模式，我们面临的问题是拥有多种不同类型的客户计算机，而且很难安装新的客户软件。但通过 Web 浏览器和客户端编程，这两类问题都可得到很好的解决。若一个信息网络局限于一家特定的公司，那么在将 Web 技术应用于它之后，即可称其为“内联网”（Intranet），以示与国际性的“因特网”（Internet）有别。内联网提供了比因特网更大的安全级别，因为可以物理性地控制对公司内部服务器的使用。说到培训，一般只要人们理解了浏览器的常规概念，就可以非常轻松地掌握网页和小程序之间的差异，所以学习新型系统的开销会大幅度减少。

安全问题将我们引入客户端编程领域一个似乎是自动形成的分支。若程序是在因特网上运行，由于无从知晓它会在什么平台上运行，所以编程时要特别留意，防范可能出现的编程错误。需作一些跨平台处理，以及适当的安全防范，比如采用某种脚本语言或者 Java。

但假如在内联网中运行，面临的一些制约因素就会发生变化。全部机器均为 Intel/Windows 平台是件很平常的事情。在内联网中，需要对自己代码的质量负责。而且一旦发现错误，就可以马上改正。除此以外，可能已经有了一些“历史遗留”的代码，并用较传统的客户机 / 服务器方式使用那些代码。但在进行升级时，每次都要物理性地安装一道客户程序。浪费在升级安装上的时间是转移到浏览器的一项重要原因。使用了浏览器后，升级就变得易如反掌，而且整个过程是透明和自动进行的。如果真的是牵涉到这样的一个内联网中，最明智的方法是采用 ActiveX，而非试图采用一种新的语言来改写程序代码。

面临客户端编程问题令人困惑的一系列解决方案时，最好的方案是先做一次投资 / 回报分析。请总结出问题的全部制约因素，以及什么才是最快的方案。由于客户端程序设计仍然要编程，所以无论如何都该针对自己的特定情况采取最好的开发途径。这是准备面对程序开发中一些不可避免的问题时，我们可以作出的最佳姿态。

1.11.3 服务器端编程

我们的整个讨论都忽略了服务器端编程的问题。如果向服务器发出一个请求，会发生什么事情？大多数时候的请求都是很简单的一个“把这个文件发给我”。浏览器随后会按适当的形式解释这个文件：作为 HTML 页、一幅图、一个 Java 小程序、一个脚本程序等等。向服务器发出的较复杂的请求通常涉及到对一个数据库进行操作（事务处理）。其中最常见的就是发出一个数据库检索命令，得到结果后，服务器会把它格式化成 HTML 页，并作为结果传回来（当然，假如客户通过 Java 或者某种脚本语言具有了更高的智能，那么原始数据就能在客户端发送和格式化；这样做速度可以更快，也能减轻服务器的负担）。另外，有时需要在数据库中注册自己的名字（比如加入一个组时），或者向服务器发出一份订单，这就涉及到对那个数据库的修改。这类服务器请求必须通过服务器端的一些代码进行，我们称其为“服务器端的编程”。在传统意义上，服务器端的编程一般是用 Perl 和 CGI 脚本完成的，但更复杂的系统已经出现。其中包括基于 Java 的 Web 服务器，它允许我们用 Java 进行所有服务器端编程，写出的程序就叫作“小服务程序”（Servlet）。正是由于小服务程序和其后期发展产物——JSP——的出现，当今许多开发 Web 站点的公司纷纷投向了 Java 的怀抱，其中最重要的原因之一便是它们免去了以后得对付各种各式不同的浏览器的麻烦（用它们搞开发时，不用关心以后使用的是哪种浏览器）。

1.11.4 一个独立的领域：应用程序

与 Java 有关的大多数争论都是与小程序有关的。Java 实际是一种常规用途的程序设计语言，可解决任何类型的问题，至少理论上如此。而且正如前面指出的，可以用更有效的方

式来解决大多数客户机 / 服务器问题。如果将视线从小程序的身上转开（同时放宽一些限制，比如禁止写盘等），就进入了常规用途的应用程序的广阔领域。这种应用程序可独立运行，毋需浏览器，就象普通的可执行程序那样。在这儿，Java 的特色并不仅仅反应在它的移植能力，也反映在编程本身上。就象贯穿全书都会讲到的那样，Java 提供了很多有用的特性，使我们能在较短的时间里写出比用从前那些语言写的更“健壮”的程序。

但要注意任何东西都不是十全十美的，我们也得为此付出一些代价。其中最明显的是执行速度放慢了（目前针对这个问题已有各种各样的应对之策——特别是 JDK 1.3，它首次引入了所谓的“热点”性能提升技术）。和任何语言一样，Java 本身也存在一些限制，使得它不十分适合解决某些特殊的编程问题。但不管怎样，Java 都是一种正在快速发展的语言。随着每个新版本的发布，它变得越来越可爱，能理解解决的问题也变得越来越多。

1.12 分析和设计

初次迈入“面向对象程序设计”（OOP）的门槛，需要换用一种全新的思路，来思考一个具体的问题。许多新手最开始都会在如何构造一个 OOP 项目上皱起了眉头。但是，只要你知道了和对象有关的方方面面，而且越来越多地接触面向对象的设计风格，便能有较大的把握，弄出一个“好”的设计，令其充分利用 OOP 提供的所有便利。

即便遇到一个相当复杂的编程问题，也能通过“方法”的概念，逐步分解掉它的复杂性。这里所说的“方法”，是指一系列小过程。我们可以一个过程接一个过程地，达成最终的目标。从 OOP 的概念诞生那一天起，许多 OOP 方法都有了自己固定的公式。我们只需“依葫芦画瓢”即可。在这一小节中，我会试着让大家理解当使用一个“方法”时，需要达成什么样的目标。

在决定采用一种方法之前，必须先弄清楚想用这个方法解决什么问题。用 Java 写程序时尤其应该如此。Java 语言本身的设计宗旨便是在你写一个程序时，对其中涉及到的复杂性加以简化（与 C 相比）。换句话说，从前需要用一个复杂方法才能达到的目标，现在用一个较为简单的方法即可实现。在 Java 中，简单的方法已足以应付现实世界中的大多数问题；这些方法能干的“活儿”，比从前那些程序化语言的简单方法所做的事情要多得多！

另一方面，大家也必须认识到：不能对“方法”寄予过高的希望！现在，当我们设计和编写一个程序时，所做的一切事情其实都是一种“方法”。它可能是你自己习惯的某种方法（甚至连你自己都没有意识到自己设计出一种“方法”）。无论如何，“方法”就是你做完一件事情的“过程”，解决一个问题的“思路”，它是一个有效的、最终确实能解决问题的过程。也许只需通过少许调整，便能用 Java 来重现你的整个“思路”。但是，假如你对自己这个方法的效率或者最终的结果感到不满，便可考虑采用一种公式化的方法，或者从这些方法中截取有用的一部分。

在整个开发过程中，要记住的最重要的事情便是：不要迷失！但事实上这种事情很容易发生。大多数方法都设计用来解决最大范围内的问题。当然，也存在一些特别困难的项目，需要作者付出更为艰辛的努力，或者付出更大的代价。但是，大多数项目都是比较“常规”的，所以一般都能作出成功的分析与设计，而且只需用到推荐的一小部分方法[⑦]。但无论多么有限，事前某些形式的分析总是有益的，这可使整个项目的开发更加容易，总比直接了当开始编码好！

[⑦] 其中一个非常典型的例子便是《UML Distilled，第 2 版》，Martin Fowler 著，Addison-Wesley 于 2000 年出版。该书将繁琐的 UML 建模过程简化为一系列非常容易完成的小步骤。

另外，也很容易陷入“分析停顿”的误区。此时，你觉得自己根本无法再前进一步，因为在当前阶段，还无法提前揭示出后续设计中的每一个细节。但请记住，无论事前进行多少分析，但除非真的进入设计阶段，否则系统的某些东西是不会真正暴露出来的。而且，有些问题要到你正式编写代码的时候才会显露出来，有的甚至要在程序完成并运行的时候才会显露。正是由于这个原因，所以在可能的前提下，应当以非常快的速度，完成分析与设计阶段，然后对这个当然很不成熟的系统进行初次测试。

对于这个问题，在此还有必要多说几句。我自己最早从事的程序化语言设计。那时，对一个开发小组来说，最值得钦佩的事情就是他们在进入设计和实施阶段之前，会小心翼翼的“前进”，争取把以后可能碰到的每一个问题、可能遇到的每一种可能性都搞清楚。显然，假如创建的是一个 DBMS（数据库管理系统），那么事先必须透彻理解客户的要求。但是，DBMS 属于那种非常容易建立模型，而且脉络非常清晰的一种系统；在许多这样的程序中，数据库的结构都是固定的。然而，本章讨论的编程问题则属于那种“不定性”（我自造的词）的系统。在这样的系统中，通常没有现成的模式可供利用，而是需要你考察一种甚至更多的“不确定因素”——这些因素通常都是首次出现在你面前，用从前的经验来“套”是不节实际的。此时，必须对它们进行深入考察[8]。假如在设计和实施阶段之前，便试图通盘分析一个“不定性”的问题，便会出现前面提及的“分析停顿”。为什么呢？因为在分析阶段，你没有足够多的信息可供彻底解决这一类的问题。问题要想真正得以解决，必然是在整个系统初步开发完成之后，再从头重新考察。显然，你事先必须作出存在一定风险的决定（真的有“风险”，因为这样的决定是“想”出来的，而不是根据实际情况调整的。当然，高风险也可能存在高收益，因为你采用的是某种“新”思路来“试图”解决问题）。从表面看，这样做似乎有些“鲁莽”，但它实际上会减轻一个“不定性”项目的开发风险，因为在很早的时候，就能知道自己想出来的一个办法是否真正可行。产品开发本身便是一种“风险管理”的过程！

以前，有项目开发背景的人都有这样一个经验：一个系统在不断成熟的过程中，需要不断抛弃老系统。采用 OOP 语言，仍然需要丢弃其中的一部分，但由于代码是封装在“类”里的，所以在第一个开发循环中，必然会生成一些有用的类设计，并摸索出一些有价值的点子，这些东西都不必“扔掉”。所以，针对特定的问题，在完成了第一次开发之后（速度非常快），我们得到的会是一些有价值的信息，利用它可着手进行下一次的分析、设计与实施循环。此外，我们也得到了一系列有用的基本代码。

总之，即使你正在仔细考察一种特殊的解决方案，预见了其中的大量细节，你也为此制订了许多步骤，准备了不少文案，但仍然很难正确判断自己该在何时停止。为此，应该时刻提醒自己注意以下几个问题：

(1) 对象是什么？（怎样将自己的项目分割成一系列单独的组件？）

(2) 它们的接口是什么？（每个对象需要接收什么消息？）

最后，假如实在想不出更多的对象和接口，便可着手写一个程序。出于多方面的原因，你可能还要用到比上述更多的说明及文档，但要求掌握的资料绝不可比这还少。

整个过程可划分为五个阶段，从阶段 0 开始，我们要开始使用某种形式的“结构”。

1.12.1 阶段 0：拟出一个计划

第一步，决定在后面的过程中要采取哪些步骤。这听起来似乎很简单（事实上，我们这儿说的一切都似乎很简单），但很常见的一种情况是：有些人根本没作什么决定，便忙忙慌

[8] 根据我的经验，大致可以采取这样的一个标准：对于一个项目，假如其中不确定的因素不止一个，那么事前甚至根本不用计划这个项目的完成时间，也不用计划它会花费多大的投资，除非正式建立了一个可以正常运作的项目原型。在此之前，你可以有极大的自由，来达成最终的目标。

慌地开始写代码。如果你的计划本来就是“直接写程序”，那样做当然也无可非议（若对自己要解决的问题已有很透彻的理解，便可考虑那样做）。但无论如何，那也算是一个计划，对吗？

在这个阶段，可能要决定一些必要的附加处理结构。但非常不幸，有些程序员写程序时喜欢随心所欲，他们认为“该完成的时候自然会完成”。这样做刚开始可能不会有什么问题，但我觉得假如能在整个过程中设置几个标志，或者“路标”，将更有益于你集中注意力。这恐怕比单纯地为了“完成整个项目”而工作好得多。至少，在达到了一个又一个的目标，经过了一个接一个的路标以后，可对自己的进度有清晰的把握，干劲也会相应地提高，不会产生“路遥漫漫无期”的感觉（项目完成只有一次，但小目标可以设置多个。相应地，大家中途也可以开多次“庆祝会”了）。

从我刚开始学习故事结构起（我想有一天能写本小说出来），就一直坚持这种做法，感觉就象简单地让文字“流”到纸上。在我写与计算机有关的东西时，发现它的结构要比小说简单得多，所以不必考虑太多“计划”问题。但我仍然制订了整个写作的结构，使自己对要写什么做到心中有数。因此，即使你的计划是一开始便写程序，仍然需要经历后续的阶段，并要对一些特定的问题做到心中有数。

任务描述

对你构建的任何程序来说，无论它本身是多么复杂，都有一个最基本的用途；它所服务的领域，便决定了它需要实现的基本用途。假如你能完全抛开用户接口、具体的硬件或系统平台、编码算法以及工作效率的问题，最终便会理解一个程序最核心的思想——简单与直接！好莱坞现在流行的一种：任何一部电影要想成功，都必须“高度浓缩”！我们的程序也是一样，它的基本用途用一、两句话便能把它描述出来。这个概括性的描述便是我们构建一个系统的起点！

“高度浓缩”的任务描述非常重要，因为它是任何一个项目的“主旋律”。最开始，可能还无法准确地对一个任务进行高度概括（等整个项目的脉络完全清晰后，再来修正也不迟），但越早定下来越好！举个例子来说，在一个飞行管制系统中，最开始可以将它的基本用途定义成：“由控制塔程序跟踪飞机。”但假如把你的系统拿到一个非常小的机场中使用，又会出现什么情况？在那里，可能没有控制塔，只有一个人负责控制；甚至连一个人都没有！因此，一个程序模型要想成功，应该尽可能地少反映一些特定的细节。比如上面那个例子，你可以这样来描述问题：“飞机到达，下人下货，机场服务，上人上货，然后起飞”。

1.12.2 阶段 1：要制作什么？

在上一代的程序设计中（即“过程化或程序化设计”），本阶段被称为“建立需求分析和系统规格”。当然，那些操作今天已经不再需要了，或者至少改换了形式。大量令人头痛的文档资料已成为历史。不过，当时的初衷是好的。需求分析的意思是：“建立一系列规则，根据它判断任务什么时候完成，以及客户怎样才能满意”。系统规格则指出：“这里是一些具体的说明，让你知道程序需要做什么（而不是怎样做），才能满足要求”。需求分析实际就是你和客户之间的一份合约（即使客户就在公司内部工作，或者是其他对象及系统）。系统规格是对所面临问题的最高级别的一种揭示，我们依据它判断任务是否完成，以及需要花多长的时间。由于这些都需要取得参与者的一致同意，所以我建议尽可能地简化它们——最好采用列表和基本图表的形式——以节省时间。不过，由于另外一些限制，也可能需要把它们扩充成为更大的文档。但是，假如把最开始的文档弄得小而精简，便可由开发小组的头头来创建它。这样一来，他不仅需要从每个人那里征求意见，同时也获得了组内每名成员的赞同与支持。或许更重要的是，它能焕发出每个人的工作热情，积极投身到最初的开发之中。

要特别注意将重点放在这一阶段的核心问题上，不要纠缠于其他细枝末节。这个核心问题便是：决定系统要具备哪些功能！

对于这个问题，最有价值的工具就是一个名为“使用场景”的集合。通过“使用场景”，我们可知道系统需要提供的一些关键性功能，从而知道自己该使用的一些基本类。具体说来，你需要对下面这一系列问题做到心中有数[9]：

■谁来使用这个系统？

■那些人能对系统进行哪些操作？

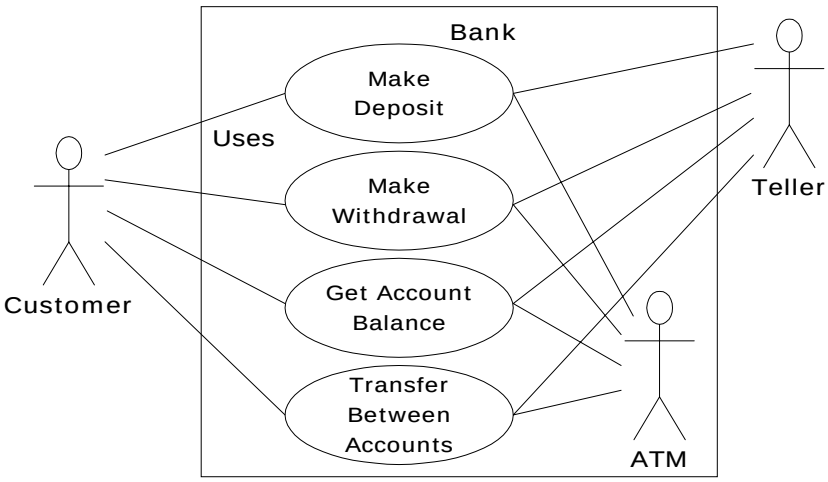
■他们具体怎么操作？

■如果有人在操作一项功能，另一个人想操作同样的功能，怎么办？或者同一个人想操作不同的功能，怎么办？（以揭示出系统的“变性”）

■对系统进行操作时，可能出现什么问题？（以揭示出“违例”）

举个例子来说，我们现在想设计一台自动取款机。该系统的基本用途是：在每种可能的情况下，自动取款机都能采取正确的行动。此时，可将“使用场景”想象成一系列可能的情况，它是所有这些情况的一个“集合”。对于每一种可能的情况，我们则可把它想象为一个简单的问题，都采用这样的格式：“假如...，系统该怎么做？”。例如：“假如客户在前 24 小时内存进了一张支票，但在他取钱的时候，除非把那张支票转帐，否则他的帐户上没有足够多的钱来取。在这种情况下，系统该怎么做？

显然，类似的系统设计细节还有很多。此时，用一幅“使用场景示意图”，可有效避免你在设计的时候误入歧途，疲于和那些细节纠缠。如下所示：

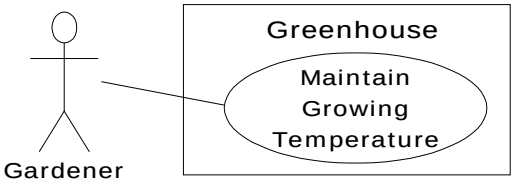


在这张图中，每个人形图标都代表一位“演员”。它通常是一个人或者其他形式的代理（在和自动取款机打交道时，甚至可能是其他计算机系统）。大方框规定了这个系统的“边界”。椭圆代表各种“使用场景”，它们是整个系统能执行的最有价值的操作。在各“演员”与“使用场景”之间，直线代表着两者间的“交互”或者“沟通”。

在这里，不必关心系统具体是如何实现的，只要知道它在用户面前能正常工作就行。

对“使用场景”来说，即便基层系统异常复杂，也不必把对它的说明搞得非常复杂。它唯一的宗旨就是让用户理解系统的功能。例如：

[9] 感谢 James H Jarrett 的帮助。



“使用场景”决定了用户以后可能与系统打的所有“交道”（或者说“交互”），根据它便能制订出系统的使用要求。我们应试着找出以后系统可能会面临的一套完整的“使用场景”。这个工作一旦完成，便能把握住系统的核心设计宗旨。将重点放在“使用场景”上，一个很好的效果就是它们总能让你将精力放在最关键的东西上，防止自己分心于和完成任务关系不大的其他事情上面。也就是说，只要总结出了一套完整的“使用场景”，就可对自己的系统作出清晰描述，并顺利转移到下一个阶段。当然，在你初次尝试时，也可能无法全面掌握系统日后需要应付的各种场合。不过没关系，只要肯花时间，所有问题都会自然而然暴露出来。不要过份在意系统规格的“完美”，否则极易产生挫败感和焦躁不安的情绪。

假如真的被“陷”进去了，这儿也有一个工具可供利用，让你尽快完成这一阶段的工作：用几个文字段落描述出自己的系统，再回过头去找出其中的所有名词和动词。其中，名词可能对应于“操作者”、“使用场景”（比如自动取款机可能摆在一个“大，厅”里）或者要在使用场景下操作的对象；而动词意味着操作者与使用场景的互动，同时规定了在使用场景下的操作步骤。另外，大家还会发现，利用这种“名词和动词”的思考方式，可获得在设计期间需要用到的对象和消息（同时请注意，使用场景描述了子系统之间的互动，因此“名词和动词”方法只能用于在你的脑海中设想一个虚拟的“使用场景”，它并不能产生真正的使用场景）[10]。

根据使用场景和一位操作者（或“演员”）之间存在的界限，也许能指出一个用户接口的存在。但是，它并未定义这样的一个用户接口。为深入理解定义和创建用户接口的问题，请参考由 larry Constantine 和 Lucy Lockwood 合著的《Software for Use》一书，Addison-Wesley Longman 于 1999 年出版，或者直接访问 www.ForUse.com 网站。

尽管仍处在初级阶段，但这时的一些日程安排也可能会非常管用。我们现在对自己要构建的东西应该有了一个较全面的认识，所以可能已感觉出了它大概会花多长的时间来完成。此时要考虑到多方面的因素。假如估算出一个较长的日程，那么和你谈判的公司也许会决定不再让你来做（或许最终会利用他们自己的资源来做——那其实是件好事情）；或者一名主管已经估算出了这个项目要花多长的时间，并会试着影响你的估计。但无论如何，最好一开始就草拟出一份“诚实”的时间表，早点儿作出一些“艰难”的决定，免得到最后对早先的日程安排产生重大干扰。目前有许多技术可帮助我们计算出准确的日程安排（就象那些预测股票市场起落的技术），但通常最好的方法还是依赖自己的经验和直觉（不要忘记，直觉也要建立在经验上）。感觉一下大概需要花多长的时间，然后将这个时间加倍，再加上 10％。你的感觉可能是正确的；“也许”能在那个时间里完成。但“加倍”使那个时间更加充裕，“10％”的时间则用于进行最后的推敲和深化[11]。但同时也要对此向上级主管作出适当的解释，无论对方有什么抱怨和修改，只要明确地告诉他们：这样的一个日程安排，只是我的一个估计！

---

[10] 有关使用场景（Use Case）的问题，更深入的讨论请见由 Schneider 和 Winters 合著的《Applying Use Cases》一书，Addison-Wesley 出版社于 1998 年出版；另见由 Rosenberg 编著的《Use Case Driven Object Modeling with UML》一书，Addison-Wesley 出版社于 1999 年出版。

[11] 我个人后来改变了这一公式。时间加倍，再加 10%，确实能够得到一个合理的、大致准确的时间（假定没有太多的不确定因素）。但是，仍然需要非常努力，才能按时完成任务。如果想让时间非常充裕，并使整个开发过程不显得太急促，正确的结果应该是 3 或 4 倍最早估计出来的时间。

1.12.3 阶段 2：如何构建？

在这一阶段，必须拿出一套设计方案，并解释其中包含的各类对象在外观上是什么样子，以及相互间是如何沟通（互动）的。在构思类和互动方式时，目前一种十分出色的工具是“类－职责－合作”（Class-Responsibility-Collaboration，简称 CRC）卡片。这种工具非常诱人的一点是它极其简单，根本不需要什么高深的技术。做法简单，你先在面前摆 3 行 5 列的空白卡片。每张卡片都代表一个类，你需要在卡片上写下：

(1) 类的名字：这个名字应当抓住该类的本质，使人一看便懂。

(2) 类的职责：说明可用这个类做什么。可以简单地罗列出它的成员函数的名字，概括出该类的功用（因为在一个好的设计中，函数名也应该能说明一切）。

(3) 类的合作：它和其他类如何打交道？“互动”、“交互”是目前一个用得非常“滥”的术语；它的意思是利用其他类，来为自己的一个对象提供服务。在这里，应当指出该类的目标用户是谁。例如，假定我们创建一个名为 Firecracker（鞭炮）的类，那么由谁来观察它，是一名 Chemist（化学家），还是一名普通的 Spectator（路人）？前者想知道它的化学构成，后者则关心它的颜色和爆炸后的式样。

大家也许会觉得这些卡片应该做得更大，以便记下和它们相关的所有信息。但是，切记不要那样做！我们故意做成小卡片的形式，正是为了保证你的类相当小，同时避免你过早接触大量细节。假如在一张小卡片上记不下一个类的所有信息，就表明这个类过于复杂了（要么是由于你草拟了过多的细节，要么应当把它分解为多个类）。一个理想的类应该是什么样子呢？它应当让人看一眼便明白。CRC 卡片的宗旨正是帮助你顺利走完第一个设计过程，让你先抓住整个项目的“纲”，以后再来慢慢修正它的“目”。

CRC 卡片最大的特色之一便是它在通信方面的功劳。最好将开发小组的人都招集到一起，现场制作这些卡片，而且不用计算机。每个人都负责几个类（最开始可以没有名字或其他信息）。换句话说，进行一次现场模拟，同时模拟出多种应用场合，决定应该向各个对象发送哪些消息，从而满足每一种场合的要求。完成这一步操作之后，便会发现自己真正需要的是哪些类，同时还有它们的职责和合作方式。记住一边做，一边填写卡片。完成了对所有应作场合的模拟后，就可对自己的设计有一个最初的总体印象了！

在我开始用 CRC 卡片之前，曾有一次非常成功的项目顾问经历。那一次，我负责对一个开发小组进行指导，他们刚刚启动一个项目的设计。这些人之前都没有做 OOP 项目的经验，所以当时是把各个对象画到白板上。一开始，我便问他们对象之间应该如何通信。根据他们的反馈，删掉了部分对象，再换成其他对象。事实上，我是把所有“CRC 卡片”都画到了白板上。这个小组（他们知道这个项目的目的是什么）实际是提前定出了设计方案。换句话说，他们自己“拥有”了设计方案，而不是让设计方案自然而然地显露出来。我在那里做的事情就是对设计进行指导，提出一些适当的问题，尝试作出一些假设，并从小组成员那里得到反馈，以便修改那些假设。这个过程中最美妙的事情就是整个小组并不是通过学习一些抽象的例子来进行面向对象的设计，而是通过实践一个真正的设计来掌握 OOP 的窍门，而那个设计正是他们当时手上的工作！

手上有了一套 CRC 卡片之后，接下来便可为自己的设计建立一个更正规的说明，此时可考虑采用 UML[12]。当然，并不一定非要用 UML，但它会对你相当有用，特别是在你想把图表挂在墙上，让每个人都考虑哪种方案更好的时候！除 UML 之外，还可考虑制作对象及其接口的文字化说明；更进一步，取决于你选用的程序语言，甚至可以直接“拿代码

[12] 对于新手，我向你推荐前面提到的《UML Distilled，第 2 版》。

说话”[13]。

UML 还提供了一套额外的图表式记号法，用于对系统的动态模型进行描述。假如一个系统或子系统的状态转变非常关键，迫切需要拥有自己的示意图（比如在一个控制系统中），那么这一套记号法便会显得相当有用。另外，我们有时需要对数据结构进行描述，因为在某些系统或子系统中，数据才是其最具制约性的一项因素（比如在一个数据库中）。

描述好对象及其接口后，阶段 2 的工作便算完成了——起码大多数工作已经完成，因为事实上，通常还会在这一阶段犯下一些小错误，要等到阶段 3 才能真正发现。但这已经足够了！到此为止，我们唯一需要关心的就是发现了自己需要的所有对象。当然，它们越早发现越好，但由于 OOP 提供了足够的支持结构，所以即便晚一点儿发现，也没有什么大不了的问题。事实上，5 个阶段中任何一个阶段（乃至程序开发的全过程）都可能有新的对象出现。

1. 对象设计 5 步走

一个对象的设计生命并不局限于你写程序的时候。相反，对象的设计通常要跨越几个阶段，而且你需要不断地考察它做的事情，以及它看起来的样子，然后相应地作出修改。这一观念应当牢牢树立，千万不要吊以轻心，否则极易出现对象运用失当的情况！设计不同类型的程序时，也要作同样的考虑；只有不断地尝试一个问题的解决方案，才有可能最终拿出最理想的程序“设计范式”（参见《Thinking in Patterns with Java》一书，可从 www.BruceEckel.com 下载）。对象也有自己的“范式”，但只有经过理解、使用以及复用的循环往复，它才能日趋成熟。

(1) 发现对象。这一阶段发生于首次对程序进行分析的时候。要想发现自己需要的对象，需要考虑大量外部因素和一些既定的限制。另外，还要考虑系统内重复的元素，以及你最小的概念化单元是什么。如果你手上已经有了一系列类库，那么某些对象是显而易见的。假如类和类之间出现了某些共通之处，便暗示着你可能需要为它们制作一个基类，同时考虑马上进行继承，或在设计过程的后期再来考虑继承。

(2) 重组对象。随着一个对象的构建，通常会发现自己需要用到一些新成员，而那些成员在“发现”阶段是没有显现出来的。对象的内部需要可能要求其他类来提供对它的支持。

(3) 系统构建。同样地，对象需要承担的更多责任要等这个较晚的阶段才显露出来。换句话说，通过你的不断调查研究，你的对象发生了“进化”，开始趋于成熟。由于需要同系统内的其他对象进行通信和连接，所以可能会改变你的类的需求，或者要求你干脆做一些新类。例如，此时可能会发现自己需要使用某些工具或助手类（比如链接列表），这些类包含了极少或根本没有状态信息，所以可以非常方便地为其他类函数提供帮助。

(4) 系统扩展。随着系统新功能的添加，最终可能发现以前的设计不大容易扩展。掌握了这方面的新资料后，你便可以重新构建系统，同时可能要添加一些新类，或者设计一些新的类结构。

(5) 对象复用。现在是对类进行实际检测的时候了。如果有人试图在一个全新的环境中复用它，便可能发现它存在的一些缺陷。随着我们修改一个类，使其适应一些更新的应用环境，类的基本思路会变得愈加清晰——直到你成功获得一个可以完美“复用”的类型为止。不过，也别指望一个系统吉的大多数对象都能“复用”。如果这些类与你的系统规格完美地相符，那么也是完全能够令人接受的。可复用的类型实际并不常用，因为它们必须解决一些更“通用”的问题，否则“复用”它也没有任何意义。

[13] Python（www.Python.org）通常作为“可执行伪代码”使用。

2. 对象开发准则

开发你的类时，请遵循以下准则：

(1) 让特定的问题来产生一个类，然后在其他问题的方案中，让类不断得以扩充和成熟。

(2) 请记住，发现你需要的类（及其接口）是系统设计时最主要的工作。只有真正得到了那些类，你的项目以后才会变得轻松。

(3) 最开始不必强求知道一切东西——在学习中进步！

(4) 尽早开始编程；让一些东西工作起来，证明你的设计是否可行。不要担心最后得到的是一系列“拿不出手”的代码——条理清晰的类会帮你区分不同的问题，并可有效控制混乱局面。

(5) 尽量简单！小而清楚的对象及功能单纯的工具要明显优于大而复杂的接口。在你面临众多选择的时候，请采用由 Occam 提出的“剃刀”方法：列出所有选择，然后选一个最简单的——因为简单的类几乎肯定是最好的。先从小和简单开始，在自己掌握了更多东西之后，再根据需要决定是否对类接口进行扩展。时间过得越久，越难将元素从一个类里剔去，所以刚开始还是简单一点儿好！

1.12.4 阶段 3：构建核心

首次从草案设计转移到编译及执行可供测试的代码主体时，便进入了这一阶段。在这一阶段，将证明我们当初设计的结构是否可行。在这儿的工作并不是做完一遍便算完——就象大家阶段 4 看到的那样，在你正式构建系统的时候，还会多次回到这里，不断地实验自己的设计。

在此，我们的目标是找出系统结构中需要实现的核心，以便在它的基础上生成一个可以实际运行的系统——无论那个系统在这个最初的阶段是多么的不完整。我们需要创建一个框架，以便据此进行其他延伸性的工作。还要这里第一次系统集成和测试工作（以后还会多次进行），并向雇自己做这个系统的“老板”汇报阶段性成绩，指出系统最终看起来是什么样子的，以及它以后还会怎样发展。理想情况下，这个时候还可以揭示出一些不利于未来发展的隐患。另外还要总结出一些可拿来对原始结构进行改进的手段——假如不真正实现一下系统，这些资料是无从拿到的。

系统构建的一部分工作是把它放到一个现实的环境中进行测试，然后参考自己的要求和系统规范进行进行分析。请测试你的系统是否能满足使用场景的要求。假如系统的核心是稳定可靠的，就可以继续开发过程，在它的基础上放心地添加其他功能。

1.12.5 阶段 4：迭代使用场景

核心框架运行起来之后，在它上面添加的每项功能都应该只是一个独立的小项目。我们经过一次“迭代”（Iteration），便完成了一系列特性的添加。每一次“迭代”都应当是一个比较短的开发周期。

那么，一次“迭代”到底要花多长时间呢？在理想情况下，每一次“迭代”都持续一至三周的时间（取决于用来实现的语言到底是什么）。每一次迭代结束后，我们都得到了一个集成的、经过测试的系统，而且又新添了不少功能。但这里特别令我们感兴趣的是，一次“迭代”的基础就是一种使用场景。每种使用场景都意味着要用到一系列相关的功能，而我们在一次迭代期间，把所有这些功能都一鼓作气地集成到系统中去。这样不仅能更好地理解一种使用场景的“作用范围”或“作用域”，而且心中也能对该使用场景更加有数。这是由于概念并不会在分析与设计之后便抛在脑后；相反，它是整个软件构建过程的一种基本开发单元。

实现了要求的功能，或者客户对当前的版本感到满意之后，便可停止这种“迭代”了（记住，软件开发是“客户说了算”）。由于这个过程是“迭代”进行的，所以有许多机会都可以

开始发布自己的产品，而不是非要制定一个固定的“目标”。那些“开放源码”的软件开发则完全就是一个不断“迭代”、频繁获得用户反馈的环境，这正是它们取得巨大的重要原因。

出于对许多原因的考虑，一个迭代式的开发过程具有相当大的价值。我们可以提早发现故障隐患，客户有更多的机会改变自己的主意，程序员会对自己的设计越来越满意，而且项目的设计水准也会越来越高。而且还有另一个好处，那便是可向客户提供充足的反馈信息，使其对整个项目有一个清晰的把握。员工和雇主之间的交流，也会变得更加令人愉快，不至于出现沟通渠道不畅的情况。

1.12.6 阶段 5：校订

事实上，整个开发周期还没有结束，现在进入的是传统意义上称为“维护”的一个阶段。“维护”是一个比较暧昧的称呼，可用它表示从“保持它按设想的轨道运行”、“加入客户从前忘了声明的功能”或者更传统的“除掉暴露出来的一切臭虫”等等意思。所以大家对“维护”这个词产生了许多误解，有的人认为：凡是需要“维护”的东西，必定不是好的，或者是有缺陷的！因为这个词说明你实际构建的是一个非常“原始”的程序，以后需要频繁地作出改动、添加新的代码或者防止它的落后、退化等。因此，我们需要用一个更合理的词语来称呼以后需要继续的工作。

这个词便是“校订”[14]。换言之，“你第一次做的东西并不完善，所以需为自己留下一个深入学习、认知的空间，再回过头去作一些改变”。对于要解决的问题，随着对它的学习和了解愈加深入，可能需要作出大量改动。进行这些工作的一个动力是随着不断的改革优化，终于能够从自己的努力中得到回报，无论这需要经历一个较短还是较长的时期。只有不断地校订，你的程序才能“好”变成“出色”。在这个过程中，最开始不甚明了的概念会变得越来越清晰。另外，你的类最终也能摆脱“只能在一个项目中使用”的尴尬境地，脱胎换骨成一种可供复用的资源。

什么时候才叫“达到理想的状态”呢？这并不仅仅意味着程序必须按要求的那样工作，并能适应各种指定的“使用条件”，它也意味着代码的内部结构应当尽善尽美。至少，我们应能感觉出整个结构都能良好地协调运作。没有笨拙的语法，没有臃肿的对象，也没有一些华而不实的东西。除此以外，必须保证程序结构有很强的生命力。由于多方面的原因，以后对程序的改动是必不可少。但必须确定改动能够方便和清楚地进行。这里没有花巧可言。不仅需要理解自己构建的是什么，也要理解程序如何不断地进化。幸运的是，面向对象的程序设计语言特别适合进行这类连续作出的修改——由对象建立起来的边界可有效保证结构的整体性，并能防范对无关对象进行的无谓干扰、破坏。也可以对自己的程序作一些看似激烈的大变动，同时不会破坏程序的整体性，不会波及到其他代码。事实上，对“校订”的支持是 OOP 非常重要的一个特点。

通过校订，可创建出至少接近自己设想的东西。然后从整体上观察自己的作品，把它与自己的要求比较，看看还短缺什么。然后就可以从容地回过头去，对程序中不恰当的部分进行重新设计和重新实现[15]。在最终得到一套恰当的方案之前，可能需要解决一些不能回避的问题，或者至少解决问题的一个方面。而且一般要多“校订”几次才行（对“设计范式”进

---

[14] 在 Martin Fowler 的《Refactoring:improving the design of existing code》一书中（Addison-Wesley 于 1999 年出版），对“校订”进行了探讨，该书完全采用 Java 例子。

[15] 这有点儿类似于“快速强制转型”。此时应着眼于建立一个简单、明了的版本，使自己能对系统有个清楚的把握。再把这个原型扔掉，并正式地构建一个。快速强制转型最麻烦的一种情况就是人们不将原型扔掉，而是直接在它的基础上建造。如果再加上程序化设计中“结构”的缺乏，就会导致一个混乱的系统，致使维护成本增加。

行一下研究在此显得颇有必要。请参考我的《Thinking in Patterns with Java》一书，它可从 www.BruceEckel.com 免费下载）。

构建一套系统时，你也几乎肯定需要对其进行“校订”。我们需要不断地对比自己的需求，了解系统是否自己实际所需要的。有时只有实际看到系统，才能意识到自己需要解决另一个不同的问题。若认为这种形式的校订必然会发生，那么最好尽快拿出自己的第一个版本，检验它是否真是自己希望的，使自己的思想不断趋于成熟。

在此，要记住的或许最重要的一件事情是：在默认情况下，假如你修改了一个类，它的超类和子类仍然在起作用。你完全不必害怕要进行的修改（特别是假如你已经有了一系列单元测试，可验证修改的正确性）。修改并不一定会使程序崩溃，结果造成的任何改变都仅仅限制在子类和（或）一些特定的、有联系的类的范围之内。

1.12.7 计划的回报

假如没有仔细拟定的设计图，当然不可能建起一所房子。即使建造的是一所狗舍，尽管设计图可以不必那么详尽，但仍然需要画一些草图，以便做到心中有数。软件开发在这方面则更是严厉而且苛刻——它的“设计图”（计划）必须详尽而完备。在很长的一段时间里，人们在他们的开发过程中并没有太多的结构，但那些大型项目很容易就会遭致失败。通过不断的摸索，人们掌握了数量众多的结构和详细资料。但它们的使用却使人提心吊胆——似乎需要把自己的大多数时间都花在编写文档上，而没有多少时间来进行实际的编程（经常如此）。我希望这里为大家讲述的一切能提供一条折衷的道路。需要采取一种最适合自己需要（以及习惯）的方法。不管制订出的计划有多么小，但与完全没有计划相比，前者总能在一定程度上改善你的项目设计。请记住：根据统计，事前没有计划项目 50％以上都会失败（有人则认为超过 70％！）。

通过按计划行事（简明扼要的计划最好），然后在正式编码之前拿出设计结构，便会发现所有事情都显得更有条理——比起一开始便埋头编程要轻松得多。由于一切似乎都在自己的掌握之一，所以你甚至能体验到一种“成就感”。就我自己的经验来看，假如最后“弄”出了一套非常出色的方案，那么从中体验到的“成就感”简直可以用“无以仑比”来形容。给我的感觉，编程更象一门艺术，而非仅仅是一种“技术活”。你的所有付出最终都会得到回报。作为真正的程序员，这并非可有可无的一种素质。全面的思考、周密的准备、良好的构造不仅使程序更易构建与调试，也使其更易理解和维护，而那正是一套软件赢利的必要条件。

1.13 极度编程

自我读研究生的时候开始，便开始学习分析与设计技术。在我见过的所有技术中，“极度编程”（Extreme Programming，XP）无疑是最“疯狂”、最“激进”的一种。而且更重要的是，它太令人兴奋，令人倍受鼓舞！《Extreme Programming Explained》是一本专门讲 XP 编程的好书，由 Kent Beck 编著，Addison-Wesley 于 2000 年出版。XP 的官方网站是 www.xprogramming.com。

XP 既是程序编写的一种基本方法论，也是一系列如何去实际做它的准则。有些准则也反映在其他方法学理论中。在它的所有准则中，我最欣赏的是两条：“先写测试代码”以及“合伙编程”。在这里，尽管我并不是说整个“极度编程”的理论都应当照搬，但至少，假如你遵守了上述两条 XP 的规则，那么往往都能显著改善你的编程效率，并增大其可靠性。

1.13.1 先写测试代码

按传统方法，“测试”通常要摆在一个项目的最后进行，在“让所有东西都正常运转”之后，最后只是检测和证实一下。也就是说，传统的“测试”并不具有较高的优先级。坚持这种做法的人认为，既然很多情况都还不明朗，为什么要急着去测试呢？但是，这样的人并不是真正的程序员，只是“保守”的编程者而已！传统的“测试小组”也往往有同样的看法，他们给人的感觉就是“幸灾乐祸”的一群。一旦发现哪里不对，便高兴得哈哈笑：“你瞧，我又挑出了一点毛病！”

XP 则完全革命了传统的“测试”理念，它为其赋予和编写程序代码同样的优先等级。事实上，你需要在正式编写程序之前先写出测试代码，而且这种测试会长伴编程左右。换句话说，你得不断地写，同时不断地测！每做一次项目集成时（亦即在项目中加入新的一套功能，每套功能与一种“使用场景”相对应。这样的事情会经常发生，有时一天甚至会进行几次项目集成），都应保证测试的成功。

先写测试代码有两个方法非常重要的优点。

首先，它强制我们对类的接口进行清晰的定义。设计系统时，我通常都会建议人们“想象能够解决特定问题的一个最完美的类是啥样”。XP 测试策略甚至比这还更进一步——它要求事先规定好一个类“必须”是啥样；以及这个类“必须”具有什么行为。从此便再也没有不确定的因素。尽管你可以写下长篇大论，制作许多示意图，对一个类的行为进行非常精确的说明——但所有这些都比不上一系列直接了当的测试！前者只是一个“构思”，后者才真正变为“现实”。这个“现实”是通过编译和运行程序而缔结出来的。除非进行测试，否则很难更具体地描述一个类！

在创建测试的过程中，我们也被迫彻底思考这个类，而且通常都能发现用 UML 图表、CRC 卡片等等“构思”时拉下的一些东西。

其次，先进行测试的重要性还表现在：每次测试时，都相当于是在构建你的软件。这一活动为我们赋予了由编译器执行的另一半测试。如果从这个角度去考察程序语言的进步，便会发现技术真正的进步实际上都是围绕“测试”展开的。汇编语言只能检查语法，但 C 可以进行一些“语义”上的限制，这些都可防范你犯下特定类型的错误。OOP 语言则引入了更严格的“语义”限制。不过你仔细想一想，便知道这种限制实际就是一个测试的过程。“这种数据类型的用法正确吗？”以及“对这个函数的调用对吗？”当编译器或运行时间系统在“思考”这些问题的时候，它们实际上是在对你的代码进行测试。事实上，我们已在语言内部看到了进行这种测试的效果：人们可以编写越来越复杂的系统，并让它们工作起来，同时需要花费的时间和精力越来越少。以前，我并不知道为什么会出现这样的进步。不过现在知道了，那是测试的功劳：假如你做错了一件事情，由语言内部各种测试构成的一张“安全网”就会捕捉那个错误，并即时向你指出。

但语言本身提供的测试机能总归是有限的。在适当的时候，我们必须自己动手，补充剩下的测试，营造一个完整的运行环境（同编译器和运行时间系统配合），从而对程序的所有方面进行检测。而且，既然反正都要测试，为什么不早一点儿进行呢？这正是我们要“先写测试代码”，在每次构建系统的时候都自动运行它们的原因。我们的测试会与语言本身提供的测试机能相得益彰，共同确保你创建出一个安全可靠的应用程序。

事实上，程序语言的功能越强大，我就越能放手在它上面做各种各样的实验。因为我现在已经知道了一点，那便是程序越强大、越优秀，它所编织的“保护网”就越严密。最起码，它可有效地防止我花上许多时间去追踪程序里的 Bug。XP 的测试方案对你的整个项目而言具有同样的作用。由于你知道自己的测试肯定会抓住一些漏洞，所以可以尽早作出较大的改动，而不是等到项目快要结束的时候，才想起来应该做那样的改动——那时恐怕已经来不及

了。

1.13.2 合伙编程

程序员给人的印象总是很“另类”——衣冠不整，独来独往，充满个人英雄主义的浪漫色彩（尽管说句老实话，写程序一点儿不“浪漫”，而且还很枯燥）。喜欢看好莱坞影片的人都知道，英雄们总喜欢自行其事，而且最终往往都能斩妖除魔，取得圆满结局[16]。程序同与此也差不多雷同，就象 Larry Constantine 常说的“牛仔程序员”（Cowboy coders）。但 XP 提出了完全与此相反的看法，它认为“个人主义要不得！”和传统做法不同，按照 XP 的理论，在每个工作站上，代码应该由两个合伙来写。为此还需要专门拿一块地，摆上一系列工作站，而且不要那种用来分割各自地段的“隔断”——换句话说，整个开发队伍需要一个真正“开放式”的环境。事实上，按照 XP 的说法，假如你真的想依着 XP 行事，那么要做的第一件事情就是下次上班的时候记得带上一把起子，走进办公室后，把所有碍事的东西都给拆了[17]——当然，你的头儿可能马上便要去安抚设备管理处的那些家伙了。

合伙编程的好处体现在一个人负责实际编程，另一个负责“思考”。在伟大的“思想者”的头脑中，应该对工作有一个总体性的把握——不仅要针对要解决的问题，还得遵循 XP 的规则。假如两个人在一起工作，便极少出现一个人走开，并说“我不想先写这个测试”的情况。假如写程序的那个人不成了，他们两个可以交换一下位置。假如两个人都不成了，他们的“沉思状”会被工作区的其他人看到，并会乐意助一臂之力。借助两个人的力量，做起事情来会更显流畅，而且更有条理。不过或许更重要的是，编程也变成了一种“社交”，有助于为你带来一定的乐趣（孤独的程序员常无乐趣可言，更不用说什么社交了）。

在我办的学习班上，做练习时便常让学生们采用合伙编程的办法，效果令人非常满意。

1.14 Java 为何取得成功

Java 之所以取得了如此大的成功，原因在于它的目标是解决当今开发者面临着的诸多问题。Java 最本质的目标便是提高编程效率。而效率的提高又通过几种途径来实现。语言本身为你提供了尽可能多的帮助，同时避免用一大堆条条框框来限制你的创作，不会规定你必须用这个、必须用那个等等。Java 是一种实用的语言，它为程序员提供了尽可能多的便利来解决手上的实际问题。

1.14.1 系统更易表达和理解

类用于更好地描述你的问题。也就是说，在你写代码的时候，相当于用问题空间的术语来描述自己的方案（“把铅笔放到铅笔盒里”），而不是用计算机的术语来描述，后者属于方案空间（“设置芯片里的一个二进制位，关闭数据中转”）。即便更高级别的概念，也往往能用单独一行代码表示出来。

易于表达之后，紧接而来的便是更易维护。根据统计（如果可靠的放），在一个程序的生命周期里，其实大多数时间都花在了对它的维护上。假如程序更易理解，那么就必然更易维护。这样同时也能减少由于维护和编制相关文档而带来的开销。

---

[16] 尽管这个例子有点儿“美国化”，但好莱坞的作品其实已受到了全球各地的认同，并非单纯的“美国货”了。

[17] 甚至还要拔掉电话头。我有段时间在一家公司上班，那里有个莫名其妙的规定：外头打给每位主管的每个电话都要广播出来。好一个“广播找人”！想想都觉得恐怖，每个电话呀！所以我们的工作经常都被打断。最后，乘人不备，我掏出一把剪子，噌噌地便把喇叭线给废了。

1.14.2 库设计更显均衡

创建程序最快的办法便是使用已经写好的代码：一个现成的库！Java 的主要目标之一便是让库的使用变得更加轻松。这是通过将库强制转型成一种新的数据类型（类）来实现的。只要你带入了一个库，便相当于为语言赋予了一种新的类型。由于 Java 编译器会自己照管好库的使用——它可保证正确的初始化和清除，保证其中的函数得到正确调用等等——所以我们可将重点放在想用库做什么上面，而不是去关心必须用它来做什么。

1.14.3 错误控制

C 的错误控制存在着许多问题，而且极易出现“漏网之鱼”——在这个时候，你只有祈祷。在构建一个大型、复杂的程序时，再也没有什么比忙得满头大汗也找不出一个莫名其妙的错误而更令人着恼了。Java 的“违例控制”则可保证发现一个错误。

1.14.4 自由扩展

许多传统语言对程序的大小和复杂程度进行了限制。例如，BASIC 在解决特定类型的问题时或许显得很方便，但只要程序变得超出几页的长度，或者脱离了语言本身面向的问题领域，那么再用它写程序就完全是在“自讨苦吃”了。不过，其实并没有一个条明显的界限告诉你一种语言最多只能支持到多大的限度；即使有，你会把它忽略。例如，你不会说：“我的 BASIC 程序太大了；得赶快用 C 来重写！”相反，你会试着再在其中添加几个新行，从而增加又一种新特性。因此，不知不觉地，它给你带来的负担越来越大，最后不得不深陷其中，难以自拔。

Java 则是一种能够“自由扩展”或者“自由伸缩”的语言，它消除了小程序和大程序的界限，你现在根本不需要关心这方面的问题。管它多大呢，用 Java 写就是了！当然，如果仅仅是为了写一个“Hello World”这样的小程序，那么根本不必用到 OOP 技术。不过，只要你真的想用它写，那么提供支持的特性也是在那里摆着的。程序无论大小，Java 都能支持，而且会“一视同仁”地加以对待。

1.15 迁移时的策略

决定选择 OOP 之后，你接下来的一个问题可能便是：“怎样带动我的主管/同事/部门/同级也开始用对象呢？”在这个时候，请你——一位独立的程序员——想一想以前刚开始转移到一种新的语言和一种新的编程模式时，是一种什么样的状况。那个时候，你首先会去参加培训，并接触一些项目实例；然后回来做一个试验性的项目，目的只是给自己练练手，不用牵涉到太多的复杂性；然后接手做一个真正的项目，让它做一些真正有用的事情。在你的初女项目期间，你通过查阅参考资料、询问专家以及同伙伴交流，实际上仍然在不断地进行学习。这同样是在你迁移到的时候，许多有经验的程序员会给你的建议。当然，要想将整个公司都迁移过来，必然牵涉到团体的协作。但针对其中的每个人来说，所经历的过程与上面说的一般无异。

1.15.1 指南

下面提供一些指南，便于你迁移至 OOP 及 Java 时作为参考：

1. 培训

第一步是接受某种形式的教育。请记住公司在代码上的投入，不要试图一下子便培训个

近一年的时间，那样会打乱现在的工作次序。选一组人，最好挑那些精明强干、有强烈兴趣的骨干，让他们去接受培训。

另一个办法是整个公司来个一次性培训。其中包括让策略主管去上综述性的课程，让负责系统构建的去上设计和编程课。对于规模较小的公司来说，这种做法值得考虑。只要操作得当，整个公司都可以一帆风顺地完成过渡。对于较大的公司来说，也至少让一个部门的人去参加培训。不过，由于代价不菲，所以有的公司还是选择先从搞项目的人开始培训，做一个试验性的项目（可能要由外部的专业人员指导）。最后，等那些人搞精通之后，让他们当公司里其他人的老师。

2. 低风险项目

首先试着做一个低风险的项目，不要怕犯错误。有了一定的经验之后，要么可把这个项目的参与者分派到其他项目中，要么抽调他们作为一个 OOP 技术支持部门的员工。第一个项目最初可能没法子真正运行起来，所以千万不要拿本公司比较关键的项目来“开刀”。这应该是一个简单的、明确的以及便于我们思考问题的项目。也就是说，它应当涉及到类的创建。这样一来，轮到公司里的其他程序员学习 Java 时，便有一个现成的东西可供参考利用。

3. 基于成功模型

先试着找一些好的面向对象设计的例子。实在找不到，再从头重新设计不迟。尽量找那些有着良好移植性，而且你的问题已有别人帮你解决好了的例子。假如尚未完全解决你的问题，便可试着利用自己学到的有关“抽象”的知识，把它应用于对原有设计的修改中去，以满足自己的需要。这是“设计范式”的一个常规性概念（参见《Thinking in Patterns with Java》一书，可从 www.BruceEckel.com 下载）。

4. 使用现成类库

当初之所以考虑迁移到 OOP，很大一部分原因就是可以节省自己的投资，易于换用类库的形式（特别是 Java 标准库，本书一直都在探讨它的问题），使用原来那些旧代码。假如能完全依赖现成的库创建及使用对象，那么应用程序的开发周期往往是相当短的。不过，有些刚入门的程序员可能还不理解这个，他们并不关心现成类库；或者被这种语言的魅力所“迷住”，一心一意地要由自己来写已经有人写过的类。在向 Java 的迁移过程中，能够越早发现并利用其他人的代码，便能越早从中受益。

5. 不要用 Java 改写原有代码

通常，我们并不将大量时间花在利用原来的、功能性的代码上，并试图用 Java 对其进行改写（如果必须把它转变成对象，那么可利用附录 B 介绍的 Java 固有接口，建立同 C 或 C++代码的接口）。你那样做可能会有一些效果（特别是在代码的“复用”能力并不高明的前提下），但在通常情况下，对于自己的头几个项目，通常都看不到什么显著的性能/效率提升——除非是一个全新的项目。只有一步一步地，亲自将一个项目从最开始的概念一直带到现实，才会最充分地体验到 Java 和 OOP 的优势。相反，用它来改写原来的代码，则通常不会有太大的意义！

1.15.2 主管的问题

假如你不幸是一位主管，那么日常的工作应该是为你的开发队伍准备足够的资源，推掉队伍前进道路上

的障碍，以及试着营造一个最富有效率、最令人愉悦的工作环境，使你的“手下”能按

时完工，而且不至于有什么“怨言”。在全体向 Java 迁移的过程中，上述三种工作都显得异常重要。假如你不至于为此而损失什么，那么就是最理想的结果。决定让一组 C 程序员全部改用某种 OOP 语言工作的时候，尽管向 Java 的迁移可能比向其他 OOP 语言的迁移简单一些，而且代价也可能运动会 低一些，但整个过程并不是靠取巧就能轻松完成的。在这个过程中，作为一名主管，你还是应该注意一些自身的问题。

1. 启动时的投入

向 Java 环境迁移时，涉及到的启动投入并非仅仅是购买 Java 编译工具所花的钱（Sun 的 Java 编译器是免费的，所以这也许不会成为障碍）。假如在培训多投资一点，那么你以后的中、长期投资就会减少（在你设计第一个项目时，或许就会多少体验到这一点）。另外，假如事先经过了仔细的考察，买回来的正是能解决自己问题的类库，而不是试着自己去构建那些类库，那么也能节省一定的费用。作计划时，这些硬性投资必须考虑在内。除此以外，还一些软性投资，比如由于要学习一种新语言（甚至一种新的编程环境），所以会耽误一定的工作。当然，通过培训可将这方面的损失减少到最低速度，但队伍中的成员必须经过自己的努力，才能理解新的技术。在这个过程中，他们也许会犯下许多错误（不过是正常的，从错误中学习），从而进一步地降低自己的工作效率。不过，这样做是值得的，因为我们的最终目标是让自己的工作更有效率。只要有合适的类、正确的开发环境，那么随着 Java 学习的深入，可能会感觉到工作越来越有效率（即使在最开始的时候，每天写的代码行越来越少，犯的错误却越来越多）。即使有这样或那样的“痛苦”，我们也再也不想重新顽固地坚守 C 阵地——因为长痛不如短痛！

2. 性能问题

一个常见的问题是：“OOP 会自动使我的程序变得更大和更慢吗？”回答是：“要视情况而定。”和 C++那样的语言相比，为了获得 Java 更高级的安全特性，也可能要付出性能退化的代价。象“热区”（Hotspot）这样的技术以及编译技术在大多数情况都能显著提高速度，而且实现更高性能努力从来便没有放弃过。

如果将自己的重点放在“快速强制转型”上，那么最开始可以一鼓作气地准备好所有组件，不必关心它们的任何效率问题。假如使用的是由其他厂家开发的库，那么它们事先通常已由厂家优化好了；只要你的宗旨是进行快速开发，那么不管在哪种情况下，效率都不是一个重要的问题。有了一个自己中意的系统原型后，假如它不仅“个头小”，而且“速度快”，那么自己的目的就算达到了。如果不是，那么可以调用一个分析工具，首先通过改写小部分代码，看看是否会产生速度的加快。如果真的有用，那么接着对底层实现进行修改，保证没有代码会用到一个马上便要修改的类。到此为止，假如你的问题还无法解决，便得考虑改变设计了。实际上，在当初计划你的设计时，便应想到一个大致的性能水准。通过快速开发，便可提早检验设计是否符合性能要求。

假如发现一个函数成为性能瓶颈，可考虑通过 Java 的“固有方法”（在附录 B 介绍），用 C/C++来改写它。

3. 常见的设计错误

迁移到 OOP 和 Java 环境时，程序员通常会经历一系列常见的设计错误。这是由于在早期项目的设计与实现时，缺乏从专业人士那儿的反馈而造成的。另外，这也可能是由于你没有打算一开始聘请相关领域内的专家。在没有“主心骨”的情况下，几乎必要会出现这样或那样的问题。有时候，一些人可能会自以为理解了 OOP 的概念。一些在专业人士眼中的小问题到一群新手中便变成地一个大问题，他们也许会花上几个小时的时间去论证——而事实

上，只要有个人稍微点拨一下就可以了。为避免这样的问题，最好的办法是从外面聘请一个人，负责培训和咨询——而不一定非让公司内部的程序员通过自行“摸索”也变成专家，那时可能有点儿晚了。

1.16 Java 还是 C++？

Java 特别象 C++；由此很自然地会得出一个结论：C++似乎会被 Java 取代。但我对这个逻辑存有一些疑问。无论如何，C++仍有一些特性是 Java 没有的。而且尽管已有大量保证，声称 Java 有一天会达到或超过 C++的速度。但这个突破迄今仍未实现（尽管 Java 的速度确实在稳步提高，但仍未达到 C++的速度）。此外，许多领域都存在为数众多的 C++爱好者，所以我并不认为那种语言很快就会被另一种语言替代（爱好者的力量是容忽视的。比如在我主持的一次“中 / 高级 Java 研讨会”上，Allen Holub 声称两种最常用的语言是 Rexx 和 COBOL）。

我感觉 Java 强大之处反映在与 C++稍有不同的领域。C++是一种绝对不会试图迎合某个模子的语言。特别是它的形式可以变化多端，以解决不同类型的问题。这主要反映在象 Microsoft Visual C++和 Borland C++ Builder（我最喜欢这个）那样的工具身上。它们将库、组件模型以及代码生成工具等合成到一起，以开发视窗化的末端用户应用（用于 Microsoft Windows 操作系统）。但在另一方面，Windows 开发人员最常用的是什么呢？是微软的 Visual Basic（VB）。当然，我们在这儿暂且不提 VB 的语法极易使人迷惑的事实——即使一个只有几页长度的程序，产生的代码也十分难于管理。从语言设计的角度看，尽管 VB 是那样成功和流行，但仍然存在不少的缺点。最好能够同时拥有 VB 那样的强大功能和易用性，同时不要产生难于管理的代码。而这正是 Java 最吸引人的地方：作为“下一代的 VB”。无论你听到这种主张后有什么感觉，请无论如何都仔细想一想：人们对 Java 做了大量的工作，使它能方便程序员解决应用级问题（如连网和跨平台 UI 等），所以它在本质上允许人们创建非常大型和灵活的代码主体。同时，考虑到 Java 还拥有我迄今为止尚未在其他任何一种语言里见到的最“健壮”的类型检查及错误控制系统，所以 Java 确实能大大提高我们的编程效率。这一点是勿庸置疑的！

但对于自己某个特定的项目，真的可以不假思索地将 C++换成 Java 吗？除了 Web 小程序，还有两个问题需要考虑。首先，假如要使用大量现有的库（这样肯定可以提高不少的效率），或者已经有了一个坚实的 C 或 C++代码库，那么换成 Java 后，反映会阻碍开发进度，而不是加快它的速度。

但若想从头开始构建自己的所有代码，那么 Java 简单易用的特点就能有效地帮助你缩短开发时间——按照某些人的说法（我亲自和一些刚刚迁移到 Java 的原 C++开发队伍进行过交流），速度可以比用 C++快上两倍！假如 Java 的性能并非主要因素，或者你有办法弥补这个问题，那么仅仅由于速度，便使我们很难选 C++而弃 Java。

最大的问题是性能。在原始的 Java 解释器中，解释过的 Java 会比 C 慢上 20 到 50 倍。尽管经过长时间的发展，这个速度有一定程度的提高，但和 C 比起来仍然很悬殊。计算机最注重的就是速度；假如在一台计算机上不能明显较快地干活，那么还不如用手做（有人建议在开发期间使用 Java，以缩短开发时间。然后用一个工具和支撑库将代码转换成 C++，这样可获得更快的执行速度）。

为使 Java 适用于大多数 Web 开发项目，关键在于速度上的改善。此时可考虑采用所谓的“准实时”（Just-In Time，或 JIT）编译器——这是 Sun 自己提供的“热区”技术。另外，甚至可以考虑固有代码编译器。当然，固有代码编译器会造成编译好的程序不能跨平台执行，但同时也带来了速度上的提升。这个速度甚至接近 C 和 C++。而且 Java 中的程序交叉编译

应当比 C 和 C++中简单得多（理论上只需重编译即可，但实际仍较难实现；其他语言也曾作出类似的保证）。

在本书第一版的附录，大家可找到与 Java / C++比较以及对 Java 现状的观察有关的内容（本书配套光盘上有，同时也可从 www.BruceEckel.com 下载）。

1.17 总 结

本章的宗旨是为大家留下对面向对象程序设计及 Java 语言本身的一个总体印象，其中包括 OOP 和 Java 各自的特点、OOP 方法学的基本概念以及在向 OOP 及 Java 迁移的时候，你可能会遇到哪些问题及其对策等等。

OOP 和 Java 也许并非面向每一个人的。我们务必要先评估好自己的需要，然后考察 Java 是否能满足那些需要；否则最好考虑其他编程系统（甚至包括你当前正在使用的）。假如你知道在可以预见的将来，自己的需要会变得非常“专业”，而且你的一些特殊限制是 Java 所不能满足的，那么请马上去调查其他方案[18]。即使最终还是选择 Java 作为自己的开发语言，至少也应该清楚自己为什么选它，并对自己以后要进行的工作，有一个明确的认识。

大家知道传统的、面向过程的程序是什么样子的，它们纯粹就是一大堆数据定义和函数调用。要想知道这样一个程序的含义，通常必须做许多工作，检查函数调用和一些低级概念，并在自己的脑海中建立一个模型等等。因此，在设计这样的程序时，我们需要一些中介性的表示。但由于具体的表达方式主要面向计算机，而不是面向我们要解决的问题，所以这些往往容易令人混淆。

Java 则属于对它的一种扩展。由于 Java 在它的顶部添加了许多新概念，所以许多人自然会认为 Java 程序里的一个 main()要比等价的一个 C 程序复杂得多。然而，事实却恰巧与此相反。只要设计得当，Java 程序实际上简单并容易理解得多！我们看到的将是一系列对象定义，它们表达出了自己那个问题空间时的概念（而不是强调在计算机上如何表示）。同时发给那些对象的消息代表着在那个空间里采取的行动。OOP 最出色的地方就在于此，在一个经过良好设计的程序中，你只需简单地读完一遍，便能理解代码的意思。另外，需要的代码量也被大幅削减了，因为我们的许多问题都是通过对原有库代码进行“复用”而解决的。

[18] 在此特别推荐 Python 语言（http://www.Python.org）

第 2 章　一切都是对象

尽管建立在 C++的基础上，但作为一种“面向对象”的语言，Java 显得更加“纯净”！

无论 C++还是 Java 都属于杂合型语言。但在 Java 中，设计者觉得这种杂合并不象在 C++里那么重要。杂合语言允许采用多种编程样式；之所以说 C++是一种杂合语言，是由于它支持与 C 语言的向后兼容能力。由于 C++是 C 的一个超集，所以包含的许多特性都是后者不具备的，这些特性使 C++在某些地方显得过于复杂。

Java 语言首先便假定了我们只希望进行面向对象的程序设计。也就是说，用它正式写程序之前，首先必须先将自己的思想转入一个“纯净”的、面向对象的世界（除非你早已习惯了这个世界的思维方式）。只有做好这个准备工作，才能真正体会到 Java 的易学易用，才能真正体会出它比起其他 OOP 语言的优势。在本章中，我们将探讨 Java 程序的基本组件，并亲自体验为什么说“Java 中的一切都是对象”——即便一个 Java 程序，它本身也是一个“对象”。

2.1　通过引用操作对象

每种编程语言都有自己的数据处理方式。有些时候，程序员必须时刻留意准备处理的是什么类型。您曾利用一些特殊语法直接操作过对象，或处理过一些间接表示的对象吗（C 或 C++里的指针）？

所有这些工作在 Java 里都得到了简化，任何东西都可看作一个“对象”。因此，我们可采用一种统一的语法，任何地方均可照搬不误。但要注意，尽管将一切都“看作”对象，但操作的标识符实际是对一个对象的“引用”（Reference）[19]。在其他 Java 参考书里，还可看到有的人将其称作一个“引用”，甚至一个“指针”。可将这一情形想象成用遥控板（引用）操纵电视机（对象）。只要握住这个遥控板，就相当于掌握了与电视机连接的通道。但一旦需要“换频道”或者“关小声音”，我们实际操纵的是遥控板（引用），再由遥控板自己操纵电视机（对象）。如果要在房间里四处走走，并想保持对电视机的控制，那么手上拿着的是遥控板，而非电视机。

此外，即使没有电视机，遥控板亦可独立存在。也就是说，只是由于拥有一个引用，并不表示必须有一个对象同它连接。所以如果想容纳一个词或句子，可创建一个 String 引用：

---

[19] 这里有必要澄清一下“引用”和“指针”这两个概念。许多人看到这里，会不假思索地说：“显然，你讲的就是指针！”但这种论断要想成立的话，就必须认为“指针”是一种最基层的实施。此外，就其语法来讲，Java 的引用和 C++的引用存在更大的相似之处。在本书第一版中，我选择了一个新术语“引用”，因为 C++的引用和 Java 的引用毕竟还是存在一些重要的差异。正是由于我最早从事的 C++的编程，所以从心里讲，并不乐意一个随便选择的名称，便把那些 C++程序员搞得晕头转向（我假定有 C++编程经验的人士是本书一个重要的读者群）。不过到第二版，我觉得“引用”已变成了一个更流行的称谓。通过和许多 C++程序员交谈，我发现他们大多克服了术语上的障碍，“引用”在他们面前变得更加熟悉和自然。就象一个已熟练掌握普通话的四川人来到北京，见到老乡说家乡话，见到其他人说普通话，不觉得有丝毫别扭。当然，还是有一些人甚至连“引用”这个称谓也不赞成。我最近读到一本书，上称：“Java 绝对不可能支持按引用传递”，因为 Java 对象标识符（那本书作者的说法）实际是“对象引用”。而且（同样是他的说法）所有东西实际上都是按值传递的。最后的结论是：你不可能按引用传递，实际是“按值传递一个对象引用”。唉，如此曲里拐弯的说法，不知又有多少人会为此展开激烈的辩论！这里我只想说明一点：我的宗旨很简单，就是简化大家对一个概念的理解，同时避免造成对任何方面的损害。达到理解的目的就成，你说呢？（一些老爱刨根问底的人可能会抱怨我正在对您撒谎。不过管它呢，我觉得自己对一个概念进行了最恰当的“抽象”，这便足够了！）

```
String s;
```

但这里创建的只是引用，并不是对象。若此时向 s 发送一条消息，就会获得一个运行时错误。这是由于 s 实际并未与任何东西连接（即“没有电视机”）。因此，一种更安全的做法是：创建一个引用时，记住无论如何都进行初始化：

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

然而，这里采用的是一种特殊类型：字串可用加引号的文字初始化。通常，必须为对象使用一种更通用的初始化类型。

2.2　必须创建所有对象

创建一个引用时，我们希望它同一个新对象建立连接。通常用 new 关键字达到这一目的。new 的意思是：“把我变成这些对象的一种新类型”。所以在上面的例子中，可以说：

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

它不仅指出“将我变成一个新字串”，也通过提供一个初始字串，指出了“如何生成这个新字串”。

当然，字串（String）并非我们能使用的唯一一种类型。尽管 Java 配套提供了数量众多的现成类型，但对我们来讲，最重要的就是记住自己能创建新类型。事实上，这应是 Java 程序设计的一项基本操作。在本书后面，你还会学到大量与此有关的内容。

2.2.1　保存到什么地方

程序运行时，我们最好对数据保存到什么地方做到心中有数。特别要注意的是内存的分配。有六个地方都可以保存数据(这些要记住)：

(1) 寄存器。这是速度最快的地方，数据位于和其他所有方式都不同的一个地方：处理器的内部。不过，寄存器的数量十分有限，所以寄存器是根据需要由编译器分配。我们对此没有直接的控制权，也不可能在自己的程序里找到寄存器存在的任何迹象。

(2) 堆栈。堆栈位于常规 RAM（随机访问存储器）内，但可通过它的“堆栈指针”获得处理器的直接支持。堆栈指针若向下移，会创建新的内存；若向上移，则会释放那些内存。这是一种特别快、特别有效的数据保存方式，仅次于寄存器。创建程序时，Java 编译器必须准确地知道堆栈内保存的所有数据的“长度”以及“存在时间”。这是由于它必须生成相应的代码，以便向上和向下移动指针。这一限制无疑影响了程序的灵活性，所以尽管有些 Java 数据要保存在堆栈里——特别是对象引用，但 Java 对象并不放到其中。

(3) 堆(或“内存堆”)。一种常规用途的内存池（也在 RAM 内），所有 Java 对象都保存在里面。和堆栈不同，“内存堆”或“堆”（Heap）最吸引人的地方在于编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据要在堆里呆多长的时间。因此，用堆保存数据时会得到更大的灵活性。要创建一个对象时，只需用 new 命令编制相关的代码即可。执行这些代码时，就会在堆里自动进行数据的保存。不过，为了获得这种灵活性，我们也必然需要付出一定的代价——假如在内存堆里分配存储空间，和分配规格存储空间相比，前者要花掉更长的时间！（和 C++不同，Java 事实上是不允许在堆栈里创建对象的。这样说，只是为了进行理论上的一种比较）。

(4) 静态存储。这儿的“静态”（Static）是指“位于固定位置”（尽管仍在 RAM 里）。程序运行期间，静态存储的数据将随时等候调用。可用 static 关键字指出一个对象的特定元素是静态的。但 Java 对象本身永远都不会~~不会~~置入静态存储空间。

(5) 常数存储。常数值通常直接置于程序代码内部。这样做是安全的，因为它们永远都

不会改变。有的常数需要严格地保护，所以可考虑将它们置入只读存储器（ROM）。

(6) 非 RAM 存储。若数据完全独立于一个程序之外，那么即使程序不运行了，它们仍可存在，并处在程序的控制范围之外。其中两个最主要的例子便是“流式对象”和“持久性对象”。对于流式对象，对象会变成字节流，通常会发给另一台机器；而对于持久性对象，我们可把它们保存在磁盘或磁带中。即使程序中止运行，它们仍可保持自己的状态不变。之所以要设计这些类型的数据存储，最主要的一个考虑便是把对象变成可在其他媒体上存在的形式；以后一旦需要，还可重新变回一个普通的、存在于 RAM 里的对象。目前，Java 只提供了有限的“持久性对象”支持。在未来的 Java 版本中，有望提供对“持久性”更完善的支持。

2.2.2 特殊情况：主类型

有一系列类型要求我们特殊对待；可将它们想象成“基本”、“主要”或者“主”（Primitive）类型，进行程序设计时会频繁用到它们。之所以要特别对待，是由于用 new 创建某些对象时（特别是那些小的、简单的变量），效率并不是特别高，因为 new 会把对象放到“堆”里。对于这些类型，Java 提供了和 C 和 C++类似的做法。也就是说，不要用 new 创建这些变量，而是创建一个“自动”变量，这种变量并非一个“引用”。为什么不是“引用”呢？因为其中容纳了具体的值，并放在堆栈（而不是内存堆）中，从而获得更高的效率。

Java 事先决定好了每种主类型的大小。就象在大多数语言中那样，这种大小并不随着你改换硬件平台、操作系统而变化。而“大小不可更改”，正是 Java 程序具有很强移植能力的原因之一。(要记住：真无聊！)

主类型	大小	最小值	最大值	封装器类型
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

所有数值类型都是有符号（正负号）的，所以和其他语言不同，在 Java 里不必费劲寻找无符号（无正负）类型。

注意 boolean（布尔）类型的大小没有明确规定；只要能容下“true”或“false”这两个文字值就 OK 了。

原始数据类型也拥有自己的“封装器”（wrapper）类。这意味着假如想让堆内一个非主对象表示那个主类型，就要使用对应的封装器。例如：

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

也可直接使用：

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

这样做的原因将在以后的章节里解释。

高精度数字

Java 提供了两个类，专门用于进行高精度运算：BigInteger 和 BigDecimal。尽管它们可大致划分到与“封装器”相同的类别里，但两者都没有对应的“主类型”。

这两个类都有自己的一系列“方法”，类似于我们针对主类型执行的操作。也就是说，能用 int 或 float 做的事情，用 BigInteger 和 BigDecimal 一样可以做。只是必须换用方法调用，而不是使用运算符。此外，由于牵涉更多，所以运算速度会慢一点。总之，我们牺牲了速度，但换来了精度。

BigInteger 支持任意精度的整数。也就是说，我们可精确表示任意大小的整数值，同时在运算过程中不会丢失任何信息。

BigDecimal 支持任意精度的定点数字。例如，可用它进行精确的币值计算。

至于调用这两个类时，可供选用的构造函数和方法，请自行参考联机帮助文档。

2.2.3　Java 的数组

几乎所有程序语言都支持数组。不过，在 C 和 C++里用数组是非常危险的，因为那些数组其实就是内存块。若程序访问自己内存块以外的数组，或者在初始化之前使用内存（颇为常见的编程错误），便会产生不可预测的后果。

Java 的一项主要设计目标就是安全性。所以在 C 和 C++里困扰程序员的许多问题都未在 Java 里出现。一个 Java 可以保证被初始化，而且不能在它的范围之外进行访问。当然，由于系统会自动进行范围检查，所以必然要付出一些代价——针对每个数组，以及在运行时间对索引的校验，都会造成少量的内存开销。但由此换回的是更高的安全性，以及更高的工作效率。为此付出少许代价应该是值得的吧？

创建对象数组时，实际创建的是由引用构成的一个数组。而且每个引用都会自动初始化成一个特殊值，并带有自己的关键字：null（空）。一旦 Java 看到 null，就知道该引用并未指向一个对象。正式使用前，必须为每个引用都分配一个对象。若试图使用依然为 null 的一个引用，就会在运行时间报告这个问题。因此，典型的数组错误在 Java 里就得到了有效的避免。

也可以创建由主类型构成的数组。同样地，编译器能担保对它进行初始化，它会将那个数组的内存设定为零。

数组问题将在以后的章节里详细讨论。

2.3　绝对不要清除对象

在大多数程序设计语言中，变量的“存在时间”（Lifetime）一直是程序员需要着重考虑的问题。变量应持续多长的时间？如果想清除它，何时进行？在变量存在时间上纠缠不清会造成大量的程序错误。在下面的小节里，我将揭示出 Java 如何帮助程序员完成所有清除工作，从而极大了简化了这个问题。

2.3.1　作用域

大多数程序设计语言都提供了“作用域”（Scope）的概念。对于在作用域里定义的每一个名字，作用域都会同时决定了它的“可见性”（谁能用它）以及“存在时间”（能用多久）。在 C、C++和 Java 里，作用域是由花括号的位置决定的。请见下面这个例子：

```
{
  int x = 12;
  /* only x available */
  {
    int q = 96;
    /* both x & q available */
  }
  /* only x available */
  /* q “out of scope” */
}
```

对于在作用域里定义的一个变量，它只有在那个作用域结束之前，才能“有所作为”。

在上面的代码清单中，我对 Java 代码进行了缩排处理，从而更易辨读。由于 Java 是一种形式自由的语言，所以额外的空格、制表位以及回车符都不会对程序结果造成影响。

注意，尽管在 C 和 C++里是合法的，但在 Java 里却不能象下面这样书写代码：

```
{
  int x = 12;
  {
    int x = 96; /* illegal */
  }
}
```

此时，编译器会认为变量 x 已被定义。也正是由于这个原因，所以 C 和 C++才能将一个变量“隐藏”在一个更大的作用域里（这句话怎么解释？）。但是，这种做法在 Java 里是不允许的，因为 Java 的设计者认为这样做使程序产生了混淆。

2.3.2 对象的作用域

Java 对象不具备与主类型一样的存在时间。用 new 关键字创建一个 Java 对象的时候，它会超出作用域的范围之外。所以假若使用下面这段代码：

```
{
  String s = new String("a string");
} /* 作用域的终点 */
```

那么引用 s 会在作用域的终点处消失。然而，s 指向的 String 对象依然占据着内存空间。在上面这段代码里，我们没有办法访问对象，因为指向它的唯一一个引用已超出了作用域的边界。在后面的章节里，大家还会继续学习如何在程序运行期间传递和复制对象引用。

这样造成的结果便是：对于用 new 创建的对象，只要我们愿意，它们就会一直保留下去。这个问题在 C 和 C++里尤其突出。看来在 C++里遇到的麻烦最大：由于不能从语言那里获得任何帮助，所以在需要对象的时候，根本无法确定它们是否可用。而且更麻烦的是，在 C++里，一旦工作完成，必须保证将对象清除。

这样便带来了一个有趣的问题。假如 Java 让对象依然故我，怎样才能防止它们大量充

斥内存，并最终造成程序“死翘翘”呢。在 C++里，这个问题最令程序员头疼。但 Java 以后，情况却发生了改观。Java 有一个特别的“垃圾收集器”，它会查找用 new 创建的所有对象，并辨别其中哪些不再被引用。随后，它会自动释放由那些闲置对象占据的内存，以便腾出地方，让新对象使用。这意味着我们根本不必操心内存的回收问题。只需简单地创建对象，一旦不再需要它们，它们就会自动离去。这样做可防止在 C++里很常见的一个编程问题：由于程序员忘记释放内存而造成“内存漏洞”（一片谁都用不了、只好看着干瞪眼的内存区域）。

2.4　新建数据类型：类

既然一切都是对象，那么用什么决定一个“类”（Class）的外观与行为呢？换句话说，是什么建立起了一个对象的“类型”（Type）呢？大家可能猜想有一个名为“type”的关键字。但从历史看来，大多数面向对象的语言都用关键字“class”表达这样一个意思：“我要告诉你一种新类型对象看起来象什么样子!”。class 关键字太常用了，以至于本书许多地方并没有用粗体字或双引号加以强调。在这个关键字的后面，应跟随新数据类型的名称。例如：

```
class ATypeName { /* 类的主体代码放在这里 */ }
```

这样就引入了一种新类型，接下来便可用 new 创建这种类型的一个新对象：

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

在上面的 ATypeName 中，类主体只包含了一条简单的注释（星号和斜杠以及其间的文字，本章后面还会对此详细讨论），所以并不能对它做太多的事情。事实上，除非为其定义了某些方法，否则根本不能指示它做任何事情。

2.4.1　字段和方法

定义一个类时（我们在 Java 里做的全部事情就是定义类、制作那些类的对象以及将消息发给那些对象等等），可在自己的类里设置两种类型的元素：数据成员（有时也叫“字段”）以及成员函数（通常叫“方法”）。其中，数据成员是一种对象（通过引用和它通信），可为任何类型。它也可以是主类型（主类型并不是“引用”）之一。如果是指向对象的一个引用，则必须初始化那个引用，用一种名为“构造函数”（第 4 章会对此详述）的特殊函数将其与一个实际对象连接起来（就象早先看到的那样，使用 new 关键字）。但若是一种主类型，则可在类的定义位置直接初始化（正如后面会讲到的那样，引用亦可在它的定义位置进行初始化）。

每个对象都为自己的数据成员保有存储空间；数据成员不会在对象之间共享。下面是定义了一些数据成员的类示例：

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

这个类并没有做任何实质性的事情，但我们可创建一个对象：

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

可将值赋给数据成员，但首先必须知道如何引用一个对象的成员。为达到引用对象成员的目的，首先要写上对象引用的名字，再跟随一个点号（句点），再跟随对象内部成员的名字：

对象引用.成员

例如：

```
d.i = 47;
d.f = 1.1f(如果写成 1.1 将会出现错误，因为浮点型数默认的是 double);
d.b = false;
```

一个对象也可能包含了另一个对象，而另一个对象里则包含了我们想修改的数据。对于这个问题，只需保持“连接句点”即可。例如：

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

除容纳数据之外，DataOnly 类再也不能做更多的事情，因为它没有成员函数（方法）。要想正确理解工作原理，首先必须知道“参数”和“返回值”的概念。我们马上就会详加解释。

主成员的默认值

若某个原始原始(翻译错误，应该是“主”)数据类型属于一个类成员，那么即使不明确（显式）进行初始化，也可保证它们获得一个默认值。(考试会考到)~~()~~

主类型(原始数据类型)	默认值
Boolean	False
Char	‘\u0000’ (null)
Byte	(byte)0
Short	(short)0
Int	0
Long	0L
Float	0.0f
Double	0.0d

一旦将变量作为“类的一个成员”使用，就要特别留意由 Java 分配的默认值是什么。由于 Java 能自动分配默认值，所以保证了主类型的成员变量肯定会得到初始化（C++不具备这一功能），从而有效遏止了多种相关的编程错误。不过，对于自己要编写的程序来说，这种初始值可能并不合适，有时甚至是“非法”的。因此，最好的做法当然是明确初始化这些变量。

然而，这种保证却并不适用于“局部”变量(这一点考试一定会考到)——那些变量并非一个类的字段。所以，假若在一个函数定义中写入下述代码：

```
int x;
```

那么 x 会得到一些随机值（这与 C 和 C++是一样的），不会自动初始化成零。我们需要在正式使用 x 前为其分配一个适当的值。假如忘记这样做，就会得到一条编译期错误，(所以就会出现编译错误)告诉我们变量可能尚未初始化。这种处理正是 Java 优于 C++的表现之一。许多 C++编译器只会对未初始化的变量未发出警告，但在 Java 里却是根本错误的。

2.5　方法、参数和返回值

迄今为止，我们一直在用“函数”（Function）这个词指代一个已命名的子例程。但在 Java 里，更常用的一个词却是“方法”（Method），代表“完成某事的途径”。假如你愿意，尽管继续用“函数”这个词来思考问题。不过，尽管它们表达的实际是同一个意思，但从现在开始，本书将一直使用“方法”，而不是“函数”。

Java 的“方法”决定了一个对象能够接收哪些消息。通过本节的学习，大家会知道方法的定义有多简单！

方法的基本组成部分包括名字、参数、返回类型以及主体代码。下面便是它最基本的形式：

```
返回类型 方法名 ( /* 参数列表 */ ) {
  /* 方法主体 */
}
```

返回类型是指调用方法之后返回的数值类型。参数列表列出了想传递给方法的那些信息的类型与名字。方法名和参数列表组合到一块儿，便“唯一”(注意：返回类型不可以用来标识方法)地标定了一个方法。

Java 的方法只能作为类的一部分创建。只能针对某个对象调用一个方法[20](静态类就不一样)，而且那个对象必须能执行那个方法调用。若试图为一个对象调用错误的方法，就会在编译时得到一条出错消息。为一个对象调用方法时，需要先列出对象的名字，在后面跟上一个句点，再跟上方法名以及它的参数列表。亦即“对象名.方法名(参数 1，参数 2，参数 3...)。举个例子来说，假设我们有一个方法名叫 f()，它没有参数，返回的是类型为 int（整数）的一个值。然后，假定有一个名为 a 的对象，可为其调用方法 f()，则代码如下：

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

返回值的类型必须保证与 x 的类型兼容(一定要注意哟)。

在我们的术语中，象这样调用一个方法的行为通常叫作“向对象发送一条消息”。在上面的例子中，消息是 f()，而对象就是 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”。

2.5.1　参数列表

参数列表规定了我们传送给方法的是什么信息。正如大家或许已猜到的那样，这些信息——如同 Java 内其他任何东西——采用的都是对象的形式。因此，我们必须在参数列表里指定要传递的对象类型，以及每个对象的名字。正如在 Java 其他地方处理对象时一样，我们传递的实际是一些“引用”[21]。(对于 21 的提示，要注意，我们经常传递的就是这些主类型)。然而，引用的类型必须正确。倘若希望参数是一个“字串”，那么传递的就必须是一个字串。

下面让我们来设想一个方法，它将一个字串作为参数使用。下面列出的是定义代码，必须将它置于一个类定义里，否则无法编译：

---

[20] 正如马上就要学到的那样，“静态”方法可针对类直接调用，不用非要针对一个对象。

[21] 对于前面提及的“特殊”数据类型 boolean，char，byte，short，int，long,，float 以及 double 来说，则是一个例外。不过，通常在传递对象时，都是指传递指向对象的一个引用。

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

这个方法告诉我们需要多少字节才能容纳一个特定字串里的信息（字串里的每个字符都是 16 位，或者说 2 个字节、长整数，以便提供对 Unicode 字符的支持）。参数的类型为 String，名字叫作 s。一旦将 s 传递给方法，就可将它当作其他对象一样处理（可向其发送消息）。在这里，我们调用的是 length()方法，它是 String 的默认方法之一，(即 String 类里面的方法)作用是返回一个字串里包含的字符数目。

通过上面的例子，也可以了解到 return 关键字的运用。它主要会做两件事情。首先，它在程序中声明：“现在离开方法吧，我已收工了”。其次，假设方法生成了一个值，那么那个值会紧跟在 return 语句的后面。在这种情况下，返回值是通过计算“s.length()*2”这个表达式而产生的。

可按自己的愿望返回任意类型，(必须和你要返回的值一致)但倘若不想返回任何东西，就可指示方法返回 void（空）。下面列出一些例子。

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

若返回类型为 void，则 return 关键字唯一的作用就是退出方法。所以一旦抵达方法末尾，该关键字便不需要了。可在任何地方从一个方法中返回（离开这个方法）。但假如已指定了一种非 void 的返回类型，那么无论从何地返回，编译器都会确保我们返回的是正确的类型。

到此为止，大家或许已得到了这样的一个印象：一个程序只是一系列对象的集合，它们的方法将其他对象作为自己的参数使用，而且将消息发给那些对象。这种说法大体正确，但通过往后的学习，大家还会知道如何在一个方法里作出决策，做一些更细致的基层工作。至于这一章，只需理解消息的传送就足够了。

2.6 开始构建 Java 程序

正式构建自己的第一个 Java 程序前，还有几个问题需要注意。

2.6.1 名字的可见性

在所有程序语言里，一个不可避免的问题是对名字或名称的控制。假设你在程序的某个模块里使用了一个名字，而另一名程序员在另一个模块里使用了相同的名字。那么，如何区分两个名字，并防止两个名字互相冲突呢？这个问题在 C 语言里尤其突出。因为程序未提供很好的名字管理方法。C++的类（亦即 Java 类的基础）会在类里对函数进行“嵌套”，使其不至于同其他类里的嵌套函数名冲突。然而，C++仍然允许使用全局数据以及全局函数，所以仍然难免冲突。为解决这个问题，C++用其他关键字引入了“命名空间”的概念。

但由于采用了全新的设计，所以 Java 能完全避免这些令人尴尬的问题。为了让库具有一个独一无二的名字，采用与 Internet 域名类似的命名方式。事实上，Java 的设计者鼓励程

序员“反转”使用自己的 Internet 域名，因为它们肯定是独一无二的。例如，我自己的域名是 BruceEckel.com，所以我的实用工具库就可命名为 com.bruceeckel.utility.foibles。反转了域名后，可将点号想象成子目录。

在 Java 1.0 和 Java 1.1 中，域扩展名 com，edu，org，net 等都约定为大写形式。所以库的样子就变成：COM.bruceeckel.utility.foibles。然而，在 Java 2 的开发过程中，设计者发现这样做会造成一些问题。所以目前的整个软件包都以小写字母为标准。

Java 的这种特殊机制意味着所有文件都自动存在于自己的命名空间里。而且一个文件里的每个类都自动获得一个独一无二的标识符。所以不必通过特殊的语言机制来解决这个问题——语言本身已帮我们照顾到这一点。

2.6.2　使用其他组件

若想在自己的程序里使用一个预先定义好的类，必须让编译器知道它的位置。当然，这个类可能就在发出调用的那个相同的源码文件里。如果是那种情况，只需简单地使用这个类即可——即使它直到文件的后面部分都还没有得到定义。Java 消除了“向前引用”的问题，所以不要关心这些事情。

但假若那个类位于其他文件里呢？你或许认为编译器应该足够“聪明”，可以自行发现它。但实情并非如此。假设我们想使用一个具有特定名称的类，但存在那个类的多个定义（可能有不同的定义）。或者更糟，假设我们准备写一个程序，但在创建它的时候，却向自己的库加入了一个新类，它与现有某个类的名字发生了冲突。

为解决这个问题，必须消除所有潜在的、纠缠不清的冲突。为达到这个目的，要用 import 关键字准确告诉 Java 编译器我们希望使用的类是什么。import 的作用是指示编译器导入一个“封装”——或者说一个“类库”（在其他语言里，可将“库”想象成一系列函数、数据以及类的集合。但请记住，Java 的所有代码都必须写入一个类中）。

大多数时候，我们直接采用来自标准 Java 库的组件（部件）即可，它们是由编译器配套提供的。使用这些组件时，没有必要关心冗长的保留域名；举个例子来说，只需象下面这样写一行代码即可：

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

它的作用是告诉编译器我们想使用 Java 的 ArrayList（数组列表）类。然而，util 包含了数量众多的类，而我们只希望使用其中的几个，并不想把所有类都明确声明出来。为达到这个目的，可使用“*”通配符。如下所示：

```
import java.util.*;(注意：*只能包含当前目录下的所有类，不能包含子目录中的类)
```

需导入一系列类时，采用的通常是这个办法。应尽量避免一个一个地导入类。

2.6.3　static 关键字

以前在创建一个类的时候，需要指出那个类的对象的外观与行为。除非用 new 创建那个类的一个对象，否则实际上并未得到任何东西。只有执行了 new 后，才会正式生成数据存储空间，并可使用相应的方法。

但在两种特殊情形下，上述方法并不堪用。一种情形是只想用一个存储区域来保存一个特定的数据(通常是常量)——无论要创建多少个对象，甚至根本不创建对象。另一种情形是我们需要一个特殊的方法，它没有与这个类的任何对象关联。也就是说，即使没有创建对象，也需要一个能调用的方法。为满足这两方面的要求，可使用 static（静态）关键字。一旦将什么东西设为 static，数据或方法就不会同那个类的任何对象实例联系到一起。所以即使从未创建那个类的一个对象，仍能调用一个 static 方法，或访问一些 static 数据。而在这之前，

对于非 static 数据和方法，我们必须创建一个对象，并用那个对象访问数据或方法。这是由于非 static 数据和方法必须知道它们操作的具体对象。//当然，在正式使用前，由于 static 方法不需要创建任何对象，所以它们不可简单地调用其他那些成员，同时不引用一个已命名的对象，从而直接访问非 static 成员或方法（因为非 static 成员和方法必须同一个特定的对象关联到一起）。看英文吧：Of course, since **static** methods don't need any objects to be created before they are used, they cannot *directly* access non-**static** members or methods by simply calling those other members without referring to a named object (since non-**static** members and methods must be tied to a particular object).

有些面向对象的语言使用了“类数据”和“类方法”这两个术语。它们意味着数据和方法只是为作为一个整体的类而存在的，并不是为那个类的任何特定对象。有时，你会在其他一些 Java 书刊里发现这样的称呼。

要想将数据成员或方法设为 static，只需把关键字放到定义前即可。例如，下述代码能生成一个 static 数据成员，并对其进行初始化：

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

现在，尽管我们生成了两个 StaticTest 对象，但它们仍然只占据 StaticTest.i 的一个存储空间。这两个对象都共享同样的 i。请考察下述代码：

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

此时，无论 st1.i 还是 st2.i 的值都是 47，因为它们引用的是同样的内存区域！

有两个办法可引用一个 static 变量。正如上面展示的那样，可通过一个对象命名它，如 st2.i。亦可直接用它的类名引用，而这在非静态成员里是行不通的（考试常考）（最好用这个办法引用 static 变量，因为它强调了那个变量的“静态”本质）。

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

其中，++运算符会使变量值递增。此时，无论 st1.i 还是 st2.i 的值都是 48。

类似的逻辑也适用于静态方法。既可象对其他任何方法那样通过一个对象引用静态方法，亦可用特殊的语法格式“类名.方法()”加以引用。我们用类似的手段来定义一个静态方法：

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

从中可以看出，StaticFun 的方法 incr()使静态数据 i 的值递增。我们可通过一个对象，通过下述典型方式来调用 incr()：

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

另外，由于 incr()是一种静态方法，所以亦可通过它的类直接调用：

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

尽管是“静态”的，但只要应用于一个数据成员之后，就会明确改变数据的创建方式（“一

个类一个成员”，对应于“每个对象一个非静态成员”）。若应用于一个方法，就没有那么戏剧化了。(好像不太懂哟)对方法来说，static 一项重要的用途就是帮助我们在不必创建对象的前提下调用那个方法。正如以后会看到的那样，这一点是至关重要的——特别是在定义程序运行所需的起点方法 **main( )**的时候。

和其他任何方法一样，static 方法也能创建自己类型的命名对象。所以经常把 static 方法作为一个“领头羊”使用，用它生成一系列自己类型的“实例”。

2.7　我们的第一个 Java 程序

最后，让我们正式编写一个程序[22]。它首先打印出一个字串，然后打印出日期（使用由 Java 标准库提供的 Date 类）。注意这里开始采用另一种注释样式：“//”。它的意思是：到本行结束之前，所有内容都是注释：

```
// 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，都必须放置一个 import 语句，导入那个文件的代码里要用到的“额外”的类。注意我们说它们是“额外”的，是由于在每个 Java 文件里，都会自动调入一个特殊的类库：java.lang。启动您的 Web 浏览器，查看由 Sun 提供的用户文档（如果尚未从 http://java.sun.com 下载，或用其他方式安装 Java 文档，请立即下载）。检查一下封装清单，应该能找到 Java 配套提供的所有类库名称。请选择其中的 java.lang。在“Class Index”下面，可找到属于那个库的全部类的列表。由于 java.lang 会默认包括到每一个 Java 代码文件里，所以这些类在任何时候都可直接使用。但在 java.lang 中，并没有列出 Date 类，所以要想使用这个“额外”的类，必须自行导入另一个库。假如不清楚自己想用的一个类到底存在于哪个类库里，或者想检视一下所有的类，请在 Java 文档中选择“Tree”。这样一来，便可找到 Java 自带的所有类。随后，可利用浏览器的“查找”命令，搜索“Date”字样。之后，你会发现它以“java.util.Date”的形式列表。换句话说，Date 类位于 util 库里，因此必须导入 java.util.*；否则便不能使用 Date。

现在回到开头，选择 java.lang，再选 System。这时可看到 System 类包含了几个字段。若选择 out，就可知道它是一个 static PrintStream 对象。由于它是“静态”的，所以不需要

[22] 在某些编程环境里，程序显示会在屏幕上一切而过，甚至没机会看到结果。可将下面这段代码置于 main()的末尾，用它暂停输出：

```
try {
   System.in.read();
} catch(Exception e) {}
```

它的作用是暂停程序输出，直到你按下“回车”键（或其他任意键）。尽管现在或许还不能理解其中的道理（它用到的技术要到接近本书末尾的时候才会探讨），但仍然请把它加到自己的程序中，因为它非常有用！

我们创建任何东西。Out(注意，out 对于 system 来说是一个数据成员，引用的是 PrintStream 类，对于该类来说，就是一个对象)对象肯定存在，所以只需直接用它即可。我们用这个 out 对象能做的事情由它的类型决定，这个类型就是“PrintStream”（打印流）。PrintStream 在说明文字中以一个超链接的形式列出，这一点做得非常方便。所以假若单击那个链接，就可看到能为 PrintStream 调用的所有方法。方法的数量不少，本书后面还会详细介绍。就目前来说，我们感兴趣的只有 println()。它的意思是“把我给你的东西打印到控制台，并用一个新行结束”。所以在任何 Java 程序中，一旦要把某些内容打印到控制台，就可条件反射地写上 System.out.println("内容")。

类名与文件名是一样的。若象现在这样创建一个独立的程序，文件中的一个类必须与文件同名（如果没这样做，编译器会及时作出反应）。类里必须包含一个名为 main()的方法，形式（签名）如下：

```
public static void main(String[] args) {(这里除了 args 可以改变之外，别的都不能改变)
```

其中，关键字“public”意味着方法可由“外部世界”调用（第 5 章会详细解释）。main()的参数是由 String 对象构成的一个数组。尽管本程序不会用到 args，但根据 Java 编译器的坚持，它们必须摆到那儿，因为它们容纳了要在命令行用到的参数：

用来打印日期的代码非常有趣：

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

请观察它的参数——一个 Date 对象。创建它的唯一目的就是将它的值发送给 println()。一旦这个语句执行完毕，Date 就不再需要。随之而来的“垃圾收集器”会发现这一情况，并在任何可能的时候将其回收。事实上，就象以前说过，我们没太大的必要关心“清除”的细节。(可是考试会考到收集的具体时间，并不是立即收集，而是等到扫描到之后，才收集)

2.7.1 编译和运行

要想编译和运行该程序，和本书其他所有程序一样，首先必须有一个 Java 编程环境。目前有大量第三方的开发环境可供选用，但在这本书中，我们只假定你使用的是由 Sun 公司提供的 JDK，它可是免费的！假如使用的是其他开发环境，则请仔细参考用户手册，了解如何编译和运行程序。

现在，请上网访问 java.sun.com。那里有大量文字和链接，让你轻松完成 JDK 的下载与安装。注意针对不同的操作系统平台，需要选择不同的 JDK。

JDK 装好之后，需设好计算机的路径信息，以便随时都能找到 javac 和 java 这两个执行程序。然后，下载和解压本书的源码包（配套光盘和 www.BruceEckel.com 处都有），从而为本书的每一章都生成一个单独的子目录。现在，进入 c02 子目录，然后键入下述命令：

```
javac HelloDate.java
```

该命令若正确执行，屏幕上不会有任何反应。但假如报告的是一条出错消息，便说明你的 JDK 尚未正确安装。请首先解决好 JDK 的安装问题！

提示光标恢复后，接着键入：

```
java HelloDate
```

便能看到程序输出的消息，以及末尾的日期。

上述两条命令分别对应于一个程序的“编译”和“运行”。针对本书用到的每个程序，都可“如法炮制”。不过，对本书的源码文件来说，同时还提供了一个名为“makefile”的文件，其中包括了“制作”命令，用于为相应的一章自动构建文件。请访问本书在 www.BruceEckel.com 的主页，了解具体如何使用这些“makefile”。

2.8　注释和嵌入文档

Java 里有两种类型的注释。第一种是传统的、C 语言风格的注释，是从 C++继承来的。这些注释用一个“/*”起头，随后是注释内容，并可跨越多行，最后用一个“*/”结束。注意许多程序员习惯在连续注释的每一行都用一个“*”开头，所以常能看到象下面这样的注释：

```
/* 这是
*  一段注释，
* 它跨越了多个行
*/
```

不过请记住，进行编译时，/*和*/之间的所有东西都会被忽略，包括那些多余的星号。因此，上述注释与下面这段注释并没有什么区别：

```
/* 这是一段注释，
 它跨越了多个行 */
```

第二种类型的注释也起源于 C++。这种注释叫作“单行注释”，以一个“//”起头，表示“//”之后、一直到行末之前的所有内容都是注释。这种类型的注释更常用，因为它书写时更方便。没有必要在键盘上寻找“/”，再寻找“*”（只需连续按“/”键两次就可以了），而且不必在注释结尾时再加一个结束标记。下面便是这类注释的一个例子：

```
// 这是一条单行注释
```

2.8.1　注释文档

对 Java 语言来说，它最体贴的一项设计就是设计者并没有打算让人们“为了写程序而写程序”——人们也需要考虑程序的文档化问题。对于程序的文档化，最大的问题莫过于对文档的维护。若文档与代码分离，那么每次改变代码后都要改变文档，这无疑会变成相当麻烦的一件事情。解决的方法看起来似乎很简单：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内容都置于同一文件。然而，为使一切都整齐划一，还必须使用一种特殊的注释语法，以便标记出特殊的文档；另外还需要一个工具，用于提取这些注释，并按照用户的需求将其表示出来。所有这一切，在 Java 中都做到了！

用于提取注释的工具叫作 javadoc。它采用了部分来自 Java 编译器的技术，查找我们置入程序的特殊注释标记。它不仅提取由这些标记指示的信息，也将毗邻注释的类名或方法名提取出来。这样一来，我们就可用最轻的工作量，生成十分专业的程序文档。

javadoc 输出的是一个 HTML 文件，可用自己的 Web 浏览器查看。该工具允许我们创建和管理单个源文件，并生动生成有用的文档。由于有了 jvadoc，所以我们能用标准的方法创建文档。而且由于它非常方便，所以我们能轻松获得所有 Java 库的文档信息。

2.8.2　具体语法

所有 javadoc 命令都只能出现在“/**”注释中。和平常一样，注释需要用一个“*/”结尾。主要通过两种方式来使用 javadoc：嵌入的 HTML，或使用“文档标记”。其中，“文档标记”（Doc tags）是一些以“@”开头的命令，置于注释行的起始处（但最前头的“*”会被忽略）。

有三种类型的注释文档，它们对应于位于注释后面的元素：类、变量或者方法。也就是

说，一个类注释正好位于一个类定义之前；变量注释正好位于变量定义之前；而一个方法定义正好位于一个方法定义的前面。如下面这个简单的例子所示：

```
/** 一个类注释 */
public class docTest {
  /** 一个变量注释 */
  public int i;
  /** 一个方法注释*/
  public void f() {}
}
```

注意 javadoc 只能为 public（公共）和 protected（受保护）成员处理注释文档。“private”（私有）和“友好”成员（详见第 5 章）的注释会被忽略，我们看不到任何输出（但是，可用-private 标记包括 private 成员）。这样做是有道理的，因为只有 public 和 protected 成员才可在文件之外使用，这也正是客户机程序员的希望。然而，所有类注释都会包含到输出结果里。

上述代码的输出是一个 HTML 文件，它与其他 Java 文档具有相同的标准格式。因此，用户不会对这种格式感到陌生，可在您设计的类中方便地“漫游”。设计程序时，请务必考虑输入上述代码，用 javadoc 处理一下，观看最终 HTML 文件的效果如何。

2.8.3 嵌入的 HTML

javadoc 将 HTML 命令传递给最终生成的 HTML 文档。这便使我们能充分利用 HTML 的巨大威力。当然，我们的最终动机是对代码进行格式化，而不是为了哗众取宠。下面列出一个例子：

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

亦可象在其他网页文件里那样运用 HTML，对普通文本进行格式化，使其更具条理、更显美观：

```
/**
* 您<em>甚至</em>可以插入一个列表:
* <ol>
* <li> 项目一
* <li> 项目二
* <li> 项目三
* </ol>
*/
```

注意在文档注释中，位于一行最开头的星号会被 javadoc 丢弃。同时丢弃的还有放在前

面的空格（专业术语称“前导空格”）。javadoc 会对所有内容进行格式化，使其与标准的文档外观相符。不要将<h1>或<hr>这样的标题当作嵌入 HTML 使用，因为 javadoc 会插入自己的标题，我们指定的标题会与之冲突。

所有类型的注释文档——类、变量和方法——都支持嵌入的 HTML。

2.8.4　@see：引用其他类

所有三种类型的注释文档（类、变量和方法）都可包含@see 标记，它允许我们对其他类里的文档进行引用（“see”本身是“参见”的意思）。javadoc 会生成相应的 HTML 代码，@see 标记会采用超链接的形式，指向其他文档。它的格式如下：

```
@see 类名
@see 完整类名
@see 完整类名#方法名
```

每种格式都会在生成的文档里自动加入一个超链接的“See Also”（参见）条目。但要注意的是，javadoc 不会检查我们指定的超链接，不会验证它们是否有效。

2.8.5　类文档标记

除了嵌入的 HTML 及@see 引用之外，类文档还可以包括用于版本信息以及作者姓名的标记。亦可为“接口”（参见第 8 章）使用类文档。

1. @version

格式如下：

```
@version 版本信息
```

其中，“版本信息”表示可对版本进行说明的任何文字资料。若在 javadoc 命令行使用了“-version”标记，就会在生成的 HTML 文档里提取到这儿指定的版本信息。

2. @author

格式如下：

```
@author 作者信息
```

其中，“作者信息”包括您的姓名、电子邮件地址或者其他任何适宜的资料。若在 javadoc 命令行使用了“-author”标记，就会在生成的 HTML 文档里专门提取出作者信息。

可为一系列作者使用多个这样的标记，但它们必须连续放置。全部作者信息都会一道混入最终生成的 HTML 代码的单独一个段落里。

3. @since

用这个标记指出一种特别的功能是从该代码的哪个版本开始启用的。大家会在 HTML Java 文档中看到它，指出使用的是哪个 JDK 版本。

2.8.6　变量文档标记

在变量文档中，只能包括嵌入的 HTML 以及@see 引用。

2.8.7　方法文档标记

除嵌入的 HTML 和@see 引用之外，方法还允许用于参数、返回值以及违例的文档标记。

1. @param

格式如下：

```
@param 参数名 说明
```

其中，“参数名”是指参数列表内的标识符，而“说明”代表一些可在后续行内延续的说明文字。一旦遇到一个新的文档标记，就认为前一个说明结束。标记数量不限，但通常每个参数一个。

2. @return

格式如下：

```
@return 说明
```

其中，“说明”是指返回值的含义。它可延续到后面的行内。

3. @throws

有关“违例”（Exception）的详细情况，我们会在第 10 章讲述。简言之，它们是一些特殊的对象，若某个方法失败，就可将它们“扔出”或“掷出”对象。调用一个方法时，尽管只有一个违例对象出现，但一些特殊的方法也许能产生任意数量的、不同类型的违例。所有这些违例都需要说明。所以，违例标记的格式如下：

```
@throws 完整类名 说明
```

其中，“完整类名”明确指定了一个违例类的名字，它应该是在其他某个地方定义好的。而“说明”（同样可以延续到下面的行）告诉我们在什么情况下，这种特定类型的违例会在方法调用中出现。

4. @deprecated

该标记用于指出一些旧功能已由改进过的新功能取代，从而提醒用户不必再使用一样特定的功能，因为未来改版时可能会完全摒弃这一功能。若将一个方法标记为@deprecated，则使用该方法时会收到编译器发出的警告。

2.8.8 文档示例

下面还是我们的第一个 Java 程序，只不过已加入了完整的文档注释：

```
//: 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());
  }
} ///:~
```

该文件的第一行是我自己采用的一项设计：把一个“:”作为特殊记号，指出这是包含了源文件名字的一个注释行。该行包含了指向文件的路径信息（此时，c02 代表第 2 章），后面紧接着文件名[23]。最后一行也用一条注释结束，它标志着源码清单的结尾。采用这样的形式，便能将一段完整的代码从本书正文中自动提取出来，并交由一个编译器检查。

2.9　编码样式

在 Java 中，对于一个类名来说，非正式的标准是对其首字母进行大写处理。若类名由几个单词构成，就把它们紧靠到一起（也就是说，不要用下划线来分隔名字）。此外，每个嵌入单词的首字母都采用大写形式。例如：

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

至于其他几乎所有东西：方法、字段（成员变量）以及对象引用名称等等，为它们采用的样式和类样式差不多，只是标识符的第一个字母采用小写形式。例如：

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

当然，要知道你的用户们也必须一个字母一个字母地输入所有这些长名字，而且不能输错，所以……发发慈悲吧！

大家在 Sun 库中看到的 Java 代码也采用一对花括号的样式编码，和本书采用的一样。

2.10　总　　结

通过本章的学习，大家已接触了足够多的 Java 编程知识，已知道如何自行编写一个简单的程序。此外，对语言的总体情况以及一些基本思想也有了一定程度的认识。然而，本章所有例子的模式都是单线形式的“这样做，再那样做，然后再做另一些事情”。如果想让程

[23] 我用 Python（见 www.Python.org）设计了一个工具，利用这些信息从正文里提取代码文件，把它们放到一个相应的子目录内，再创建“makefile”工具。

序作出一项选择，又该如何设计呢？例如，“假如这样做的结果是红色，就那样做；如果不是，就做另一些事情”。对于这种基本的编程方法，下一章会详细说明在 Java 里是如何实现的。

2.11 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 参照本章的 HelloDate.java 例子，创建一个“Hello，World”程序，在屏幕上简单地显示这句话。注意在自己的类里只需一个方法（“main”方法会在程序启动时执行）。记住要把它设为 static 形式，并包括参数列表——即使根本不会用到这个列表。用 javac 编译这个程序，再用 java 运行它。如果用的是一种不同的编程环境（非 JDK），那么先要了解如何在那种环境中进行程序的编译和运行。

(2) 找出和 ATypeName 有关的代码段，把它们转变成一个程序，编译并运行之。

(3) 将 DataOnly 代码段转变成一个程序，编译并运行之。

(4) 修改练习(3)，只将 DataOnly 中的数据值赋给 main()，并在 main 中打印。

(5) 写一个程序，在其中包括和调用本章中已定义成一个代码段的 storage()方法。//好题目，按照答案进行多次使用 static 方法，可以学习 static 方法。

(6) 将 StaticFun 代码段变成一个可实际工作的程序。//tys,今天到此为止，明天继续。

(7) 写一个程序，打印从命令行提取的三个参数。要想做到这一点，必须对字串构成的命令行数组进行索引。

(8) 将 AllTheColorsOfTheRainbow 例子变成一个正式的程序，编译和运行之。

(9) 找到 HelloDate.java 第二版的代码，它是一个简单的注释文档例子。请对这个文件执行 javadoc，用 Web 浏览器看看结果。

(10) 将 docTest 变成一个文件，先编译，再通过 javadoc 处理一遍。用 Web 浏览器观看结果文档。

(11) 在练习(10)的文档里，增加一个 HTML 项目列表。

(12) 针对练习(1)的程序，为其增添注释文档。用 javadoc 把它提取到一个 HTML 文件里，然后用 Web 浏览器观看结果。

第 3 章　控制程序流程

就象任何理性生物一样，程序必须能操纵自己的世界，并在执行过程中作出判断与选择。

在 Java 里，我们利用运算符操纵对象和数据，并用执行控制语句作出选择。Java 是建立在 C++基础上的，所以对 C 和 C++程序员来说，对 Java 这方面的大多数语句和运算符都应是非常熟悉的。当然，Java 也进行了自己的一些改进与简化工作。

如感觉本章的内容很难，请事先看看本书配套提供的多媒体光盘：“Thinking in C:Foundations for Java and C++”。其中包括了语音授课、幻灯片、练习以及答案，专门让你熟悉 C 基本语法，以便更容易地进军 Java 的世界。

3.1　使用 Java 运算符

运算符以一个或多个参数为基础，可生成一个新值。参数采用与原始方法调用不同的一种形式，但效果是相同的。如果以前有写程序的经验，那么运算符的常规概念应该不难理解。加（+）、减和负（-）、乘（*）、除（/）以及赋值（=）的用法与其他所有程序语言都是相似的。

所有运算符都能根据自己的运算对象生成一个值。除此以外，一个运算符还可改变运算对象的值，这叫作“副作用”（Side Effect）。对于一个会修改运算对象的值的运算符来说，它最常见的作用便是产生“副作用”。但要注意的是，和没有副作用的运算符一样，我们同样可以使用生成的值。

几乎所有运算符都只能操作“主类型”（Primitives）。唯一的例外是“=”、“==”和“!=”，它们能操作所有对象(要记住)（也也是对象易令人混淆的一个地方）。除此以外，String 类支持“+”和“+=”。

3.1.1　优先级

运算符的优先级决定了存在多个运算符时，一个表达式各部分的计算顺序。Java 对计算顺序作出了特别的规定。其中，最简单的规则就是乘法和除法在加法和减法之前完成。程序员经常都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);
```

3.1.2　赋值

赋值是用等号运算符（=）进行的。它的意思是“取得右边的值（通常简称为右值），把它复制到左边（通常简称为左值）”。右值可以是任何常数、变量或者表达式，只要能产生一个值就行。但左值必须是一个明确的、已命名的变量。也就是说，它必须有一个物理性的空间来保存右边的值。举个例子来说，可将一个常数值赋给一个变量（A=4;），但不可将任何东西赋给一个常数，常数永远也不能成为“左值”，比如不能说：4=A;。

对主原始数据类型的赋值是非常直接的。由于主类型容纳了实际的值，而且并非指向一个对象的引用，所以在为其赋值的时候，可将来自一个地方的内容复制到另一个地方。例如，假设为主类型使用“A=B”，那么 B 处的内容就复制到 A。若接着又修改了 A，那么 B 根本不会受这种修改的影响。作为一名程序员，这应成为自己的常识。

但在为对象“赋值”的时候，情况却发生了变化。对一个对象进行操作时，我们真正操作的是它的引用。所以倘若“从一个对象到另一个对象”赋值，实际就是将引用从一个地方复制到另一个地方。这意味着假若为对象指定“C=D”，那么 C 和 D 最终都会指向最初只有 D 才指向的那个对象。下面这个例子将向大家阐示这一点。

```
//: c03:Assignment.java
// Assignment with objects is a bit tricky.

class Number {
  int i;
}

public class Assignment {
  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);
  }
} ///:~
```

Number 类非常简单，它的两个实例（n1 和 n2）是在 main()里创建的。每个 Number 中的 i 值都被赋予了一个不同的值。随后，将 n2 赋给 n1，n1 发生了改变。在许多程序设计语言中，我们都希望 n1 和 n2 任何时候都相互独立。但由于我们赋予的是一个引用，所以下面才是真实的输出：

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

看来改变 n1 的同时也改变了 n2！这是由于无论 n1 还是 n2 都包含了相同的引用，它指向相同的对象（最初的引用位于 n1 内部，指向容纳了值 9 的一个对象。在赋值过程中，那

个引用实际已丢失；它的对象会由“垃圾收集器”自动清除）。

这种特殊的现象通常也叫作“别名”，是 Java 操作对象的一种基本方式。但假若不愿意在这种情况下出现别名，又该怎么操作呢？可放弃赋值，并写入下述代码：

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

这样便可保留两个独立的对象，而不是将 n1 和 n2 绑定到相同的对象。但你很快就会意识到，这样做会使对象内部的字段处理发生混乱，并与标准的面向对象设计准则相悖。由于这并非一个简单的话题，所以留待附录 A 详细论述，那一部分是专门讨论别名的。不过，大家同时还是应该注意到，为对象赋值的时候，会发生一些“令人惊讶”的事情。

方法调用中的别名处理

将一个对象传递到方法内部时，也会产生别名现象。

```
//: c03:PassObject.java
// Passing objects to methods may not be what
// you're used to.

class Letter {
  char c;
}

public class PassObject {
  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);
  }
} ///:~
```

在许多程序语言中，f()方法表面上是要在方法的作用域内制作自己的参数 Letter y 的一个副本。但同样地，实际传递的是一个引用。所以下面这个程序行：

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

实际改变的是 f()之外的对象。输出结果如下：

```
1: x.c: a
2: x.c: z
```

（值得重视的一个问题，就是说 f 带有的‘z’被传递到对象 x 中去了）

别名及其对策是非常复杂的一个问题。尽管必须等到附录 A 才可获得所有答案，但从现在起就应加以重视，以便提早熟悉它的运用，掌握它的一些“窍门”。

3.1.3 算术运算符

Java 的基本算术运算符与其他大多数程序设计语言是相同的。其中包括加（+）、减（-）、除（/）、乘（*）以及模数（%，从整数除中获得余数）。整数除会直接砍掉小数，而不是进位。

Java 也允许用一种简化形式执行运算，并同时进行赋值操作。这是由等号前的一个运算符指定的，而且对于语言中的所有运算符都是固定的。例如，要想把 4 加到变量 x 上，再将结果赋还给 x，可用：x+=4。

下面这个例子展示了算术运算符的各种用法：

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

public class MathOps {
  // Create a shorthand to save typing:
  static void prt(String s) {
    System.out.println(s);
  }
  // shorthand to print a string and an int:
  static void pInt(String s, int i) {
    prt(s + " = " + i);
  }
  // shorthand to print a string and a float:
  static void pFlt(String s, float f) {
    prt(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;
    // '%' limits maximum value to 99:
    j = rand.nextInt() % 100;
    k = rand.nextInt() % 100;
    pInt("j",j);  pInt("k",k);
    i = j + k; pInt("j + k", i);
    i = j - k; pInt("j - k", i);
    i = k / j; pInt("k / j", i);
    i = k * j; pInt("k * j", i);
    i = k % j; pInt("k % j", i);
    j %= k; pInt("j %= k", j);
    // Floating-point number tests:
    float u,v,w;  // applies to doubles, too
```

```
    v = rand.nextFloat();
    w = rand.nextFloat();
    pFlt("v", v); pFlt("w", w);
    u = v + w; pFlt("v + w", u);
    u = v - w; pFlt("v - w", u);
    u = v * w; pFlt("v * w", u);
    u = v / w; pFlt("v / w", u);
    // the following also works for
    // char, byte, short, int, long,
    // and double:
    u += v; pFlt("u += v", u);
    u -= v; pFlt("u -= v", u);
    u *= v; pFlt("u *= v", u);
    u /= v; pFlt("u /= v", u);
  }
} ///:~
```

我们注意到的第一件事情就是用于打印（显示）的一些快捷方法：prt()方法打印一个字串；pInt()先打印一个字串，再打印一个整数；而 pFlt()先打印一个字串，再打印一个浮点数。当然，它们最终都要用 System.out.println()结尾。

为了生成数字，程序首先会创建一个 Random（随机）对象。由于参数是在创建过程中传递的，所以 Java 将当前时间作为一个“种子值”，由随机数生成器利用。通过 Random 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()，nextLong()，nextFloat()或者 nextDouble()。

若随同随机数生成器的结果使用，模数运算符（%）可将结果限制到运算对象减 1 的上限（本例是 99）之下。

一元加、减运算符

一元减号（-）和一元加号（+）与二元加号和减号都是相同的运算符。根据表达式的书写形式，编译器会自动判断使用哪一种。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);
```

一元减号得到的是运算对象的负值。一元加号的含义与一元减号相反，虽然它实际并不做任何事情。

3.1.4　自动递增和递减

和 C 类似，Java 提供了丰富的快捷运算方式。这些快捷运算可使代码更清爽，更易录入，也更易读者辨读。

两种很不错的快捷运算方式是递增和递减运算符（常称作“自动递增”和“自动递减”运算符）。其中，递减运算符是“--”，意为“减少一个单位”；递增运算符是“++”，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）。递增和递减运算符结果生成的是变量的值。

对每种类型的运算符，都有两个版本可供选用；通常将其称为“前缀版”和“后缀版”。“前递增”表示++运算符位于变量或表达式的前面；而“后递增”表示++运算符位于变量或表达式的后面。类似地，“前递减”意味着--运算符位于变量或表达式的前面；而“后递减”意味着--运算符位于变量或表达式的后面。对于前递增和前递减（如++a 或--a），会先执行运算，然后生成值。而对于后递增和后递减（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.

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    prt("i : " + i);
    prt("++i : " + ++i); // Pre-increment
    prt("i++ : " + i++); // Post-increment
    prt("i : " + i);
    prt("--i : " + --i); // Pre-decrement
    prt("i-- : " + i--); // Post-decrement
    prt("i : " + i);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

该程序的输出如下：

```
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。它们是唯一具有“副作用”的运算符（除那些涉及赋值的以外）。也就是说，它们会改变运算对象，而不仅仅是使用自己的值。

递增运算符正是对“C++”这个名字的一种解释，暗示着“超越 C 的一步”。在早期的一次 Java 演讲中，Bill Joy（始创人之一）声称“Java=C++--”（C 加加减减），意味着 Java 已去除了 C++一些没来由折磨人的地方，变成一种更精简的语言。不过，正象大家会在这本

书中学到的那样，尽管 Java 在许多地方都进行了简化，但并不是说它会比 C++“简单得多”！

3.1.5 关系运算符

关系运算符生成的是一个“布尔”（Boolean）结果。它们评价的是运算对象值之间的关系。若关系是真实的，关系表达式会生成 true（真）；若关系不真实，则生成 false（假）。关系运算符包括小于（<）、大于（>）、小于或等于（<=）、大于或等于（>=）、等于（==）以及不等于（!=）。等于和不等于适用于所有内建的数据类型，但其他比较不适用于 boolean 类型。(仔细想想就是肯定的)

检查对象是否相等

关系运算符==和!=也适用于所有对象，但它们的含义通常会使初涉 Java 领域的人找不到北。下面是一个例子：

```
//: c03:Equivalence.java

public class Equivalence {
  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);
  }
} ///:~
```

其中，表达式 **System.out.println(n1 == n2)**可打印出内部的布尔比较结果。一般人都会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。但尽管对象的内容相同，引用却是不同的，而==和!=比较的正好就是对象引用(一定要注意这一点)。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的人感到惊奇。

若想对比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“主类型”，那些类型直接使用==和!=即可。下面举例加以说明：

```
//: c03:EqualsMethod.java

public class EqualsMethod {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1.equals(n2));
  }
} ///:~(这里的 Integer 是系统提供的类)
```

正如我们预计的那样，此时得到的结果是 true。但事情并未到此结束！假设你创建了自己的类，就象下面这样：

```
//: c03:EqualsMethod2.java

class Value {
  int i;
}

public class EqualsMethod2 {
  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));
  }
} ///:~
```

此时的结果又变回了 false！这是由于 equals()的默认行为是**对引用进行对比**。所以除非在自己的新类中改变了 equals()，否则不可能出现我们希望的行为。不幸的是，要到第 7 章才会学习如何改变行为。但要注意 equals()的这种行为方式同时或许能避免一些“灾难”性的事件。

大多数 Java 类库都实现了 equals()，所以它实际比较的是对象的内容，而非它们的引用。(以上的要注意理解)

3.1.6 逻辑运算符

逻辑运算符 AND（&&）、OR（||）以及 NOT（!）能生成一个布尔值（true 或 false）——以参数的逻辑关系为基础。下面这个例子向大家展示了如何使用关系和逻辑运算符。

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

public class Bool {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt() % 100;
    int j = rand.nextInt() % 100;
    prt("i = " + i);
    prt("j = " + j);
    prt("i > j is " + (i > j));
    prt("i < j is " + (i < j));
    prt("i >= j is " + (i >= j));
    prt("i <= j is " + (i <= j));
    prt("i == j is " + (i == j));
    prt("i != j is " + (i != j));
```

```
    // Treating an int as a boolean is
    // not legal Java
//! prt("i && j is " + (i && j));
//! prt("i || j is " + (i || j));
//! prt("!i is " + !i);

    prt("(i < 10) && (j < 10) is "
      + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) is "
      + ((i < 10) || (j < 10)) );
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

只可将 AND、OR 或 NOT 应用于布尔值。与在 C 及 C++中不同，不可将一个非布尔值当作布尔值在逻辑表达式中使用。若这样做，就会发现尝试失败，并用一个“//!”标出。然而，后续的表达式利用关系比较生成布尔值，然后对结果进行逻辑运算。

输出的列表看起来象下面这个样子：

```
i = 85
j = 4
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is true
```

注意假若程序希望得到的是一个字串，那么布尔值会自动转换成适当的文本形式。

在上述程序中，可将对 int 的定义替换成除 boolean 以外的其他任何原始数据类型。但要注意，对浮点数字的比较是非常严格的。即使一个数字仅在小数部分与另一个数字存在极微小的差异，仍然认为它们是“不相等”的。即使一个数字只比零大一点点（例如 2 不停地开平方根），它仍然属于“非零”值。

短路

操作逻辑运算符时，我们会遇到一种名为“短路”的情况。这意味着只有明确得出整个表达式真或假的结论，才会对表达式进行逻辑求值。因此，一个逻辑表达式的所有部分都有可能不进行求值。下面是演未这种现象的一个例子：

```
//: c03:ShortCircuit.java
// Demonstrates short-circuiting behavior.
// with logical operators.

public class ShortCircuit {
  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");
  }
} ///:~
```

每次测试都会比较参数，并返回真或假。它不会显示与准备调用什么有关的资料。测试在下面这个表达式中进行：

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

很自然地，你也许认为所有这三个测试都会得以执行。但希望输出结果不至于使你大吃一惊：

```
test1(0)
result: true
test2(2)
result: false
expression is false
```

第一个测试生成一个 true 结果，所以表达式求值会继续下去。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剩余的表达式呢？这样做只会徒劳无益。事实上，“短路”一词的由来正种因于此。如果一个逻辑表达式的所有部分都不必执行下去，那么潜在的性能提升将是相当可观的。

3.1.7　按位运算符

按位运算符允许我们操作一个整数主原始数据类型中的单个“比特”，即二进制位。按位运算符会对两个参数中对应的位执行布尔代数，并最终生成一个结果。

按位运算来源于 C 语言的低级操作。我们经常都要直接操纵硬件，需要频繁设置硬件寄存器内的二进制位。Java 的设计初衷是嵌入机顶盒内，所以这种低级操作仍被保留下来了。然而，由于操作系统的进步，现在也许不必过于频繁地进行按位运算。

若两个输入位都是 1，则按位 AND 运算符（&）在输出位里生成一个 1；否则生成 0。若两个输入位里至少有一个是 1，则按位 OR 运算符（|）在输出位里生成一个 1；只有在两个输入位都是 0 的情况下，它才会生成一个 0。若两个输入位的某一个是 1，但不全都是 1，那么按位 XOR（^，异或）在输出位里生成一个 1。按位 NOT（~，也叫作“非”运算符）属于一元运算符；它只对一个参数进行操作（其他所有运算符都是二元运算符）。按位 NOT 生成与输入位的相反的值——若输入 0，则输出 1；输入 1，则输出 0。

按位运算符和逻辑运算符都使用了同样的字符，只是数量不同。因此，我们能方便地记忆各自的含义：由于“位”是非常“小”的，所以按位运算符仅使用了一个字符。(很好记忆)

按位运算符可与等号（=）联合使用，以便同时进行运算和赋值：&=，|=和^=都是合法的（由于~是一元运算符，所以不可与=联合使用）。

我们将 boolean（布尔）类型当作一种“单位”或“单比特”值对待，所以它多少有些独特的地方。我们可执行按位 AND、OR 和 XOR，但不能执行按位 NOT（大概是为了避免与逻辑 NOT 混淆）。对于布尔值，按位运算符具有与逻辑运算符相同的效果，只是它们不会中途“短路”。(为什么？)此外，针对布尔值进行的按位运算为我们新增了一个 XOR 逻辑运算符，它并未包括在“逻辑”运算符的列表中。在移位表达式中，我们不能被禁止使用布尔运算，原因将在下面解释。

3.1.8　移位运算符

移位运算符面向的运算对象也是二进制的“位”。可单独用它们处理整数类型（主类型的一种）。左移位运算符（<<）能将运算符左边的运算对象向左移动运算符右侧指定的位数（在低位补 0）。“有符号”右移位运算符（>>）则将运算符左边的运算对象向右移动运算符右侧指定的位数。“有符号”右移位运算符使用了“符号扩展”：若值为正，则在高位插入 0；若值为负，则在高位插入 1。Java 也添加了一种“无符号”右移位运算符（>>>），它使用了“零扩展”：无论正负，都在高位插入 0。这一运算符是 C 或 C++没有的。

若对 char、byte 或者 short 进行移位处理，那么在移位进行之前，它们会自动转换成一个 int。只有右侧的 5 个低位才会用到。这样可防止我们在一个 int 数里移动不切实际的位数。若对一个 long 值进行处理，最后得到的结果也是 long。此时只会用到右侧的 6 个低位，防止移动比 long 值里实际位数还要多的位。

移位可与等号（<<=或>>=或>>>=）组合使用。此时，运算符左边的值会移动由右边的值指定的位数，再将结果赋还给左边的值。但在进行“无符号”右移位时，假如同时还要赋值，会出现一个问题：若对 byte 或 short 值进行右移位运算，那么肯定得不到正确的结果。此时，它们会先自动转换成 int 类型，再进行右移位。但在赋回它们的变量时，会进行“截尾”处理。这样一来，我们便得到了-1 的结果。下面这个例子向大家演示了这个问题：

```
//: c03:URShift.java
// Test of unsigned right shift.
```

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

在最后一行，结果值没有赋还回 b，而是直接打印出来，所以我们得到了正确的结果。

下面这个例子向大家阐示了如何应用涉及“按位”操作的所有运算符，以及它们的实际效果：

```
//: c03:BitManipulation.java
// Using the bitwise operators.
import java.util.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt();
    int j = rand.nextInt();
    pBinInt("-1", -1);
    pBinInt("+1", +1);
    int maxpos = 2147483647;
    pBinInt("maxpos", maxpos);
    int maxneg = -2147483648;
    pBinInt("maxneg", maxneg);
    pBinInt("i", i);
    pBinInt("~i", ~i);
    pBinInt("-i", -i);
    pBinInt("j", j);
    pBinInt("i & j", i & j);
```

```
    pBinInt("i | j", i | j);
    pBinInt("i ^ j", i ^ j);
    pBinInt("i << 5", i << 5);
    pBinInt("i >> 5", i >> 5);
    pBinInt("(~i) >> 5", (~i) >> 5);
    pBinInt("i >>> 5", i >>> 5);
    pBinInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    pBinLong("-1L", -1L);
    pBinLong("+1L", +1L);
    long ll = 9223372036854775807L;
    pBinLong("maxpos", ll);
    long lln = -9223372036854775808L;
    pBinLong("maxneg", lln);
    pBinLong("l", l);
    pBinLong("~l", ~l);
    pBinLong("-l", -l);
    pBinLong("m", m);
    pBinLong("l & m", l & m);
    pBinLong("l | m", l | m);
    pBinLong("l ^ m", l ^ m);
    pBinLong("l << 5", l << 5);
    pBinLong("l >> 5", l >> 5);
    pBinLong("(~l) >> 5", (~l) >> 5);
    pBinLong("l >>> 5", l >>> 5);
    pBinLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void pBinInt(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 pBinLong(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：pBinInt()和 pBinLong()。它们分别操作一个 int 和 long 值，并用一种二进制格式输出，同时附有简要的说明文字。就目前来说，大家可以暂时不要管它们在程序里具体是如何实现的。

大家要注意的是 System.out.print()的使用，而不是 System.out.println()。print()方法不会产生一个新行，所以它允许我们把一行的内容分隔为几个部分。

除展示所有按位运算符针对 int 和 long 的效果之外，本例也展示了 int 和 long 的最小值、最大值、+1 和-1 值，使大家能体会出它们的实际效果。注意高位代表正负号：0 为正，1 为负。下面列出 int 部分的输出：(这个例子一定要仔细自己算一遍)

```
-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:
  00000011100000000000000110000100
i | j, int: 199212028, binary:
  00001011110111111011101111111100
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
```

数字的二进制形式表现为“有符号 2 的余数”。

3.1.9 三元 if-else 运算符

这种运算符比较罕见，因为它有三个运算对象。但它确属于运算符的一种，因为我们最终也能从它那里得到一个值。这与本章下一节要讲到的普通 if-else 语句是不同的。表达式采取下述形式：

```
布尔表达式 ? 值0 : 值1
```

若“布尔表达式”的结果为 true，就计算“值 0”，而且它的结果成为最终由运算符产生的值。但若“布尔表达式”的结果为 false，计算的就是“值 1”，结果同样成为最终由运算符产生的值。

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

可将条件运算符用于自己的“副作用”，或用于它生成的值。但通常都应将其用于值，因为那样做可将运算符与 if-else 明确区别开。下面便是一个例子：

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

可以看出，假设用普通的 if-else 结构写上述代码，代码量会比上面多出许多。如下所示：

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

但第二种形式更易理解，而且不要求更多的录入。所以在决定用三元运算符之前，务必权衡一下利弊。

3.1.10 逗号运算符

在 C 和 C++里，逗号不仅作为函数参数列表的分隔符使用，也作为进行后续计算的一个运算符使用。在 Java 里需要用到逗号的唯一场所就是 for 循环，本章稍后会对此详加解释。

3.1.11 字串运算符+

这个运算符在 Java 里有一项特殊用途：连接不同的字串。这一点已在前面的例子中展示过了。尽管与+的传统意义不符，但用+来做这件事情仍然是非常自然的。在 C++里，这一功能看起来非常不错，所以引入了一项“运算符重载”机制，以便 C++程序员为几乎所有运算符增加特殊的含义。但非常不幸，由于 C++存在的另外一些限制，所以运算符重载其实是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周到的考虑。与 C++相比，尽管运算符重载在 Java 里更易实现，但迄今为止仍然认为这一特性过于复杂。所以 Java 程序员不能象 C++程序员那样设计自己的重载运算符。

我们注意到运用“String +”时一些有趣的现象。若表达式以一个 String 起头，那么后续所有运算对象都必须是字串（请记住，编译器会将一个加上引号的字符序列返还到一个字串里）。如下所示：

```
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 转换成它们的字串形式，而不是先把它们加到一起。然而，如果使用的是下述语句：

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

那么 Java 会将 x 转换成一个字串。

3.1.12 运算符常规操作规则

使用运算符的一个缺点是括号的运用经常容易搞错。即使你对对一个表达式如何运算有丝毫搞不清楚的地方，都会很容易弄混淆括号的用法。这个问题在 Java 里依然存在。

在 C 和 C++中，一个特别常见的错误如下：

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

程序原先的意图是测试是否“相等”（==），而不是进行赋值操作。在 C 和 C++中，只要 y 是一个非零值，那么这种赋值的结果肯定是 true。这样便会得到一个死循环。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，而且不会自动将一个 int 值转换成布尔值。所以在编译时，系统便会提示出现错误，有效地阻止我们进一步运行程序。所以上述缺点在 Java 里永远不会造成更严重的后果。唯一不会得到编译错误的情况便是 x 和 y 都为布尔值。在这种情况下，x = y 属于合法表达式。而在上述情况下，则可能是一个错误。

在 C 和 C++里，类似的一个问题是使用按位 AND 和 OR，而不是逻辑 AND 和 OR。按位 AND 和 OR 使用两个字符之一（&或|），而逻辑 AND 和 OR 使用两个相同的字符（&&或||）。就象“=”和“==”一样，键入一个字符当然要比键入两个简单。在 Java 里，编译器同样可防止这一点，因为它不允许我们强行使用一种它并不属于的类型。

3.1.13 强制转型运算符

“强制转型”（Cast）的作用是“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，假设我们为浮点变量分配一个整数值，计算机会将 int 自动转换成 float。通过强制转型，我们可明确设置这种类型的转换，或者在一般没有可能进行的时候强迫它进行。

要想进行一次强制转型，要将括号中希望的数据类型（包括所有修改符）置于其他任何值的左侧。下面是一个例子：

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

正如你看到的那样，既可对一个数值进行强制转型处理，亦可对一个变量进行强制转型处理。但在这儿展示的两种情况下，强制转型均是多余的，因为编译器在必要的时候会自动进行 int 值到 long 值的转换。当然，仍可设置一个强制转型，提醒自己留意，也使程序更清楚。在其他情况下，强制转型只有在代码编译时才会显出其重要性。

在 C 和 C++中，强制转型有时会让人头痛。在 Java 里，强制转型则是一种比较安全的操作。但是，若进行一种名为“缩小转换”（Narrowing Conversion）的操作（也就是说，脚本是能容纳更多信息的数据类型，将其转换成容量较小的类型），此时就可能面临信息丢失的危险。此时，编译器会强迫我们进行强制转型，就好象说：“这可能是一件危险的事情——如果您想让我不顾一切地做，那么对不起，请先明确地造一个型吧！”而对于“放大转换”（Widening conversion），则不必进行明确强制转型，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 允许我们将任何主类型“强制转型”为其他任何一种主类型，只是布尔值（bollean）除外，后者根本不允许进行任何强制转型处理。“类”不允许进行强制转型。要想将一种类转换成另一种，必须采用特殊的方法（字串是一种特殊的情况，本书后面会讲到将对象强制转型到一个类型“家族”里；例如，“橡树”可强制转型为“树”；反之亦然。但对于其他任何外来类型，如“岩石”，则不能强制转型为“树”）。

1. 文字值

最开始的时候，若在一个程序里插入“文字值”（Literal），编译器通常能准确知道要生成什么样的类型。但在有些时候，对于类型却是暧昧不清的。若发生这种情况，必须对编译器加以适当的“指导”。方法是用与文字值关联的字符形式加入一些额外的信息。下面这段代码向大家展示了这些字符。

```
//: c03:Literals.java

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
  long n3 = 200;
  //! long l6(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（Base 16）——它适用于所有整数数据类型——用一个前置的 0x 或 0X 指示。并在后面跟随采用大写或小写形式的 0-9 以及 a-f。若试图将一个变量初始化成超出自身能力的一个值（无论这个值的数值形式如何），编译器就会向我们报告一条出错消息。注意在上述代码中，最大的十六进制值只会在 char、byte 以及 short 身上出现。若超出这一限制，编译器会将值自动变成一个 int，并告诉我们需要对这一次赋值进行“缩小强制转型”。这样一来，我们便可清楚获知自己已经“越界”了。

八进制（Base 8）是用数字中的一个前置 0 以及 0-7 的数位指示的。在 C、C++或者 Java 中，对二进制数字没有相应的“文字”表示方法。

文字值后的尾随字符标志着它的类型。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。

指数总是采用一种我们认为很不直观的记号方法：1.39e-47f。在科学与工程学领域，“e”代表自然对数的基数，约等于 2.718（Java 一种更精确的 double 值采用 Math.E 的形式）。它在象“1.39×e 的-47 次方”这样的指数表达式中使用，意味着“1.39×2.718 的-47 次方”。然而，自 FORTRAN 语言发明后，人们自然而然地觉得 e 代表“10 多少次幂”。这种做法显得颇为古怪，因为 FORTRAN 最初面向的是科学与工程设计领域。理所当然，它的设计者应对这样的混淆概念持谨慎态度[24]。但不管怎样，这种特别的表达方法在 C、C++以及现在的 Java 中顽固地保留下来了。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，从程序设计的角度思考它；它真正的含义是“1.39×10 的-47 次方”。

[24] John Kirkham 这样写道：“我最早是在 1962 年用一部 IBM 1620 机器使用 FORTRAN II。那时——包括 60 年代以及 70 年代的早期，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，那种码并不具备小写能力。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’，代表计算系统的基数——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，就会把它认作 Base 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。所以说，这个问题完全是自己‘潜入’FORTRAN 里去的，并非一开始就有，也并非有意想把程序员搞糊涂。如果你真的想使用自然对数的基数，实际有现成的函数可供利用，但它们都是大写的。”

注意假如编译器能正确识别类型，就不必使用尾随字符。对于下述语句：

```
long n3 = 200;
```

它并没有含混不清的地方，所以200后面的一个L大可省去。然而，对于下述语句：

```
float f4 = 1e-47f; // 10的幂数
```

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，告诉我们应该用一个“强制转型”将 double 转换成 float。

2. 转型

大家会发现假若对原始数据类型执行任何算术或按位运算，只要它们“比 int 小”（即char、byte 或者 short），那么在正式执行运算之前，那些值会自动转换成 int。这样一来，最终生成的值就是 int 类型。所以只要把一个值赋回较小的类型，就必须使用“强制转型”。此外，由于是将值赋回给较小的类型，所以可能出现信息丢失的情况）。通常，表达式中最大的数据类型是决定了表达式最终结果大小的那个类型。若将一个 float 值与一个 double 值相乘，结果就是 double；如将一个 int 和一个 long 值相加，则结果为 long。

3.1.14 Java 没有“sizeof”

在 C 和 C++中，sizeof()运算符能满足我们的一项特殊需要——获知为数据项目分配的字符数量。在 C 和 C++中，size()最常见的一种应用就是“移植”。不同的数据在不同的机器上可能有不同的大小，所以在进行一些对大小敏感的运算时，程序员必须对那些类型有多大做到心中有数。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。正如你可能已经想到的那样，在不同平台间的移植正是令 C 和 C++程序员颇为头疼的一个问题。

Java 不需要 sizeof()运算符来满足这方面的需要，因为所有数据类型在所有机器的大小都是相同的。我们根本不必考虑移植问题——Java 本身就是一种“与平台无关”的语言。

3.1.15 复习计算顺序

在我举办的一次培训班中，有人抱怨运算符的优先顺序太难记了。一名学生推荐用一句话来帮助记忆：“Ulcer Addicts Really Like C A lot”，即“溃疡患者特别喜欢（维生素）C”。

助记词	运算符类型	运算符
Ulcer（溃疡）	Unary：一元	+ - ++ --
Addicts（患者）	Arithmetic(shift)：算术（和移位）	* / % + - << >>
Really（特别）	Relational：关系	> < >= <= == !=
Like（喜欢）	Logical(bitwise)：逻辑（和按位）	&& || & | ^
C	Conditional(ternary)：条件（三元）	A>B ? X:Y
A Lot	Assignment：赋值	=（以及复合赋值，如*=）

当然，对于移位和按位运算符（以及其他非英语语系的学生——译注），上表可不是完美的助记方法；但对于其他运算来说，它确实很管用。

3.1.16 运算符总结

下面这个例子向大家展示了如何随同特定的运算符使用主原始数据类型。从根本上说，它是同一个例子不断重复地执行，只是用了不同的主原始数据类型。文件编译时不会报错，因为那些会导致错误的代码行已用//!变成了注释内容。

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

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;
  }
} ///:~
```

注意布尔值（boolean）的能力非常有限。我们只能为其赋予 true 和 false 值。而且可测试它为真还是为假，但不可为它们再添加布尔值，或进行其他其他任何类型运算。

在 char、byte 和 short 中，我们可看到算术运算符的“转型”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。必须将其明确“强制转型”回原来的类型（缩小转换会造成信息的丢失），以便将值赋回那个类型。但对于 int 值，却不必进行强制转型处理，因为所有数据都已经属于 int 类型。然而，不要放松警惕，认为一切事情都是安全的。假如对两个足够大的 int 值执行乘法运算，结果值就会溢出。下面这个例子向大家展示了这一点：

```
//: c03:Overflow.java
// Surprise! Java lets you overflow.

public class Overflow {
  public static void main(String[] args) {
    int big = 0x7fffffff; // max int value
    prt("big = " + big);
    int bigger = big * 4;
    prt("bigger = " + bigger);
  }
  static void prt(String s) {
    System.out.println(s);
  }
```

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

输出如下：

```
big = 2147483647
bigger = -4
```

而且不会从编译器那里收到出错或警告消息，在运行时间也不会有“违例”产生。Java 确实是个很好的东西，只是没有一般想像的“那样”好！

对于 char、byte 或者 short，混合赋值并不需要强制转型。即使它们执行转型操作，也会获得与直接算术运算相同的结果。而在另一方面，将强制转型略去可使代码显得更加简练。

大家可以看到，除 boolean 以外，任何一种主类型都可通过强制转型变为其他主类型。同样地，当强制转型成一种较小的类型时，必须留意“缩小转换”的后果。否则会在强制转型过程中不知不觉地丢失信息。

3.2 执行控制

Java 沿用了 C 的全部控制语句，所以假如你以前用过 C 或 C++编程，其中大多数都应是非常熟悉的。大多数程序化的编程语言都提供了某种形式的控制语句，这在语言间通常是共通的。在 Java 里，我们要用到的关键字包括 if-else、while、do-while、for 以及一个名为 switch 的选择语句。注意，Java 并不推荐非常有害的 goto（尽管它仍是解决某些特殊问题的权宜之计）。尽管仍然可以进行象 goto 那样的跳转，但和典型的 goto 比起来要局限多了。

3.2.1 真和假

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。条件表达式的一个例子是 A==B。它用条件运算符“==”来判断 A 值是否等于 B 值。该表达式返回 true 或 false。本章早些时候接触到的所有关系运算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，即使它在 C 和 C++里是允许的（真是非零，而假是零）。若想在一次布尔测试中使用一个非布尔值——比如在 if(a)里，那么首先必须用一个条件表达式将其转换成一个布尔值，例如 if(a!=0)。

3.2.2 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(布尔表达式)
  语句
```

或者

```
if(布尔表达式)
  语句
else
  语句
```

条件必须产生一个布尔结果。“语句”要么是用分号结尾的一个简单语句，要么是一个复合语句——封闭在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，

就有可能包括简单或复合语句。

作为 if-else 的一个例子，用下面这个 test()方法可知道自己猜测的一个数字位于正确数字之上、之下还是相等：

```
//: c03:IfElse.java
public class IfElse {
  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));
  }
} ///:~
```

最好将流程控制语句缩进排列，使读者能方便地看出起点与终点。

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
public class IfElse2 {
  static int test(int testval, int target) {
    int result = 0;
    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));
```

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

不必加上 else，因为方法在遇到 return 后便不再继续。(这一点非常重要，别以为这样是错误的)

3.2.3 迭代

while、do-while 和 for 控制着循环，有时将其划分到“迭代语句”这一类。除非用于控制迭代的布尔表达式得到“假”的结果，否则语句会一直重复执行下去。while 循环的格式如下：

```
while(布尔表达式)
  语句
```

循环刚开始时，会计算一次“布尔表达式”的值。而对于后来每一次额外的循环，都会在开始前重新计算一次。

——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.

public class WhileTest {
  public static void main(String[] args) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
    }
  }
} ///:~
```

它用到了由 Math 库提供的 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(双精度)值。while 条件表达式的意思是说：“一直循环下去吧，直到数字等于或大于 0.99 为止”。由于它的随机性，每运行一次这个程序，都会获得大小不同的随机数列表。

3.2.4 do-while

do-while 的格式如下：

```
do
  语句
while(布尔表达式);
```

while 和 do-while 唯一的区别就是 do-while 至少都会执行一次；也就是说，它至少会将其中的语句“过一遍”——即便表达式第一次的计算结果就为 false。而在 while 循环结构中，

若条件第一次就为 false，那么后续的语句根本不会执行。在实际应用中，while 比 do-while 更常用一些。

3.2.5 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，而且在每一次迭代的时候，进行某种形式的“步进”（Stepping）。for 循环的形式如下：

```
for(初始表达式; 布尔表达式; 步进)
  statement
```

```
for(初始表达式; 布尔表达式; 步进)
  语句
```

无论初始表达式，布尔表达式，还是步进，都可以置空（省去）。每次迭代前，都要测试一下布尔表达式。若结果是 false，就会继续执行紧跟在 for 语句后面的那行代码。在每次循环的末尾，会计算一次步进。

for 循环通常用于执行“计数”任务，就象下面这样：

```
//: c03:ListCharacters.java
// Demonstrates "for" loop by listing
// all the ASCII characters.

public class ListCharacters {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // ANSI Clear screen(好像不是，我的结果是右箭头)
      System.out.println(
        "value: " + (int)c +
        " character: " + c);
  }
} ///:~
```

注意变量 c 是在需要用到它的时候定义的——在 for 循环的控制表达式内部，而非在由起始花括号标记的代码块的最开头。c 的作用域就是由 for 控制的表达式。

对于象 C 这样传统的程序化语言，要求所有变量都在一个块的开头定义。这样一来，编译器创建一个块的时候，便可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。

另外，你也可在 for 语句里定义多个变量，但它们必须具有同样的类型：

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
  /* 循环主体 */;
```

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。

逗号运算符

早在第 1 章，我们便提到了逗号运算符——注意不是逗号分隔符；后者用于分隔函数需要的不同参数。Java 里唯一用到逗号运算符的地方就是 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项能力，下面则是另一个例子：

```
//: c03:CommaOperator.java
public class CommaOperator {
  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);
    }
  }
} ///:~
```

输出如下：

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

大家可看到，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。

3.2.6 中断和继续

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

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

```
//: c03:BreakAndContinue.java
// Demonstrates break and continue keywords.

public class BreakAndContinue {
  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);
    }
  }
} ///:~
```

在这个 for 循环中，i 的值永远不会到达 100。因为一旦 i 循环到 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，才需象这样使用 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执行（所以使 i 值递增）。如果能够整除，就将值显示出来。

第二部分向大家揭示了一个“无限循环”的情况。然而，循环内部有一个 break 语句，可中止循环。除此以外，大家还会看到用 continue 移回循环顶部，不完成剩余的内容（所以只有在 i 值能被 9 整除时才打印出值）。输出结果如下：

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

之所以显示 0，是由于 0%9 等于 0。

无限循环的第二种形式是 **for(;;)**。编译器将 **while(true)**与 **for(;;)**看作同一码事儿。所以具体选用哪个取决于自己的习惯。

臭名昭著的“goto”

goto 关键字很早就在程序设计语言中出现。事实上，goto 是汇编语言的程序控制结构的始祖：“若条件 A，则跳到这里；否则跳到那里”。看看由几乎所有编译器生成的汇编代码，就会发现程序控制里包含了大量跳转。只不过，goto 是在源码这一级跳转的，所以招致了很不好的声誉。若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出代码的流程呢？随着 Edsger Dijkstra 著名的“Goto 有害”论的问世，goto 的地位便如江河日下。

不过，凡事都不能走极端。事实上，真正的问题并不在于使用 goto，而在于 goto 的滥用。在一些少见的情况下，goto 甚至是组织控制流程的最佳手段呢！

尽管 goto 仍是 Java 的一个保留字，但并未在语言中得到正式使用；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。

“标签”是后面跟一个冒号的标识符，就象这样：

```
label1:
```

对 Java 来说，唯一用到标签的地方是在循环语句之前。进一步说，它实际需要紧靠在循环语句的前方——在标签和循环之间置入任何语句都是不明智的。而在循环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，它们就会中断到存在标签的地方。如下所示：

```
label1:
外部循环{
  内部循环{
    //…
    break; // 1
    //…
    continue;  // 2
    //…
    continue label1; // 3
    //…
    break label1;  // 4
  }
}
```

在条件 1 中，break 中断内部循环，并在外部循环结束。在条件 2 中，continue 移回内部循环的起始处。但在条件 3 中，continue label1 却同时中断内部循环以及外部循环，并移至 label1 处。随后，它实际是继续循环，但却从外部循环开始。在条件 4 中，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 for 循环的一个例子：

```
//: c03:LabeledFor.java
// Java's "labeled for" loop.

public class LabeledFor {
  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++) {
        prt("i = " + i);
        if(i == 2) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("break");
          i++; // Otherwise i never
               // gets incremented.(如果没有这句话，break 之后的 i 变量还没有来得及增加!!! 就会导致死循环)
          break;
        }
        if(i == 7) {
          prt("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
        }
        if(i == 8) {
          prt("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            prt("continue inner");
            continue inner;
          }
        }
      }
    }
    // Can't break or continue
    // to labels here
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~(这个例子有点让人转晕)
```

这里用到了在其他例子中已经定义的 prt()方法。

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，所以递增会在 i==3 的情况下直接执行。在 i==7 的情况下，continue outer 语句也会到达循环顶部，而且也会跳过递增，所以它也是直接递增的。

下面是输出结果：

```
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
```

如果没有 break outer 语句，就没办法从一个内部循环里直接跳出外部循环。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，若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.

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      prt("Outer while loop");
      while(true) {
        i++;
        prt("i = " + i);
        if(i == 1) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("continue outer");
          continue outer;
```

```
        }
        if(i == 5) {
          prt("break");
          break;
        }
        if(i == 7) {
          prt("break outer");
          break outer;
        }
      }
    }
  }
  static void prt(String s) {
    System.out.println(s);
  }(问一个简单问题，如果这里的 prt 方法没有 static 关键字，应该怎么办可以同样有效？)
} ///:~
```

同样的规则亦适用于 while：

(1) 简单的一个 continue 会退回最内层循环的开头（顶部），并继续执行。

(2) 带有标签的 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) break 会中断当前循环，“退到循环的底部之外”。

(4) 带标签的 break 会中断当前循环，移到由标签指定的循环底部之外。

这个方法的输出结果是一目了然的：

```
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
```

大家要记住的重点是：在 Java 里唯一要用到标签的地方就是打算使用嵌套循环，并想在多个嵌套级内中断或继续的时候。

在 Dijkstra 的“Goto 有害”论中，他最反对的就是标签，而非 goto。随着标签在一个程序里数量的增多，他发现产生错误的机会也越来越多。标签和 goto 使我们难于对程序作静

态分析。这是由于它们在程序的执行流程中引入了许多“怪圈”。但幸运的是，Java 标签不会造成这方面的问题，因为它们的活动场所已被限死，不可通过特别的方式四处传递程序的控制权。由此也看出了 Java 一个有趣的地方——通过限制语句的能力，反而能使一项语言特性更加有用！

3.2.7 开关

“开关”（Switch）有时也被划分为一种“选择语句”。根据一个整数表达式的值，switch 语句可从一系列代码选出一段执行。它的格式如下：

```
switch(整数选择因子) {
  case 整数值 1 : 语句; break;
  case 整数值 2 : 语句; break;
  case 整数值 3 : 语句; break;
  case 整数值 4 : 语句; break;
  case 整数值 5 : 语句; break;
        // ...
  default: 语句;
}
```

其中，“整数选择因子”是一个特殊的表达式，能产生整数值。switch 能将整数选择因子的结果与每个整数值比较。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（条件）均以一个 break 结尾。这样可使执行流程跳至 switch 主体的末尾。这是构建 switch 语句的一种传统方式，但 break 在这里实际是可选的。若省去 break，会继续执行后面的 case 语句的代码，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，也许仍然能够善加利用。(特别有用，)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的一种简便方式（比如从一系列执行路径中挑选一个）。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，若将一个字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.

public class VowelsAndConsonants {
  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");
      }
    }
  }
} ///:~
```

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最大随机数（对于英语字母，这个数字是 26），再加上一个偏移量，便得到最小的随机数。

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，用单引号封闭起来的字符也会产生整数值(注意这一点，char 也可以)，以便我们进行比较。

请注意 case 语句相互间是如何聚合在一起的，它们依次排列，为一部分特定的代码提供了多种匹配模式。也请注意要将 break 语句置于一个特定 case 的末尾，否则控制流程会简单地下移，并继续判断下一个条件是否相符。

具体的计算

应特别留意下面这个语句：

```
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。利用一个“强制转型”，double 结果会转换回 char。

那么，强制转型会对 char 作什么样的处理呢？换言之，假设一个值是 29.7，我们把它强制转型成一个 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?

public class CastingNumbers {
  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));
  }
} ///:~
```

输出如下：

```
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 值强制转型成整数值后，总是将小数部分“砍掉”，不作任何进位处理。(需要记住这一点)

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？用正统的数学语言表达，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?

public class RandomBounds {
  static void usage() {
    System.out.println("Usage: \n\t" +
      "RandomBounds lower\n\t" +
      "RandomBounds 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();
  }
} ///:~
```

为了运行这个程序，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。但这只是一项实验而已。试想 0 和 1 之间有 2 的 128 次方那么多个不同的双精度小数，假如要全部产生这些数字，花费的时间会远远超过一个人的生命！当然，最后的结果是在 Math.random()的输出中包括了 0.0；用数学语言表达，输出值范围则是[0,1)。

3.3 总　　结

本章总结了大多数程序设计语言都具有的基本特性：计算、运算符优先顺序、类型转换以及选择和迭代等等。现在，我们作好了相应的准备，可继续向面向对象的程序设计领域迈进。在下一章里，我们将讨论对象的初始化与清除问题，再后面则讲述隐藏的基本实现方法。

3.4 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 在本章早先的“优先级”一节里，有两个表达式。请将这些表达式放到一个程序里，看看它们是不是会输出不同的结果。

(2) 在一个可实际工作的程序中，使用 ternary()和 alternative()这两个方法。

(3) 找到“if-else”和“return”这一节，在一个能实际工作的程序中使用 test()和 test2()。

(4) 自己写一个程序，打印出从 1 到 100 的值。

(5) 修改练习(4)，利用 break 关键字，在 47 这个值处，退出程序。再换用 return 试试。(结果虽然一样，但是他们退出的方式却不一样)

(6) 自己写一个函数，令其接收两个字串作为参数，并利用全部布尔比较方式，对比这两个字串，打印出结果。对于==和!=，还要执行 equals()测试。在 main()中，用一些不同的字串对象来调用该函数。

(7) 写一个程序，令其生成 25 对个随机整数值。对于每对个值，都用 if-then-else 语句进行处理，判断它比随机生成的另一个值大、小还是相等。

(8) 修改练习(7)，把代码装到一个“无限”while 循环里。保证它连续运行，除非用键盘中断（通常是按 Ctrl+C）。

(9) 写一个程序，在其中使用两个嵌套循环和模数运算符（%），检查和打印出质数（只能被自己或 1 整除的一种特殊整数）。

(10) 创建一个开关语句（switch），在每个条件（case）下都打印出一条消息，然后将开关语句放到对每种条件进行测试的一个 for 循环里。在每个条件之后，都放一个 break 语句。最后删去所有 break，看看结果会有什么不同。(还应该部分地删除 break，(比如只删除其中的任意一个)你会发现情况很有趣)

第 4 章　初始化和清除

随着计算机的进步，‘不安全’的程序设计已成为编程代价高昂的罪魁祸首之一。

“初始化”和“清除”正是这一系列安全问题中非常重要的两个。许多 C 程序的错误都是由于程序员忘了初始化一个变量造成的。对于现成的库来说，若用户不知道如何初始化库的一个组件，就往往会出现这一类的错误。清除是另一个特殊的问题，因为用完一个元素后，由于不再关心，所以很容易把它忘记。这样一来，那个元素占用的资源会一直保留下去，极易产生资源（主要是内存）耗尽的恶果。

C++为我们引入了“构造函数”的概念。这是一种特殊的方法，在一个对象创建之后自动调用。Java 也沿用了这个概念，但新增了自己的“垃圾收集器”，能在资源不再需要的时候自动释放它们。本章将讨论初始化和清除的问题，以及 Java 如何提供它们的支持。

4.1　用构造函数自动初始化

对于自己写的每个类，都可假想同时创建了一个名为 initialize()的方法（initialize 是“初始化”的意思）。这个名字提醒我们在使用对象之前，应首先调用它，进行初始化。但不幸的是，这也意味着用户千万不能忘了调用这个方法。而在 Java 中，由于提供了名为“构造函数”的一种特殊方法，所以类的设计者可担保每个对象都会得到正确的初始化。若某个类有一个构造函数，那么在创建对象时，Java 会自动调用那个构造函数——甚至在用户毫不知觉的情况下。所以说这是可以担保的！

接着的一个问题是如何命名这个方法。此时有两方面的因素需要考虑。第一个是我们使用的任何名字都可能与打算为某个类成员使用的名字冲突。第二是由于编译器的责任是调用构造函数，所以它必须知道要调用是哪个方法。C++采取的方案看来是最简单的，且更有逻辑性，所以也在 Java 里得到了应用：构造函数的名字与类名相同。这样一来，便可保证象这样的一个方法会在初始化期间自动调用。

下面是带有构造函数的一个简单的类：

```
//: c04:SimpleConstructor.java
// Demonstration of a simple constructor.

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

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
```

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

现在，一旦创建一个对象：

```
new Rock();
```

就会分配相应的存储空间，并调用构造函数。这样就可保证在我们正式使用之前，对象得以正确的初始化。

请注意所有方法首字母小写的编码规则并不适用于构造函数。这是由于构造函数的名字必须与类名完全相同！

和其他任何方法一样，构造函数也能使用参数，以便我们指定对象的具体创建方式。可非常方便地改动上述例子，以便为构造函数赋予自己的参数（参数）。如下所示：

```
//: c04:SimpleConstructor2.java
// Constructors can have arguments.

class Rock2 {
  Rock2(int i) {
    System.out.println(
      "Creating Rock number " + i);
  }
}

public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock2(i);
  }
} ///:~
```

利用构造函数的参数，我们可为一个对象的初始化设定相应的参数。举个例子来说，假设类 Tree 有一个构造函数，它用一个整数参数标记树的高度，那么就可象下面这样创建一个 Tree 对象：

```
Tree t = new Tree(12);  // 12 英尺高的树
```

若 Tree(int)是我们唯一的构造函数，那么编译器不会允许我们以其他任何方式创建一个 Tree 对象。

构造函数有助于消除大量涉及类的问题，并使代码更易阅读。例如在前述的代码段中，我们并未看到对 initialize()方法的明确调用——它在概念上是和定义分离开的。在 Java 中，定义和初始化属于统一的概念——两者缺一不可。

构造函数属于一种较特殊的方法类型，因为它没有返回值。(考试经常考到)这与 void 返回值存在着明显的区别。对于 void 返回值，尽管方法本身不会自动返回什么，但仍然可以让它返回另一些东西。构造函数则不同，它不仅什么也不会自动返回，而且根本不能有任何选择。若存在一个返回值，而且假设我们可以自行选择返回内容，那么编译器多少要知道如何对那个返回值作什么样的处理。(什么意思？)

4.2　方法重载

在任何程序语言中，一项重要的特性便是名字的运用。我们创建一个对象时，相当于为一个存储区域指定了名字；而方法相当于为一项具体的行动指定了名字。通过用名字描述自己的系统，可使自己的程序更易人们理解和修改。它非常象写散文——目的是和读者沟通。

我们利用名字来引用或描述所有对象与方法。若名字选得好，自己和其他人都能更容易地理解代码。

将人类语言中存在细致差别的概念“映射”到一种程序设计语言中时，会出现一些特殊的问题。在日常生活中，我们用相同的词表达多种不同的含义——即词的“重载”。我们可以说“洗衬衫”（wash the shirt）、“洗车”（wash the car）以及“洗狗”（wash the dog）。但若象下面这样说，就显得很不自然：“shirtWash the shirt”、“carWash the car”及“dogWash the dog”。这是由于听众根本不需要对执行的行动作任何明确的区分。人类的大多数语言都具有很强的“冗余”性，所以即使漏掉了几个词，仍可正确推断出含义。我们不需要独一无二的标识符——从具体的语境中，便能推断出含义。

但是，大多数程序语言（特别是 C）都要求我们为每个函数都设定一个独一无二的标识符。所以绝对不能用一个名为 print()的函数来显示整数，再用另一个 print()显示浮点数——每个函数都要求具备唯一的名字。

在 Java 中，另一项因素强迫方法名出现重载情况——构造函数。由于构造函数的名字由类名决定，所以只能有一个构造函数名称。但假若我们想用多种方式创建一个对象呢？例如，假设我们想创建一个类，令其用标准方式进行初始化，另外从文件里读取信息来初始化。此时，我们需要两个构造函数，一个没有参数（默认构造函数），另一个将字串作为参数——用于初始化对象的那个文件的名字。由于都是构造函数，所以它们必须有相同的名字，亦即类名。所以为了让相同的方法名伴随不同的参数类型使用，“方法重载”是非常关键的一项措施。同时，尽管方法重载是构造函数必需的，但它亦可应用于其他任何方法，且用法非常方便。

在下面这个例子里，我们向大家同时展示了重载的构造函数和重载的原始方法：

```
//: c04:Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import java.util.*;

class Tree {
  int height;
  Tree() {
    prt("Planting a seedling");
    height = 0;
  }
  Tree(int i) {
    prt("Creating new Tree that is "
        + i + " feet tall");
    height = i;
  }
```

```
  void info() {
    prt("Tree is " + height
        + " feet tall");
  }
  void info(String s) {
    prt(s + ": Tree is "
        + height + " feet tall");
  }
  static void prt(String s) {
    System.out.println(s);
  }
}

public class Overloading {
  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();
  }
} ///:~
```

Tree 对象既可创建成一颗种子，不含任何参数；亦可创建成生长在苗圃中的植物。为支持这种创建，共用了两个构造函数，一个没有参数（我们把没有参数的构造函数称作“默认构造函数”[25]，），另一个采用现成的高度。

我们也有可能希望通过多种途径调用 info()方法。例如，假设我们有一条额外的消息想显示出来，就使用 String 参数；而如果没其他的话可说，就不使用。由于为显然相同的概念赋予了两个独立的名字，所以看起来可能有些古怪。幸运的是，方法重载允许我们为两者使用相同的名字。

4.2.1 区分重载方法

若方法有同样的名字，Java 怎样知道我们指的是哪一个方法呢？这里有一个简单的规则：每个重载的方法都必须采取独一无二的参数类型列表。

有心人马上就会想到：除根据参数的类型，程序员还有其他办法区分两个同名方法吗？

事实上，即使参数的顺序，也足可使我们区分出两个方法（尽管我们通常不愿意采用这种方法，因为它会产生难以维护的代码）：

[25] 在 Sun 公司出版的一些 Java 资料中，用简陋但很说明问题的词语称呼这类构造函数——“无参数构造函数”（no-arg constructors）。但“默认构造函数”这个称呼已沿用了多年，所以我在这里选用了它。

```
//: c04:OverloadingOrder.java
// Overloading based on the order of
// the arguments.

public class OverloadingOrder {
  static void print(String s, int i) {
    System.out.println(
      "String: " + s +
      ", int: " + i);
  }
  static void print(int i, String s) {
    System.out.println(
      "int: " + i +
      ", String: " + s);
  }
  public static void main(String[] args) {
    print("String first", 11);
    print(99, "Int first");
  }
} ///:~
```

两个 print()方法有完全一致的参数，但顺序不同，所以可据此区分它们。

4.2.2　主类型的重载//tys,今天到这里

主（数据）类型能从一个“较小”的类型自动转变成一个“较大”的类型。涉及重载问题时，这会稍微造成一些混乱。下面这个例子揭示了将主类型传递给重载的方法时发生的情况：

```
//: c04:PrimitiveOverloading.java
// Promotion of primitives and overloading.

public class PrimitiveOverloading {
  // boolean can't be automatically converted
  static void prt(String s) {
    System.out.println(s);
  }

  void f1(char x) { prt("f1(char)"); }
  void f1(byte x) { prt("f1(byte)"); }
  void f1(short x) { prt("f1(short)"); }
  void f1(int x) { prt("f1(int)"); }
  void f1(long x) { prt("f1(long)"); }
  void f1(float x) { prt("f1(float)"); }
  void f1(double x) { prt("f1(double)"); }
```

```
  void f2(byte x) { prt("f2(byte)"); }
  void f2(short x) { prt("f2(short)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(long x) { prt("f2(long)"); }
  void f2(float x) { prt("f2(float)"); }
  void f2(double x) { prt("f2(double)"); }

  void f3(short x) { prt("f3(short)"); }
  void f3(int x) { prt("f3(int)"); }
  void f3(long x) { prt("f3(long)"); }
  void f3(float x) { prt("f3(float)"); }
  void f3(double x) { prt("f3(double)"); }

  void f4(int x) { prt("f4(int)"); }
  void f4(long x) { prt("f4(long)"); }
  void f4(float x) { prt("f4(float)"); }
  void f4(double x) { prt("f4(double)"); }

  void f5(long x) { prt("f5(long)"); }
  void f5(float x) { prt("f5(float)"); }
  void f5(double x) { prt("f5(double)"); }

  void f6(float x) { prt("f6(float)"); }
  void f6(double x) { prt("f6(double)"); }

  void f7(double x) { prt("f7(double)"); }

  void testConstVal() {
    prt("Testing with 5");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
  }
  void testChar() {
    char x = 'x';
    prt("char argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testByte() {
    byte x = 0;
    prt("byte argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testShort() {
    short x = 0;
```

```
    prt("short argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testInt() {
    int x = 0;
    prt("int argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testLong() {
    long x = 0;
    prt("long argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testFloat() {
    float x = 0;
    prt("float argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testDouble() {
    double x = 0;
    prt("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();
  }
} ///:~
```

观察这个程序的输出，就会发现常数值 5 被当作一个 int 值处理。也就是说，假如存在一个重载的方法，它要求接收 int 值，那么就会使用那个方法。在其他所有情况下，若我们的数据类型“小于”方法中使用的参数，就会对那种数据类型进行“转型”处理。char 的效果稍有不同，这是由于假若它没有发现一个准确的 char 匹配，就会转型为 int。

若我们的参数“大于”重载方法期望的参数，又会出现什么情况呢？对上述程序修改一下，便可看出答案：

```
//: c04:Demotion.java
// Demotion of primitives and overloading.

public class Demotion {
  static void prt(String s) {
    System.out.println(s);
  }

  void f1(char x) { prt("f1(char)"); }
  void f1(byte x) { prt("f1(byte)"); }
  void f1(short x) { prt("f1(short)"); }
  void f1(int x) { prt("f1(int)"); }
  void f1(long x) { prt("f1(long)"); }
  void f1(float x) { prt("f1(float)"); }
  void f1(double x) { prt("f1(double)"); }

  void f2(char x) { prt("f2(char)"); }
  void f2(byte x) { prt("f2(byte)"); }
  void f2(short x) { prt("f2(short)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(long x) { prt("f2(long)"); }
  void f2(float x) { prt("f2(float)"); }

  void f3(char x) { prt("f3(char)"); }
  void f3(byte x) { prt("f3(byte)"); }
  void f3(short x) { prt("f3(short)"); }
  void f3(int x) { prt("f3(int)"); }
  void f3(long x) { prt("f3(long)"); }

  void f4(char x) { prt("f4(char)"); }
  void f4(byte x) { prt("f4(byte)"); }
  void f4(short x) { prt("f4(short)"); }
  void f4(int x) { prt("f4(int)"); }

  void f5(char x) { prt("f5(char)"); }
  void f5(byte x) { prt("f5(byte)"); }
  void f5(short x) { prt("f5(short)"); }

  void f6(char x) { prt("f6(char)"); }
  void f6(byte x) { prt("f6(byte)"); }

  void f7(char x) { prt("f7(char)"); }

  void testDouble() {
```

```
    double x = 0;
    prt("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();
  }
} ///:~
```

在这里，方法采用了容量更小、范围更窄的主类型值。如果我们的参数范围比它宽，就必须用括号中的类型名将其转为适当的类型。如果不这样做，编译器就会报错。

大家就注意到，这实际是一种“缩小转换”。也就是说，在强制转型或转型过程中可能丢失一些信息。这正是编译器强迫我们明确定义的原因——我们要明确表达出缩小转换的愿望。

4.2.3　返回值重载

我们很易对下面这些问题感到迷惑——为什么只列出了类名和方法参数？为什么不根据返回值对方法加以区分？

比如对下面这两个方法来说，虽然它们有同样的名字和参数，但其实是很容易区分的：

```
void f() {}
int f() {}
```

若编译器可根据上下文（语境）明确判断出含义，比方说在 int x=f()中，那么这样做完全没有问题。然而，我们也可能调用一个方法，同时忽略返回值；我们通常把这称为“为副作用而调用一个方法”，因为我们关心的不是返回值，而是方法调用的其他效果（副作用）。所以假如我们象下面这样调用方法：

```
f();
```

Java 怎样判断 f()的具体调用方式呢？而且别人如何识别并理解你的代码呢？由于存在这一类的问题，所以不能根据返回值类型来区分重载的方法。(需要注意的一点)

4.2.4　默认构造函数

正如早先指出的那样，默认构造函数是没有参数的。它们的作用是创建一个“空对象”。若创建一个没有构造函数的类，编译器会帮我们自动创建一个默认构造函数。例如：

```
//: c04:DefaultConstructor.java

class Bird {
  int i;
}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird nc = new Bird(); // default!
```

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

对于下面这一行：

```
new Bird();
```

它的作用是新建一个对象，并调用默认构造函数——即使尚未明确定义一个象这样的构造函数。若没有它，便没方法可供调用，无法建立起我们的对象。然而，假如已定义了一个构造函数（无论是否有参数），编译器就不会帮我们自动合成一个：

```
class Bush {
  Bush(int i) {}
  Bush(double d) {}
}
```

现在，假若使用下述代码：

```
new Bush();
```

编译程序就会报告自己找不到一个相符的构造函数。这就好象在我们没有设置任何构造函数的时候，编译程序会说："你看来好需要一个构造函数，就让我给你造一个吧。"但假如我们已经写了一个构造函数，编译程序就会说："啊，你已经有一个构造函数了！所以说，你知道自己想干什么了？即使你没有设置一个默认的，那也是你自己的意愿啊！"

4.2.5 this 关键字

如果有两个同类型的对象，分别叫作 a 和 b，大家也许不知道如何为这两个对象同时调用一个 f()方法：

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

若只有一个叫 f()的方法，它怎样才能知道自己是为对象 a 还是为 b 调用的呢？

为了能用简便的、面向对象的语法来书写代码——亦即"将消息发给对象"，编译器为我们完成了一些幕后工作。其中的秘密就是第一个参数传递给方法 f()，然后那个参数成为即将操作的那个对象的引用。所以前述的两个方法调用就变成了下面这样的形式：

```
Banana.f(a,1);
Banana.f(b,2);
```

不过，这是系统内部的表达形式，我们自己不能这样书写表达式，并试图让编译器接受它。但是，通过它可理解幕后到底发生了什么事情。

假定我们在一个方法的内部，并希望获得当前对象的引用。由于那个引用是由编译器"秘密"传递的，所以没有标识符可用。然而，针对这一目的有个专用的关键字：this。this 关键字（注意只能在方法内部使用）可为已调用了其方法的那个对象生成相应的引用。可象对待其他任何对象引用一样对待这个引用。但要注意，如果准备从自己某个类的另一个方法内部调用一个类方法，就不必用 this，只需简单地调用那个方法即可。当前的 this 引用会自动

应用于其他方法。所以我们能用下面这样的代码：

```
class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
}
```

在 pit()内部，我们可以说 this.pick()，但事实无此必要。编译器能帮我们自动完成。this 关键字只能用于那些特殊的类——需明确使用当前对象的引用。例如，在我们希望将引用返回给当前对象的时候，便经常要在 return 语句中使用：

```
//: c04:Leaf.java
// Simple use of the "this" keyword.

public class Leaf {
  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();
  }
} ///:~
```

由于 increment()通过 this 关键字返回当前对象的引用，所以可以方便地对同一对象执行多项操作。

1. 在构造函数里调用构造函数

若为一个类写了多个构造函数，那么经常都要在一个构造函数里调用另一个构造函数，以免写重复的代码。可用 this 关键字做到这一点。

通常，当我们说 this 的时候，都是指“这个对象”或者“当前对象”。而且它本身会产生当前对象的一个引用。在一个构造函数中，若为其赋予一个参数列表，那么 this 关键字会具有不同的含义：它会对与那个参数列表相符的构造函数进行明确的调用。这样一来，我们就可通过一条直接的途径来调用其他构造函数。如下所示：

```
//: c04:Flower.java
// Calling constructors with "this."

public class Flower {
```

```
  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();
  }
} ///:~
```

(注意：这里的 this.s 是类里面的 s，而等号后面的 s 是参数 s，二者是不同的，可以用语句 System.out.println(this.s)和 System.out.printlln(s)来确认一个打印 null，一个打印 hi)

(这个例子非常好，值得仔细研究，可以修改各种不同的构造函数来查看结果)

其中，构造函数 Flower(String s,int petals)向我们揭示出这样一个问题：尽管可用 this 调用一个构造函数，但不可调用两个。除此以外，构造函数调用必须是我们做的第一件事情，否则编译程序会报错。(在 print 中就不可以调用，调用 this 语句必须是在构造函数的第一条语句!!)

这个例子也向大家展示了 this 的另一项用途。由于参数 s 的名字以及成员数据 s 的名字

是相同的，所以会出现混淆。为解决这个问题，可用 this.s 来引用成员数据。经常都会在 Java 代码里看到这种形式的应用，本书也多处采用了这种做法。

在 print()中，我们发现编译器不让我们从除了一个构造函数之外的其他任何方法内部调用一个构造函数。(!!)

2. static 的含义（讲得太少了）

理解了 this 关键字后，我们可更完整地理解 static（静态）方法的含义。它意味着一个特定的方法没有 this。我们不可从一个 static 方法内部发出对非 static 方法的调用[26]，尽管反过来说是可以的。而且在没有任何对象的前提下，我们可针对类本身发出对一个 static 方法的调用。事实上，那正是 static 方法最基本的意义。它就好象我们创建一个全局函数的等价物（在 C 语言中）。除了全局函数不允许在 Java 中使用以外，若将一个 static 方法置入一个类的内部，它就可以访问其他 static 方法以及 static 字段。

有些人抱怨 static 方法并不是“面向对象”的，因为它们具有全局函数的某些特点；利用 static 方法，我们不必向对象发送一条消息，因为不存在 this。这个抱怨也许是正确的，如果你发现自己使用了大量静态方法，就应重新思量自己的策略。然而，static 的概念是非常实用的，许多时候都要用到它。所以至于它们是否真的“面向对象”，应该留给那些理论家们去争论。事实上，即便是 Smalltalk 语言，它在自己的“类方法”里也有类似于 static 的东西。

4.3　清除：收尾和垃圾收集(好难懂，以后再看)

程序员都知道“初始化”的重要性，但往往把“清除”撇到一边。毕竟，谁需要清除一个 int 呢？但对于库来说，用完后简单地“释放”一个对象并非总是安全的。当然，Java 可用垃圾收集器回收由不再使用的对象占据的内存。现在考虑一种非常特殊且不多见的情况。假定我们的对象分配了一个“特殊”内存区域，没有使用 new。由于垃圾收集器只知释放那些由 new 分配的内存，所以不知道如何释放对象的“特殊”内存。为解决这个问题，Java 提供了一个名为 finalize()的方法，可为自己的类定义它（finalize 是“收尾”的意思）。在理想情况下，它的工作原理应该是这样的：一旦垃圾收集器准备好释放由对象占用的存储空间，便首先调用 finalize()，而且只有在下一次垃圾收集过程中，才会真正回收对象的内存。所以假如使用 finalize()，就可在垃圾收集期间进行一些重要的清除或清扫工作。

但也是一个潜在的编程陷阱，因为有些程序员（特别是在 C++开发背景的）刚开始可能会错误认为它就是在 C++中为“破坏器”（Destructor）使用的 finalize()——破坏（清除）一个对象的时候，肯定会调用这个函数。但在这里有必要区分一下 C++和 Java 的区别，因为 C++的对象肯定会被清除（排除编程错误的因素），而 Java 对象并非肯定能作为垃圾被“收集”去的。或者换句话说：

垃圾收集并不等于“破坏”！

若能时刻牢记这一点，踩到陷阱的可能性就会大为减少。它意味着在我们不再需要一个对象之前，有些行动是必须采取的，而且必须由自己来采取这些行动。Java 并未提供“破坏

[26] 有可能发出这类调用的一种情况是：我们将一个对象引用传到 static 方法内部，随后，通过引用（此时实际是 this），便可调用非 static 方法，并访问非 static 字段。但一般地，如果真的想要这样做，只要制作一个普通的、非 static 方法即可。

器”或类似概念，所以必须创建一个原始的方法，用它来进行这种清除。例如，假设在对象创建过程中，它会将自己描绘到屏幕上。如果不从屏幕上明确删除它的图像，那么它可能永远都不会被清除。若在 finalize()里置入某种删除机制，那么假设对象被当作垃圾收掉了，图像首先会将自身从屏幕上移去。但若未被收掉，图像就会保留下来。所以要记住的第二个重点是：

我们的对象可能不会被当作垃圾收掉！

有时可能发现一个对象的存储空间永远都不会释放，因为自己的程序永远都接近于用光空间的临界点。若程序执行结束，而且垃圾收集器一直都没有释放我们创建的任何对象的存储空间，那么随着程序的退出，那些资源会返回给操作系统。这是一件好事情，因为垃圾收集本身也要消耗一些开销。如永远都不用它，那么永远也不用支出这部分开销。

4.3.1 finalize()用途何在

此时，大家可能已相信了自己应该将 finalize()作为一种常规用途的清除方法使用。它有什么好处呢？

要记住的第三个重点是：

垃圾收集只跟内存有关！

也就是说，垃圾收集器存在的唯一理由就是为了回收程序不再使用的内存。所以对于与垃圾收集有关的任何活动来说，其中最值得注意的是 finalize()方法，它们也必须同内存以及它的回收有关。

但这是否意味着假如对象包含了其他对象，finalize()就应该明确释放那些对象呢？答案是否定的——垃圾收集器会负责释放所有对象占据的内存，无论这些对象是如何创建的。它将对 finalize()的需求限制到特殊的情况。在这种情况下，我们的对象可采用与创建对象时不同的方法分配一些存储空间。但大家或许会注意到，Java 中的所有东西都是对象，所以这到底是怎么一回事呢？

之所以要使用 finalize()，看起来似乎是由于有时需要采取与 Java 的普通方法不同的一种方法，通过分配内存来做一些具有 C 风格的事情。这主要可以通过“固有方法”来进行，它是从 Java 里调用非 Java 方法的一种方式（固有方法的问题在附录 B 讨论）。C 和 C++是目前唯一获得固有方法支持的语言。但由于它们能调用通过其他语言编写的子程序，所以能有效地调用任何东西。在非 Java 代码内部，也许能调用 C 的 malloc()系列函数，用它分配存储空间。而且除非调用了 free()，否则存储空间不会得到释放，从而造成内存“漏洞”的出现。当然，free()是一个 C 和 C++函数，所以我们需要在 finalize()内部的一个固有方法中调用它。

读完上述文字后，大家或许已弄清楚了自己不必过多地使用 finalize()。这个思想是正确的；它并不是进行普通清除工作的理想场所。那么，普通的清除工作应在何处进行呢？

4.3.2 必须明确执行清除

要想清除一个对象，那个对象的用户必须在希望进行清除的地点调用一个清除方法。这听起来似乎很容易做到，但却与 C++“破坏器”的概念稍有抵触。在 C++中，所有对象都会破坏（清除）。或者换句话说，所有对象都“应该”破坏。若将 C++对象创建成一个本地对象，比如在堆栈中创建（在 Java 中是不可能的），那么清除或破坏工作就会在“结束花括号”所代表的、创建这个对象的作用域的末尾进行。若对象是用 new 创建的（类似于 Java），

那么当程序员调用 C++的 delete 命令时（Java 没有这个命令），就会调用相应的破坏器。若程序员忘记了，那么永远不会调用破坏器，我们最终得到的将是一个内存“漏洞”，另外还包括对象的其他部分永远不会得到清除。

相反，Java 不允许我们创建本地（局部）对象——无论如何都要使用 new。但在 Java 中，没有“delete”命令来释放对象，因为垃圾收集器会帮助我们自动释放存储空间。所以如果站在比较简化的立场，我们可以说正是由于存在垃圾收集机制，所以 Java 没有破坏器。然而，随着以后学习的深入，就会知道垃圾收集器的存在并不能完全消除对破坏器的需要，或者说不能消除对破坏器代表的那种机制的需要（而且由于永远都不能直接调用 finalize()，所以它也不是一种合适的方案）。若希望执行除释放存储空间之外的其他某种形式的清除工作，仍然必须调用 Java 中的一个方法。它等价于 C++的破坏器，只是没后者方便。

finalize()最有用处的地方之一是观察垃圾收集的过程。下面这个例子向大家展示了垃圾收集所经历的过程，并对前面的陈述进行了总结。

```
//: c04:Garbage.java
// Demonstration of the garbage
// collector and finalization

class Chair {
  static boolean gcrun = false;
  static boolean f = false;
  static int created = 0;
  static int finalized = 0;
  int i;
  Chair() {
    i = ++created;
    if(created == 47)
      System.out.println("Created 47");
  }
  public void finalize() {
    if(!gcrun) {
      // The first time finalize() is called:
      gcrun = true;
      System.out.println(
        "Beginning to finalize after " +
        created + " Chairs have been created");
    }
    if(i == 47) {
      System.out.println(
        "Finalizing Chair #47, " +
        "Setting flag to stop Chair creation");
      f = true;
    }
    finalized++;
    if(finalized >= created)
```

```
      System.out.println(
        "All " + finalized + " finalized");
  }
}

public class Garbage {
  public static void main(String[] args) {
    // As long as the flag hasn't been set,
    // make Chairs and Strings:
    while(!Chair.f) {
      new Chair();
      new String("To take up space");
    }
    System.out.println(
      "After all Chairs have been created:\n" +
      "total created = " + Chair.created +
      ", total finalized = " + Chair.finalized);
    // Optional arguments force garbage
    // collection & finalization:
    if(args.length > 0) {
      if(args[0].equals("gc") ||
         args[0].equals("all")) {
        System.out.println("gc():");
        System.gc();
      }
      if(args[0].equals("finalize") ||
         args[0].equals("all")) {
        System.out.println("runFinalization():");
        System.runFinalization();
      }
    }
    System.out.println("bye!");
  }
} ///:~
```

上面这个程序创建了许多 Chair（椅子）对象，而且在垃圾收集器开始运行后的某些时候，程序会停止创建 Chair。由于垃圾收集器可能在任何时间运行，所以我们不能准确知道它在何时启动。因此，程序用一个名为 gcrun 的标记来指出垃圾收集器是否已开始运行。利用第二个标记 f，Chair 可告诉 main()它应停止对象的生成。这两个标记都是在 finalize()内部设置的，后者在垃圾收集的时候调用。

另两个 static 变量——created 以及 finalized——分别用于跟踪已创建的对象数量以及由垃圾收集器完成收尾工作的对象的数量。最后，每个 Chair 都有它自己的（非 static）int i，所以能跟踪了解它具体的编号是多少。编号为 47 的 Chair 进行完收尾工作后，标记会设为 true，最终结束 Chair 对象的创建。

所有这些都在 main()的内部进行——在下面这个循环里：

```
while(!Chair.f) {
  new Chair();
  new String("To take up space");
}
```

大家可能会疑惑这个循环什么时候停下来，因为内部没有任何改变 Chair.f 值的语句。然而，finalize()进程最后会改变这个值，在它完成了对 47 号的收尾之后。

每次循环过程中创建的 String 对象只是属于额外的垃圾，用于吸引垃圾收集器——一旦垃圾收集器对可用内存的容量感到“紧张不安”，就会开始关注它。

运行这个程序的时候，需要指定一个命令行参数（参数）：gc、finalize 或者 all。其中，gc 参数会调用 System.gc()方法（以强制执行垃圾收集器）；使用 finalize 参数，会调用 System.runFinalization()方法，从而（在理论上）对所有尚未收尾的对象进行收尾处理；而 all 用于同时调用这两个方法。

本书第一版提供了该程序的第一个版本。通过对比两个程序，大家可注意到随着 JDK 版本的升级，垃圾收集和收尾机制也在发生着变化，变得越来越成熟。事实上，到你读到本书的时候，这个程序的行为恐怕又发生了某些变化。

如果调用了 System.gc()，那么会对所有对象进行“收尾”。不过在以前版本的 JDK 中，实情却并非如此（尽管文档上是这么说的）。此外，就当时的情况来看，不管是不是调用了 System.runFinalization()，结果都没有任何分别。

然而，大家会注意到，在所有对象都创建并丢弃之后，只有调用 System.gc()，才会导致所有收尾器的调用。假如事先没有调用 System.gc()，那么只有部分对象才会收尾。在 Java 1.1 中，出现了一个名为 System.runFinalizersOnExit()的方法，可在程序退出的时候运行所有收尾器。然而，这样的设计容易出错，所以许多人都反对用它。事实上，这件事情正好反映出 Java 的设计者们正在尝试各种各样的方法，来试图解决垃圾收集和收尾问题。到了 Java 2 中，所有问题才有望得以圆满解决。

上面的例子显示出，“收尾器无论如何都会运行”的许诺确实已得到了实现。但是，我们仍然要明确地指示这一情况发生。如果不调用 System.gc()，得到的输出就会象下面这样：

```
Created 47
Beginning to finalize after 3486 Chairs have been created
Finalizing Chair #47, Setting flag to stop Chair creation
After all Chairs have been created:
total created = 3881, total finalized = 2684
bye!
```

也就是说，在程序结束运行的时候，并不是所有收尾器都得到了调用。而只要调用了 System.gc()，它就会收尾和破坏到当前为止不再使用的所有对象。

请记住，无论垃圾收集还是收尾都并非肯定会运行。只要 Java 虚拟机（JVM）还没达到内存即将用光的地步，那么通常不会浪费时间通过垃圾收集机制来回收内存空间。

4.3.3　死亡条件

通常，我们不能依赖对 finalize()的调用，而是应当创建单独的“清除”函数，并明确调

用它们。这样一来，finalize()的用途似乎不是很大，大多数程序员应该很少用到它才是。但是，finalize()实际上还有一种非常有趣的用途，而且这种用途并不是依赖每次对它的调用而发挥出来的。这一用途便是：检验一个对象的“死亡条件”（注释③）。

③：在 Bill Venners（www.artima.com）和我联合举办的一个学习班上，由他“杜撰”了这个术语不过非常贴切。

假如不再对一个对象感兴趣（嗯，它可以被清除掉了），它应当处在内存可以安全清除（回收）的状态。例如，假定该对象代表的是一个打开的文件，那么在把它当作垃圾“收”走之前，应当由程序员关闭那个文件。只要对象的任何部分未被安全清除掉，你的程序就会出现一个“Bug”，而且往后很难发现。finalize()的价值在这时便体现出来了，它可用来发现这种情况，“揪”出隐藏在其中的 Bug——即使并不一定每次都要调用它。只要任何一次通过“收尾”（finalize()）发现了这个 Bug，我们都可以察觉并纠正问题，这才是我们最关心的！

下面是一个简单的例子，阐述了如何用它：

```
//: c04:DeathCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.

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 DeathCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.checkIn();
    // Drop the reference, forget to clean up:
    new Book(true);
    // Force garbage collection & finalization:
    System.gc();
  }
} ///:~
```

这儿的死亡条件是：所有 Book 对象都必须登记完毕！只有所有书都完成了登记，它们才能正式宣告“死亡”，才可对它们实行“垃圾处理”。但是，由于程序员的一处疏忽，main()中的一个错误造成有一本书未被登记。假如这个时候没有 finalize()对死亡条件进行检查，就会导致一个以后极难发现的 Bug。

在这里要注意的是，我们用 System.gc()来进行强制性收尾（在程序开发阶段就应该这样做，以便加快调试速度）。但即使没有这样做，通过程序的迭代执行，最终也有很大可能发现错误的 Book（假定程序分配了足够大的空间，垃圾收集器最后“坐不住”了而不得不开始回收不用的内存）。

4.3.4　垃圾收集器的工作原理

假如你以前学过的程序语言认为在堆中分配对象是一种“代价高昂”的操作，那么很自然地，你会认为 Java 这种“将所有东西都分配到堆中（原始数据类型除外）”的做法未免显得过于草率。

然而，这个问题要从两方面来看。正是由于垃圾收集器的存在，所以大大加快地对象创建的速度！乍一看，这似乎有点儿矛盾——存储空间的释放居然会影响存储空间的分配？但这事实上正是 JVM 的某种工作原理，由此造成的结果便是：在 Java 中为堆对象分配存储空间可达到其他语言在堆栈中创建存储空间差不多的速度。

例如，我们可将 C++的内存堆想象成一个院子，每个对象都拥有自己的场地。这块地以后可以废弃不用，必须加以回收。但在某些 JVM 中，Java 的内存堆却采取了全然不同的做法。它变得更象一条传送带——每次分配了一个新对象，传送带都会向前移动一格。这样一来，对象的分配速度就可以快上许多。此时，“堆指针”只需向前移至最初的地段就可以了。最后的结果便是，它能达到与 C++堆栈分配差不多的工作效率。当然，记录还需涉及一定的开销，但和搜索存储空间相比，却又是微不足道的。

不过，大家现在或许已注意到，堆栈实际并不是一条真正的“传送带”。如果真的把它当传送带来操作，就需要不断地进行内存交换（内存转移到磁盘上，再从磁盘上取回），从而造成极大的性能问题。垃圾收集器在这儿采用了一个非常有用的技巧：收集好垃圾之后，它会对堆内所有对象进行压缩，“堆指针”最后实际是移到了接近传送带起点的地方，比出现页面错误的地方远得多。总之，垃圾收集器对所有东西进行了重新安排，从而实现了一个高速度、无限可用的堆模型，同时还完成了内存空间的分配。

为理解它具体是如何工作的，有必要先对不同的垃圾收集器（GC）机制作一番了解。最简单、速度也最慢的一种 GC 技术叫作“引用计数”。它的原理是：每个对象都包含了一个引用计数器，每次有一个引用连接到一个对象，计数都会递增；而每次有一个引用超出作用域，或被设为 null 之后，引用计数都会递减。因此，对引用计数的管理只需牵涉到极小的连续性开销，它会在你的程序运行期间一直进行。垃圾收集器会对整个对象列表进行搜索。只要找到一个计数为 0 的对象，就会释放掉它的存储空间。这种技术的缺点在于，假如对象相互间进行了循环引用，那么尽管没有非零的引用计数存在，它们仍有可能是真正的垃圾！对垃圾收集器来说，要想发现这种“自己引用自己”的对象组可不容易，需要做大量耗时、耗力的工作。引用计数通常只是理论上的一种垃圾收集机制，但实际的 JVM 方案中，还没有真正用到它的！

垃圾收集要想速度更快，千万不要指望引用计数。相反，它应该基于这样的一个思想：对任何还没“死”的对象来说，归根结底，都可找出它存在于堆栈或静态存储空间里的一个引用。由于这是一种“回溯追踪”的方式，所以往往要经过对象的几个层次。因此，假如从堆栈和静态存储区域开始，遍历所有引用，最后便能找出所有“存活”的对象。对于自己找

到的每一个引用，都必须追踪到它指向的那个对象，再对那个对象中存在的所有引用进行追踪，查出它们指向的对象，然后再对其中的引用进行追踪……以此类推，直到编织出从堆栈或静态存储空间的引用开始的一张“大网”为止。在这期间，我们经历的每个对象肯定仍然处于“存活”状态。要注意的是，那些“自己引用自己”的对象组不会被编到这张由存活对象构成的“大网”里来——我们根本就发现不了它们，所以会被自动地当垃圾看待。

基于上述思路，JVM 采用了一种“自适应”的垃圾收集机制。它对存活对象采取的操作，取决于当前采用的各种变体技术。其中一种变体技术便是“停止和拷贝”。换言之——由于以后大家都会理解的原因——程序首先停止运行（这并不是一种后台收集机制）。然后，每个存活的对象都从一个堆拷贝到另一个堆，将所有垃圾都隔离起来。除此以外，在对象拷贝到新堆的时候，它们也会一个接一个地封装起来（打包），这样便实现了对新堆空间的压缩（而且就象前面说的那样，这样也可使新的存储空间从末尾抽离出去）。

当然，对象从一个地方挪到另一个地方的时候，对该对象的所有引用都必须改变。其中，从堆或静态存储区域到对象的引用马上就可以改变，但可能还有指向该对象的另一些引用，它们要在以后“遍历”的时候才会遇到。那些引用要等到发现的时候才会修改（想象一张表吧，它将老地址对应成新地址）。

不过，考虑到两方面的问题，这种所谓的“拷贝收集器”的效率并不是很高。第一个问题是对多余内存的维护。我们此时有两个堆，需在两个独立的堆之间不断地移动所有对象，同时要维持两倍于我们需要的内存空间，所以显得很“累”。有些 JVM 的做法是根据需要，“成块”地分配一系列堆，以后只需从一个“块”拷贝到另一个“块”就可以了。

第二个问题牵扯到拷贝操作本身。一个程序经调试稳定运行之后，它就可能只产生极少的垃圾，或者根本没有垃圾产生。但拷贝收集器可不管这些，仍然“坚持不懈”地将所有内存从一个地方拷贝到另一个地方，这显然是徒劳无益的。为避免这种情况的发生，有些 JVM 能侦测到没有新垃圾产生的情况，然后改为采用另一种垃圾收集机制（这正是“自适应”最大的魅力所在）。另一种机制叫作“标记和清除”，早期版本的 Sun JVM 一直都在使用这一机制。当然对常规用途来说，“标记和清除”的速度未免太慢了。但假如知道自己已经没有垃圾或者只有少量垃圾，换用它便可得到更高的效率。

“标记和清除”仍然要求从堆栈和静态存储区域开始，对所有引用进行追踪，最后找出所有“活着”的对象。只不过，它每次找到一个存活的对象时，都会为它设置一个“旗帜”，把它标记出来，但此时还不会收集（清除）任何对象。只有在标记过程全部结束之后，才会进行实际的清除工作。清除期间，那些“死掉”的对象会被释放。但请注意，其间不会采取任何拷贝行动，所以即使收集器决定对一个已成碎片的堆进行“压缩”，对象原来的位置也不会改变。

若选择“停止和拷贝”，意味着这种形式的垃圾收集不会在后台进行（都停下来了，还怎么“后台”？）一旦需要 GC，程序就会停止运行。在 Sun 公司提供的资料中，我发现许多地方都说明“垃圾收集是一种低优先级的后台操作”。这当然是不正确的，GC 并没有象那样实现，至少在 Sun JVM 的早期版本中没这样做。相反，在内存不足的时候，Sun 的垃圾收集器才会运行。除此以外，“标记和清除”也要求程序停止运行先。

就象前面指出的那样，在这儿说的 JVM 中，内存是“大块大块”地分配的。假如分配的是一个大型对象，它就会拥有完全属于自己的一个块。而严格的“停止和拷贝”要求将每一个“活着”的对象从原始堆拷贝到一个新堆，然后才能释放掉老堆的空间。显然，其间会消耗掉大量宝贵的内存。而采用“块”技术，GC 在收集的时候，通常可将对象拷贝到那些“死掉”的块中。每个块都有一个“使用计数”（Generation count），以便确认它是否还“活着”。正常情况下，只有在上一次垃圾收集之后创建的块才会得到压缩；对其他所有块来说，只要是被某个地方引用了的，它们的“使用计数”都会递增，表明它“用过”或者“死”了。

这种技术特别适合有大量短期存在的临时对象需要处理的情况。其间会定期进行一次完全清除——大对象仍然不会拷贝（只是递增它们的使用计数），而那些包含着小对象的块会被拷贝和压缩。JVM 会主动监视 GC 的效率。如发现所有对象都有很长的存活时间，那么为了不浪费时间，就改为采用“标记和清除”机制。类似地，JVM 会对成功的“标记和清除”操作进行跟踪。至此为止，“自适应”的全部要旨都被发挥出来，我们得到一套非常令人满意、效率非常高的“垃圾收集”（GC）系统。

在 JVM 中，还可对垃圾收集进行更多的优化，进一步提高速度。其中特别重要的一项技术涉及装载器和“准实时”（Just-In-Time，简称 JIT）编译器的运用。在必须装入一个类的时候（通常是我们第一次想创建属于那个类的一个对象时），就会载入.class 文件，并将那个类的字节码载入内存。在这个时候，一个办法是直接对所有代码进行 JIT 处理，但这样做存在两个缺点：要花更长的时间（随着程序的运行，积累下来的时间还会更长）；另外，它会增大可执行程序文件的长度（字节码可比扩展后的 JIT 代码精简得多），由此可能带来页面交换（数据从内存交换到硬盘上），从而显著影响程序的效率。另一个办法是“暂缓执行”。换句话说，若非必需，否则不对代码进行 JIT 编译。这样一来，那些永远不会执行的代码便永远没有 JIT 编译的机会！

4.4　成员初始化

Java 已尽自己的全力保证所有变量都能在使用前得到正确的初始化。若被定义成相对于某个方法的“局部”或“内部”变量，那么通过编译期的报错，我们就可以理解到 Java 许诺的这一“保证”。因此，假如使用下述代码：

```
void f() {
  int i;
  i++;
}
```

就会收到一条出错提示消息，告诉你 i 可能尚未初始化。当然，编译器也可为 i 赋予一个默认值，但它看起来更象一个程序员的失误，此时默认值反而会“帮倒忙”。若强迫程序员提供一个初始值，就往往能够帮他 / 她纠出程序里的“臭虫”。

然而，若将基类型（主类型）设为一个类的数据成员，情况就会变得稍微有些不同。由于任何方法都可初始化或使用那个数据，所以在正式使用数据前，若还是强迫程序员将其初始化成一个适当的值，就可能不是一种实际的做法。然而，若为其赋予一个垃圾值，同样是非常不安全的。因此，一个类的所有基类型数据成员都会保证获得一个初始值。可用下面这段小程序看到这些值：

```
//: c04:InitialValues.java
// Shows default initial values.

class Measurement {
  boolean t;
  char c;
  byte b;
  short s;
```

```
  int i;
  long l;
  float f;
  double d;
  void print() {
    System.out.println(
      "Data type     Initial value\n" +
      "boolean       " + t + "\n" +
      "char          [" + c + "] "+ (int)c +"\n"+
      "byte          " + b + "\n" +
      "short         " + s + "\n" +
      "int           " + i + "\n" +
      "long          " + l + "\n" +
      "float         " + f + "\n" +
      "double        " + d);
  }
}

public class InitialValues {
  public static void main(String[] args) {
    Measurement d = new Measurement();
    d.print();
    /* In this case you could also say:
    new Measurement().print();
    */
  }
} ///:~
```

输出如下：

```
Data type     Initial value
boolean       false
char          [ ] 0
byte          0
short         0
int           0
long          0
float         0.0
double        0.0
```

其中，char 值是一个零，打印成一个空格。

稍后大家就会看到：在一个类的内部定义一个对象引用时，如果不将其初始化成新对象，那个引用就会获得一个特殊的 null 值（null 是 Java 的一个关键字）。

大家可以发现，即便没有指定值，也会自动得到初始化。也就是说，我们以后不用担心

自己处理的会是一个未被初始化的变量！(这只有在类里面的变量才可以，在方法内的就不行，如上所示)

4.4.1　规定初始化

如果想自己为变量赋予一个初始值，又会发生什么情况呢？为达到这个目的，一个最直接的做法是在类内部定义变量的同时也为其赋值（注意在 C++里可不能这样做，尽管 C++的新手们总是“想”这样做）。在下面，Measurement 类内部的字段定义已发生了变化，提供了初始值：

```
class Measurement {
  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;
  //. . .
```

亦可用相同的方法初始化非基本（主）类型的对象。若 Depth 是一个类，那么可象下面那样插入一个变量并进行初始化：

```
class Measurement {
  Depth o = new Depth();
  boolean b = true;
  // . . .
```

若尚未为 o 指定一个初始值，但又不顾一切地提前用它，就会得到一条运行时错误提示，告诉你产生了名为“违例”（Exception）的一个运行时间错误（在第 10 章详述）。

甚至可通过调用一个方法来提供初始值：

```
class CInit {
  int i = f();
  //...
}
```

当然，这个方法亦可拥有参数，但那些参数不可是尚未初始化的其他类成员。因此，尽管可以这样做：

```
class CInit {
  int i = f();
  int j = g(i);
  //...
```

```
}
```

但却不能这样做：

```
class CInit {
  int j = g(i);
  int i = f();
  //...
}
```

这正是编译器对“向前引用”感到不适的一个反映，因为它与初始化的顺序有关，而不是与程序的编译方式有关。

这种初始化方法非常简单和直观。它的一个限制是类型 Measurement 的每个对象都会获得相同的初始化值。有时，这正是我们希望的结果，但另一些时候，我们却盼望着能有更大的灵活性。

4.4.2 构造函数初始化

可考虑用构造函数进行初始化，这样便可在编程时获得更大的灵活性，因为我们可在运行时间调用方法和采取某些行动，从而“当场”决定初始化值。但要注意的是：我们不能防止自动初始化的进行，它在进入构造函数之前就会发生。因此，假如使用下述代码：

```
class Counter {
  int i;
  Counter() { i = 7; }
  // . . .
```

那么 i 首先会初始化成零，然后变成 7。对于所有基类型以及对象引用——包括在定义时已进行了明确初始化的那一些——这种情况都是成立的。考虑到这个原因，编译器不会强迫我们在某个特定的地方对构造函数的元素进行初始化，也不强迫我们在使用它们之前初始化——初始化早已得到了保证[27]。

1. 初始化顺序

在一个类里，初始化的顺序是由变量在类内的定义顺序决定的。即使变量定义可能散布在方法定义之间，那些变量仍会在调用任何方法之前得到正确的初始化——甚至在构造函数调用之前。例如：

```
//: c04:OrderOfInitialization.java
// Demonstrates initialization order.

// When the constructor is called to create a
```

[27] 相反，C++有自己的一个“构造函数初始模块列表”，能在进入构造函数主体之前进行初始化，而且它对于对象来说是强制进行的。请参见我的《Thinking in C++》第二版，本书配套光盘和 www.BruceEckel.com 处都有。

```
// 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 {
  public static void main(String[] args) {
    Card t = new Card();
    t.f(); // Shows that construction is done
  }
} ///:~(该例子很好)
```

在 Card 中，Tag 对象的定义故意到处散布，以证明它们全都会在进入构造函数或发生其他任何事情之前得到初始化。除此之外，t3 还在构造函数内进行了重新初始化。输出如下：

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

也就是说，t3 引用会被初始化两道，一次在构造函数调用前，一次在调用期间（第一个对象会被丢弃，所以它后来可被当作垃圾收掉）。从表面看，这样做似乎效率低下，但它能保证正确的初始化——假期定义了一个重载的构造函数，它既没有初始化 t3；在它的定义里也没有 t3 的一个“默认”初始化，那么会产生什么样可怕的后果？

2. 静态数据的初始化

若数据是静态的（static），那么同样的事情就会发生；如果它属于一个基类型（主类型），而且未对其初始化，就会自动获得自己的标准基类型初始值；如果它是指向一个对象的引用，那么除非新建一个对象，并将引用同它连接起来，否则就会得到一个 null 值。

如果想在定义的同时进行初始化，采取的方法与非静态值表面看起来是相同的。针对一个静态值，无论你创建了多少个对象，都只有一个存储空间。但在初始化了静态存储空间之后，我们就会遇到问题。下面这个例子可将问题说得更清楚一些：

```
//: c04:StaticInitialization.java
// Specifying initial values in a
// class definition.

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 {
  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);
  }
  static Table t2 = new Table();
  static Cupboard t3 = new Cupboard();
} ///:~
```

Bowl 允许我们检查一个类的创建过程，而 Table 和 Cupboard 能创建散布于类定义中的 Bowl 的静态成员。注意在静态定义之前，Cupboard 先创建了一个非静态的 Bowl b3。它的输出结果如下：

```
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)(好晕啊!!! )
```

static 初始化只有在必要的时候才会进行。如果不创建一个 Table 对象，而且永远都不引用 Table.b1 或 Table.b2，那么 static Bowl b1 和 b2 永远都不会创建。然而，只有在创建了第一个 Table 对象之后（或者发生了第一次 static 访问），它们才会得到初始化。在这以后，static 对象都不会重新初始化。

初始化顺序首先是 static（如果它们尚未由前一次对象创建过程初始化），接着是非 static 对象。大家可从输出结果中找到相应的证据。

在这里有必要总结一下对象的创建过程。请考虑一个名为 Dog 的类：

(1) 类型为 Dog 的一个对象首次创建时，或者首次访问 Dog 类的静态方法 / 静态字段时，Java 解释器必须找到 Dog.class（在事先设好的类路径里搜索）。

(2) 载入 Dog.class 后（创建一个 Class 对象，这在以后还会学到），它的所有静态初始化模块都会运行。因此，静态初始化仅发生一次——在 Class 对象首次载入的时候。

(3) 创建一个 new Dog()时，Dog 对象的构建进程首先会在内存堆（Heap）里为一个 Dog 对象分配足够多的存储空间。

(4) 这种存储空间会清为零，将 Dog 中的所有基类型设为它们的默认值（数字设为零，boolean 和 char 值设为等价的值）。

(5) 字段定义时的所有初始化都会执行。

(6) 执行构造函数。正如第 6 章将要讲到的那样，这实际可能要求进行相当多的操作，特别是在涉及到继承的时候。

3. 明确进行的静态初始化

Java 允许我们将其他 static 初始化工作划分到类内一个特殊的“static 构建从句”（有时也叫作“静态块”）里。它看起来象下面这个样子：

```
class Spoon {
  static int i;
  static {
    i = 47;
  }
  // . . .
```

尽管看起来象个方法，但它实际只是一个 static 关键字，后面跟随一个方法主体。与其他 static 初始化一样，这段代码仅执行一次——在首次生成属于那个类的一个对象时，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.

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 {
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Cups.c1.f(99);  // (1)
  }
  // static Cups x = new Cups();  // (2)
  // static Cups y = new Cups();  // (2)
} ///:~
```

在标记为(1)的行内访问 static 对象 c1 的时候，或在行(1)标记为注释，同时(2)行不标记成注释的时候，就会针对 Cups 进行静态初始化。若(1)和(2)都被标记成注释，那么针对 Cups 的静态初始化操作永远都不会发生。另外，标记为(2)的那两行是不是变成注释都没有关系——静态初始化只分发生一次！

4. 非静态实例的初始化

针对每个对象的非静态变量的初始化，Java 提供了一种类似的语法格式。下面是一个例子：

```
//: c04:Mugs.java
// Java "Instance Initialization."

class Mug {
  Mug(int marker) {
    System.out.println("Mug(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

public class Mugs {
  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();
  }
} ///:~
```

其中的实例初始化从句是：

```
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
  }
```

它看起来和静态初始化从句极其相似，只是 static 关键字从里面消失了。为支持对“匿名内部类”的初始化（参见第 8 章），必须采用这一语法格式。

4.5 数组初始化

在 C 中初始化数组极易出错，而且相当麻烦。C++通过“集合初始化”使其更安全[28]。Java 则没有象 C++那样的“集合”概念，因为 Java 中的一切都是对象。但它确实提供了数组，通过数组初始化来支持。

数组是一系列对象或者原始数据类型，它们都具有相同的类型，并封装到同一个标识名下。数组的定义和使用是通过方括号索引运算符进行的（[]）。为了定义一个数组，只需在类型名后简单地放上一对空的方括号即可：

```
int[] a1;
```

也可将方括号放在标识符后面，意思是完全一样的：

```
int a1[];
```

这种格式与 C 和 C++程序员习惯的格式是一致的。然而，最“通顺”的也许还是前一种语法，因为它指出类型是“一个 int 数组”。本书将沿用那种格式。

编译器不允许我们告诉它一个数组有多大。这样便使我们回到了“引用”的问题上。此

[28] 参见《Thinking in C++》第二版，其中有对 C++集合初始化的完整说明。

时，我们拥有的一切就是指向数组的一个引用，而且尚未给数组分配任何空间。为了给数组创建相应的存储空间，必须编写一个初始化表达式。对于数组，初始化工作可在代码的任何地方出现，但也可使用一种特殊的初始化表达式，它必须在数组创建的地方出现。这种特殊的初始化是一系列由花括号封闭起来的值。在这种情况下，将由编译器负责存储空间的分配（相当于用 new）。例如：

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

那么为什么还要定义没有数组的一个数组引用呢？

```
int[] a2;
```

事实上，在 Java 中，我们可以将一个数组分配给另一个，所以能使用下述语句：

```
a2 = a1;
```

我们真正准备做的是复制一个引用，就象下面演示的那样：

```
//: c04:Arrays.java
// Arrays of primitives.

public class Arrays {
  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]);
  }
} ///:~
```

大家看到 a1 获得了一个初始值，而 a2 没有；a2 将在以后赋值——这种情况下是赋给另一个数组。

这里也出现了一些新东西：所有数组都包含了一个核心成员（无论是对象数组还是基类型数组），可对其进行查询——但不能修改——以便了解数组内包含了多少个元素。这个成员便是 length。和 C 和 C++类似，由于 Java 数组从元素 0 开始计数，所以能索引的最大元素编号是“length-1”。如超出边界，C 和 C++的做法便是“没有做法”，它会“默默”地接受，对我们胡乱使用内存视而不见，而这正是许多程序错误的根源！然而，Java 可保护我们免受这一问题的损害，它的做法是一旦超过边界，就产生一个运行时间错误（即一个“违例”，这是第 10 章的主题）。当然，由于需要检查每个数组的访问，所以会消耗一定的时间和多余的代码量，而且没有办法把它关闭。这意味着数组访问可能成为程序效率低下的重要原因——如果它们在某些关键性场合进行。但考虑到因特网访问的安全，以及程序员的编程效率，Java 设计人员还是应该把它看作是值得的。

程序编写期间，如果事先不知道自己的数组需要多少元素，那又该怎么办呢？此时，只需简单地用 new 在数组里创建元素。在这里，即使创建的是一个由原始数据类型构成的数组，new 也能正常地工作（new 不会创建非数组的基类型）：

```
//: c04:ArrayNew.java
// Creating arrays with new.
import java.util.*;

public class ArrayNew {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pRand(20)];
    System.out.println(
      "length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println(
        "a[" + i + "] = " + a[i]);
  }
} ///:~
```

由于数组的大小是随机决定的（使用 pRand()方法），所以非常明显，数组真正创建实际是在运行时间完成的。除此以外，从这个程序的输出中，大家可看到原始数据类型的数组元素会自动初始化成“空”值（对于数值，空值就是零；对于 char，它是 null；而对于 boolean 值，它却是 false）。

当然，数组也可在同样的语句中当场完成定义和初始化，如下所示：

```
int[] a = new int[pRand(20)];
```

若操作的是一个由非基类型对象构成的数组，那么无论如何都要使用 new。在这里，我们会再一次遇到引用问题，因为实际创建的是一个引用数组。请大家观察封装器类型 Integer，它是一个类，而非原始数据类型：

```
//: c04:ArrayClassObj.java
// Creating an array of nonprimitive objects.
import java.util.*;

public class ArrayClassObj {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    Integer[] a = new Integer[pRand(20)];
    System.out.println(
      "length of a = " + a.length);
    for(int i = 0; i < a.length; i++) {
```

```
      a[i] = new Integer(pRand(500));
      System.out.println(
        "a[" + i + "] = " + a[i]);
    }
  }
} ///:~
```

在这儿，甚至要在 new 调用后才开始创建数组：

```
Integer[] a = new Integer[pRand(20)];
```

它只是一个引用数组，而且除非通过创建一个新的 Integer 对象，从而初始化了对象引用，否则初始化工作不会结束：

```
a[i] = new Integer(pRand(500));
```

但如果忘了创建对象，那么到了运行时间后，一旦试图读取空数组的位置，就会获得一个“违例”错误。

下面让我们看看打印语句中 String 对象的构成情况。大家可看到指向 Integer 对象的引用会自动转换，从而产生一个 String，它代表着位于对象内部的值。

亦可用花括号封闭列表来初始化对象数组。这里可采用两种形式：

```
//: 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),
    };
  }
} ///:~
```

这种做法大多数时候都很有用，但受到的限制也是最大的，因为数组的大小是在编译时决定的。初始化列表的最后一个逗号是可选的（这便使长列表的维护变得更加容易）。

数组初始化的第二种形式提供了一种更简便的语法，可让我们创建和调用方法，获得与 C 的“变量参数列表”（C 通常把它简称为“变参表”一即 varargs）一致的效果。这些效果包括未知的参数（参数）数量以及未知的类型。由于所有类最终都是从统一的根类 Object 继承来的，所以我们能创建一个方法，令其取得一个 Object 数组，然后象下面这样调用它：

```
//: c04:VarArgs.java
// Using the array syntax to create
// variable argument lists.

class A { int i; }

public class VarArgs {
  static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    f(new Object[] {
       new Integer(47), new VarArgs(),
       new Float(3.14), new Double(11.11) });
    f(new Object[] {"one", "two", "three" });
    f(new Object[] {new A(), new A(), new A()});
  }
} ///:~
```

此时，我们对这些未知的对象并不能采取太多的操作，而且这个程序利用自动 String 转换来对每个 Object 做一些有用的事情。在第 12 章（讨论运行时间类型标识或 RTTI），大家还会学到如何查出这类对象的准确类型，使自己能对它们采取一些更有趣的操作。

4.5.1 多维数组

在 Java 里可以方便地创建多维数组：

```
//: c04:MultiDimArray.java
// Creating multidimensional arrays.
import java.util.*;

public class MultiDimArray {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  static void prt(String s) {
    System.out.println(s);
  }
  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++)
    prt("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++)
      prt("a2[" + i + "][" +
          j + "][" + k +
          "] = " + a2[i][j][k]);
// 3-D array with varied-length vectors:
int[][][] a3 = new int[pRand(7)][][];
for(int i = 0; i < a3.length; i++) {
  a3[i] = new int[pRand(5)][];
  for(int j = 0; j < a3[i].length; j++)
    a3[i][j] = new int[pRand(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++)
      prt("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++)
    prt("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++)
        prt("a5[" + i + "][" + j +
            "] = " + a5[i][j]);
  }
} ///:~
```

在用来打印的代码里，使用了 length，所以它不必依赖一个固定的数组大小。

第一个例子展示了由原始数据类型构成的一个多维数组。我们可用花括号定出数组内每个矢量的边界：

```
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
```

每对方括号都将我们移至数组的下一级。

第二个例子展示了用 new 分配的一个三维数组。在这里，整个数组都是立即分配好的：

```
int[][][] a2 = new int[2][2][4];
```

但第三个例子却向大家揭示出构成矩阵的每个矢量都可以有任意的长度：

```
    int[][][] a3 = new int[pRand(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
    }
```

对于第一个 new 创建的数组来说，它的第一个元素的长度是随机的，其他元素的长度则没有定义。for 循环内的第二个 new 则会填写元素，但保持第三个索引的未定状态——直到碰到第三个 new。

根据输出结果，大家可以看到：假若没有明确指定初始化值，数组值就会自动初始化成零。

可用类似方式处理非基类型对象数组。这从第四个例子可以看出，它向我们演示了用花括号收集多个 new 表达式的能力：

```
    Integer[][] a4 = {
      { new Integer(1), new Integer(2)},
      { new Integer(3), new Integer(4)},
      { new Integer(5), new Integer(6)},
    };
```

第五个例子则展示了如何一点一点地构建非基类型的对象数组：

```
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 只是在 Integer 里置了一个有趣的值。

4.6 总　　结

作为初始化的一种具体操作形式，构造函数应使大家明确感受到在语言中进行初始化的重要性。与 C++的程序设计一样，判断一个程序效率如何，关键是看是否由于变量的初始化不正确而造成了严重的编程错误（臭虫）。这些形式的错误很难发现，而且类似的问题也适用于不正确的清除或收尾工作。由于构造函数使我们能保证正确的初始化和清除（若没有正确的构造函数调用，编译器不允许对象创建），所以能获得完全的控制权和安全性。

在 C++中，与“构建”相反的“破坏”（Destruction）工作也是相当重要的，因为用 new 创建的对象必须明确地清除。在 Java 中，垃圾收集器会自动为所有对象释放内存，所以 Java 中等价的清除方法并不是经常都需要用到的。如果不需要类似于构造函数的行为，Java 的垃圾收集器可以极大简化编程工作，而且在内存的管理过程中带来更大的安全性。有些垃圾收集器甚至能清除其他资源，比如图形和文件引用等。然而，垃圾收集器确实也增大了运行时间的开销。但这种开销到底造成了多大的影响却是很难看出的，因为到目前为止，Java 解释器的总体运行速度仍然是比较慢的。随着这一情况的改观，我们应该能判断出垃圾收集器的开销是否使 Java 不适合做一些特定的工作（其中一个问题是垃圾收集器具有行为不可预测的特性）。

由于所有对象都肯定能正确构建，所以同这儿讲到的相比，构造函数实际还能做更多的事情。特别地，当我们通过“合成”或“继承”生成新类的时候，对“肯定构建”的保证仍然有效，同时另外一些语法来提供对它的支持。大家将在以后的章节里详细了解创作、继承以及它们对构造函数造成的影响。

4.7 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 用默认构造函数创建一个类（没有参数或参数），用它任意打印一条消息。然后，创建属于这个类的一个对象。

(2) 在练习 1 的基础上增加一个重载的构造函数，令其采用一个 String 参数，并和你的消息一道打印出来。

(3) 以练习 2 创建的类为基础，创建属于它的、由对象引用构成的一个数组，但不要实

际创建对象并分配到数组里。运行程序时，注意是否会打印出来自构造函数调用的初始化消息。

(4) 实际创建同引用数组联系起来的对象，最终完成练习 3。

(5) 创建一个由 String 对象构成的数组，为每个元素都分配一个字串。用一个 for 循环打印出数组。

(6) 用一个重载的 bark()方法创建一个名叫 Dog 的类。这个方法应当根据不同的原始数据类型进行重载。然后，打印出狗不同类型的吠叫（Barking）、咆哮（howling）等等——具体由调用的那个重载版本决定。写一个 main()，用它调用所有不同的版本。

(7) 修改练习(6)，使两个重载的方法有两个参数（属于两种不同的类型），但相互间顺序相反。检查它能不能工作。

(8) 创建一个没有构造函数的类，再在 main()中创建该类的一个对象，核查默认构造函数是否能自动同步。

(9) 创建一个类，其中带有两个方法。在第一个方法内，调用第二个方法两次：第一次不用 this，第二次用 this。

(10) 创建一个类，其中有两个（重载的）构造函数。利用 this，在第一个构造函数中调用第二个构造函数。

(11) 用一个 finalize()方法创建一个类，用它打印一条消息。在 main()中，创建该类的一个对象。请解释程序的行为。

(12) 修改练习(11)，保证 finalize()无论如何都会被调用。

(13) 创建一个名为 Tank 的类，它可以填充，也可以清空。同时指定一个“死亡条件”，在对象清除之后，必须把它清空。写一个 finalize()，用它检查该死亡条件是否满足。在 main()中，测试在实际使用 Tank 时，可能遇到的每一种情况。

(14) 创建一个类，在其中包含一个 int 和一个 char，均未初始化。然后打印出它们的值，检查 Java 是否执行了默认初始化。

(15) 创建一个类，在其中包含一个未初始化的字串引用。实际体验 Java 会把这个引用初始化成 null。

(16) 创建一个类，在其中包含两个 String 字段。一个在定义时便完成初始化，另一个由构造函数进行初始化。两者有什么不同？

(17) 创建一个类，在其中包含两个 static String 字段。一个在定义时便完成初始化，另一个由 static 块进行初始化。增加一个 static 方法，用它打印这两个字段，并演示它们在使用之前，均已完成了初始化。

(18) 创建一个类，其中一个 String 用“实例初始化”手段完成自己的初始化。请举出该特性的一项实际应用（除本书提到的之外）。

(19) 写一个方法，用它创建并初始化一个由 double（双精度）值构成的数组。数组的大小由该方法的一个参数决定，而元素初始值是一个由起始值和结束值（也是该方法的参数）限定的取值范围。创建另一个方法，用它打印由第一个方法生成的数组。在 main()中，创建和打印几个不同大小的数组，对自己的方法加以测试。

(20) 把练习(19)修改成一个三维数组。

(21) 找到本章的 ExplicitStatic.java 例子，找到标记为(1)的那一行，把它变成注释，验证静态初始化从句此时并没有调用。接下来，取消对标记为(2)的某一行的注释，把它变成实际的代码，验证静态初始化从句会被调用。接下来，再撤消对标记为(2)的另一行的注释，把它变成实际的代码，验证静态初始化只会进行一次。

(22) 用 Garbage.java 做一些实验，具体做法是组合 gc、finalize 或 all 这几个参数来运行程序。重复这项操作，直至最终看出程序输出的某些规律。修改代码，使

System.runFinalization()在 System.gc()之前就得到调用，并观察输出。

第 5 章 隐藏实现

进行面向对象的设计时，一项基本的着眼点是：如何将要发生变化的东西与保持不变的东西区分开。

这一点对于库来说是特别重要的。那个库的用户（客户程序员）必须能依赖自己能用的那一部分完成工作，并希望即使有库的新版本发布，自己也不需要修改原来的代码。而与此相反，库的创建者必须能自由地进行修改与改进，同时保证客户程序员的代码不会受到那些变动的影响。

为达到这个目的，需遵守一定的约定或规则。例如，库程序员在修改库内的一个类时，必须保证不删除已有的方法，因为那样做会造成客户程序员代码出现断点。然而，相反的情况却是令人痛苦的。对于一个数据成员，库的创建者怎样才能知道哪些数据成员已受到客户程序员的访问呢？若方法是某个类唯一的部分，而且并不打算提供给客户程序员直接使用，那么同样会遇到这个问题。假如库的创建者想删除一种旧有的实现，并置入新代码，此时又该怎么办呢？对那些成员进行的任何改动都有可能“废”掉客户程序员的代码。所以库的创建者处在一个尴尬的境地，似乎根本动弹不得。

为了解决这个问题，Java 推出了“访问指示符”的概念，允许库创建者声明哪些东西是客户程序员可以使用的，哪些则是不可使用的。这种访问控制的级别在“最大访问”和“最小访问”的范围之间，分别包括：public、“友好的”（无关键字）以及 private。根据前一段的描述，大家此时或许已总结出：作为一名库设计者，应将所有东西都尽可能保持为“private”（私有），并只展示出那些想让客户程序员使用的方法。这种思路是完全正确的，尽管它有点儿违背那些用其他语言（特别是 C）编程的人的直觉，那些人习惯于在没有限制的情况下访问任何东西。不过到这一章结束的时候，大家应该能够深刻体验出 Java 访问控制的价值。

然而，组件库以及控制谁能访问那个库的组件的概念现在仍不是完整的。仍存在这样一个问题：如何将组件绑定到单独一个统一的库单元里。这是通过 Java 的 package（封装）关键字来实现的，而且访问指示符要受到类在相同的封装还是在不同的封装里的影响。所以在本章开头，大家首先要学习库组件是如何放在封装里的。然后才能真正理解访问指示符的完整含义。

5.1 封装：库单元

我们用 import 关键字导入一个完整的库时，就会获得一个“封装”（Package）。例如：

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

它的作用是导入完整的实用工具（Utility）库，该库属于标准 Java 开发工具包的一部分。举个例子来说，由于 ArrayList 位于 java.util 内，所以现在要么可以指定一个完整名称“java.util.ArrayList”（可省略 import 语句），要么简单地指定一个“ArrayList”（由于已经有了 import）。

若想导入单独一个类，可在 import 语句里指定那个类的名字：

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

现在，我们就可自由地使用 ArrayList。然而，java.util 中的其他任何类仍是不可用的。

之所以要进行这样的导入，是为了提供一种特殊的机制，以便管理“命名空间”（Name

Space）。我们所有类成员的名字相互间都会隔离起来。位于类 A 内的一个方法 f()不会与位于类 B 内的、拥有同样“签名”（参数列表）的 f()发生冲突。但类名会不会冲突呢？假设我们创建了一个 stack 类，把它安装到已经装了一个 stack 类（其他人写的）的机器上，这时会出现什么情况呢？对于因特网中的 Java 应用，这种情况会在用户毫无知觉的时候发生，因为运行一个 Java 程序的时候，类就会自动下载。

正是名字存在潜在的冲突，所以特别有必要对 Java 中的命名空间进行全面控制，而且需要创建一个完全独一无二的名字，无论因特网存在什么样的限制。

迄今为止，本书的大多数例子都仅存在于单个文件中，而且设计成局部（本地）使用，根本就没有牵涉到封装名的问题（在这种情况下，类名实际是放在“默认封装”内）。这是一种有效的做法，而且为使问题简化，本书余下的部分也会尽可能地采用它。然而，若计划创建一个“对因特网友好”或者说“适合在网上使用”的程序，就必须考虑如何防止类名的重复。

为 Java 创建一个源码文件的时候，它通常叫作一个“编译单元”（有时也叫作“翻译单元”）。每个编译单元都必须有一个以.java 结尾的名字。而且在编译单元的内部，可以有一个公共（public）类，它必须拥有与文件相同的名字（包括大小写形式，但“.java”这个文件扩展名除外）。如果不这样做，编译器就会报错。每个编译单元内都只能有一个 public 类（同样地，否则编译器会报错）。编译单元剩下的其他类（如果有的话）就在那个封装外面的世界面前隐藏起来，因为它们并非“公共”的，而且它们包含了用于主 public 类的“支撑”类。

编译一个.java 文件时，我们会获得一个名字完全相同的输出文件，只是针对.java 文件中的每个类，它们都有一个.class 扩展名。因此，我们在少量.java 文件的基础上，最终有可能获得大量.class 文件。如以前用一种汇编语言写过程序，那么可能已习惯编译器先分割出一种过渡形式（通常是一个.obj 文件），再用一个链接器将其与其他东西封装到一起（生成一个可执行文件），或者与一个库封装到一起（生成一个库）。但那并不是 Java 的工作方式。在 Java 中，一个有效的程序就是一系列.class 文件，它们可以封装和压缩到一个 JAR 文件里（使用 Java 提供的 jar 工具）。Java 解释器则负责对这些文件的查找、装载和解释[29]。

“库”也由一系列类文件构成。每个文件都有一个 public 类（并没强迫使用一个 public 类，但这种情况是最典型的），所以每个文件都有一个组件。如果想将所有这些组件（在它们各自独立的.java 和.class 文件里）都归纳到一起，那么 package 关键字就可以发挥作用。

若在一个文件的开头使用下述代码：

```
package mypackage;
```

那么 package 语句必须作为文件的第一个非注释语句出现。该语句的作用是指出这个编译单元属于名为 mypackage 的一个库的一部分。或者换句话说，它表明这个编译单元内的 public 类名位于 mypackage 这个名字的下面。如果其他人想使用这个名字，要么指出完整的名字，要么与 mypackage 联合使用 import 关键字（使用前面给出的选项）。注意根据 Java 封装的约定，名字的所有字母都应小写，甚至那些中间单词亦要如此。

例如，假定文件名是 MyClass.java。它意味着在那个文件有一个、而且只能有一个 public 类。而且那个类的名字必须是 MyClass（包括大小写形式）：

```
package mypackage;
public class MyClass {
```

[29] Java 并没有强制一定要用解释器。一些固有代码的 Java 编译器可生成一个单一的可执行文件。

```
  // . . .
```

现在，如果有人想使用 MyClass，或者想使用 mypackage 内的其他任何 public 类，就必须用 import 关键字激活 mypackage 内的名字，使它们能够使用。另一个办法则是指定完整的名称：

```
mypackage.MyClass m = new mypackage.MyClass();
```

import 关键字则可将其变得简洁得多：

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

作为一名库设计者，一定要记住 package 和 import 关键字允许我们做的事情就是对个全局的命名空间进行分割，保证不会出现名字冲突的情况——无论有多少人在网上，也无论有多少人用 Java 编写自己的类。

5.1.1 创建独一无二的封装名

大家或许已注意到这样一个事实：由于一个封装永远不会真的“打包”到单独一个文件里面，它可由多个.class 文件构成，所以局面可能稍微有些混乱。为避免这个问题，最合理的一种做法就是将某个特定封装使用的所有.class 文件都放到单个目录里。也就是说，我们要利用操作系统的分级文件结构避免出现混乱局面。而这正是 Java 所采取的办法之一；以后在讲到 jar 工具的时候，大家还会学到另一个办法。

将封装文件收集到一个子目录里，同时也解决了另外两个问题：创建独一无二的封装名以及找出那些可能深藏于目录结构某处的类。正如我们在第 2 章讲述的那样，为达到这个目的，需要将.class 文件的位置路径编码到 package 的名字里。但根据约定，编译器强迫 package 名的第一部分是类创建者的因特网域名。由于因特网域名肯定是独一无二的，所以假如按这一约定行事，package 的名称就肯定不会重复，所以永远不会遇到名字冲突的问题。换句话说，除非将自己的域名转让给其他人，而且对方也按照相同的路径名编写 Java 代码，否则名字的冲突是永远不会出现的。当然，如果你没有自己的域名，那么必须创造一个非常生僻的封装名（例如自己的英文姓名），以便尽最大可能创建一个独一无二的封装名。如决定发行自己的 Java 代码，那么强烈推荐去申请自己的域名，它所需的费用是相当低廉的。

这个技巧的另一部分是将 package 名解析成自己机器上的一个目录。这样一来，Java 程序运行并需要装载.class 文件的时候（这是动态进行的，在程序需要创建属于那个类的一个对象，或者首次访问那个类的一个 static 成员时），它就可以找到.class 文件所在的那个目录。

Java 解释器的工作程序如下：首先，它找到环境变量 CLASSPATH（将 Java 或者具有 Java 解释能力的工具——如浏览器——安装到机器中时，通过操作系统进行设定）。CLASSPATH 包含了一个或多个目录，它们作为一种特殊的“根”使用，从这里展开对.class 文件的搜索。从那个根开始，解释器会寻找包名，并将每个点号（句点）替换成一个斜杠，从而生成从 CLASSPATH 根开始的一个路径名（所以 package foo.bar.baz 会变成 foo\bar\baz 或者 foo/bar/baz；具体是正斜杠还是反斜杠由操作系统决定）。随后将它们连接到一起，成为 CLASSPATH 内的各个条目（入口）。以后搜索.class 文件时，就可从这些地方开始查找与准备创建的类名对应的名字。此外，它也会搜索一些标准目录——这些目录与 Java 解释器驻留的地方有关。

为进一步理解这个问题，下面以我自己的域名为例，它是 bruceeckel.com。将其反转过

来后，com.bruceeckel 就为我的类创建了独一无二的全局名称（com、edu、org、net 等扩展名以前在 Java 封装中都是大写的，但自 Java 2 以来，这种情况已发生了变化。现在整个包名都是小写的）。由于决定创建一个名为 util 的库，我可以进一步扩展它，最后得到的封装名如下：

```
package com.bruceeckel.simple;
```

现在，可将这个封装名作为下述两个文件的“命名空间”使用：

```
//: com:bruceeckel:simple:Vector.java
// Creating a package.
package com.bruceeckel.simple;

public class Vector {
  public Vector() {
    System.out.println(
      "com.bruceeckel.util.Vector");
  }
} ///:~
```

创建自己的封装时，要求 package 语句必须是文件中的第一个“非注释”代码。第二个文件表面看起来是类似的：

```
//: com:bruceeckel:simple:List.java
// Creating a package.
package com.bruceeckel.simple;

public class List {
  public List() {
    System.out.println(
      "com.bruceeckel.util.List");
  }
} ///:~
```

这两个文件都置于我自己系统的一个子目录中：

```
C:\DOC\JavaT\com\bruceeckel\simple
```

若通过它往回走，就会发现封装名 com.bruceeckel.util，但路径的第一部分又是什么呢？这是由 CLASSPATH 环境变量决定的。在我的机器上，它是：

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

可以看出，CLASSPATH 里能包含大量备用的搜索路径。然而，使用 JAR 文件时要注意一个问题：除了它所在路径之外，还要将 JAR 文件的名字放到类路径里。所以对一个名为 grape.jar 的 JAR 文件来说，我们的类路径需要包括：

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

正确设好类路径后，便可将下面这个文件放在任何目录里：

```
//: c05:LibTest.java
// Uses the library.
import com.bruceeckel.simple.*;

public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} ///:~
```

编译器遇到 import 语句后，它会搜索由 CLASSPATH 指定的目录，查找子目录 com\bruceeckel\util，然后查找有恰当名字的已编译文件（对于 Vector 来说是 Vector.class，对于 List 则是 List.class）。注意 Vector 和 List 内无论类还是需要的方法都必须设为 public。

对 Java 新手来说，类路径的设置多少有些麻烦（特别是我，在我刚开始学 Java 的时候，它显得特别“烦人”）。为此，从 Java 2 开始，Sun 让 JDK 稍微简化了这方面的工作，把它变得更“聪明”了一些。大家不妨注意一下，在你安装 JDK 的时候，即使没有设置 CLASSPATH，也能正常编译和运行基本的 Java 程序。不过，要想编译和运行本书用到的源代码封装（配套光盘或 www.BruceEckel.com 处都有），仍然需要对自己的 CLASSPATH 作一番修改（源码封装里已提供了具体解释）。

冲突

若通过*导入了两个库，而且它们包括相同的名字，这时会出现什么情况呢？例如，假定一个程序使用了下述导入语句：

```
import com.bruceeckel.simple.*;
import java.util.*;
```

由于 java.util.*也包含了一个 Vector 类，所以这会造成潜在的冲突。然而，只要冲突并不真的发生，那就不会产生任何问题——这当然是最理想的情况，因为否则的话，就需要进行大量编程工作，防范那些可能可能永远也不会发生的冲突。

现在试着生成一个 Vector，. 那么肯定会发生冲突。如下所示：

```
Vector v = new Vector();
```

它引用的到底是哪个 Vector 类呢？编译器对这个问题没有答案，读者也不可能知道。所以编译器会报告一个错误，强迫我们进行明确的说明。例如，假设我想使用标准的 Java Vector，那么必须象下面这样编程：

```
java.util.Vector v = new java.util.Vector();
```

由于它（与 CLASSPATH 一起）完整指定了那个 Vector 的位置，所以不再需要 import java.util.*语句，除非你还想使用来自 java.util 的其他东西。

5.1.2 自定义工具库

掌握前述的知识后，接下来就可以开始创建自己的工具库，以便减少或者完全消除重复性代码。例如，可为 System.out.println()创建一个别名，减少重复键入的代码量。它可以是名为 tools 的一个封装的一部分：

```
//: 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);
  }
} ///:~
```

可用它打印一个字串，既可要一个新行（P.rintln()），亦可不要新行（P.rint()）。

大家可能已经想到了，该文件所在的目录必须从某个 CLASSPATH 位置开始，然后继续 com/bruceeckel/tools。编译完毕后，利用一个 import 语句，即可在自己系统内的任何地方使用 P.class 文件。如下所示：

```
//: c05:ToolTest.java
// Uses the tools library.
import com.bruceeckel.tools.*;

public class ToolTest {
  public static void main(String[] args) {
    P.rintln("Available from now on!");
    P.rintln("" + 100); // Force it to be a String
    P.rintln("" + 100L);
    P.rintln("" + 3.14159);
  }
} ///:~
```

注意，所有对象都可被强制成字串（String）表示，方法是将它们放在一个 String 表达式中。在上面的例子中，用一个空 String 作为表达式的开头，便可达到目的。但这样做也会产生一个有趣的效果。假如调用 System.out.println(100)，那么即使不把它强制转型为 String，仍然可以正常工作。进行一些额外的重载操作，就可让 P 类做同样的事情（这会成为本章最后的一道练习题）

所以从现在开始，无论什么时候只要做出了一个有用的新工具，就可将其加入 tools 目录（或者自己的个人 util 或 tools 目录）。

5.1.3　利用导入改变行为

Java 已取消的一种特性是 C 的“条件编译”，它允许我们改变参数，获得不同的行为，同时不必改变其他任何代码。Java 之所以抛弃了这一特性，可能是由于该特性经常在 C 里用于解决跨平台问题——代码的不同部分根据具体的平台进行编译，否则不能在特定的平台上运行。由于 Java 的设计思想是成为一种自动跨平台的语言，所以这种特性是没有必要的。

然而，条件编译还有另一些非常有价值的用途。一种很常见的用途就是代码调试。调试

特性可在开发过程中使用，但在发行的产品中却无此功能。Alen Holub（www.holub.com）提出了利用封装来模仿条件编译的概念。根据这一概念，它创建了 C“断定机制”一个非常有用的 Java 版本。之所以叫作“断定机制”，是由于我们可以说“它应该为真”或者“它应该为假”。如果语句不同意你的断定，我们马上就可以知道。这种工具在调试过程中显得特别有用。

可用下面这个类进行程序调试：

```
//: com:bruceeckel:tools:debug:Assert.java
// Assertion tool for debugging.
package com.bruceeckel.tools.debug;

public class Assert {
  private static void perr(String msg) {
    System.err.println(msg);
  }
  public final static void is_true(boolean exp) {
    if(!exp) perr("Assertion failed");
  }
  public final static void is_false(boolean exp){
    if(exp) perr("Assertion failed");
  }
  public final static void
  is_true(boolean exp, String msg) {
    if(!exp) perr("Assertion failed: " + msg);
  }
  public final static void
  is_false(boolean exp, String msg) {
    if(exp) perr("Assertion failed: " + msg);
  }
} ///:~
```

这个类只是简单地封装了布尔测试。如果失败，就打印出错消息。在第 10 章，大家还会学习一个更高级的错误控制工具，名为“违例控制”。但在目前这种情况下，perr()方法已经可以很好地工作。

输出结果会写入 System.err，从而打印到控制台的“标准错误”流内。

如果想使用这个类，只需在自己的程序中加入下面这一行：

```
import com.bruceeckel.tools.debug.*;
```

如果想把断定功能从自己的程序中剔去，以便能发行最终的代码，我们创建了第二个 Assert 类，但却是在一个不同的封装里：

```
//: com:bruceeckel:tools:Assert.java
// Turning off the assertion output
// so you can ship the program.
package com.bruceeckel.tools;
```

```
public class Assert {
  public final static void is_true(boolean exp){}
  public final static void is_false(boolean exp){}
  public final static void
  is_true(boolean exp, String msg) {}
  public final static void
  is_false(boolean exp, String msg) {}
} ///:~
```

现在，假如将前一个 import 语句变成下面这个样子：

```
import com.bruceeckel.tools.*;
```

程序便不再打印断定消息。下面是个例子：

```
//: c05:TestAssert.java
// Demonstrating the assertion tool.
// Comment the following, and uncomment the
// subsequent line to change assertion behavior:
import com.bruceeckel.tools.debug.*;
// import com.bruceeckel.tools.*;

public class TestAssert {
  public static void main(String[] args) {
    Assert.is_true((2 + 2) == 5);
    Assert.is_false((1 + 1) == 2);
    Assert.is_true((2 + 2) == 5, "2 + 2 == 5");
    Assert.is_false((1 + 1) == 2, "1 +1 != 2");
  }
} ///:~
```

通过改变导入的封装，我们就可将自己的代码从调试版变成最终的发行版。这种技术可应用于任何种类的条件判断代码。

5.1.4　封装的停用

大家应注意这样一个问题：每次创建一个封装后，都在为封装取名的时候间接指定了一个目录结构。这个封装必须放在由它的名字规定的那个目录内。而且这个目录必须能从 CLASSPATH 开始搜索并发现。最开始的时候，package 关键字的运用可能会令人迷惑，因为除非坚持遵守根据目录路径指定包名的规则，否则就会在运行时获得大量莫名其妙的消息，指出找不到一个特定的类——即使那个类明明就在相同的目录中。若得到象这样的一条消息，请试着将 package 语句作为注释标记出去。如果这样做行得通，就可知道问题到底出在哪儿。

5.2 Java 访问指示符

针对类内每个成员的每个定义，poublic、protected 以及 private 这几个 Java 访问指示符都必须放在它们的最前面——无论这些成员是字段还是方法。每个访问指示符都只控制着对那个特定定义的访问。这与 C++存在着显著不同。在 C++中，访问指示符控制着它后面的所有定义，直到又一个访问指示符加入为止。

通过千丝万缕的联系，程序为所有东西都指定了某种访问形式。在后面的小节里，大家要学习与各类访问有关的知识。首次，让我们拿默认访问“开刀”。

5.2.1 “友好的”

如果根本不指定访问指示符，就象本章之前的所有例子那样，那么会出现什么情况呢？默认访问没有关键字，但它通常称为“友好”（Friendly）访问。这意味着当前封装内其他所有类都能访问“友好”成员，但对封装外的所有类来说，这些成员却又是“私有”（Private）的，外界无法访问到它们。由于一个编译单元（一个文件）只可从属于一个封装，所以对一个编译单元内的所有类来说，它们相互间都是默认为“友好”的。因此，我们也说友好的元素拥有“封装访问”的权限（只能在封装的范围内访问）。

友好访问允许我们将相关的类都组合到一个封装里，使它们相互间能方便地沟通。将类组合到一个封装里之后（这样便允许友好成员相互访问，让它们“交朋友”），我们便“拥有”了那个包内的代码。显然，这样做是有道理的，因为只有我们已经“拥有”的代码才能友好地访问我们“拥有”的其他代码。我们可以这样认为：正是由于“友好访问”的存在，才使得把类组合到一个封装里有了意义。在其他许多语言中，我们在文件内对定义进行组织的方式往往显得有些牵强。但在 Java 中，却强制用一种颇有意义的形式进行组织。除此以外，我们有时也可以方便地排除掉一些类，不让它们访问当前在封装内定义的类。

类控制着哪些代码能访问它的成员。没有任何秘诀可以强行访问。来自另一个封装的代码绝对不能跳出来嚷嚷道：“嗨，我是 Bob 的朋友！”，并指望这样就能看到 Bob 类的 protected（受保护的）、友好的以及 private（私有）成员。要想真正访问到一个成员，只有：

(1) 使成员成为“public”（公共的）。这样所有人从任何地方都可以访问它。

(2) 使成员成为“友好”的，取消任何访问指示符，并将其他类放到相同的封装内。这样一来，其他类就可访问到成员。

(3) 如同大家在第 6 章会学到的那样（从那一章开始引入“继承”概念），一个继承的类既可以访问一个“受保护的”成员，也可以访问一个公共成员（但绝不可访问“私有”成员）。只有两个类都在相同的包内，它才能访问友好成员。不过，现在不必关心这方面的问题。

(4) 提供“访问器／变化器”方法（亦称为“获取／设置”方法），以便对值进行读取和修改。这是 OOP 环境最正规的一种做法，也是 JavaBeans 的基础——详情见第 13 章。

5.2.2 public：接口访问

使用 public 关键字时，它的意思是紧随在 public 后面的成员可由所有人使用，特别是以后要用到这个库的客户程序员。假定我们定义了一个名为 dessert（饭后甜品）的封装，其中包含下述编译单元：

```
//: 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"); }
} ///:~
```

请记住，Cookie.java 必须放在名为 dessert 的一个子目录内，而这个子目录又必须位于某个 CLASSPATH 目录的 C05 目录下面（C05 代表本书第 5 章）。不要错误地以为 Java 无论如何都会将当前目录作为搜索的起点看待。如果不将一个“.”包括到 CLASSPATH 里，Java 根本就不会考虑当前目录。

现在，假若我们创建了要用到 Cookie 的一个程序，如下所示：

```
//: c05:Dinner.java
// Uses the library.
import c05.dessert.*;

public class Dinner {
  public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
  }
} ///:~
```

那么就可以创建一个 Cookie 对象，因为它的构造函数是 public 的，而且类也是 public 的（公共类的概念稍后还会详细讨论）。然而，bite()成员不可在 Dinner.java 内访问，因为 bite()只有在 dessert 包内才是“友好”的。

默认封装

大家可能会惊讶地发现下面这些代码居然能顺利编译——尽管表面看起来似乎与规则不符：

```
//: c05:Cake.java
// Accesses a class in a
// separate compilation unit.

class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
```

```
    x.f();
  }
} ///:~
```

在位于相同目录的第二个文件里：

```
//: c05:Pie.java
// The other class.

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

最初可能会把它们看作完全不相干的两个文件，而 Cake 居然能创建一个 Pie 对象，并调用它的 f()方法（注意必须在类路径中设置一个'.'，使文件能顺利编译）！通常会认为 Pie 和 f()是“友好的”，所以为什么居然能由 Cake 调用呢？在这里，它们的确是友好的——我们对这个结论没有异议。但它们之所以仍能在 Cake.java 中使用，是由于它们位于相同的目录中，而且没有指定明确的封装名。Java 把这样的文件看作那个目录的“默认封装”的一部分，所以它们对于目录内的其他文件来说都是“友好”的。

5.2.3 private：你不能碰它！

private 关键字意味着没有谁能访问那个成员——除非由那个特定的类，而且是从那个类的方法里访问。同一个封装内的其他类都不能访问 private 成员。因此，这就好象就连自己都不能访问到那个类！不过另一方面，通常也不会由几个人合作来创建一个封装。所以，private 允许我们自由地修改那个成员，同时毋需关心它是否会影响同一个封装内的另一个类。

默认的“友好”封装访问通常已足够让我们实现代码隐藏；请记住，对于一个封装的用户来说，他们是无法访问到一个“友好”成员的。这个效果非常好，因为我们通常都会采用默认访问（即使忘了设置任何访问控制，也能达到这样的效果）。因此，我们一般只需考虑如何为那些想设成 public 的成员设置访问权限。这样一来，private 这个关键字就变得似乎有些“古怪”——即使没有它，好象也无关痛痒嘛！（这一点与 C++形成鲜明对比。）但是，随着学习的深入，大家就会发现 private 仍然有非常重要的用途，特别是在涉及到多线程的时候（详见第 14 章）。

下面是应用了 private 的一个例子：

```
//: 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();
  }
} ///:~
```

这个例子向大家展示了 private 的方便之处：有时可能想控制对象的创建方式，并防止有人直接访问一个特定的构造函数（或所有构造函数）。在上面的例子中，我们不可通过构造函数来创建一个 Sundae 对象；相反，必须调用 makeASundae()方法来实现[30]。

对一个类来说，只要肯定了一个方法是它的“助手”或“辅助”方法，便可将其设为 private，从而保证自己不会在封装内的其他地方误用到它，从而防范自己不慎更改或删除这个方法。将一个方法的属性设为 private 后，便可一直保证具有这种效果。

对类内的一个 private 字段来说，它的道理是一样的。除非必须将最基层的代码实现暴露出来（这种情况会很少见，我可以保证），否则应将所有字段都设为 private。但是，尽管类内对一个对象的引用是 private 的，但这并不能说明其他对象不能拥有对同一个对象的一个 public 引用（有关“别名”的问题，详见附录 A）。

5.2.4　protected：有点儿“友好”

对 protected（受保护的）访问指示符来说，大家要注意的是，即使没有搞懂本节的知识，仍然可以顺利进行以后的学习，直到“继承”为止（第 6 章）。不过为了保持本章内容的完整性，这里仍然对它进行了简要描述，并提供了相关的例子。

protected 关键字为我们引入了一种名为“继承”的概念，它以一个现成的类为基础，可在其中加入新的成员，同时不会对原来的类产生影响——我们将这种现有的类称为“基类”或者“基本类”（Base Class）。亦可改变那个类现有成员的行为。要想从一个现有的类继承，我们要声明自己的新类“扩展”（extends）了一个现有的类：

```
class Foo extends Bar {
```

类定义其余的部分看起来是完全相同的。

若新建一个封装，并从另一个封装的某个类继承，那么唯一能够访问的成员就是原来那个封装的 public 成员。当然，如果在相同的封装里进行继承，就可以获得对所有“友好”成员的封装访问权限。有些时候，基类的创建者喜欢提供一个特殊的成员，允许它访问派生类，但不允许访问“整个世界”。这正是 protected 的工作。回过头去看看 Cookie.java 文件，大家就会发现下面这个类不能访问“友好”成员：

```
//: c05:ChocolateChip.java
// Can't access friendly member
// in another class.
import c05.dessert.*;
```

[30] 此时还会产生另一个效果：由于默认构造函数是唯一获得定义的，而且它的属性是 private，所以可防止对这个类的继承（这是第 6 章要重点讲述的主题）。

```
public class ChocolateChip extends Cookie {
  public ChocolateChip() {
   System.out.println(
     "ChocolateChip constructor");
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    //! x.bite(); // Can't access bite
  }
} ///:~
```

对于继承，一个有趣的地方值得注意：假若方法 **bite( )**存在于 **Cookie** 类中，那么它也会存在于从 **Cookie** 继承的所有类中。但由于 **bite( )**在外部的封装面前是“友好”的，所以我们不能使用它。当然，亦可将其变成 **public**。但这样一来，由于所有人都能自由访问它，所以可能并非我们所希望的局面。如果象下面这样修改了 **Cookie** 类：

```
public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
}
```

那么仍然能在 **dessert** 封装里“友好”地访问 **bite( )**；另一方面，它同时也能由自 **Cookie** 继承的任何类访问。然而，它此时并非“**public**”（公共）的。

5.3 接口与实现

我们通常把访问控制叫作“隐藏实现”。将数据和方法组合到类里，同时加上实现的隐藏，我们把这一过程统一称作“封装”[31]。结果得到的就是一个特殊的数据类型，它有自己的特征与行为。

考虑到两方面重要目的，访问控制在一个数据类型里规定了“边界”。第一个目的是规定客户程序员哪些能够使用，哪些不能。这样一来，我们就可在结构里构建起坚不可摧的“堡垒”，不必担心客户程序员误将其当作接口的一部分使用。

这个目的直接导致了我们的第二个目的：将接口同实现区分开。假定结构以后会在一系列程序中使用，但由于用户除了将消息发给 public 接口之外，不能做其他任何事情，所以我们就可改变不属于 public 的所有东西（如“友好的”、protected 以及 private），同时不要求用户对他们的代码作任何修改。

[31] 然而，人们通常也把实现的隐藏单独叫作“封装”。换句话说，“封装”其实是个很广义的概念。它既能作动词，也能作名词。有时还有“打包”一说。

现在，我们是在一个面向对象的编程环境中，其中的一个类（class）实际指的是“一类对象”，就象我们说“鱼类”或“鸟类”那样。从属于这个类的所有对象都共享一系列相同的特征与行为。“类”是对这一类所有对象的外观及行为进行的一种描述。

在一些早期 OOP 语言中，如 Simula-67，关键字 class 的作用是描述一种新的数据类型。同样的关键字在大多数面向对象的编程语言里都得到了应用。它其实是整个语言的重点：需要新建数据类型的场合比那些只是用于容纳数据和方法的“容器”多得多。

在 Java 中，类是最基本的 OOP 概念。它是本书未采用粗体印刷的关键字之一——如果“class”的数量太多，会造成页面排版的严重混乱。

为清楚起见，可考虑用特殊的样式创建一个类：将 public 成员置于最开头，后面跟随 protected、友好以及 private 成员。这样做的好处是类的使用者可从上向下依次阅读，并首先看到对自己来说最重要的内容（即 public 成员，因为它们可从文件的外部访问），并在遇到非公共成员后停止阅读，后者已经进入内部实现的“地界”了：

```
public class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
  public void pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

由于接口和实现仍然混合在一起，所以只是部分容易阅读。也就是说，仍然能看到源码（实现），因为它们需要保存在类里面。除此以外，由于我们可利用 javadoc（已在第 2 章介绍）来支持注释文档，它从另一方面使代码的可读性显得并不那么重要。要想向一个类的使用者显示出接口，这实际是“类浏览器”的工作。这种工具能查找所有可用的类，总结出可对它们采取的全部操作（比如可使用哪些成员等），并用一种清爽悦目的形式显示出来。到大家读到这本书的时候，所有优秀的 Java 开发工具都应推出了自己的“类浏览器”。

5.4　类 访 问

在 Java 中，亦可用访问指示符判断出一个库内的哪些类可由那个库的用户使用。若希望一个类能由客户程序员调用，可在类主体的起始花括号前面某处放置一个 public 关键字。它决定客户程序员是否能创建属于这个类的一个对象。

为控制一个类的访问，指示符必须在关键字 class 之前出现。所以我们能够使用：

```
public class Widget {
```

也就是说，假设我们的库名是 mylib，那么所有客户程序员都能访问 Widget——通过下述语句：

```
import mylib.Widget;
```

或者

```
import mylib.*;
```

然而，我们同时还要注意到另一些限制：

(1) 每个编译单元（文件）都只能有一个 public 类。每个编译单元有一个公共接口的概念是由那个公共类表达出来的。根据自己的需要，它可拥有任意多个提供支撑的“友好”类。但若在一个编译单元里使用了多个 public 类，编译器就会向我们提示一条出错消息。

(2) public 类的名字必须与包含了编译单元的那个文件的名字完全相符，甚至包括它的大小写形式。所以对于 Widget 来说，文件的名字必须是 Widget.java，而不应是 widget.java 或者 WIDGET.java。同样地，如出现不符，就会报告一个编译时间错误。

(3) 可能（但并常见）有一个编译单元根本没有任何公共类。此时，可按自己的意愿任意指定文件名。

如果已获得了 mylib 内部的一个类，准备用它完成由 Widget 或者 mylib 内部的其他某些 public 类执行的任务，此时又会出现什么情况呢？我们不希望花费力气为客户程序员编制文档，并预计以后某个时候也许会进行大幅修改，并将自己的类一起删掉，换成另一个不同的类。为获得这种灵活处理的能力，需要保证没有客户程序员能够依赖自己隐藏于 mylib 内部的特定实现。为达到这个目的，只需将 public 关键字从类中剔除即可，这样便把类变成了“友好的”（类仅能在那个封装内使用）。

注意不可将类设成 private（那样会使除类之外的其他东西都不能访问它），也不能设成 protected[32]。因此，我们现在对于类的访问只有两个选择：“友好的”或者 public。若不愿其他任何人访问那个类，可将所有构造函数都设为 private。这样一来，在类的一个 static 成员内部，除自己之外的其他所有人都无法创建属于那个类的一个对象[33]。如下例所示：

```
//: 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() {}
}
```

[32] 实际上，内部类既可以是“受保护的”，也可以是“私有的”，但那属于特别情况。第 7 章会详细解释这个问题。

[33] 亦可通过从那个类继承（第 6 章）来实现。

```
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();
  }
} ///:~
```

迄今为止，我们创建过的大多数方法都是要么返回 void，要么返回一个原始数据类型。所以对下述定义来说：

```
  public static Soup access() {
    return ps1;
  }
```

它最开始多少会令人感到迷惑。位于方法名（access）前的单词指出方法到底返回什么。在这之前，我们看到的都是 void，它意味着“什么也不返回”（void 在英语里是“虚无”的意思）。但亦可返回指向一个对象的引用，此时出现的就是这个情况。该方法返回一个引用，它指向 Soup 类的一个对象。

Soup 类向我们展示出如何通过将所有构造函数都设为 private，从而防止直接创建一个类。请记住，假若不明确地至少创建一个构造函数，就会自动创建默认构造函数（没有参数）。若自己编写默认构造函数，它就不会自动创建。把它变成 private 后，就没人能为那个类创建一个对象。但别人怎样使用这个类呢？上面的例子为我们揭示出了两个选择。第一个选择，可创建一个 static 方法，再通过它创建一个新的 Soup，然后返回指向它的一个引用。如果想在返回前对 Soup 进行一些额外的操作，或者想了解准备创建多少个 Soup 对象（有可能是为了限制它们的个数），这种方案无疑是特别有用的。

第二个选择是采用“设计范式”（Design Pattern）技术——详见《Thinking in Patterns with Java》一书，可从 www.BruceEckel.com 下载。这时需要采取一种叫做“独身”（Singleton）的范式，因为它仅允许创建一个对象。Soup 类的对象被创建成 Soup 的一个 static private 成员，所以有一个而且只能有一个。除非通过 public 方法 access()，否则根本无法访问它。

正如早先指出的那样，如果不针对类的访问设置一个访问指示符，那么它会自动默认为“友好的”。这意味着那个类的对象可由封装内的其他类创建，但不能从封装外创建。请记住，对于相同目录内的所有文件，假如没有明确地进行 package 声明，那么它们都默认为那个目录的默认封装的一部分。然而，若那个类一个 static 成员的属性是 public，那么客户程序员仍然能够访问那个 static 成员——即使它们不能创建属于那个类的一个对象。

5.5 总 结

对于任何关系，最重要的一点都是事先规定好所有方面都必须遵守的界限或规则。创建一个库时，相当于建立了同那个库的用户（即“客户程序员”）的一种关系——那些用户也是程序员，他们可能用我们的库自行构建一个应用程序，或者用我们的库构建一个更大的库。

没有规矩，不成方圆。假如不订出规则，客户程序员就能随心所欲操作一个类的所有成员，无论我们本来愿不愿意其中的一些成员被直接操作。所有东西在别人面前都会暴露无遗！

本章讲述了如何构建类，从而制作出理想的库。首先，我们讲述如何将一组类封装到一个库里。其次，我们讲述类如何控制对自己成员的访问。

一般情况下，一个 C 程序项目会在 50K 到 100K 行代码之间的某个地方开始莫名其妙地出错。这是由于 C 仅有一个“命名空间”，所以纷繁的名字会开始互相抵触，从而造成额外的管理性开销。而在 Java 中，package 关键字、封装命名方案以及 import 关键字为我们提供对名字的完全控制，所以命名冲突的问题可以很轻易地避免。

有两方面的原因要求我们控制对成员的访问。第一个是防止用户接触那些他们不应碰的工具。对于数据类型的内部机制，那些工具是必需的。但它们并不属于用户接口的一部分，用户不必用它来解决自己的特定问题。所以将方法和字段变成“私有”（private）之后，可极大方便用户。因为他们能轻易看出哪些对于自己来说是最重要的，以及哪些是自己需要忽略的。这样便简化了用户对一个类的理解。

进行访问控制的第二个、也是最重要的一个原因是：允许库设计者改变类的内部工作机制，同时不必担心它会对客户程序员产生什么影响。最开始的时候，可用一种方法构建一个类，后来发现需要重新构建代码，以便达到更快的速度。如接口和实现早已进行了明确的分隔与保护，就可以轻松地达到自己的目的，不用强迫用户改写他们的代码。

利用 Java 中的访问指示符，可有效控制类的创建者。那个类的用户可确切知道哪些是自己能够使用的，哪些则是可以忽略的。但更重要的一点是，它可确保没有任何用户能依赖一个类的基础实施机制的任何部分。作为一个类的创建者，我们可自由修改基础的实现，这一改变不会对客户程序员产生任何影响，因为他们不能访问类的那一部分。

有能力改变基础的实现后，除了能在以后改进自己的设置之外，也同时拥有了“犯错误”的自由。无论当初计划与设计时有多么仔细，仍然有可能出现一些失误。由于知道自己能相当安全地犯下这种错误，所以可以放心大胆地进行更多、更自由的试验。这对自己编程水平的提高是很有帮助的，使整个项目最终能更快、更好地完成。

一个类的公共接口是所有用户都能看见的，所以在进行分析与设计的时候，应尽量保证它的准确性。不过也不必过于紧张，少许的误差仍是允许的。若最初设计的接口存在少许问题，可考虑添加更多的方法，只要保证不删除客户程序员正在他们的代码里使用的东西。

5.6 练 习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 写一个程序，用它创建一个 ArrayList 对象，同时不明确导入 java.util.*。

(2) 在“封装：库单元”一节内，将 mypackage 代码段转变成一个可以编译和运行的 Java 文件集。

(3) 在“冲突”一节内，取出其中的代码段，把它们转变成一个程序，并验证是否真的

产生冲突。

(4) 修改类 P，增大它的适用范围。具体做法是增加 rint()和 rintln()的所有重载版本，以对所有不同的基本 Java 类型进行控制。

(5) 改变 TestAssert.java 中的导入语句，允许和禁止断定机制。

(6) 用 public、private、protected 以及“友好的”数据成员及方法成员创建一个类。创建这个类的一个对象，并观察在试图访问所有类成员时会获得哪种类型的编译器错误提示。注意同一个目录内的类属于“默认”封装的一部分。

(7) 用 protected 数据创建一个类。在相同的文件里创建第二个类，用一个方法操纵第一个类里的 protected 数据。

(8) 根据“protected：有点儿′友好′”一节的指示，修改 Cookie 类。验证 bite()并非“公共”的。

(9) 在“类访问”一节里，找到对 mylib 和 Widget 进行描述的代码段。创建这个库，再在并非 mylib 封装一部分的一个类里，创建一个 Widget。

(10) 新建一个目录，并编辑自己的 CLASSPATH，把那个目录包括到其中。将 P.class 文件（通过编译 com.bruceeckel.tools.P.java 而生成）复制到自己的新目录，然后改变文件名、内部的 P 类以及方法名（亦可考虑添加额外的输出，观察它的运行过程）。在一个不同的目录里创建另一个程序，令其使用刚才创建的新类。

(11) 根据 Lunch.java 的思路，创建一个名为 ConnectionManager 的类，用它对一个固定的 Connection 对象数组进行管理。客户程序员绝对不能明确地创建 Connection 对象，而是只能通过 ConnectionManager 中的一个 static 方法来获得它们。当 ConnectionManager 用光了对象之后，它会返回一个 null 引用。请在 main()中对类进行测试。

(12) 在 c05/local 目录（假定在自己的 CLASSPATH 里）创建下述文件：

```
///: c05:local:PackagedClass.java
package c05.local;
class PackagedClass {
  public PackagedClass() {
    System.out.println(
      "Creating a packaged class");
  }
} ///:~
```

然后在 c05 之外的另一个目录里创建下述文件：

```
///: c05:foreign:Foreign.java
package c05.foreign;
import c05.local.*;
public class Foreign {
   public static void main (String[] args) {
      PackagedClass pc = new PackagedClass();
   }
} ///:~
```

解释编译器为什么会产生一个错误。将 Foreign 类变成 c05.local 封装的一部分之后，难

道改变了什么东西吗？

第 6 章　复用类

Java 最引人注目的一项特性是代码的重复使用或者复用。但最具革命意义的是，这样除了能简单地复制和修改代码之外，我们还能做其他多得多的事情！

在象 C 那样的程序化语言里，代码的重复使用早已可行，只是效果并非特别显著。和 Java 的所有东西一样，这个方案解决的也是与类有关的问题。我们通过创建新类来重复使用代码，但却用不着重新创建，可直接使用别人已建好并调试好的现成类。

但这样做必须保证不会干扰原来的代码。在这一章，我们将介绍两个达到这一目标的方法。第一个最简单：在新类里简单地创建原有类的对象。我们把这种方法叫作“合成”，因为新类由现有类的对象合并而成。我们只是简单地重复利用代码的功能，而不是采用它的形式。

第二种方法则显得更加“巧妙”。它创建一个新类，将其作为现有类的一个“类型”。我们可原样采取现有类的形式，并在其中加入新代码，同时用不着对原有的类进行修改。这种魔术般的行为便叫作“继承”（Inheritance），涉及的大多数工作都是由编译器完成的。对于面向对象的程序设计，“继承”是最重要的基础概念之一，而且它对我们在第 7 章要讲述的内容紧密相关。

对于合成和继承来说，大多数语法和行为都是类似的（因为它们都要根据现有的类型来生成新类型）。在本章，我们将深入学习这些代码复用机制。

6.1　合成的语法

就以前的学习情况来看，事实已进行了多次“合成”操作。为进行合成，只需在新类里简单地置入对象引用即可。举个例子来说，假定要在一个对象里容纳几个 String 对象、两种原始数据类型以及属于另一个类的一个对象。对于非基类型的对象来说，只需将引用置于新类即可；但对原始数据类型来说，则需直接定义：

```
//: c06:SprinklerSystem.java
// Composition for code reuse.

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
```

```
  WaterSource source;
  int i;
  float f;
  void print() {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }
  public static void main(String[] args) {
    SprinklerSystem x = new SprinklerSystem();
    x.print();
  }
} ///:~
```

WaterSource 内定义了一个非常特殊的方法：toString()。大家不久就会知道，每种非基类型的对象都有一个 toString()方法。若编译器本来希望得到的是一个 String，但却获得了这样的一个对象，便会调用这个方法。所以在下面这个表达式中：

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

编译器会发现我们试图向一个 WaterSource 添加一个 String 对象（"source ="）。这对它来说是不可接受的，因为我们只能将一个字串“添加”到另一个字串，所以它会说：“我要调用 toString()，把 source 转换成字串!”经这样处理后，它就能编译两个字串，并将结果字串传递给一个 System.out.println()。每次想让自己的一个类允许这样的行为，都只需写一个 toString()方法。

如果不深究，可能会草率地认为编译器会为上述代码中的每个引用都自动构造对象（由于 Java 一贯具有安全和谨慎的形象）。例如，可能以为它会为 WaterSource 调用默认构造函数，以初始化 source。然而，打印语句的输出事实是：

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null
```

在类内作为字段使用的基本数据会初始化成零，就象第 2 章指出的那样。但对象引用会初始化成 null。而且假如试图为它们中的任何一个调用方法，就会产生一个“违例”。这种结果实际是相当好的（而且很有用），我们可在不“掷出”一次违例的前提下，仍然把它们打印出来。

编译器并不只是为每个引用创建一个默认对象，因为那样在许多情况下都会带来不必要

的开销。如希望引用得到初始化，可在下面这些地方进行：

(1) 在对象定义的时候。这意味着它们在构造函数调用之前肯定能得到初始化。

(2) 在那个类的构造函数中。

(3) 就在真正使用对象之前。这通常叫作“暂缓初始化”。它可减少不必要的开销——只要对象并不需要每次都创建。

下面向大家展示了所有这三种方法：

```
//: c06:Bath.java
// Constructor initialization with composition.

class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class Bath {
  private String
    // Initializing at point of definition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  void print() {
    // Delayed initialization:
    if(s4 == null)
      s4 = new String("Joy");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
```

```
    System.out.println("toy = " + toy);
    System.out.println("castille = " + castille);
  }
  public static void main(String[] args) {
    Bath b = new Bath();
    b.print();
  }
} ///:~
```

请注意在 Bath 构造函数中，语句会在任何初始化进行之前得以执行。如果未在定义时便进行初始化，那么在将一条消息发给一个对象引用之前，仍然不能保证会进行任何初始化操作——除了不可避免的运行时间违例。

下面是该程序的输出：

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

调用 **print( )**时，它会填充 **s4**，使所有字段在使用前都获得正确的初始化。

6.2 继承的语法

继承与 Java（以及其他 OOP 语言）非常紧密地结合在一起。创建一个类时，肯定会进行继承。因为若不是从其他某些类里明确继承，便会默认为从 Java 的标准根类 Object 中继承。

用于合成的语法是非常简单且直观的。但为了进行继承，必须采用一种全然不同的形式。需要继承的时候，我们会说：“这个新类和那个旧类差不多。”为了在代码里表示这一概念，需要象往常一样指定类名。只是在类主体的起始花括号之前，需要放置一个关键字 extends，在后面跟上“基类”的名字。采取这种做法，就可自动获得基类的所有数据成员以及方法。下面是一个例子：

```
//: c06:Detergent.java
// Inheritance syntax & properties.

class Cleanser {
  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 void print() { System.out.println(s); }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }
}

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();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
} ///:~
```

这个例子向大家展示了大量特性。首先，在 Cleanser append()方法里，字串同一个 s 连接起来。这是用“+=”运算符实现的。利用+=和+，Java 实现了对字串的“重载”。

其次，无论 Cleanser 还是 Detergent 都包含了一个 main()方法。我们可为自己的每个类都创建一个 main()。通常都建议大家象这样写程序，使自己的测试代码能和类封装到一起。即使程序里包含了数量众多的类，但根据在命令行指定的类，只有它的 main()才会得到调用（只要 main()是“公共”的，类是不是“公共”的都无所谓）。所以在这种情况下，当我们在命令行执行“java Detergent”的时候，调用的是 Degergent.main()。但是，也可以执行“java Cleanser”，从而调用 Cleanser.main()——即使 Cleanser 并非一个 public 类。采用这种将 main()置入每个类的做法，可方便地为每个类都进行单元测试。而且在完成测试以后，毋需将 main()删去；可把它保留下来，用于以后的测试。

在这儿，大家可看到 Deteregent.main()对 Cleanser.main()的调用是明确进行的，从命令行里向其传递同样的参数（事实上，可向它传递任何字串数组）。

需着重强调的是，Cleanser 中的所有方法都是“公共”的。请记住，若省去了所有访问指示符，那么成员会默认为“友好的”。这样一来，就只有封装成员才能访问它。换言之，在这个封装内，任何人都可使用那些没加访问指示符的方法。在这里，Detergent 不会遇到任何麻烦。然而，假设来自另外某个封装的类准备从 Cleanser 继承，它就只能访问那些 public 成员。所以在计划继承的时候，一个比较好的规则是将所有字段都设为 private，再将所有方法都设为 public（protected 成员也允许派生出来的类访问它；以后还会深入探讨这一问题）。当然，在一些特殊场合，我们仍需作一些调整，不过这确实是一条非常有用的指导性规则。

注意 Cleanser 在它的接口内含有一系列方法：append()、dilute()、apply()、scrub()以及 print()。由于 Detergent 是从 Cleanser 派生出来的（通过 extends 关键字），所以它会自动获得接口内的所有这些方法——即使我们在 Detergent 里并未看到对它们的明确定义。这样一来，就可将继承想象成“接口复用”（具体的代码实现也随着它一道“复用”了，不过那并非我们强调的重点）。

正如在 scrub()里看到的情况那样，我们可获得在基类里定义的一个方法，并对其进行修改。此时，我们可能是打算在新版本里调用来自基类的方法。但在 scrub()里，不可只是简单地发出对 scrub()的调用。那样会造成递归调用，我们可不愿看到这种事情。为解决这个问题，Java 提供了一个 super（超）关键字，它指出这是一个“超类”（Superclass），当前类是从这个超类中继承的。所以，super.scrub()这个表达式调用的是 scrub()方法的基类版本。

进行继承时，我们并不限于只能使用基类的方法。亦可在派生类里再加入新方法。这时的做法与在普通类里添加其他任何方法是完全一样的：只需简单地定义它就行——foam()正是一例！

在 Detergent.main()里，我们可看到对于 Detergent 对象，可调用 Cleanser 以及 Detergent 内可以使用的所有方法（如 foam()）。

6.2.1 初始化基类

由于这儿已经牵扯到两个类——基类和派生类，而不再象以前那样只有一个，所以在想象由一个派生类产生的结果对象时，可能会产生一些迷惑。从外部看，似乎新类拥有与基类相同的接口，而且可包含一些额外的方法和字段。但继承并非仅仅简单地复制基类的接口了事。创建派生类的一个对象时，它在其中包含了基类的一个“子对象”。可将这个子对象想象成我们根据基类本身而创建了它的一个对象。从外部看，结果不过就是将基类的子对象封装到派生类的对象里罢了。

当然，很重要的一点在于，基类子对象应得到正确的初始化。而且，此时只有一个办法能保证这一点：通过调用基类构造函数，在构造函数中完成初始化。基类构造函数有足够的能力和权限来执行对基类的初始化。在派生类的构造函数中，Java 会自动插入对基类构造函数的调用。下面这个例子向大家展示了这种三级继承的实际应用：

```
//: c06:Cartoon.java
// Constructor calls during inheritance.

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}
```

```
class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} ///:~
```

程序的输出证明了自动调用的进行：

```
Art constructor
Drawing constructor
Cartoon constructor
```

可以看出，构建是在基类的“外部”进行的，所以基类会在派生类访问它之前得到正确的初始化。

即使没有为 Cartoon()创建一个构造函数，编译器也会为我们自动合成一个默认构造函数，并发出对基类构造函数的调用。

1. 含有参数的构造函数

上述例子有自己默认的构造函数；也就是说，它们不含任何参数。编译器可以很容易地调用它们，因为不存在要具体传递什么参数的问题。如果类没有默认的参数，或者想调用含有一个参数的某个基类构造函数，就必须明确地编写对基类的调用代码。这是用 super 关键字以及相应的参数列表实现的，如下所示：

```
//: c06:Chess.java
// Inheritance, constructors and arguments.

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 {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
} ///:~
```

假如不调用 BoardGames()内的基类构造函数，编译器就会报告自己找不到 Games()形式的一个构造函数。除此以外，在派生类构造函数中，对基类构造函数的调用是必须做的第一件事情（如操作失当，编译器会向我们指出）。

2. 捕获基本构造函数的违例

正如刚才指出的那样，编译器会强迫我们在派生类构造函数的主体中首先设置对基类构造函数的调用。这意味着在它之前不能出现任何东西。正如大家在第 10 章会看到的那样，这同时也会防止派生类构造函数捕获来自一个基类的任何违例事件。显然，这有时会为我们造成不便。

6.3 合成与继承的结合

许多时候都要求将合成与继承两种技术结合起来使用。下面这个例子展示了如何同时采用继承与合成技术，从而创建一个更复杂的类，同时进行必要的构造函数初始化工作：

```
//: c06:PlaceSetting.java
// Combining composition & inheritance.

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 {
  Spoon sp;
  Fork frk;
  Knife kn;
  DinnerPlate pl;
  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);
  }
} ///:~
```

尽管编译器会强迫我们对基类进行初始化，并要求我们在构造函数最开头做这一工作，但它并不会监视我们是否正确初始化了成员对象。所以必须对此特别留意。

6.3.1 确保正确的清除

Java 不具备象 C++“破坏器”那样的概念。在 C++中，一旦破坏（清除）一个对象，就会自动调用破坏器方法。之所以将其省略，大概是由于在 Java 中只需简单地忘记对象，不必强行“破坏”它们。垃圾收集器会在必要的时候自动回收内存。

垃圾收集器可满足大多数时候的要求，但在某些情况下，我们的类可能在自己活动期间采取一些行动，而这些行动要求必须进行明确的清除工作。正如第 4 章已经指出的那样，我们并不知道垃圾收集器何时才会“动手”，或者说不知它何时会被调用。所以一旦希望为一个类清除什么东西，就必须写一个特别的方法，明确、专门地来做这件事情。同时，还要让客户程序员知道他们必须调用这个方法。而在所有这一切的后面，就如第 10 章要详细解释的那样，必须将这样的清除代码置于一个 finally 从句中，从而防范任何可能出现的违例事件。

下面以一个简单的计算机辅助设计（CAD）系统为例，它能在屏幕上描绘出图形：

```
//: c06:CADSystem.java
// Ensuring proper cleanup.
import java.util.*;

class Shape {
  Shape(int i) {
    System.out.println("Shape constructor");
  }
  void cleanup() {
    System.out.println("Shape cleanup");
  }
}

class Circle extends Shape {
  Circle(int i) {
    super(i);
    System.out.println("Drawing a Circle");
```

```
  }
  void cleanup() {
    System.out.println("Erasing a Circle");
    super.cleanup();
  }
}

class Triangle extends Shape {
  Triangle(int i) {
    super(i);
    System.out.println("Drawing a Triangle");
  }
  void cleanup() {
    System.out.println("Erasing a Triangle");
    super.cleanup();
  }
}

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 a Line: " +
          start + ", " + end);
  }
  void cleanup() {
    System.out.println("Erasing a Line: " +
          start + ", " + end);
    super.cleanup();
  }
}

public class CADSystem extends Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[10];
  CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
```

```
    System.out.println("Combined constructor");
  }
  void cleanup() {
    System.out.println("CADSystem.cleanup()");
    // The order of cleanup is the reverse
    // of the order of initialization
    t.cleanup();
    c.cleanup();
    for(int i = lines.length - 1; i >= 0; i--)
      lines[i].cleanup();
    super.cleanup();
  }
  public static void main(String[] args) {
    CADSystem x = new CADSystem(47);
    try {
      // Code and exception handling...
    } finally {
      x.cleanup();
    }
  }
} ///:~
```

这个系统中的所有东西都属于某种 Shape（几何形状）。Shape 本身是一种对象，因为它是从根类明确继承的。每个类都重新定义了 Shape 的 cleanup()方法，同时还用 super 调用那个方法的基类版本。尽管对象活动期间调用的所有方法都可负责做一些要求清除的工作，但对于特定的 Shape 类来说——Circle（圆）、Triangle（三角形）以及 Line（直线）——它们都拥有自己的构造函数，能完成“作图”（draw）任务。每个类都有它们自己的 cleanup()方法，用于在对象退出之前，将不需记忆的东西恢复回原状。

在 main()中，大家可看到两个新的关键字：try 和 finally。我们要到第 10 章才会向大家正式介绍它们。其中，try 关键字指出后面跟随的块（由花括号定界）是一个“警戒区”。也就是说，它会受到特别的待遇。其中一种待遇就是：该警戒区后面跟随的 finally 从句的代码肯定会得以执行——不管 try 块以何种方式退出（通过违例控制技术，try 块可有多种不寻常的应用）。在这里，finally 从句的意思是:“总是为 x 调用 cleanup()，无论会发生什么事情”。这些关键字将在第 10 章进行全面、完整的解释。

在自己的清除方法中，必须注意对基类以及成员对象清除方法的调用顺序——假若一个子对象要以另一个为基础。通常，应该和 C++编译器对它的“破坏器”采取的做法一样：首先执行与自己的类有关的所有清除工作——按照与它们创建时相反的顺序（通常，这要求基类元素仍然可用）。然后，就象这里演示的那样，再调用基类清除方法。

许多情况下，清除可能并不是个大问题——只要把这些活儿让给垃圾收集器去干就可以了。但在必须由自己明确动手清除的时候，就必须特别谨慎，并考虑周全。

垃圾收集的顺序

不能指望自己能确切知道何时会开始垃圾收集；有些时候，垃圾收集器甚至永远都不会调用。即使调用，它也可能按自己乐意的任何顺序回收对象。总之，除了回收内存空间之外，

最好都不要依靠垃圾收集。如果想进行清除，请使用自己的清除方法，不要依赖 finalize()（就象第 4 章讲到的那样，可强迫 Java 调用所有“收尾器”）。

6.3.2　名字的隐藏

初次接触 Java 的名字隐藏时，恐怕只有 C++程序员才会大惊小怪，因为它的工作方式和 C++的截然不同！假定 Java 基类有一个方法名被“重载”使用多次，那么在派生类里重新定义那个方法名的时候，并不会隐藏任何一个基类版本。换言之，不管方法是在这一级还是在基类中定义的，重载都会正常进行：

```
//: c06:Hide.java
// Overloading a base-class method name
// in a derived class does not hide the
// base-class versions.

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) {}
}

class Hide {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1); // doh(float) used
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
  }
} ///:~
```

正如下一章会讲到的那样，很少会用与基类里完全一致的签名和返回类型来覆盖同名的方法，否则会使人感到迷惑（这正是 C++不允许那样做的原因，它能防止产生一些不必要的错误）。

6.4 到底选择合成还是继承

无论合成还是继承，都允许我们将子对象置于自己的新类中。大家或许会对两者间的差异感到迷惑，到底该如何选择呢？

如果想利用新类内部一个现有类的特性，而不想用它的接口，通常应选择合成。也就是说，我们可嵌入一个对象，使自己能用它实现新类的特性。但新类的用户会看到我们已定义的接口，而不是来自嵌入对象的接口。考虑到这种效果，我们需在新类里嵌入现有类的 private 对象。

有些时候，我们想让类用户直接访问合成的新类。也就是说，需将成员对象的属性变为 public。成员对象会将自身隐藏起来，所以这是一种安全的做法。而且在用户知道我们准备合成一系列组件时，接口就更容易理解。car（汽车）对象便是一个很好的例子：

```
//: 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);
  }
} ///:~
```

由于汽车的装配是故障分析时需要考虑的一项因素（并非仅仅是基础设计的一部分），所以将成员设为 public 之后，有助于客户程序员理解如何使用类，而且作为类的创建者，我们的编程复杂程度也会大幅降低。不过，要注意这只是一种特殊情况，通常还是应该将字段设为 private。

在我们选择继承的时候，需要取得一个现成的类，然后生成它的一个特殊版本。通常，这意味着我们准备使用一个常规用途的类，并根据要求对其进行定制。只需稍加想象，就知道自己无法用一个车辆对象来合成一辆汽车——汽车并不“包含”车辆；相反，它“属于”车辆的一类。“属于”关系是用继承来表达的，而“包含”关系是用合成来表达的。

6.5　受保护的

现在我们已理解了继承的概念，protected（受保护的）这个关键字最后终于有了意义。在理想情况下，private 成员无论如何都是“私有”的，任何人都不得访问。但在实际应用中，我们却常常不这样干，而是一方面想把某些东西在大多数人面前隐藏起来，一方面又允许派生类的成员访问它。protected 关键字便可帮助我们做到这一点。它的意思是“对类用户来说，它绝对是私有的；但是，对于从这个类继承的任何东西来说，或者对同一封装内的其他东西来说，却是可以访问它的。”换言之，Java 中的 protected 会自动变成“友好的”。

通常，我们应保持数据成员的 private 状态——作为类的创建者，我们无论如何都该有权对基础实现进行修改吧？在这一前提下，再通过 protected 方法，允许对类的继承者进行受控制的、有限的访问：

```
//: c06:Orc.java
// The protected keyword.
import java.util.*;

class Villain {
  private int i;
  protected int read() { return i; }
  protected void set(int ii) { i = ii; }
  public Villain(int ii) { i = ii; }
  public int value(int m) { return m*i; }
}

public class Orc extends Villain {
  private int j;
```

```
  public Orc(int jj) { super(jj); j = jj; }
  public void change(int x) { set(x); }
} ///:~
```

我们看到，change()拥有对 set()的访问权限，因为它的属性是 protected（受保护的）。

6.6 累积开发

继承的一个好处是它支持“累积开发”——允许我们引入新代码，同时不会在老代码里造成新问题。也就是说，即使出现了新的程序错误，那也肯定是在新代码中。通过从一个现有的、能正常工作的类继承，同时增添新的数据成员和方法（并重定义现有方法），我们就可保持老代码原封不动（或许另外有人正在用它），不会在其中带来新的编程错误。一旦出现错误，我们就知道它肯定是由于新代码造成的。这样一来，由于不必修改老代码，所以改正错误所需的时间和精力就可以少上许多。

类和类之间的隔离效果非常好，这是许多初次接触 Java 的程序员事先没有预料到的。甚至不需要方法的源码，就可实现代码的“重复利用”——至多只需导入一个封装（对继承和合成来说都是适用的）。

大家要记住这样一个重点：程序开发是一个不断递增或者累积的过程；就象人们学习知识一样，一个程序也是在不断进步的。当然可根据实际要求，事先进行尽可能多的分析。但在一个项目的设计之初，谁都不可能提前获知所有问题的答案。最好能将自己的项目看作一个有机的、能不断学习进步的生物，从而不断地发展和改进它——而不是象盖房子那样，一次盖完了事——这样才有望获得更大的成功以及更为直接的反馈。

尽管继承是一种非常有用的技术，但在某些情况下，特别是在项目稳定下来以后，仍需从新角度考察自己的类结构，将其收缩成一个更灵活的结构。请记住，继承表达的乃是一种特殊关系：“这个新类是老类的一种类型”。我们的程序不应纠缠于细枝末节，而应着眼于大局，规划出如何创建和操作各类对象，用来自问题空间的术语，表达出一个模型。

6.7 向上强制转型

继承最值得注意的地方就是它没有为新类提供方法。继承是对新类和基类之间关系的一种表达。可这样来总结关系：“新类是老类的一种类型”。

听起来有些“拗口”，但这样的表述可并不是文字上的一种“取巧”——语言本身便直接提供了对它的支持。作为一个例子，大家可考虑一个名为 Instrument 的基类，它用于表示“乐器”；同时还有一个派生类，叫作 Wind，用于表示“管乐器”。由于继承意味着基类的所有方法亦可在派生出来的类中使用，所以我们发给基类的任何消息亦可发给派生类。假如 Instrument 类有一个 play()方法（play 是“演奏”的意思——译注），那么 Wind 也肯定有这个方法。这意味着我们可认定一个 Wind 对象同时也是 Instrument 的一种类型。下面这个例子揭示出编译器如何提供对这一概念的支持：

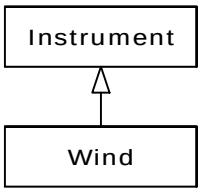
```
//: c06:Wind.java
// Inheritance & upcasting.
import java.util.*;
```

```
class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
} ///:~
```

在这个例子中，最有趣的无疑是 tune()方法，它能接受一个 Instrument 引用。但在 Wind.main()中，tune()方法是通过为其赋予一个 Wind 引用来调用的。由于 Java 对类型检查特别严格，所以大家可能会觉得奇怪，为什么接受了一种类型的方法同时也能接受另一种类型呢？在这里，大家一定要牢记一个 Wind 对象同时也是一个 Instrument 对象，而且对于 tune()能为 Instrument 调用的一个方法来说，它同时也必须在 Wind 中。在 tune()中，代码适用于 Instrument 以及从 Instrument 派生出来的任何东西。在这里，我们将从一个 Wind 引用转换成一个 Instrument 引用的行动就叫作“向上强制转型”。

6.7.1　何谓“向上强制转型”？

之所以叫作这个名字，除了有一定历史原因外，也是由于在传统意义上，类继承示意图的画法是：根位于最顶部，再逐渐向下扩展（当然，可根据自己的习惯用任何方法描绘这种图）。因此，Wind.java 的继承图就象下面这个样子：



可以看到，在继承图中，从派生类强制转型成基类时，是“向上”移动的，所以通常把它叫作“向上强制转型”（Upcasting）。向上强制转型肯定是安全的，因为我们是从一个较特殊的类型到一个更通用的类型（从管乐器到乐器）。换言之，派生类是基类的一个超集。它可以包含比基类更多的方法，但它至少包含了基类的方法。进行向上强制转型时，类接口可能出现的唯一问题便是它可能丢失现有方法，而不是获得更多的方法。这便是在没有任何明确强制转型指令或其他特殊标注的情况下，编译器仍然允许向上强制转型的原因所在。

当然，也可以执行向下强制转型，但这时会面临第 12 章要详细讲述的一种困境。

再论合成与继承

在面向对象的程序设计中，创建和使用代码最可能采取的一种做法是：将数据和方法统一封装到一个类里，并使用那个类的对象。有些时候，需通过“合成”技术根据现有的类来构造新类。而继承是最少见的一种做法。因此，尽管继承在学习 OOP 的过程中得到了大量的强调，但并不意味着应该尽可能地到处用它。相反，想用它时需要特别慎重。只有清楚地知道了在所有方法中，只有继承最有效，才可考虑它。要想判断自己到底该用合成还是继承，一个最简单有效的办法就是问问自己是否需要从新类向上强制转型回基类。如必须向上强制转型，便肯定要继承；但假如不需要向上强制转型，就应该警惕了，此时通常都用不着继承。在下一章里（多态），会向大家介绍必须进行向上强制转型的另一种场合。但只要记住经常问自己“我需要向上强制转型吗”，合成还是继承的选择就不该是个难题。

6.8 final 关键字

由于语境（应用环境）不同，final 关键字的含义可能会稍微产生一些差异。但它最一般的意思就是声明“这个东西不能改变”。之所以要禁止改变，可能是考虑到两方面的因素：设计或效率。由于这两个原因颇有些区别，所以也许会造成 final 关键字的误用。

在接下去的小节里，我们将讨论 final 关键字的三种应用场合：数据、方法以及类。

6.8.1 final 数据

许多程序设计语言都有自己的办法告诉编译器某个数据是“常数”。常数主要应用于下述两个方面：

(1) 编译时间常数，它永远不会改变

(2) 在运行时间初始化的一个值，初始化后便不希望改变

对于编译时间常数，在需要用到它的任何计算中，编译器都可将它的值提前“代入”；也就是说，计算可在编译时间提前执行，从而节省运行时间的一些开销——常数嘛，反正以后又不会变化！在 Java 中，这些形式的常数必须属于原始数据类型，而且要用 final 关键字进行表达。在对这样的一个常数进行定义的时候，必须当场给出一个值。

无论 static 还是 final 字段，都只能存储一个数据，而且不得改变。

若随对象引用使用 final，而不是用于原始数据类型，它的含义就有点儿让人迷糊了。对于原始数据类型，final 会将值变成一个常数；但对于对象引用，final 会将引用变成一个常数。引用初始化成一个对象之后，便永远不能改动它，令它指向另一个对象。不过，对象本身却是可以修改的。Java 并未提供任何手段，可将一个对象变成常数（但是，我们可自己编写一个类，使其中的对象具有“常数”效果）。这一限制也适用于数组，它们也是对象。

下面是演示 final 字段用法的一个例子：

```
//: c06:FinalData.java
// The effect of final on fields.

class Value {
  int i = 1;
}

public class FinalData {
  // Can be compile-time constants
  final int i1 = 9;
```

```
  static final int VAL_TWO = 99;
  // Typical public constant:
  public static final int VAL_THREE = 39;
  // Cannot be compile-time constants:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);

  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();
  // Arrays:
  final int[] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
    System.out.println(
      id + ": " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
    FinalData fd1 = new FinalData();
    //! fd1.i1++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object isn't constant!
    //! fd1.v2 = new Value(); // Error: Can't
    //! fd1.v3 = new Value(); // change reference
    //! fd1.a = new int[3];

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");
  }
} ///:~
```

由于 i1 和 VAL_TWO 都是具有 final 属性的原始数据类型，并指定了编译时间的值，所以它们都可作为编译时间常数使用。而且在各种主流应用中，它们并无什么区别。VAL_THREE 则不同，它的常数定义方式显得要“特别”一些：public 表示它可在封装的外部使用；static 强调只有一个；而 final 表明它是一个常数。注意根据规则，对于含有固定初始化值（即编译时间常数）的 fianl static 原始数据类型来说，它们的名字需要全部采用大写形式。如名字内包含了多个单词，那么相互间必须用下划线连接（这和 C 常数是一样的——它是制订该规则的“老祖宗”）。另外要注意的是，i5 的值在编译时间是未知的，所以它没

有大写。

不能由于某样东西的属性是 final，便认定它的值可在编译时间知道。i4 和 i5 向大家证明了这一点。它们在运行时间使用随机生成的数字。例子的这一部分也向大家揭示出将 final 值设为 static 和非 static 之间的差异。只有值在运行时间初始化的前提下，这种差异才会被揭示出来——因为编译时间的值被编译器认为是相同的（而且不存在提前优化的问题）。这种差异可从一次运行的输出结果中看出：

```
fd1: i4 = 15, i5 = 9
Creating new FinalData
fd1: i4 = 15, i5 = 9
fd2: i4 = 10, i5 = 9
```

注意对 fd1 和 fd2 来说，i4 的值是唯一的，但 i5 的值不会由于创建了另一个 FinalData 对象而发生改变。那是因为它的属性是 static，而且在载入时初始化，而非在每创建一个对象时初始化。

从 v1 到 v4 的变量向我们揭示出 final 引用的含义。正如大家在 main()中看到的那样，并不能认为由于 v2 属于 final，所以就不能再改变它的值。然而，我们确实不能再将 v2 绑定到一个新对象，因为它的属性是 final。这便是 final 对一个引用来说的确切含义。我们会发现同样的含义亦适用于数组，后者只不过是另一种类型的引用而已。将引用变成 final，看来似乎不如将原始数据类型变成 final 那么有用！

1. 空白 final

Java 允许我们创建“空白 final”。它们是一类非常特殊的字段——尽管被声明成 final，但却未得到一个初始值。无论在哪种情况下，空白 final 都必须在实际使用前得到正确的初始化。而且编译器会主动保证这一规定得以贯彻。然而，对于 final 关键字的各种应用来说，空白 final 具有最大的灵活性。举个例子来说，位于类内的一个 final 字段现在对每个对象都可以有所不同，同时依然保持其“不变”的本质。下面列出一个例子：

```
//: c06:BlankFinal.java
// "Blank" final data members.

class Poppet { }

class BlankFinal {
  final int i = 0; // Initialized final
  final int j; // Blank final
  final Poppet p; // Blank final reference
  // Blank finals MUST be initialized
  // in the constructor:
  BlankFinal() {
    j = 1; // Initialize blank final
    p = new Poppet();
  }
  BlankFinal(int x) {
```

```
    j = x; // Initialize blank final
    p = new Poppet();
  }
  public static void main(String[] args) {
    BlankFinal bf = new BlankFinal();
  }
} ///:~
```

现在，我们被强制为 final 执行赋值——要么在定义字段时用一个表达式，要么在每个构造函数中进行。这样就可以确保 final 字段在使用前获得正确的初始化。

2. final 参数

Java 允许我们将参数设成 final 属性，方法是在参数列表中对它们进行适当的声明。这意味着在一个方法的内部，我们不能改变参数引用指向的东西。如下所示：

```
//: 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);
  }
} ///:~
```

注意此时仍可为 final 参数分配一个 null（空）引用，编译器不会报错，这和使用非 final 参数时是一样的。

方法 f()和 g()向我们展示出基类型参数为 final 时会发生什么情况——只能读取参数，不可改变它。

6.8.2 final 方法

之所以要使用 final 方法，可能是由于两方面的原因。第一个是为方法“加锁”，防止任何继承类改变它的本来含义。设计程序时，若希望一个方法的行为在继承期间仍然保持不变，而且不可被覆盖或改写，就可采取这种做法。

采用 final 方法的第二个理由是提高程序运行的效率。将一个方法设成 final 后，编译器就可把对那个方法的所有调用都置入“嵌入”调用里。只要编译器发现一个 final 方法调用，就会（根据它自己的判断）忽略为执行方法调用机制而采取的常规代码插入方法（将参数压入堆栈；跳至方法代码并执行它；又跳回来；清除堆栈参数；最后对返回值进行处理）。相反，它会用方法主体内实际代码的一个副本来替换方法调用。这样做可避免方法调用时的系统开销。当然，若方法体积太大，程序会变得臃肿不堪，嵌入调用对性能提升可能也不会有什么帮助，因为任何提升都被花在方法内部的时间抵消了。Java 编译器能自动侦测这些情况，并颇为“明智”地决定是否嵌入一个 final 方法。然而，最好还是不要完全相信编译器能正确作出所有判断。通常，只有在方法代码量非常少，或者想明确禁止方法被别人覆盖的时候，才应考虑将一个方法设为 final。

final 和 private

类内所有 private 方法都会默认为 final。由于不能访问一个 private 方法，所以它绝不可能被其他方法覆盖（若强行覆盖，尽管编译器不会报错，但此时并没有真正“覆盖”，而是新建了一个方法）。当然，可为一个 private 方法明确添加 final 指示符，但实际效果却并无什么分别。

这个问题有时会为程序员带来一些迷惑，因为假如试着覆盖一个 private 方法（它默认为 final 的），似乎也是可行的：

```
//: c06:FinalOverridingIllusion.java
// It only looks like you can override
// a private or private final method.

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 {
  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();
  }
} ///:~
```

“覆盖”（Overriding）是个不大准确的用词，它的确切含义是指：为老东西赋予新含义，而不是用一个全新的东西来替代原有的东西！在 Java 中，只有在基类接口的一部分得以保留的前提下，才会发生真正意义上的“覆盖”。换言之，若想发生真正的“覆盖”，那么一个对象必须能向上强制转型回它的基类型，并能调用相同的方法（这个问题在下一章会变得更加明确）。在这时，只是某些代码在类的内部被隐藏起来了，而且刚好使用了相同的名字。但假如我们在派生类中创建了一个 public、protected 或者“友好”方法，那么它与基类中可能同名的方法是没有丝毫联系的。由于 private 方法是用不到的，所以实际就是“隐形”的，除了和类的代码组织方式有关之外，它和其他任何东西都不会扯上联系。

6.8.3　final 类

如果说整个类都是 final 的（在它的定义前冠以 final 关键字），就表明自己不希望再从

这个类继承，也不允许其他任何人采取这种操作。换言之，出于这样或那样的理由，我们的类以后肯定不需要进行任何改变；或者出于安全方面的理由，我们不希望对它进行子类化（子类处理）。另外，我们这样做，或许也是考虑到执行效率的问题，想确保牵涉到这个类的各个对象的任何行动都尽可能地具有高效率。如下所示：

```
//: 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++;
  }
} ///:~
```

注意数据成员既可以是 final，也可以不是，取决于我们具体选择。应用于 final 的规则同样适用于数据成员，无论类是否被定义成 final。将类定义成 final 后，结果只是禁止进行继承——没有更多的限制。然而，由于它禁止了继承，所以一个 final 类中的所有方法都默认为 final。因为此时再也无法覆盖它们。所以与我们将一个方法明确声明为 final 一样，编译器此时有相同的效率选择。

可为 final 类内的一个方法添加 final 指示符，但这样并没有什么特殊的含义。

6.8.4 final 的注意事项

设计一个类时，往往要考虑是否将一个方法设为 final。大家可能会觉得，在别人用自己的类时，效率的重要性应摆在第一位，所以其他任何人都不应该对自己的方法进行“覆盖”。这种想法在某些时候是正确的。

但要慎重作出自己的假定。通常，我们很难预测一个类以后会以什么样的形式复用。常规用途的类尤其如此。若将一个方法定义成 final，就可能杜绝了在其他程序员的项目中对自己的类进行继承的途径，因为我们根本没有想到它会象那样使用。

标准 Java 库是我们的最佳范例。特别是由 Java 1.0/1.1 提供的 Vector 类，它的应用非常

广泛。考察“执行效率”，大家便会发现，假如所有方法都不是设为 final，那么 Vector 类可能还会更加有用！我们很容易就会得出结论：假如能继承和覆盖如此有用的一个基类，那岂不是“锦上添花”的一桩美事？但我们的美好愿望被 Java 的设计者否决了！真搞不懂他们当时是怎么想的，因为至少可以用两条有力的理由来反驳他们。首先，在 Java 中，Stack（堆栈）正是从 Vector 继承来的。难道说，Stack“是”一个 Vector 吗？从逻辑角度讲，这种的说法显得是站不住脚的。其次，对于 Vector 许多重要的方法来说，如 addElement()以及 elementAt()等，它们都变成了 synchronized（同步的）。正如在第 14 章要讲到的那样，这会造成严重的性能开销，从而将 final 本来便不多的优势抵销得干干净净。因此，程序员完全有理由感到迷惑，不知道该用方式对自己的程序加以优化。在一个语言的标准库里居然采用了如此笨拙的设计，应该把 Java 的设计人员拿来“打屁股”才是！幸运的是，Java 2 的容器库已将 Vector 换成了 ArrayList，它的表现要令人满意得多；另外不幸的是，由于习惯思维的影响，现在仍有许多新代码在使用老的容器库。

另一个值得注意的是 Hashtable（散列表），它是另一个重要的标准类库。该类根本没用任何 final 方法！正如大家在本书其他地方还会注意到的那样，有些类显然是由不同的人设计的（注意 Hashtable 的方法名要比 Vecor 的方法名精简得多）。对于类库用户来说，这样的事情绝对不应让他们轻易就看得出来。产品设计的不一致，会为用户带来负担，增大他们的工作量。这也从另一个方面强调了项目设计与代码检查的重要性。作为设计者，需要有很强的责任心，保证自己的产品整齐划一（注意 Java 2 容器库用 HashMap 替代了 Hashtable）。

6.9　初始化和类装载

在许多传统语言里，程序都是作为启动过程的一部分一次性载入的。随后进行的是初始化，再接着是程序启动。在这些语言中，必须小心翼翼地对初始化过程进行控制，保证 static 数据的初始化顺序不会带来麻烦。比如在 C++中，假如静态值 a 希望静态值 b 是一个有效值，但此时 b 值尚未初始化，那么肯定会出现问题。

Java 则没有这样的问题，因为它采用了不同的装载方法。由于 Java 中的一切都是对象，所以许多活动变得更加简单。初始化和装载便是一例。正如下一章会讲到的那样，每个对象的代码都存在于独立的文件中。除非真的需要代码，否则那个文件是不会载入的。通常，我们可认为除非那个类的一个对象构造完毕，否则代码不会真的载入。通常，我们可以大而化之地说：“类代码将在首次使用时载入”。但事实上，除非那个类的第一个对象构建完毕，否则类代码通常还是不会载入的。另外，在访问一个 static 字段或 static 方法的时候，也会进行装载。

首次使用的地方同时也是 static 初始化发生的地方。装载时，所有 static 对象和 static 代码块都会按“文字”顺序初始化（亦即它们在类定义代码里写入的顺序）。当然，static 数据只会初始化一次。

6.9.1　通过继承初始化

我们有必要对整个初始化过程有所认识，其中包括继承，从而对这个过程中发生的事情有一个整体性概念。请观察下述代码：

```
//: c06:Beetle.java
// The full process of initialization.

class Insect {
```

```
  int i = 9;
  int j;
  Insect() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    prt("static Insect.x1 initialized");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Beetle extends Insect {
  int k = prt("Beetle.k initialized");
  Beetle() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 =
    prt("static Beetle.x2 initialized");
  public static void main(String[] args) {
    prt("Beetle constructor");
    Beetle b = new Beetle();
  }
} ///:~
```

程序输出如下：

```
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
```

执行 java Beetle 命令时，我们事实上是首先试着访问 Beetle.main()——这是一个静态方法。因此，装载器会到外面去，找到为 Beetle 类编译好的代码（这些代码恰恰保存在一个名为 Beetle.class 的文件中）。载入过程中，装载程序会注意它有一个基类（即 extends 关键字要表达的意思），所以随之将其载入。无论是否准备生成那个基类的一个对象，这个过程都会发生（请试着将对象的创建代码当作注释标注出来，自己去证实）。

若基类含有另一个基类，则另一个基类随即也会载入，以此类推。接下来，会在根基类

（此时是 Insect）执行 static 初始化，再在下一个派生类执行，以此类推。保证这个顺序是非常关键的，因为派生类的初始化可能要依赖于对基类成员的正确初始化。

此时，必要的类已全部装载完毕，所以就可以开始创建对象了。首先，这个对象中的所有原始数据类型都会设成它们的默认值，同时将对象引用设为 null——将对象占用的内存一次性全部清为二进制 0。随后会调用基类构造函数。在这种情况下，调用是自动进行的，但也完全可以用 super 来自行指定构造函数调用（就象 Beetle()构造函数中的第一个操作一样）。基类的构建采用与派生类构造函数完全相同的过程，采用完全相同的顺序。基类构造函数完成以后，实例变量会按其“文字”顺序得以初始化。最后，执行构造函数主体剩余的部分。

6.10　总　　结

无论继承还是合成，我们都可在现有类型的基础上创建一个新类型。但在典型情况下，我们通过合成来实现现有类型的“复用”，将其作为新类型基础实现的一部分使用。但如果想实现接口的“复用”，就应使用继承。由于派生出来的类拥有基类的接口，所以能将其“向上强制转型”为基类。对于下一章要讲述的多态问题，这一点是至关重要的。

尽管继承在面向对象的程序设计中得到了特别的强调，但在实际进行一个设计时，最好还是先考虑采用合成技术。只有在特别必要的时候，才应考虑继承。合成显得更加灵活。此外，通过对自己的成员类型应用一些继承技巧，可在运行时间精确修改那些成员对象的类型，由此改变它们的行为。换句话说，在运行时间，我们可对复合对象的行为加以改变。

尽管对于快速项目开发来说，通过合成和继承实现的代码复用具有很大的帮助作用。但在允许其他程序员完全依赖它之前，一般都希望能重新设计自己的类结构。我们理想的类结构应该是每个类都有自己特定的用途。它们不能过大（如集成的功能太多，很难实现它的复用），也不能过小（就连自己也无法使用，或者无法再新增更多的功能）。

6.11　练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 用默认构造函数（空参数列表）创建两个类：A 和 B，令它们自己声明自己。从 A 继承一个名为 C 的新类，并在 C 内创建一个成员 B。不要为 C 创建构造函数。创建类 C 的一个对象，并观察结果。

(2) 修改练习 1，使 A 和 B 都拥有包括了参数的构造函数，而不是使用默认构造函数。为 C 写一个构造函数，并在 C 的构造函数中执行所有初始化工作。

(3) 创建一个简单的类。在另一个类里，为第一个类的一个对象定义一个字段。用“暂缓初始化”的方式，对该对象进行例示。

(4) 从 Detergent 类继承一个新类。覆盖 scrub()，并新增一个名为 sterilize()的方法（单词说明：Detergent－清洁剂；scrub－清洗；sterilize－消毒——译注）。

(5) 找到文件 Cartoon.java，将 Cartoon 类的构造函数代码变成注释内容标注出去。解释这样做的后果。

(6) 找到文件 Chess.java，将 Chess 类的构造函数代码作为注释标注出去。同样解释后果。

(7) 想办法证实编译器会为我们创建默认构造函数。

(8) 想办法证实基类构造函数：(a)肯定会被调用；(b)而且是在派生类构造函数之前调用。

(9) 创建一个基类，它只有一个非默认构造函数；再创建一个派生类，同时有一个默认和非默认构造函数。在派生类构造函数中，请调用基类构造函数。

(10) 创建一个名为 Root 的类，其中包括下列所有类（也要由你创建）的一个实例：Component1、Component2 和 Component3。从 Root 派生出一个类，名为 Stem，其中也包括每个“Component”的一个实例。所以类都应该有默认构造函数，用于打印和那个类有关的一条消息。

(11) 修改练习(10)，使每个类都只有非默认的构造函数。

(12) 为练习(11)的所有类增加一个正确的 cleanup()方法结构。

(13) 创建一个类，在其中包括一个重载三次的方法。再继承出一个新类，增加一个新的重载方法，证明所有这四个方法都可在派生类中使用。

(14) 在 Car.java 中，为 Engine 增加一个 service()方法，然后在 main()中调用该方法。

(15) 在一个封装内创建一个类。该类应包括一个 protected 方法。在封装外部，试着调用 protected 方法，并对结果进行解释。接下来，从这个类继承，并从派生类的一个方法内部来调用 protected 方法。

(16) 创建一个名为 Amphibian（两栖动物）的类，从它继承一个名为 Frog（青蛙）的类。在基类中加上合适的方法。在 main()中，创建一个 Frog，并将其向上强制转型为 Amphibian，证明所有方法仍能正常工作。

(17) 修改练习(16)，使 Frog 覆盖基类的同名方法定义（用同样的方法签名，提供新的定义）。注意此时在 main()中会发生什么。

(18) 创建一个类，其中同时包括一个 static final 字段和一个 final 字段，演示两者的差异。

(19) 创建一个类，其中包括指向某对象的一个空白 final 引用。在一个方法（而不是构造函数）内部，刚好就在用它之前，对空白 final 执行初始化。请由此证实两点：Java 保证 final 会在使用之前得到初始化；而且一旦初始化，便不能改变。

(20) 创建一个类，其中含有一个 final 方法。从那个类继承，并试着对那个方法进行覆盖。

(21) 创建一个 final 类，试着从它继承。

(22) 证实类的装载只会进行一次。证实在创建那个类的第一个实例时，或在访问一个 static 成员时，都会造成类的装载。

(23) 在 Beetle.java 中，从 Beetle 类继承一个特定的 Beetle（甲虫）类型，采用与现有类相同的格式。请跟踪和解释结果。

第 7 章　多态

对于面向对象的程序语言，多态是其第三种最基本的特征——前两种是数据抽象与继承。

“多态”（Polymorphism）采用另一种方式，将接口与具体的实现分离开，亦即实现了“是什么”与“怎么做”这两个模块的分离。利用多态的概念，代码的组织以及可读性均可获得较大改善。此外，还能用它创建出“易于扩展”的程序。无论在项目的创建过程中，还是在往后需要添加新特性的时候，这些程序都可方便地“成长”。

通过合并特征与行为，封装技术可创建出新的数据类型。将所有细节都设为“private”（私有），将具体实现隐藏起来，就可将接口与实现分开。但这种方式显得非常初级，只有那些具有程序化编程背景的人才会感觉舒适。“多态”是另一种比较高级的方式，它涉及到对“类型”的分离。通过上一章的学习，大家知道通过继承，可将一个对象当作它自己的类型或它自己的基类型对待。这一能力是十分重要的，因为它使多个类型（从相同基类型派生出来）能被当作同一种类型对待。而且只需一段代码，即可对所有不同的类型进行同样的处理。利用多态方法调用，一种类型可将自己与另一种相似的类型区分开，只要它们都是从相同的基类型派生出来的。这种区分是通过各种方法在行为上的差异实现的，可通过基类实现对那些方法的调用。

在这一章中，大家要由浅入深地学习有关多态的问题（也叫作动态绑定、推迟绑定或运行时间绑定）。同时还有一些简单的例子，其中所有无关的部分都已剥除，只保留与多态有关的代码。

7.1　再论向上强制转型

通过第 6 章的学习，大家知道可将一个对象作为它自己的类型使用，或作为它的基类型的一个对象使用。取得一个对象引用，并将其作为基类型引用使用的行为就叫作“向上强制转型”——因为继承树的画法是基类位于最上方。

但这样做也会遇到一个问题，如下例所示：

```
//: c07:music:Music.java
// Inheritance & upcasting.

class Note {
  private int value;
  private Note(int val) { value = val; }
  public static final Note
    MIDDLE_C = new Note(0),
    C_SHARP  = new Note(1),
    B_FLAT   = new Note(2);
} // Etc.
```

```
class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
  // Redefine interface method:
  public void play(Note n) {
    System.out.println("Wind.play()");
  }
}

public class Music {
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Upcasting
  }
} ///:~
```

其中，方法 Music.tune()要求接收一个 Instrument（乐器）引用，同时也接收从 Instrument 派生出来的所有东西。在 main()中，当一个 Wind（管乐器）引用传递给 tune()的时候，就会出现这种情况，此时没有强制转型的必要。这样做是可以接受的；Instrument 的接口肯定也存在于 Wind 中，因为 Wind 是从 Instrument 继承来的。从 Wind 到 Instrument 的向上强制转型可能会“缩小”那个接口，但不可能变得比 Instrument 的完整接口还要小。

7.1.1 忘了对象类型

看起来，这个程序似乎有点儿奇怪：为什么要有目的地忘记一个对象的类型呢？进行向上强制转型时，我们就应该这样做。但是，假如让 tune()简单地取得一个 Wind 引用，将其作为自己的参数使用，难道不是简单、直观得多吗？但请注意的是，假如那样做，就需为系统内 Instrument 的每种类型都写一个全新的 tune()。现在，根据这一推论，让我们来亲自尝试一下不有意忘记对象类型的后果——假设我们要加入 Stringed（弦乐）和 Brass（铜管）这两种 Instrument（乐器）：

```
//: c07:music2:Music2.java
// Overloading instead of upcasting.

class Note {
```

```
  private int value;
  private Note(int val) { value = val; }
  public static final Note
    MIDDLE_C = new Note(0),
    C_SHARP = new Note(1),
    B_FLAT = new Note(2);
} // Etc.

class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

class Wind extends Instrument {
  public void play(Note n) {
    System.out.println("Wind.play()");
  }
}

class Stringed extends Instrument {
  public void play(Note n) {
    System.out.println("Stringed.play()");
  }
}

class Brass extends Instrument {
  public void play(Note n) {
    System.out.println("Brass.play()");
  }
}

public class Music2 {
  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);
  }
} ///:~
```

尽管行得通，但却存在一个极大的弊端——你必须为每种新增的 Instrument 类都编写自己的专用方法。这意味着第一次便需要多得多的编码量。以后，假如想添加一个象 tune()那样的新方法，或者为 Instrument 添加一个新类型，还要进行大量编程工作。此外，即使忘记对自己的某个方法进行重载设置，编译器也不会提示任何错误。这样一来，类型的整个操作过程就显得极难管理，有失控的危险。

假如可以只写一个方法，将基类作为参数或参数使用，而不是用那些特定的派生类，结果岂不是会好得多吗？也就是说，假如我们能不顾派生类，只让自己的代码同基类打交道，那么节省的工作量将是难以估计的！

这正是“多态”大显身手的地方。然而，大多数程序员（特别是有程序化编程背景的）对于多态的工作原理仍然显得有些生疏。

7.2 深入理解

对于 Music.java 的麻烦，可通过运行程序实际加以体会。输出是 Wind.play()。这当然是我们希望的输出，但它看起来似乎并不愿按我们的希望行事。请观察一下 tune()方法：

```
public static void tune(Instrument i) {
  // ...
  i.play(Note.MIDDLE_C);
}
```

它接收 Instrument 引用。所以在这种情况下，编译器怎样才能知道 Instrument 引用指向的是一个 Wind，而不是一个 Brass 或 Stringed 呢？事实上，编译器无从得知！为了深入了理解这个问题，有必要探讨一下“绑定”这个主题。

7.2.1 方法调用的绑定

将一个方法调用同一个方法主体连接到一起就称为“绑定”（Binding）。若在程序运行前执行绑定（由编译器和链接程序完成——如果有的话），就叫作“早期绑定”。大家以前或许从未听说过这个术语，因为任何程序化语言都没有提供这一选项。C 编译器只有一种方法调用，那就是“早期绑定”。

上述程序最令人迷惑不解的地方全与早期绑定有关，因为在只有一个 Instrument 引用的前提下，编译器不知道具体该调用哪个方法。

解决的办法就是“后期绑定”，它意味着绑定是在运行时间进行，以对象的类型为基础。后期绑定也叫作“动态绑定”或“运行时间绑定”。若一种语言实现了后期绑定，那么同时还要提供一些机制，以便在运行时间正确判断对象类型，并调用适当的方法。也就是说，编

译器此时仍然不知道对象的类型，但方法调用机制能自己去调查，找到正确的方法主体。不同的语言对后期绑定的实现方法是有所区别的。但我们至少可以这样认为：它们都是要在对象中安插某些特殊类型的信息。

Java 的所有方法绑定都采用“后期绑定”技术，除非一个方法已被明确声明成 final。也就是说，我们通常不必关心是否会进行后期绑定——它是自动进行的。

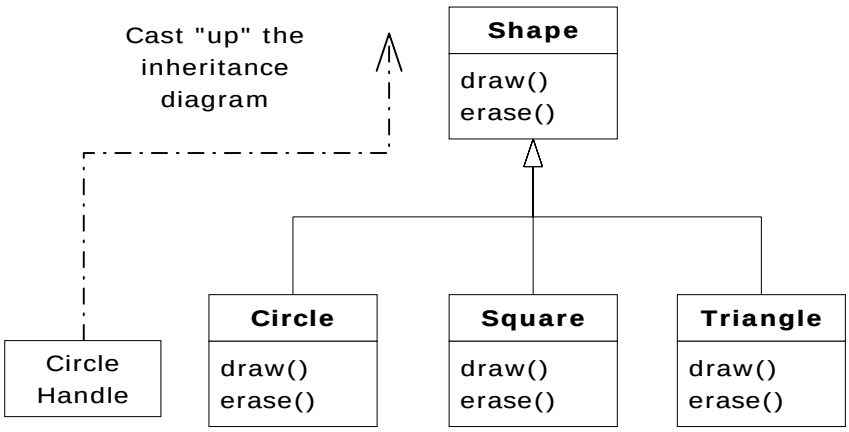
为什么要把一个方法声明成 final 呢？正如上一章指出的那样，它能防止其他人覆盖（改写）那个方法。但也许更重要的一点是，它可有效地“关闭”动态绑定，或者告诉编译器不需要进行动态绑定。这样一来，编译器就可为 final 方法调用生成效率更高的代码。不过在大多数情况下，我们都不会注意到程序的总体运行效率有多大的提高，因此 final 最好还是只作为一种设计时的选项加以考虑，而不要指望能依赖它“显著”改善性能。

7.2.2　产生正确的行为

知道 Java 里绑定的所有方法都通过后期绑定具有多态效果以后，就可相应地编写自己的代码，令其与基类沟通。此时，所有的派生类都保证能用相同的代码正常工作。或者换用另一种方法，我们可以“将一条消息发给一个对象，让对象自行判断要做什么事情。”

在面向对象的程序设计中，有一个经典的“几何形状”例子。由于它很容易用可视化的形式表现出来，所以常常用它说明问题。但不幸的是，它可能会误导初学者将 OOP 想象成是为图形化编程设计的。这一认识当然是错误的！

在几何形状的例子中，我们一个基类，名为 Shape；另外还有大量派生类型：Circle（圆），Square（正方形），Triangle（三角形）等等。大家之所以喜欢这个例子，是由于很容易就能理解象“圆是一类几何形状”这样的概念。下面这幅继承图向我们展示了它们的关系：



向上强制转型可用下面这个简单的语句进行：

```
Shape s = new Circle();
```

在这里，我们创建了 Circle 对象，并将结果引用立即赋给一个 Shape。这表面看起来似乎属于错误操作（将一种类型赋给另一个），但实际是完全可行的——因为根据继承关系，Circle 属于 Shape 的一种。所以编译器认可了上述语句，不会向我们报错。

调用其中一个基类方法时（已在派生类里覆盖）：

```
s.draw();
```

同样地，大家也许希望调用的是 Shape 的 draw()，因为这毕竟是一个 Shape 引用——所以，编译器怎样知道自己该做其他事情呢？此时，同样由于后期绑定（多态）的功劳，才会

调用正确的 Circle.draw()。

下面这个例子从一个稍微不同的角度说明了问题：

```
//: c07:Shapes.java
// Polymorphism in Java.

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()");
  }
}

public class Shapes {
  public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
      default:
      case 0: return new Circle();
      case 1: return new Square();
```

```
      case 2: return new Triangle();
    }
  }
  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] = randShape();
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++)
      s[i].draw();
  }
} ///:~
```

针对从 Shape 派生出来的所有东西，基类 Shape 建立了它们的一个通用接口——也就是说，所有（几何）形状都可以描绘和删除。派生类覆盖了这些定义，为每种特殊类型的几何形状都提供了独一无二的行为。

在主类 Shapes 里，包含了一个 static 方法，名为 randShape()。它的作用是在每次调用它时为某个随机选择的 Shape 对象生成一个引用。请注意向上强制转型是在每个 return 语句里发生的。这个语句取得指向一个 Circle、Square 或者 Triangle 的引用，再将其作为返回类型 Shape 发给方法。所以无论什么时候调用这个方法，都绝对没有机会知道它的具体类型是什么，因为返回的肯定是一个纯 Shape 引用。

main()包含了 Shape 引用的一个数组，其中的数据通过对 randShape()的调用填入。在这个时候，我们只知道自己拥有 Shape，但除此之外不知道任何更具体的情况（编译器同样不知）。然而，当我们在这个数组里步进，并为每个元素都调用 draw()的时候，与各类型有关的正确行为就会魔术般地发生，就象下面这个程序输出展示的那样：

```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

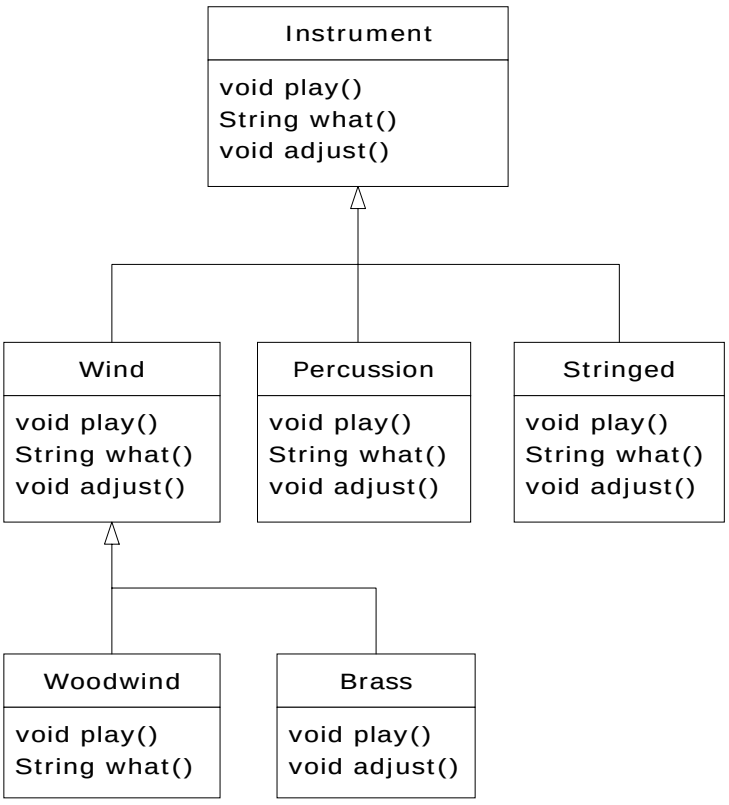
当然，由于几何形状是每次随机选择的，所以每次运行都可能有不同的结果。之所以要突出形状的随机选择，是为了让大家深刻体会这一点：为了在编译时发出正确的调用，编译器毋需获得任何特殊情报。对 draw()的所有调用都是通过动态绑定进行的。

7.2.3　扩展性

现在，让我们仍然返回乐器（Instrument）的例子。由于有了多态，所以可根据自己的需要向系统里加入任意多的新类型，同时毋需更改 true()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从 tune()的模型，而且只与基类接口通信。我们说这样的程序具有“扩展性”，因为可从通用的基类继承新的数据类型，从而新添一些功能。如果是为了适应新类的要求，那么对基类接口进行操纵的方法根本不需要改变，

对于乐器例子，假设我们要在基类里加入更多的方法，以及一系列新类，那么会出现什么情况呢？下面是示意图：



所有这些新类都能与老类——tune()默契地工作，毋需对 tune()作任何调整。即使 tune()位于一个独立的文件里，而将新方法添加到 Instrument 的接口，tune()也能正确地工作，不需要重新编译。下面这个程序是对上述示意图的具体实现：

```
//: c07:music3:Music3.java
// An extensible program.
import java.util.*;

class Instrument {
  public void play() {
    System.out.println("Instrument.play()");
  }
  public String what() {
    return "Instrument";
  }
  public void adjust() {}
}
```

```
class Wind extends Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
```

```
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

新增的方法是 what()和 adjust()。前者返回一个 String 引用，同时返回对那个类的说明；后者使我们能对每种乐器进行调整。

在 main()中，当我们将某样东西放入 Instrument 数组时，就会自动向上强制转型到 Instrument。

可以看到，在围绕 tune()方法的其他所有代码都发生变化的同时，tune()方法却丝毫不受它们的影响，依然故我地正常工作。这正是利用多态希望达到的目标。我们对代码进行修改后，不会对程序中不应受到影响的部分造成影响。此外，我们认为多态是一种至关重要的技术，它允许程序员“将要改变的东西同不会改变的东西分开”。

7.3 覆盖与重载

现在让我们用不同的眼光来看看本章的第一个例子。在下面这个程序中，方法 play()的接口会在被覆盖的过程中发生变化。这意味着我们实际并没有“覆盖”（Override）方法，而是使其“重载”（Overload）。由于编译器允许我们对方法进行重载，所以不会报错。但这种行为可能并不是我们所希望的。下面是一个例子：

```
//: c07:WindError.java
// Accidentally changing the interface.

class NoteX {
  public static final int
    MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}
```

```
class InstrumentX {
  public void play(int NoteX) {
    System.out.println("InstrumentX.play()");
  }
}

class WindX extends InstrumentX {
  // OOPS! Changes the method interface:
  public void play(NoteX n) {
    System.out.println("WindX.play(NoteX n)");
  }
}

public class WindError {
  public static void tune(InstrumentX i) {
    // ...
    i.play(NoteX.MIDDLE_C);
  }
  public static void main(String[] args) {
    WindX flute = new WindX();
    tune(flute); // Not the desired behavior!
  }
} ///:~
```

这里还向大家引入了另一个易于混淆的概念。在 InstrumentX 中，play()方法采用了一个 int（整数）数值，它的标识符是 NoteX。也就是说，即使 NoteX 是一个类名，也可以把它作为一个标识符使用，编译器不会报告出错。但在 WindX 中，play()采用一个 NoteX 引用，它有一个标识符 n。即便我们使用“play(NoteX NoteX)”，编译器也不会报错。这样一来，看起来就象是程序员有意覆盖 play()的功能，只是对方法的类型定义有点儿不对头。然而，编译器此时假定的是程序员有意进行“重载”，而非“覆盖”。请仔细体会这两个术语的区别。“重载”是指同一样东西在不同的地方具有多种含义；而“覆盖”是指它随时随地都只有一种含义，只是原先的含义完全被后来的含义取代了。请注意如果遵守标准的 Java 命名规范，参数标识符就应该是 noteX（n 要小写），这样可把它与类名区分开。

在 tune 中，play()消息会发给“InstrumentX i”，同时将某个 NoteX 成员（MIDDLE_C）作为一个参数使用。由于 NoteX 包含了 int 定义，所以重载的 play()方法的 int 版本会得到调用。同时由于它尚未被“覆盖”，所以使用的是基类版本。

输出如下：

```
InstrumentX.play()
```

显然，这并不象是一个多态方法调用。当然，假如大家能够理解背后发生的事情，就可以非常轻松地解决问题。但是，请想象一下吧，假如错误深深隐藏在一个大型程序的某个地方，那么怎样才能找出问题出在哪里？你可能一辈子都做不到！

7.4 抽象类和方法

在我们所有乐器（Instrument）例子中，基类 Instrument 里面的方法肯定是“伪”方法。如果去调用这些方法，肯定会错。那是由于 Instrument 的设计宗旨是为从它派生出去的所有类都创建一个通用接口。

之所以要建立这个通用接口，唯一的原因就是它能为不同的子类型作出不同的表示。它为我们建立了一种基本形式，使我们能定义在所有派生类里“通用”的一些东西。为阐述这个观念，另一个方法是把 Instrument 称为“抽象基类”（简称“抽象类”）。若想通过该通用接口处理一系列类，就需要创建一个抽象类。对所有与基类声明的签名相符的派生类方法，都可以通过动态绑定机制进行调用（然而，正如上一节指出的那样，如果方法名与基类相同，但参数或参数不同，就会出现重载现象，那或许并非我们所愿意的）。

如果你有一个象 Instrument 那样的抽象类，那么它的对象几乎肯定没什么意义。换言之，Instrument 的作用仅仅是表达接口，而不是表达一些具体的实现。所以创建一个 Instrument 对象是没有意义的，而且我们通常都应禁止用户那样做。为达到这个目的，可令 Instrument 内的所有方法都显示出错消息。但那样做要到运行时间才会看到结果，而且要求在用户那一方面进行全面、可靠的测试。不管怎样，我们最好的做法都是在编译时间便捕捉到问题！

针对这个问题，Java 专门提供了一种机制，名为“抽象方法”。它属于一种不完整的方法，只含有一个声明，没有方法主体。下面是抽象方法声明时采用的语法：

```
abstract void f();
```

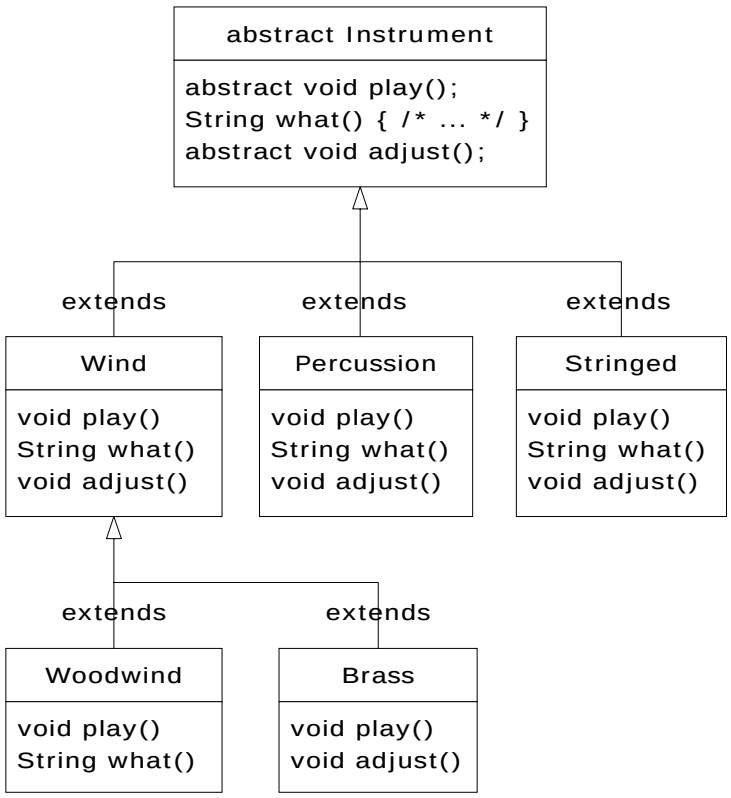
包含了抽象方法的一个类叫作“抽象类”。假如一个类里包含了一个或多个抽象方法，类就必须指定成 abstract（抽象）。否则，编译器会向我们报告一条出错消息。

如果一个抽象类是不完整的，那么一旦有人试图生成那个类的一个对象，编译器会采取什么行动呢？由于不能安全地为一个抽象类创建属于它的对象，所以会从编译器那里获得一条出错提示。通过这种方法，编译器可保证抽象类的“纯洁性”，我们不必担心会误用它。

如果从一个抽象类继承，而且想生成新类型的一个对象，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（你完全可以可以选择），那么派生类也会是抽象的，而且编译器会强迫我们用 abstract 关键字来标志那个类的“抽象”本质。

即使不包括任何 abstract 方法，亦可将一个类声明成“抽象类”。如果一个类没必要拥有任何抽象方法，而且我们想禁止那个类的所有实例，就可选择那样做。

Instrument 类可很轻松地转换成一个抽象类。只有其中一部分方法会变成抽象方法，因为使一个类抽象以后，并不会强迫我们将它的所有方法都同时变成抽象。下面是它看起来的样子：



下面是我们修改过的“管乐器”例子，其中采用了抽象类以及方法：

```
//: c07:music4:Music4.java
// Abstract classes and methods.
import java.util.*;

abstract class Instrument {
  int i; // storage allocated for each
  public abstract void play();
  public String what() {
    return "Instrument";
  }
  public abstract void adjust();
}

class Wind extends Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument {
```

```
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music4 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
```

```
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

可以看出，除了在基类里面，其他地方实际并未发生变化。

创建抽象类和方法有时对我们非常有用，因为它们使一个类的抽象变成明显的事实，可明确告诉用户和编译器自己打算如何用它。

7.5　构造函数和多态

同往常一样，构造函数和其他种类的方法是有所区别的。涉及到多态问题后，这一点同样成立。尽管构造函数并不是多态的（即便可以使用一种“虚拟构造函数”——将在第 12 章介绍），但仍有必要理解构造函数在复杂分级结构中的工作方式，以及它和多态的关系。这是为了给大家打好基础，免得以后碰到一些令人尴尬的问题。

7.5.1　构造函数的调用顺序

构造函数的调用顺序已在第 4 章和第 6 章进行了简要说明，但那是在多态问题引入之前说的话。

用于基类的构造函数肯定会在用于一个派生类的构造函数中调用，而且在继承结构中，逐渐向上“走”，使每个基类使用的构造函数都得以调用。之所以要这样做，是由于构造函数负有一项特殊任务：检查对象是否得到了正确的构建。一个派生类只能访问它自己的成员，不能访问基类的成员（这些成员通常都具有 private 属性）。只有基类的构造函数在初始化自己的元素时才知道正确的方法以及拥有适当的权限。所以，必须令所有构造函数都得到调用，否则整个对象的构建就可能不正确。那正是编译器为什么要强迫对派生类的每个部分进行构造函数调用的原因。在派生类的构造函数主体中，若我们没有明确指定对一个基类构造函数的调用，它就会“默默”地调用默认构造函数。如果不存在默认构造函数，编译器就会报告一个错误（若某个类没有构造函数，编译器会自动合成一个默认构造函数）。

下面让我们来看一个例子，它展示了按构建顺序进行合成、继承以及多态的效果：

```
//: c07:Sandwich.java
// Order of constructor calls.

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()");
  }
}

class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
} ///:~
```

这个例子在其他类的外部创建了一个复杂的类，而且每个类都有一个构造函数对自己进行了宣布。其中最重要的类是 Sandwich，它反映出了三个级别的继承（若将从 Object 的默认继承算在内，就是四级）以及三个成员对象。在 main()里创建了一个 Sandwich 对象后，输出结果如下：

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
```

```
Sandwich()
```

这意味着对于一个复杂的对象，构造函数的调用遵照下面的顺序：

(1) 调用基类构造函数。这个步骤会不断重复下去，首先构建的是分级结构的根部，然后是下一个派生类，等等……直至走得最远的一级派生类。

(2) 按声明顺序调用成员初始化模块。

(3) 调用派生构造函数主体。

构造函数调用的顺序是非常重要的。进行继承时，我们知道有关基类的一切，并且能访问基类的任何 public 和 protected 成员。这意味着当我们在派生类的时候，必须能假定基类的所有成员都是有效的。采用一种标准方法，构建行动已经进行，所以对象所有部分的成员均已得到构建。但在构造函数内部，必须保证使用的所有成员都已构建。为达到这个要求，唯一的办法就是首先调用基类构造函数。然后在进入派生类构造函数以后，我们在基类能够访问的所有成员都已得到初始化。此外，所有成员对象（亦即通过合成放到类里的对象）在类内进行定义的时候（比如上例中的 b、c 和 l），由于我们应尽可能地对它们进行初始化，所以也应保证构造函数内部的所有成员均为有效。若坚持按这一规则行事，会有助于我们确定所有基类成员以及当前对象的成员对象均已获得正确的初始化。但不幸的是，这种做法并不适用于所有情况，这将在下一节具体说明。

7.5.2　继承和 finalize()

通过“合成”方法创建新类时，永远不必担心那个类成员对象的收尾工作。每个成员都是一个独立的对象，所以会得到正常的垃圾收集以及收尾处理——无论它是不是不自己某个类一个成员。但在进行初始化时，必须覆盖派生类中的 finalize()方法——如果已设计了某个特殊的清除进程，要求它必须作为垃圾收集的一部分进行。覆盖派生类的 finalize()时，务必记住调用 finalize()的基类版本。否则，对基类的收尾根本不会进行。下面这个例子便是明证：

```
//: c07:Frog.java
// Testing finalize with inheritance.

class DoBaseFinalization {
  public static boolean flag = false;
}

class Characteristic {
  String s;
  Characteristic(String c) {
    s = c;
    System.out.println(
      "Creating Characteristic " + s);
  }
  protected void finalize() {
    System.out.println(
      "finalizing Characteristic " + s);
  }
}
```

```
class LivingCreature {
  Characteristic p =
    new Characteristic("is alive");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void finalize() throws Throwable {
    System.out.println(
      "LivingCreature finalize");
    // Call base-class version LAST!
    if(DoBaseFinalization.flag)
      super.finalize();
  }
}

class Animal extends LivingCreature {
  Characteristic p =
    new Characteristic("has heart");
  Animal() {
    System.out.println("Animal()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Animal finalize");
    if(DoBaseFinalization.flag)
      super.finalize();
  }
}

class Amphibian extends Animal {
  Characteristic p =
    new Characteristic("can live in water");
  Amphibian() {
    System.out.println("Amphibian()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Amphibian finalize");
    if(DoBaseFinalization.flag)
      super.finalize();
  }
}

public class Frog extends Amphibian {
  Frog() {
```

```
    System.out.println("Frog()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Frog finalize");
    if(DoBaseFinalization.flag)
      super.finalize();
  }
  public static void main(String[] args) {
    if(args.length != 0 &&
       args[0].equals("finalize"))
       DoBaseFinalization.flag = true;
    else
      System.out.println("Not finalizing bases");
    new Frog(); // Instantly becomes garbage
    System.out.println("Bye!");
    // Force finalizers to be called:
    System.gc();
  }
} ///:~
```

其中，DoBaseFinalization 类只是简单地容纳了一个标志，向分级结构中的每个类指出是否应调用 super.finalize()。这个标志的设置建立在命令行参数的基础上，所以能在进行和不进行基类收尾工作的前提下查看行为。

分级结构中的每个类也包含了 Characteristic 类的一个成员对象。大家可以看到，无论是否调用了基类收尾模块，Characteristic 成员对象都肯定会得到收尾（清除）处理。

每个被覆盖的 finalize()至少要拥有对 protected 成员的访问权力，因为 Object 类中的 finalize()方法具有 protected 属性，而编译器不允许我们在继承过程中消除访问权限（“友好的”比“受到保护的”具有更小的访问权限）。

在 Frog.main()中，DoBaseFinalization 标志会得到配置，而且会创建单独一个 Frog 对象。请记住垃圾收集（特别是收尾工作）可能不会针对任何特定的对象发生，所以为了强制采取这一行动，对 System.gc()的调用会主动触发垃圾收集，因而进行收尾。如果没有基类收尾，那么输出是：

```
Not finalizing bases
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
Bye!
Frog finalize
finalizing Characteristic is alive
```

```
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

从中可以看出确实没有为 Frog 的基类调用收尾模块（但成员对象确实进行了收尾）。但假如在命令行加入“finalize”参数，便会获得下述结果：

```
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
Amphibian finalize
Animal finalize
LivingCreature finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

尽管成员对象用与它们创建时相同的顺序进行收尾，但从技术角度说，并没有指定对象收尾的顺序。但对于基类，我们可对收尾顺序进行控制。采用的最佳顺序正是在这里采用的顺序，它与初始化顺序正好相反。按照与 C++中用于“破坏器”相同的形式，我们应该首先执行对派生类的收尾，再是基类的收尾。这是由于派生类的收尾可能调用基类中相同的方法，要求基类组件仍然处于活动状态。所以，千万不能过早地把它们“破坏”掉了。

7.5.3 构造函数内部的多态方法的行为

构造函数调用的分级结构（顺序）为我们带来了一个有趣的问题，或者说让我们进入了一种进退两难的局面。若当前位于一个构造函数的内部，同时调用准备构建的那个对象的一个动态绑定方法，那么会出现什么情况呢？在原始的方法内部，我们完全可以想象会发生什么——动态绑定的调用会在运行期间进行解析，因为对象不知道它到底从属于方法所在的那个类，还是从属于从它派生出来的某些类。为保持一致性，大家也许会认为这应该在构造函数内部发生。

但实际情况并非完全如此。若调用构造函数内部一个动态绑定的方法，会使用那个方法被覆盖的定义。然而，产生的效果可能并不如我们所愿，而且可能造成一些难于发现的程序错误。

从概念上讲，构造函数的职责是让对象实际进入存在状态。在任何构造函数内部，整个对象可能只是得到部分组织——我们只知道基类对象已得到初始化，但却不知道哪些类已经继承。然而，一个动态绑定的方法调用却会在分级结构里“向前”或者“向外”前进。它调用位于派生类里的一个方法。如果在构造函数内部做这件事情，那么对于调用的方法，它要操纵的成员可能尚未得到正确的初始化——这显然不是我们所希望的。

通过观察下面这个例子，这个问题便会昭然若揭：

```
//: c07:PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect.

abstract class Glyph {
  abstract void draw();
  Glyph() {
    System.out.println("Glyph() before draw()");
    draw();
    System.out.println("Glyph() after draw()");
  }
}

class RoundGlyph extends Glyph {
  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 {
  public static void main(String[] args) {
    new RoundGlyph(5);
  }
} ///:~
```

在 Glyph 中，draw()方法是“抽象的”（abstract），所以它可以被其他方法覆盖。事实上，我们在 RoundGlyph 中也不得不对其进行覆盖。但 Glyph 构造函数会调用这个方法，而且调用会在 RoundGlyph.draw()中止，这看起来似乎是有意的。但请看看输出：

```
Glyph() before draw()
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

当 Glyph 的构造函数调用 draw()时，radius 的值甚至不是默认的初始值 1，而是 0。这

可能是由于一个点号或者屏幕上根本什么都没有画而造成的。这样就不得不开始查找程序中的错误，试着找出程序不能工作的原因。

前一节讲述的初始化顺序并不十分完整，那正是关键之所在！初始化的实际过程是这样的：

(1) 在采取其他任何操作之前，为对象分配的存储空间初始化成二进制 0。

(2) 就象前面叙述的那样，调用基类构造函数。此时，被覆盖的 draw()方法会得到调用（的确是在 RoundGlyph 构造函数调用之前），此时会发现 radius 的值为 0，这是由于步骤(1)造成的。

(3) 按照原先声明的顺序调用成员初始化代码。

(4) 调用派生类构造函数的主体。

采取这些操作要求有一个前提，那就是所有东西都至少要初始化成零（或者某些特殊数据类型与“零”等价的值），而不是仅仅留作垃圾。其中包括通过“合成”技术嵌入一个类内部的对象引用。如果假若忘记初始化那个引用，就会在运行期间出现违例事件。其他所有东西都会变成零，这在观看结果时通常是一个严重的警告信号。

另一方面，应对这个程序的结果提高警惕。从逻辑角度说，我们似乎已进行了无懈可击的设计，所以它的错误行为令人感到非常不可思议。况且，我们根本没从编译器那里收到任何报错（C++在这种情况下会表现出更合理的行为）。象这样的错误会很轻易地被人忽略，而且要花很长的时间才能找出。

因此，设计构造函数时一个特别有用的规则是：用尽可能简单的方法使对象进入就绪状态；如果可能，避免调用任何方法。在构造函数内唯一能安全调用的是在基类中具有 final 属性的那些方法（也适用于 private 方法，它们自动具有 final 属性）。这些方法不能被覆盖，所以不会出现上述问题。

7.6 通过继承进行设计

学习了多态的知识后，由于多态是如此“聪明”的一种工具，所以看起来似乎所有东西都应该继承。但假如过度使用继承技术，也会使自己的设计变得不必要地复杂起来。事实上，当我们以一个现成类为基础建立一个新类时，如首先选择继承，会使情况变得异常复杂。

一个更好的思路是首先选择“合成”——如果不能十分确定自己应使用哪一个。合成不会强迫我们的程序设计进入继承的分级结构中。同时，合成显得更加灵活，因为可以动态选择一种类型（以及行为），而继承要求在编译时间准确地知道一种类型。下面这个例子对此进行了阐释：

```
//: c07:Transmogrify.java
// Dynamically changing the behavior of
// an object via composition.

abstract class Actor {
  abstract void act();
}

class HappyActor extends Actor {
  public void act() {
    System.out.println("HappyActor");
```

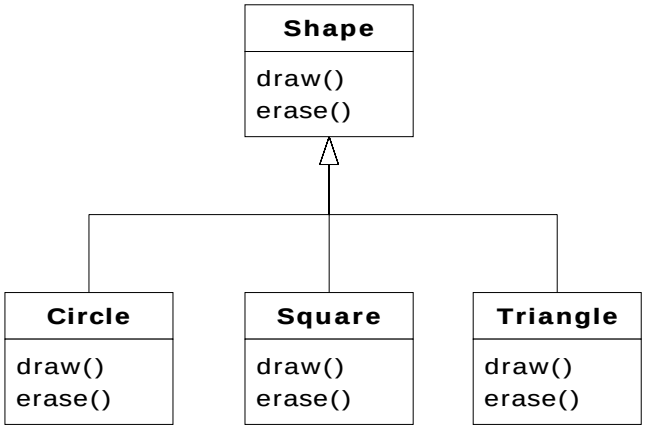
```
  }
}

class SadActor extends Actor {
  public void act() {
    System.out.println("SadActor");
  }
}

class Stage {
  Actor a = new HappyActor();
  void change() { a = new SadActor(); }
  void go() { a.act(); }
}

public class Transmogrify {
  public static void main(String[] args) {
    Stage s = new Stage();
    s.go(); // Prints "HappyActor"
    s.change();
    s.go(); // Prints "SadActor"
  }
} ///:~
```

在这里，一个 Stage 对象包含了指向一个 Actor 的引用，后者被初始化成一个 HappyActor 对象。这意味着 go()会产生特定的行为。但由于引用在运行时间可以重新与一个不同的对象绑定或结合起来，所以 SadActor 对象的引用可在 a 中得到替换，然后由 go()产生的行为发生改变。这样一来，我们在运行时间就获得了很大的灵活性（也把它叫作“状态范式”。详见《Thinking in Patterns》一书，可从 www.BruceEckel.com 下载）。与此相反，我们不能在运行时间换用不同的形式来进行继承；它要求在编译时间完全决定下来。

一条常规的设计准则是：用继承表达行为间的差异，用字段表达状态的变化。在上述例子中，两者都得到了应用：继承了两个不同的类，用于表达 act()方法的差异；而 Stage 通过合成技术允许它自己的状态发生改变。此时，状态的改变同时也产生了行为的变化。

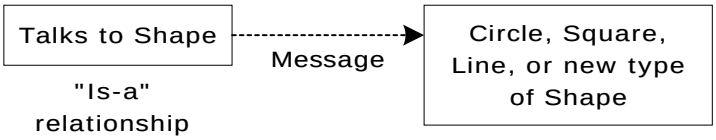
7.6.1　纯继承与扩展

学习继承时，为了创建继承分级结构，看来最明显的方法是采取一种“纯粹”的手段。也就是说，只有在基类或“接口”中已建立的方法才可在派生类中被覆盖，如下面这张图所示：



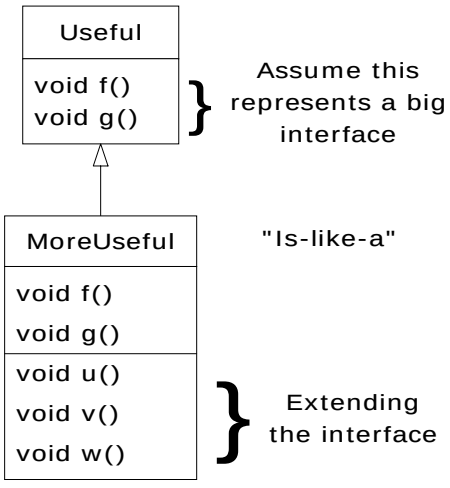
可将其描述成一种纯粹的“属于”关系，因为一个类的接口已规定了它到底“是什么”或者“属于什么”。通过继承，可保证所有派生类都只拥有基类的接口。如按上述示意图操作，派生出来的类除了有基类的接口之外，不会再拥有其他什么。

可将其想象成一种“纯替换”，因为派生类对象可针对基类完美地替换掉。使用它们的时候，我们根本没必要知道与子类有关的任何额外信息。如下所示：

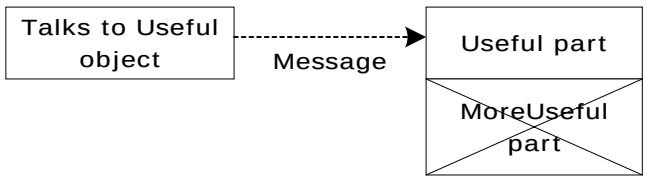


也就是说，基类可接收我们发给派生类的任何消息，因为两者拥有完全一致的接口。我们要做的全部事情就是从派生类开始向上强制转型，而且永远不需要回过头来检查对象的准确类型是什么。所有细节都已通过多态获得了完美的控制。

若按这种思路考虑问题，那么一个纯粹的“属于”关系似乎是唯一明智的设计方法，其他任何设计方法都会导致混乱不清的思路，而且在定义上存在很大的困难。但这相当于又走向了另一个极端。经过细致研究，我们发现假如对接口进行扩展（不幸的是，似乎是在鼓励使用 extends 关键字），那么对一些特定问题来说仍然是一个完美的方案。可将其称为“类似于”关系，因为派生类“类似于”基类——它们有相同的基础接口——但它增加了另外一些特性，要求用其他方法加以实现。如下所示：



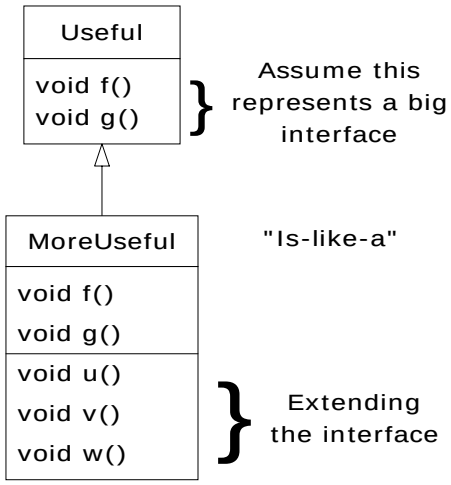
尽管这是一种有用和可行的做法（由具体的环境决定），但它也有一个缺点：派生类中对接口扩展的那一部分不可在基类中使用。所以一旦向上强制转型，就不可再调用新方法：



若此时不进行向上强制转型，就不会碰到此类问题。但在许多情况下，都需要重新核实对象的准确类型，使自己能访问到那个类型的扩展方法。在后面的小节里，我们具体讲述了这是如何实现的。

7.6.2　向下强制转型与运行时间类型标识

由于我们在向上强制转型（在继承结构中向上移动）时丢失了具体的类型信息，所以为了获取具体的类型信息——亦即在分级结构中向下移动——我们必须使用“向下强制转型”技术。然而，我们知道一个向上强制转型肯定是安全的——基类不可能拥有一个比派生类更大的接口。因此，我们通过基类接口发送的每一条消息都肯定能够收到。但在进行向下强制转型时，我们（举个例子来说）并不真的知道一个几何形状实际是一个圆，它完全可能是一个三角形、正方形或者其他形状。



为解决这个问题，必须有一种办法能保证向下强制转型的正确进行。只有这样，我们才不会冒然强制转型成一种错误的类型，然后发出一条对象不可能收到的消息。那样做是非常不安全的。

在某些语言中（如 C++），为了进行“类型安全”的向下强制转型，必须采取特殊的操作。但在 Java 中，所有强制转型都会自动得到检查和核实！所以即使我们只是进行一次普通的括弧强制转型，进入运行时间以后，仍然会毫不留情地对这个强制转型进行检查，保证它的确是我们希望的那种类型。如果不是，就会得到一个 ClassCastException（类强制转型违例）。在程序运行时对类型进行检查的行为叫作“运行时间类型标识”（Run-Time Type Identification，RTTI）。下面这个例子向大家演示了 RTTI 的行为：

```
//: c07:RTTI.java
// Downcasting & Run-time Type
```

```
// Identification (RTTI).
import java.util.*;

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
  }
} ///:~
```

和在示意图中一样，MoreUseful（更有用的）对 Useful（有用的）的接口进行了扩展。但由于它是继承来的，所以也能向上强制转型回一个 Useful。我们可看到这会在对数组 x（位于 main()中）进行初始化时发生。由于数组中的两个对象都属于 Useful 类，所以可将 f()和 g()方法同时发给它们两个。而且假如试图调用 u()（它只存在于 MoreUseful 中），就会收到一条编译时间出错提示。

若想访问一个 MoreUseful 对象的扩展接口，可试着进行向下强制转型。如果它是正确的类型，这一行动就会成功。否则，就会得到一个 ClassCastException 违例。我们不必为这个违例写任何特殊的代码，因为它指出的是一个可能在程序内任何地方发生的一个编程错误。

RTTI 的意义远不仅仅反映在强制转型上。例如，在试图向下强制转型之前，可通过一种方法了解自己处理的是什么类型。事实上，本书整个第 12 章都在讲述 Java 运行时间类型标识的方方面面。

7.7 总　　结

“多态”意味着我们可采取“不同的形式”。在面向对象的程序设计中，我们可以有一个相同的外表（基类的通用接口），同时可以有那个外表的不同使用形式——亦即动态绑定方法的不同版本。

通过这一章的学习，大家已知道假如不利用数据抽象以及继承技术，就不可能理解、甚至创建出多态的一个例子。多态是一种不可独立应用的特性（就象一个 switch 语句），只可与其他元素协同使用。我们应将其当作类总体关系的一部分来看待。人们经常混淆 Java 其他的、非面向对象的特性，如方法重载等，这些特性有时也具有面向对象的某些特征。但不要被愚弄：如果没有后期绑定，多态便无从谈起！

要想使用多态乃至面向对象的各种技术，特别是在自己的程序中，必须将自己的编程视野扩展到不仅包括单独一个类的成员和消息，也要包括类与类之间的一致性以及它们的关系。尽管这要求学习时付出更多的精力，但却是非常值得的，因为只有这样才可真正有效地加快自己的编程速度、更好地组织代码、更容易做出包容面广的程序以及更易对自己的代码进行维护与扩展。

7.8 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 在 Shapes.java 的基类中，新增一个方法，用它打印一条消息，但不在任何派生类对其进行覆盖。请解释此时发生的事情。接着，只在某一个派生类中对其进行覆盖，看看会发生什么事情。最后，在所有派生类中都对其进行覆盖。

(2) 在 Shapes.java 中，添加一种新类型的 Shape，请在 main()中证实多态除了适用于老类型之外，也会对你的新类型发生作用。

(3) 修改 Music3.java，将 what()变成根对象方法 toString()。试着用 System.out.println()打印出 Instrument 对象（不进行任何强制转型）。

(4) 为 Music3.java 添加一种新类型的 Instrument，证实多态可对新类型产生作用。

(5) 修改 Music3.java，令其象 Shapes.java 那样，能够随机地创建 Instrument 对象。

(6) 创建 Rodent（啮齿动物）的一个继承结构：Mouse（老鼠）、Gerbil（鼹鼠）、Hamster（大颊鼠）等等。在基类中，提供通用于所有 Rodent 的方法，并在派生类中覆盖它们，从而根据不同类型的 Rodent 采取不同的行动。创建一个 Rodent 数组，在其中填充不同类型的 Rodent，然后调用自己的基类方法，看看会有什么情况发生。

(7) 修改练习(6)，使 Rodent 成为一个抽象类，将 Rodent 的方法尽可能变成“抽象”的。

(8) 创建一个抽象类，但不在其中包括任何抽象方法，证实自己的确不能再创建那个类的任何实例。

(9) 为 Sandwich.java 增添一个名为 Pickle 的类。

(10) 修改练习(6)，演示出基类和派生类的初始化顺序。接下来，同时为基类和派生类增添成员对象，并演示出在构建过程中，它们被初始化的顺序。

(11) 创建一个三级继承结构。其中每个类都应该有一个 finalize()方法，而且应该正确地调用 finalize()的基类版本。请证实自己的分级结构能正常工作。

(12) 创建一个基类，其中有两个方法。请在第一个方法中，调用第二个方法。继承一

个类，并覆盖第二个方法。创建派生类的一个对象，将其向上强制转型成基类型，并调用第一个方法。解释其间发生的事情。

(13) 创建一个基类，在其中包括一个 abstract print()方法，后者已在一个派生类中覆盖。该方法被覆盖的版本可打印出一个 int 变量的值，而那个变量是在派生类中定义的。定义该变量时，请为其赋一个非零值。在基类构造函数中，调用该方法。在 main()中，创建派生类型的一个对象，然后调用它的 print()方法。请对结果进行解释。

(14) 参照 Transmogrify.java 的例子，创建一个 Starship 类，其中包括一个 AlertStatus 引用，可指示出三种不同的状态。加入适当的方法，对状态进行改变。

(15) 创建一个不含任何方法的抽象类。派生一个类，并加入一个方法。创建一个静态方法，令其取得指向基类的一个引用，将基类向下强制转型为派生类，再调用方法。在 main()中，请证实这样做是否可行。接着，将方法的抽象声明放在基类中，从而免去向下强制转型的必要。

第 8 章　接口和内部类

接口（Interface）和内部类（Inner Class）让我们以更高级的形式，对系统内的对象进行组织和控制。

而在其他语言中（如 C++），并未提供这样的机制，尽管一些水平高超的程序员或许能“模拟”出同样的效果。不过到了 Java 后，它已成为语言的一个重要部分。Java 通过专门的关键字，直接提供了对它的支持。

在第 7 章，大家已学习了 abstract（抽象）关键字，用它可在一个类里创建一个乃至更多个没有定义的方法。事实上，这相当于我们提供了一部分接口，但却没有提供具体的实现，那些细节是由继承者来创建的。另一方面，interface（接口）关键字产生的是一个完全抽象的类，根本没有进行任何具体的实现。大家不久便会知道，如果极端地讲，“接口”不过是一个抽象类而已，因为它可实现与 C++的“多重继承”相似的效果——创建一个类，它可向上强制转型成多种基类型！

最开始，“内部类”好象只不过是一种简单的代码隐藏机制——把一个类放在其他类里。但大家不久就会知道，内部类并非只有这么一点儿作用——它还能调查自己周围的类，并与它们通信。假如能在自己的代码中妥善地运用内部类，肯定会显得更有条理——尽管它的概念对许多人来说都还是全新的。你刚开始可能要多花一些时间，来慢慢习惯用内部类进行设计。

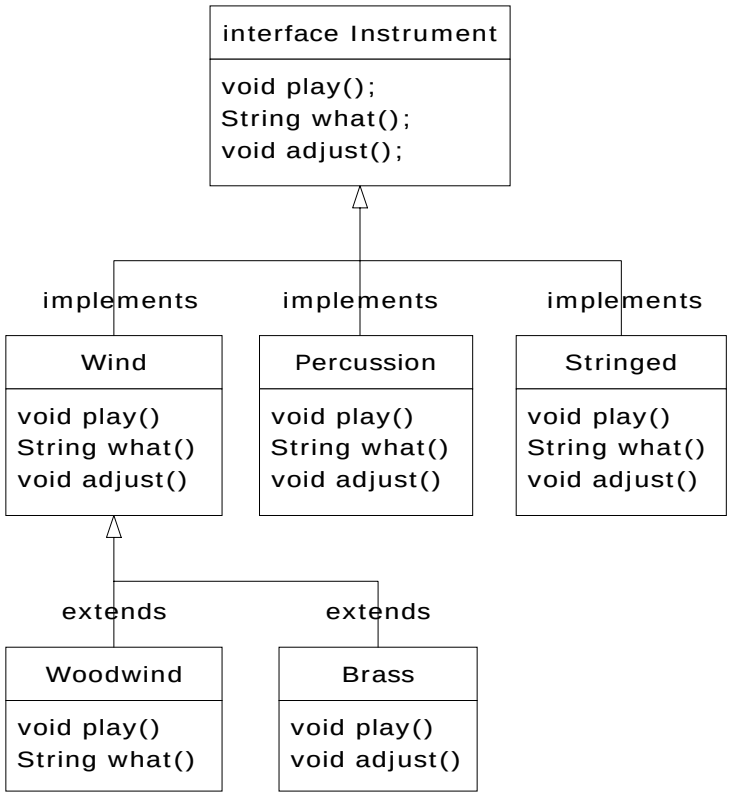
8.1　接　　口

“interface”（接口）关键字使抽象的概念更深入了一层。我们可将其想象为一个“纯”抽象类。它允许创建者规定一个类的基本形式：方法名、参数列表以及返回类型等等，只是不规定方法主体。接口也包含了原始数据类型的数据成员，但它们都默认为 static 和 final。接口只提供一种形式，并不提供实施的细节。

接口这样描述自己：“对于实现了我的所有类，看起来都应该象我现在这个样子”。因此，采用了一个特定接口的所有代码都知道对于那个接口可能会调用什么方法。这便是接口的全部含义。所以我们常把接口用于建立类和类之间的一个“协议”。有些面向对象的程序语言采用了一个名为“protocol”（协议）的关键字，它做的便是和接口同样的事情。

要想创建一个接口，需使用 interface 关键字，而不要用 class。与类相似，我们可在 interface 关键字的前面增加一个 public 关键字（但要求那个接口在一个同名文件内定义）；或干脆省略 public，达成一种“友好”状态，使其只能在同一个封装内使用。

为了生成与一个特定的接口（或一组接口）相符的类，要使用 implements（实现）关键字。用它表达的意思是：“接口看起来就象那个样子了，我这里要说的是它具体是如何工作的”。除这些之外，我们其他的工作都与继承极为相似。下面是乐器例子的示意图：



实现了一个接口以后，就得到了一个正常的类，可用标准方式对其进行扩展。

可决定将一个接口中的方法声明明确定义为“public”。但即便不明确定义，它们也会默认为 public。所以在实现一个接口的时候，来自接口的方法必须定义成 public。否则的话，它们会默认为“友好的”，这样会在继承期间“缩小”一个方法的访问范围——Java 编译器可不允许这样的情况发生。

在 Instrument 例子的修改版本中，大家可明确地看出这一点。注意接口中的每个方法都严格地是一个声明，它是编译器唯一允许的。除此以外，Instrument 中没有一个方法被声明为 public，但它们都会自动获得 public 属性。如下所示：

```
//: c08:music5:Music5.java
// Interfaces.
import java.util.*;

interface Instrument {
  // Compile-time constant:
  int i = 5; // static & final
  // Cannot have method definitions:
  void play(); // Automatically public
  String what();
  void adjust();
}

class Wind implements Instrument {
  public void play() {
```

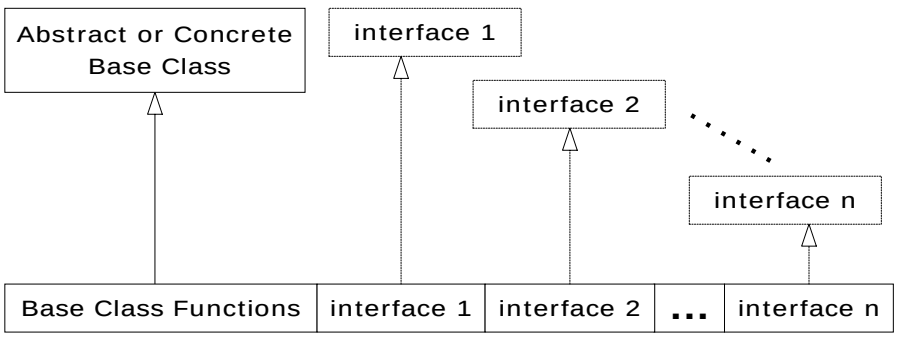
```
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion implements Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed implements Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music5 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play();
```

```
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

代码剩余的部分按相同的方式工作。我们可自由决定向上强制转型成一个名为 Instrument 的“普通”类，一个名为 Instrument 的“抽象”类，或者一个名为 Instrument 的“接口”。所有行为都是相同的。事实上，我们在 tune()方法中可以发现，事实上没有任何证据显示 Instrument 到底是个“普通”类、“抽象”类还是一个“接口”。这是做是故意的：便于程序员以不同的方式，控制对象的创建和使用方式。

8.1.1 Java 的“多重继承”

接口并不只是比抽象类“更纯”的一种形式。它还一些更高级的用途。由于接口根本没有具体的实现——也就是说，没有与存储空间与“接口”关联在一起——所以没有任何办法可以防止多个接口合并到一起。这一点是至关重要的，因为我们经常都需要表达这样一个意思：“x 从属于 a，也从属于 b，也从属于 c”。在 C++中，将多个类合并到一起的行动称作“多重继承”，而且操作起来极为不便，因为每个类都可能有一套自己的实现。在 Java 中，我们可采取同样的行动，但只有其中一个类拥有具体的实现。所以在合并多个接口的时候，C++的“悲剧”不会在 Java 中重演。如下所示：



在一个派生类中，我们并不一定要拥有一个抽象或具体（没有抽象方法）的基类。如果确实想从一个非接口继承，那么只能从一个继承。剩余的所有基本元素都必须是“接口”。

我们将所有接口名置于 implements 关键字的后面，并用逗号分隔它们。可根据需要使用多个接口，而且每个接口都会成为一个独立的类型，可对其进行向上强制转型。下面这个例子展示了一个“具体”类同几个接口合并的情况，它最终生成了一个新类：

```
//: c08:Adventure.java
// Multiple interfaces.
import java.util.*;

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 {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  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
  }
} ///:~
```

从中可以看到，Hero 将 ActionCharacter 这个“具体类”与 CanFight、CanSwim 以及 CanFly 接口合并起来。注意合并时，具体类必须首先出现，然后才是接口（否则编译器会报错）。

请注意 fight()的签名在 CanFight 接口中和在 ActionCharacter 类中是相同的，而且没有在 Hero 中为 fight()提供一个具体的定义。接口的规则是：我们可以从它继承（稍后就会看到），但这样得到的将是另一个接口。如果想创建新类型的一个对象，它就必须是已提供所有定义的一个类。尽管 Hero 没有为 fight()明确地提供一个定义，但定义是随 ActionCharacter 来的，所以这个定义已自动提供了，我们可顺利创建 Hero 的对象。

在类 Adventure 中，我们可看到共有四个方法，它们将不同的接口和具体类作为自己的参数使用。创建一个 Hero 对象后，它可以传递给这些方法中的任何一个。这意味着它们会依次向上强制转型到每一个接口。由于接口是用 Java 设计的，所以这样做不会有任何问题，而且程序员不必对此加以任何特别的关注。

注意上述例子已向我们揭示了接口最关键的作用，也是使用接口最重要的一个原因：能向上强制转型至多个基类。然而，使用接口的第二个原因与使用抽象基类的原因是一样的：防止客户程序员生成这个类的一个对象，以及规定它仅仅是一个接口。这样便带来了一个问题：到底应该使用一个接口还是一个抽象类呢？若使用接口，我们可同时获得抽象类以及接口的好处。所以假如想创建的基类没有任何方法定义或者成员变量，那么无论如何都更愿意用接口，而不要选择抽象类。事实上，如果事先知道某种东西会成为基类，那么第一个选择就是把它变成一个接口。只有在必须使用方法定义或成员变量的时候，才应考虑采用抽象类。

解决合并接口时的名字冲突

不过，在实现多个接口时，也可能会遇遇到一点儿小麻烦。在上述例子中，CanFight 和 ActionCharacter 都有一个一模一样的 void fight()方法。正是由于它们完全一致，所以这里不会出现问题。但是，假如两个方法不一样，又会发生什么问题呢？请看下面这个例子：

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

这时麻烦就来了，因为覆盖、实施和重载非常别扭地混合到一起，而且重载的函数不能仅由返回类型来加以区分。如撤消最后两行代码的注释身份，出错消息便向我们说明了一切：

```
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
```

如果不同的接口注定要合并到一起，但却在其中使用了相同的方法名，通常还会给代码的可读性带来问题。因此，请尽量避免吧！

8.1.2　通过继承扩展接口

利用继承技术，可方便地为一个接口添加新的方法声明，也可将几个接口合并成一个新接口。在这两种情况下，最终得到的都是一个新接口，如下例所示：

```
//: 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 HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  public static void main(String[] args) {
    DragonZilla if2 = new DragonZilla();
    u(if2);
    v(if2);
  }
} ///:~
```

其中，DangerousMonster 是对 Monster 的一个简单的扩展，最终生成了一个新接口。这是在 DragonZilla 里实现的。

Vampire 的语法仅在继承接口时才可使用。通常，只能对单独一个类应用 extends（扩展）关键字。但由于接口可能由多个其他接口构成，所以在构建一个新接口时，extends 可能引用多个基础接口。正如大家看到的那样，接口的名字只是简单地使用逗号分隔。

8.1.3 常数分组

由于置入一个接口的所有字段都自动具有 static 和 final 属性，所以接口是对常数值进行分组的好工具，它具有与 C 或 C++的 enum 非常相似的效果。如下例所示：

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

注意根据 Java 命名规则，假如 static final 中含有常数初始化模块，那么应该全部使用大写字母（同时用下划线分隔一个标识符里的多个单词）。

接口中的字段会自动具备 public 属性，所以没必要专门指定。

现在，通过导入 c08.*或 c08.Months，我们就可从封装外部使用常数——就象对其他任

何封装进行的操作那样。此外，也可以用类似 Months.JANUARY 的表达式对值进行引用。当然，我们获得的只是一个 int，所以不象 C++的 enum 那样拥有额外的类型安全性。但同将数字强行编码（硬编码）到自己的程序中相比，这种技术无疑已经是一个巨大的进步（该技术也得到了我们的普遍使用）。我们通常把采用“硬编码”形式的数字称为“魔法数”，因为它产生的代码是非常难于维护的，令人头晕！

如果确实不想放弃额外的类型安全性，也可构建象下面这样的一个类[34]：

```
//: c08:Month2.java
// A more robust enumeration system.
package c08;

public final class Month2 {
  private String name;
  private Month2(String nm) { name = nm; }
  public String toString() { return name; }
  public final static Month2
    JAN = new Month2("January"),
    FEB = new Month2("February"),
    MAR = new Month2("March"),
    APR = new Month2("April"),
    MAY = new Month2("May"),
    JUN = new Month2("June"),
    JUL = new Month2("July"),
    AUG = new Month2("August"),
    SEP = new Month2("September"),
    OCT = new Month2("October"),
    NOV = new Month2("November"),
    DEC = new Month2("December");
  public final static Month2[] month =  {
    JAN, JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
  };
  public static void main(String[] args) {
    Month2 m = Month2.JAN;
    System.out.println(m);
    m = Month2.month[12];
    System.out.println(m);
    System.out.println(m == Month2.DEC);
    System.out.println(m.equals(Month2.DEC));
  }
} ///:~
```

[34] 是 Rich Hoffarth 的一封 E-mail 触发了我编写这样一个程序的灵感。

类名叫作 Month2，因为标准 Java 库里已经有一个 Month。它是一个 final 类，并含有一个 private 构造函数，所以没人能从它继承，或生成它的一个实例。唯一的实例就是那些 final static 对象，它们是在类的内部创建的，包括、JAN（一月）、FEB（二月）、MAR（三月）……等等。这些对象也在 month 数组中使用，后者让我们能按数字挑选月份，而不是按名字（注意数组中填写了两个 JAN，使偏移量增 1，也使 December 确实成为 12 月）。在 main()中，我们可注意到类型的安全性：m 是一个 Month2 对象，所以只能将其分配给 Month2。而在前面的 Months.java 例子中，由于只提供了 int 值，所以本来想用来代表一个月份的 int 变量最后可能获得一个整数值，那样便不是特别安全。

这儿介绍的方法也允许我们交换使用==或者 equals()，就象 main()尾部展示的那样。

8.1.4 初始化接口中的字段

接口中定义的字段会自动具有 static 和 final 属性。它们不能是“空白 final”，但可初始化成非常数的表达式。例如：

```
//: c08:RandVals.java
// Initializing interface fields with
// non-constant initializers.
import java.util.*;

public interface RandVals {
  int rint = (int)(Math.random() * 10);
  long rlong = (long)(Math.random() * 10);
  float rfloat = (float)(Math.random() * 10);
  double rdouble = Math.random() * 10;
} ///:~
```

由于字段是 static 的，所以它们会在类首次载入时得到初始化；而首次访问任何字段时，便会发生类的装载。下面是一个简单的测试：

```
//: c08:TestRandVals.java

public class TestRandVals {
  public static void main(String[] args) {
    System.out.println(RandVals.rint);
    System.out.println(RandVals.rlong);
    System.out.println(RandVals.rfloat);
    System.out.println(RandVals.rdouble);
  }
} ///:~
```

当然，字段并非接口的一部分，而是保存于那个接口的 static 存储区域中。

8.1.5　嵌套接口

[35]接口可在类内嵌套使用，亦可在其他接口中嵌套。这样一来，我们便可实现一系列非常有趣的功能：

```
//: c08:NestingInterfaces.java

class A {
  interface B {
    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();
  }
```

[35] 本书第一版并无此节，感谢 Martin Danner 在一堂课上提出了这个问题。

```
}

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());
  }
} ///:~
```

想在一个类里嵌套使用一个接口时，要用到的语法应该是非常明了的。和非嵌套接口一样，它也可设为“公共的”或者“友好的”。另外，大家还可看到，无论“公共的”还是“友好的”嵌套接口，它们都可实现为公共的、友好的以及私有的嵌套类。

作为一种新手法，接口也可以是“私有的”（private），就象大家在 A.D 中看到的那样（嵌套接口采用和嵌套类相同的限制语法）。

那么，一个私有的嵌套接口有什么好处呢？大家可能会猜想，就象在 DImp 中那样，它只能象一个“私有的”嵌套类那样实现。但通过 A.DImp2，大家也可以看出，它也能象一个“公共的”类那样实现。但是，A.DImp2 只能“作为自己”使用。我们不能指出它实现了私有接口这一事实。因此，通过实现私有接口，我们可强制那个接口中的方法定义不增添任何类型信息（换言之，不允许进行任何形式的向上强制转型）。

getD()这个方法为我们带来了有关“私有接口”的第二点困惑——它是一个公共方法，返回指向一个私有接口的引用。针对这个方法的返回值，我们可以做什么呢？在 main()中，大家可见有几处均在尝试使用返回值，但均以失败告终。唯一可行的是：将返回值传递给一个有权用它的对象——就目前来说，是通过 received()方法，传递给另一个 A。

接口 E 显示出接口与接口也可相互嵌套。但是，围绕接口制订的规则在这里也必须严格遵守——最重要的一条规则是：所有接口元素都必须设为“public”（公共的）。因此，一个接口嵌套到另一个接口后，它会自动成为“公共的”，不可将其转变成“私有的”。

NestingInterfaces 向我们演示了嵌套接口的各种实现方式。特别要注意的是，在我们具体实现一个接口时，并不需要同时实现其中嵌套的接口。另外，私有接口不可在定义它的那个类外实现。

从表面看，这些特性似乎为我们加了不少的条条框框，有一种让人“束手缚脚”的感觉。但是，只要你真正理解和掌握了一种特性，那么往往都能找到它的用武之地。

8.2　内部类

可将一个类定义置入另一个类定义中。这就叫作“内部类”。内部类对我们非常有用，因为利用它可对那些逻辑上相互联系的类进行分组，并可控制一个类在另一个类里的“可见性”。然而，我们必须认识到内部类与以前讲述的“合成”方法存在着根本的区别。

通常，对内部类的需要并不是特别明显的，至少不会立即感觉到自己需要使用内部类。在本章末尾，介绍完内部类的所有语法之后，大家会发现一个特别的例子。通过它应该可以清晰地认识到内部类的好处。

创建内部类的过程是平淡无奇的，只需将类定义置入外部围绕着它的类内：

```
//: c08:Parcel1.java
// Creating inner classes.
```

```
public class Parcel1 {
  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");
  }
} ///:~
```

若在 ship()内使用，内部类的用法看起来和其他任何类都没什么分别。在这里，唯一明显的区别就是它的名字是嵌套在 Parcel1 里面的。但大家不久就会知道，这其实并非唯一的区别。

更典型的一种情况是，一个外部类拥有一个特殊的方法，它会返回指向一个内部类的引用。就象下面这样：

```
//: 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");
  }
} ///:~
```

如果想在任何一个地方（外部类的非静态方法内部除外）生成内部类的一个对象，必须将那个对象的类型设为“外部类名.内部类名”，就象 main()中展示的那样。

8.2.1　内部类和向上强制转型

迄今为止，内部类看起来仍然没什么特别的地方。毕竟，用它实现隐藏显得有些大题小做。Java 已经有一个非常优秀的隐藏机制——只要让类成为“友好的”就可以了（只在一个封装内部可见）——用不着把它创建成一个内部类。

然而，当我们准备向上强制转型到一个基类（特别是到一个接口）的时候，内部类就开始发挥其关键作用（在实现了接口的一个对象的基础上，产生一个对应的接口引用，其效果和向上强制转型至一个基类是一样的）。这是由于内部类（即接口的具体实现）可以完全进入不可见或不可用状态——对任何人都将如此。所以我们可以非常方便地隐藏实现。最后，我们得到的一切就是指向基类或者接口的一个引用。

刚开始，一系列通用接口可先在它们各自的文件中定义好，以便在后续所有例子中使用：

```
//: c08:Destination.java
public interface Destination {
  String readLabel();
} ///:~
```

```
//: c08:Contents.java
```

```
public interface Contents {
  int value();
} ///:~
```

现在，Contents 和 Destination 代表可由客户程序员使用的接口（记住接口会自动将自己的所有成员都变成 public 属性）。

得到指向基类或接口的一个引用后，我们有可能无法知道准确的类型是什么，就象下面这样：

```
//: c08:Parcel3.java
// Returning a reference to an inner class.

public 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();
  }
}

class Test {
  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();
  }
} ///:~
```

注意由于 main()位于 Test 中，所以在运行该程序时，不能执行 Parcel3，而要用下述命

令行：

```
java Test
```

在这个例子中，main()必须放在一个单独的类中，否则无法演示出 PContents 这个类的“私有”特征。

在 Parcel3 中，一些新东西已被加入：内部类 PContents 是“私有的”，所以除了 Parcel3 之外，其他任何人都不能访问它。PDestination 是“受保护的”，所以除了 Parcel3、Parcel3 封装内的类（因为在“受保护”后，也为封装赋予了访问权；也就是说，“受保护的”相当于“友好的”）以及 Parcel3 的继承者之外，其他任何人都不能访问 PDestination。这意味着客户程序员无法知道这些成员的详情，对它们的访问将受到限制。事实上，你甚至不能向下强制转型到一个私有内部类（或者一个受保护的内部类，除非你自己便是一个继承者），因为这时根本访问不到名字——就象在 class Test 里展示的那样。所以，利用私有内部类，类设计人员可完全禁止其他人依赖类型编码，并可将具体的实现完全隐藏起来。除此以外，从客户程序员的角度来看，一个接口的范围没有意义的，因为他们不能访问不属于公共接口类的其他任何方法。由此，Java 编译器也有机会生成执行效率更高的代码。

普通（非内部）类则不可设为“私有的”或者“受保护的”——只允许是“公共的”或者“友好的”。

8.2.2　方法和作用域中的内部类

至此，我们已基本理解了内部类的典型用途。对那些涉及内部类的代码来说，它们表达的通常都是“纯”内部类——非常简单，也非常容易理解。然而，内部类本身的设计非常全面。在必要的时候，还可选择它们的一些比较“冷僻”的用法。例如，内部类可在一个方法内部创建，甚至可在任意一个作用域内创建——有两方面的原因促使我们这样做：

(1) 正如前面展示的那样，我们准备实现某种形式的接口，使自己能创建和返回一个引用。

(2) 要解决一个复杂的问题，并希望创建一个类，用它为自己的程序方案提供辅助，但同时不愿把它公开。

在下面的一系列例子里，我们打算修改以前的代码，以便用到：

(1) 在一个方法内定义的类

(2) 在方法的一个作用域内定义的类

(3) 一个匿名类，用于实现一个接口

(4) 一个匿名类，用于扩展拥有非默认构造函数的一个类

(5) 一个匿名类，用于执行字段初始化

(6) 一个匿名类，通过实例初始化进行构建（匿名内部类不可拥有构造函数）

一方面，Wrapping 是一个进行了具体实现的普通类；另一方面，Wrapping 也是其派生类的一个通用“接口”：

```
//: c08:Wrapping.java
public class Wrapping {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
} ///:~
```

从中可以看出，Wrapping 含有一个需要指定参数的构造函数，所以情况就变得有点儿

“微妙”了。

第一个例子展示了如何在一个方法的作用域内（而不是另一个类的作用域内）创建一个完整的类：

```
//: 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");
  }
} ///:~
```

PDestination 类属于 dest()的一部分，而不是 Parcel4 的一部分（同时请注意，我们可为相同子目录内每个类内的一个内部类使用类标识符 PDestination，这样做不会发生名字的冲突）。因此，PDestination 不可从 dest()的外部访问。请注意在返回语句中发生的向上强制转型——除指向基类 Destination 的一个引用之外，没有任何东西超出 dest()的边界之外。当然，不能由于类 PDestination 的名字在 dest()内部，就认为在 dest()返回之后 PDestination 不是一个有效的对象。

下面这个例子展示了如何在任意作用域内嵌套一个内部类：

```
//: 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 类嵌套在一个 if 语句的作用域内。这并不意味着类是有条件创建的——它会随其他所有东西一起得到编译。然而，在定义它的那个作用域之外，它是不可用的。除这些之外，它和一个普通类实际并无什么区别。

8.2.3　匿名内部类

下面这个例子看起来有些奇怪：

```
//: 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()方法不仅创建了返回值，也对代表那个返回值的类进行了定义！另外，这个类还是匿名的——居然没有名字。而且更让人摸不着头脑的是，它看起来似乎是先要创建一个 Contents 对象：

```
return new Contents()
```

但在这之后，在分号之前，程序又改变了注意：“等一等，让我先到一个类定义里去一下”：

```
return new Contents() {
  private int i = 11;
  public int value() { return i; }
};
```

这种奇怪的语法要表达的意思是：“创建从 Contents 派生出来的匿名类的一个对象”。由 new 表达式返回的引用会自动向上强制转型成一个 Contents 引用。匿名内部类的语法其实要表达的是：

```
class MyContents implements Contents {
  private int i = 11;
  public int value() { return i; }
}
return new MyContents();
```

在匿名内部类中，Contents 是用一个默认构造函数创建的。下面这段代码展示了假如基类需要一个带有参数的构造函数，那么该如何操作：

```
//: c08:Parcel7.java
// 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) {
      public int value() {
        return super.value() * 47;
      }
    }; // Semicolon required
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Wrapping w = p.wrap(10);
  }
} ///:~
```

也就是说，我们只需将适当的参数传递给基类构造函数，在这里就是在“new Wrapping(x)”中传递 x。匿名类不能拥有一个构造函数，这和在调用 super()时的常规做法不同。

在前述的两个例子中，分号并不标志着类主体的结束（和 C++不同）。相反，它标志着用于包含匿名类的那个表达式的结束。因此，它完全等价于在其他任何地方使用分号。

若想对匿名内部类的一个对象进行某种形式的初始化，此时会出现什么情况呢？由于它是匿名的，没有名字赋给构造函数，所以我们不能拥有一个构造函数。然而，我们可在定义自己的字段时进行初始化：

```
//: c08:Parcel8.java
// An anonymous inner class that performs
// initialization. A briefer version
// of Parcel5.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");
  }
} ///:~
```

若试图定义一个匿名内部类，并想使用在匿名内部类外部定义的一个对象，则编译器要求外部对象为 final 属性。这正是我们将 dest()的参数设为 final 的原因。如果忘记这样做，就会得到一条编译期出错提示。

如果只是想分配一个字段，上述方法便肯定可行。但假如需要采取一些类似于构造函数的行动，又应该怎样操作呢？通过实例初始化，我们事实上可为一个匿名内部类创建一个构造函数：

```
//: c08:Parcel9.java
// Using "instance initialization" to perform
// construction on an anonymous inner class.

public class Parcel9 {
  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);
  }
} ///:~
```

在实例初始化模块中，我们可看到代码不能作为类初始化模块（即 if 语句）的一部分执行。所以实际上，一个实例初始化模块就是一个匿名内部类的构造函数。当然，它的功能是有限的；我们不能对实例初始化模块进行重载处理，所以只能拥有这些构造函数的一个。

8.2.4 链接到外部类

迄今为止，我们见到的内部类好象仅仅是一种名字隐藏以及代码组织方案。尽管这些功能非常有用，但似乎并没有什么特别引人注目的地方。然而，我们还忽略了另一个重要事实。创建一个内部类时，那个类的对象同时拥有指向包围自己的那个对象（外部封装对象或“封套对象”）的一个链接。所以它们能访问那个封装对象的成员——毋需进行任何限定。除此以外，内部类拥有对封装类所有元素的访问权限[36]。下面这个例子阐示了这个问题：

```
//: c08:Sequence.java
// Holds a sequence of Objects.

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private Object[] obs;
  private int next = 0;
  public Sequence(int size) {
    obs = new Object[size];
  }
  public void add(Object x) {
    if(next < obs.length) {
      obs[next] = x;
      next++;
    }
```

[36] 这与 C++“嵌套类”的设计颇有不同，后者只是一种单纯的名字隐藏机制。在 C++中，没有指向一个封装对象的链接，也不存在默认的访问权限。

```
  }
  private class SSelector implements Selector {
    int i = 0;
    public boolean end() {
      return i == obs.length;
    }
    public Object current() {
      return obs[i];
    }
    public void next() {
      if(i < obs.length) i++;
    }
  }
  public Selector getSelector() {
    return new SSelector();
  }
  public static void main(String[] args) {
    Sequence s = new Sequence(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.getSelector();
    while(!sl.end()) {
      System.out.println(sl.current());
      sl.next();
    }
  }
} ///:~
```

其中，Sequence 只是一个大小固定的对象数组，有一个类将其封装在内部。我们调用 add()，以便将一个新对象添加到 Sequence 末尾（如果还有地方的话）。为了取得 Sequence 中的每一个对象，要使用一个名为 Selector 的接口，它使我们能够知道自己是否位于最末尾（end()），能查看当前对象（current() Object），并能移至 Sequence 内的下一个对象（next() Object）。由于 Selector 是一个接口，其他许多类都能用它们自己的方式实现接口，而且许多方法都能将接口作为一个参数使用，从而创建出具有通用性的代码。

在这里，SSelector 是一个私有类，它提供了 Selector 功能。在 main()中，大家可看到 Sequence（序列）的创建过程，在它后面是一系列字串对象的添加。随后，通过对 getSelector() 的一个调用，生成一个 Selector。并用它在 Sequence 中移动，同时选中每一项。

从表面看，SSelector 似乎只是另一个内部类。但不要被表面现象所迷惑。请注意观察 end()，current()以及 next()，它们每个方法都引用了 obs。obs 是个不属于 SSelector 一部分的引用，而是位于外部封装类的一个 private 字段。然而，内部类可以从封装类访问方法与字段，就好象已经“拥有”了它们一样。这一特征对我们来说是非常方便的，就象在上面的例子中看到的那样。

因此，我们现在知道一个内部类可以访问封装类的成员。这是如何实现的呢？内部类必须拥有对封装类的特定对象的一个引用，而封装类的作用就是创建这个内部类。随后，当我

们引用封装类的一个成员时，就利用那个（隐藏）的引用来选择那个成员。幸运的是，编译器会帮助我们照管所有这些细节。但我们现在也可以理解内部类的一个对象只能与封装类的一个对象联合创建。在这个创建过程中，要求对封装类对象的引用进行初始化。若不能访问那个引用，编译器就会报错。进行所有这些操作的时候，大多数时候都不要求程序员的任何介入。

8.2.5 静态内部类

假如不需在内部类对象与外部类对象之间建立一个连接，那么可将内部类设为“静态的”（static）。为正确理解 static 在应用于内部类时的含义，大家必须回忆起我们以前指出的一个要点：对一个普通内部类来说，它的对象会默认持有指向当初创建它的那个外部封装类对象的一个引用。但在将一个内部类设为“静态”之后，这一说法便不再成立了。一个“静态”内部类的含义是：

(1) 要想创建静态内部类的一个对象，我们不需要一个外部类对象。

(2) 不能从静态内部类的一个对象中访问一个外部类对象。

静态内部类和非静态内部类还有另一个方面的差别。对于非静态内部类中的字段和方法来说，它们只能处在一个类的外部级别上，所以非静态内部类不能拥有静态数据、静态字段或者静态内部类。但与此相反的是，所有这些东西都可在静态内部类中拥有：

```
//: c08:Parcel10.java
// Static inner classes.

public class Parcel10 {
  private static class PContents
  implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected static class PDestination
      implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
    // Static inner 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 PDestination(s);
  }
  public static Contents cont() {
    return new PContents();
  }
  public static void main(String[] args) {
    Contents c = cont();
    Destination d = dest("Tanzania");
  }
} ///:~
```

在 main()中，我们不需要 Parcel10 的对象；相反，我们用常规的语法来选择一个静态成员，以便调用将引用返回给 Contents 和 Destination 的方法。

就象大家不久便会学到的那样，在一个普通的（非静态）内部类中，指向外部类对象的链接是用一个特殊的 this 引用来完成的。但对静态内部类来说，却没有这个特殊的 this 引用，这使它看起来就象一个静态方法。

通常，我们不在一个接口里放置任何代码，但 static 内部类可以成为接口的一部分。由于类是“静态”的，所以它并没有违背为接口制订的规则——static 内部类只能放在接口的命名空间内：

```
//: c08:IInterface.java
// Static inner classes inside interfaces.

interface IInterface {
  static class Inner {
    int i, j, k;
    public Inner() {}
    void f() {}
  }
} ///:~
```

在本书早些时候，我建议大家在每个类里都设置一个 main()，将其作为那个类的“测试”床使用。这样做的一个缺点是会产生大量额外的编译码。若不愿如此，可考虑用一个静态内部类来容纳自己的测试代码。如下所示：

```
//: c08:TestBed.java
// Putting test code in a static inner class.

class TestBed {
  TestBed() {}
  void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      TestBed t = new TestBed();
```

```
      t.f();
    }
  }
} ///:~
```

这样可生成一个独立的、名为 TestBed$Tester 的类（要运行程序，请用“java TestBed$Tester”命令）。可将这个类用于测试目的，但不必在程序的最终发行版本中包括它。

8.2.6 引用外部类对象

若想生成对外部类对象的引用，就要在外部类的名字后面加上一个点号以及一个“this”字样。举个例子来说，在 Sequence.SSelector 类中，它的所有方法都能产生外部类 Sequence 的存储引用——只需采用 Sequence.this 的形式。结果获得的引用会自动具备正确的类型（这会在编译时间查实，所以不会产生运行时间的开销）。

有些时候，我们想告诉其他某些对象创建它某个内部类的一个对象。为达到这个目的，必须在 new 表达式中提供指向其他外部类对象的一个引用，就象下面这样：

```
//: 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");
  }
} ///:~
```

要想直接创建内部类的一个对象，不可采用同样的形式，简单地引用外部类名 Parcel11 了事。和许多人猜想的不同，此时必须利用外部类的一个对象来生成内部类的一个对象：

```
Parcel11.Contents c = p.new Contents();
```

因此，除非已拥有外部类的一个对象，否则不可能创建内部类的一个对象。这是由于内部类的对象已同创建它的外部类的对象“默默”地连接到一起。然而，如果生成一个 static 内部类，便不需要指向外部类对象的一个引用。

8.2.7　从一个多重嵌套类中访问外面

[37]不管一个内部类嵌套得有多“深”，都没有关系——它可透明地访问包裹它的所有类的所有成员。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();
      }
    }
  }
}

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，g()和 f()这两个方法不需任何限定便可调用（除了它们都是“私有的”以外）。该例也向大家演示出在创建多重嵌套内部类的对象时（假如在一个不同的类中创建对象），需要采用什么样的语法。其中，“.new”语法可产生正确的作用域，所以不必在构造函数调用中对类名加以限定。

---

[37] 同样感谢 Martin Danner 的建议。

8.2.8 从内部类继承

由于内部类构造函数必须同封装类对象的一个引用联系到一起，所以从一个内部类继承的时候，情况会稍微变得有些复杂。这儿的问题是：对封装类的“秘密”引用必须先获得初始化，而在派生类中不再有一个默认对象可供连接。解决这个问题的办法是采用一种特殊的语法，以便明确建立这种关联：

```
//: 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);
  }
} ///:~
```

从中可以看出，InheritInner 只对内部类进行了扩展，没有扩展外部类。但在需要创建一个构造函数的时候，默认对象已经没有意义，我们不能只是传递封装对象的一个引用。此外，必须在构造函数中采用下述语法：

```
enclosingClassReference.super();
```

它提供了必要的引用，使程序能顺利编译。

8.2.9 内部类可以覆盖吗？

如果先创建一个内部类，再从封装类（外部包裹它的类）继承，然后重新定义内部类，那么会出现什么情况呢？也就是说，我们有可能“覆盖”一个内部类吗？这看起来似乎是一个非常有用的概念，但当你真正去“覆盖”一个内部类时——好象它是外部类的另一个方法，就会发现这一概念实际不能做任何事情：

```
//: c08:BigEgg.java
// An inner class cannot be overriden
// like a method.

class Egg {
  protected class Yolk {
```

```
    public Yolk() {
      System.out.println("Egg.Yolk()");
    }
  }
  private Yolk y;
  public Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  public class Yolk {
    public Yolk() {
      System.out.println("BigEgg.Yolk()");
    }
  }
  public static void main(String[] args) {
    new BigEgg();
  }
} ///:~
```

默认构造函数是由编译器自动合成的，而且会调用基类的默认构造函数。大家或许会认为由于准备创建一个 BigEgg，所以会使用 Yolk 的“被覆盖”版本。但实际情况并非如此。输出如下：

```
New Egg()
Egg.Yolk()
```

这个例子简单地揭示出当我们从外部类继承的时候，没有任何额外的内部类继续下去。两个内部类是两个完全独立的实体，每个都在自己的命名空间内。不过，仍有可能从内部类中明确地继承：

```
//: c08:BigEgg2.java
// Proper inheritance of an inner class.

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 {
  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();
  }
} ///:~
```

现在，BigEgg2.Yolk 明确扩展了 Egg2.Yolk，而且覆盖了它的方法。方法 insertYolk()允许 BigEgg2 将它自己的某个 Yolk 对象向上强制转型至 Egg2 的 y 引用。所以当 g()调用 y.f()的时候，就会使用 f()被覆盖版本。输出结果如下：

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

对 Egg2.Yolk()的第二个调用是 BigEgg2.Yolk 构造函数的基类构造函数调用。调用 g()的时候，可发现使用的是 f()的被覆盖版本。

8.2.10 内部类标识符

由于每个类都会生成一个.class 文件，用于容纳与如何创建这个类型的对象有关的所有信息（这种信息产生了一个名为 Class 对象的“元类”），所以大家或许会猜想：内部类也肯定会生成相应的.class 文件，用来容纳与它们的 Class 对象有关的信息。这些文件或类的名字遵守一种严格的形式：先是封装类的名字，跟一个$，再跟上内部类的名字。例如，由 InheritInner.java 创建的.class 文件包括：

```
InheritInner.class
```

```
WithInner$Inner.class
WithInner.class
```

如果内部类是匿名的，那么编译器会简单地生成数字，把它们作为内部类标识符使用。若内部类嵌套在其他内部类中，则它们的名字会简单地追加在一个$以及外部类标识符的后面。

这种生成内部名称的方法除了非常简单和直观以外，也非常“健壮”，可适应大多数场合的要求[38]。由于它是 Java 的标准命名机制，所以产生的文件会自动具备“与平台无关”的能力（注意 Java 编译器会根据情况改变内部类，使其在不同的平台上都能正常工作）。

8.2.11　为什么要用内部类

到目前为止，大家已接触了对内部类的运作进行描述的大量语法与概念。但这些并不能真正说明内部类存在的原因。为什么 Sun 公司要费如此多的周折，增加这种基本的语言特性呢？

通常，内部类是从一个类继承来的，或者实现了一个接口，而且内部类中的代码可对外部类对象进行操作（内部类就是在这个外部类对象中创建的）。因此，我们可认为一个内部类提供了对外部类的情况进行反映的某种“窗口”。

直逼内部类要害的一个问题是：“如果我需要的只是对一个接口的引用，为何不就让外部类来实现那个接口呢？”答案在于：“如果你要的就是这些，尽管那样做无妨！”既然如此，“实现了接口的内部类和实现了同一个接口的外部类还有什么分别呢？”答案在于：“你不可能永远都能获得接口提供的方便——有时，你还得同具体的实施代码打交道。所以对内部类来说，我们决定用它的最重要的理由是：

“每个内部类都能从一个实施中独立地继承。所以，无论外部类是不是先从一个实施中继承来的，都不关内部类的事！”

假如内部类不具备从多个具体或抽象类继承的能力，那么某些设计和编程问题就会变得非常棘手。所以，对内部类来说，我们也可把它当作“实施多个继承”问题的一个补充方案来看待。接口虽能解决部分问题，但内部类实际是“实现了多个实施继承”。也就是说，内部类实际允许我们从多个“非接口”中继承。

为了更具体地理解这个问题，大家可考虑一种有两个接口的情况，它们必须在一个类内实现。由于接口存在的灵活性，所以此时共有两个选择：单独一个类，或者一个内部类：

```
//: c08:MultiInterfaces.java
// Two ways that a class can
// implement multiple interfaces.

interface A {}
interface B {}

class X implements A, B {}
```

[38] 但另一方面，由于“$”也是 Unix 外壳的一个元字符，所以有时会在列出.class 文件时遇到麻烦。对一家以 Unix 为基础的公司——Sun——来说，采取这种方案显得有些奇怪。我的猜测是他们根本没仔细考虑这方面的问题，而是认为我们自然会将全部注意力都放在源文件上。

```
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，这样做的前提是我们的代码结构不管怎样都富有逻辑性。但是，根据要解决的问题的本质，通都可根据某种准则，指导自己决定是用单独一个类，还是使用一个内部类。但在没有其他任何限制的前提下，在上述的例子中，从实施的角度看，选用的方式实际不会造成太大的差异。两者都可行！

但是，假如此时拥有的是一个抽象或具体类，而不是接口，那么只要你的类需要同时实现两者，就只能使用内部类。如下所示：

```
//: c08:MultiImplementation.java
// With concrete or abstract classes, inner
// classes are the only way to produce the effect
// of "multiple implementation inheritance."

class C {}
abstract class D {}

class Z extends C {
  D makeD() { return new D() {}; }
}

public class MultiImplementation {
  static void takesC(C c) {}
  static void takesD(D d) {}
  public static void main(String[] args) {
    Z z = new Z();
```

```
    takesC(z);
    takesD(z.makeD());
  }
} ///:~
```

当然，假如不需要解决“多个实施继承”的问题，那么随便用什么方法都可以，不必非要使用内部类。不过假如使用了内部类，还可以获得下面这些好处：

(1) 内部类可以有多个实例，每个均有自己的状态信息，且独立于外部类对象的信息。

(2) 在单独一个外部类中，我们可同时使用几个内部类，每个都可实现相同的接口，或者以不同的形式，从同一个类继承。大家不久便会看到一个具体的例子。

(3) 内部类对象并不一定非要在创建外部类对象的时候一并创建。

(4) 内部类并不存在什么默认的、令人混淆的“属于”关系；它是一种独立的实体。

例如，假定 Sequence.java 没有使用内部类，那么我们必须指出：“Sequence 属于 Selector 的一种”。而且对一个特定的 Sequence 来说，只能存在一个 Selector。另外，我们还可以使用另一个名为 getRSelector()的方法，用它生成一个 Selector，以便在序列中向后移动。只有在使用内部类的前提下，才可获得所有这些便利！

封闭和回调

这儿说的“封闭”（Closure）是一种可供调用的对象，保持着来自创建它的作用域的信息。根据这一定义，我们可得出结论：内部类实际就是一种面向对象的“封闭”，因为它除了包含着来自外部类对象（即“创建它的作用域”）的所有信息之外，还自动持有返回整个外部类对象的一个引用，在那里它有权对所有对象进行操作，甚至那些“私有”对象。

由 Java 引发的一场著名争论是在 Java 中到底要不要提供某种形式的指针机制，从而实现“回调”。通过回调，只要为对象指定一点儿信息，以后便可回过头去调用原始对象。这是非常有用的一个概念，大家在第 13 和第 16 章还会进一步认识到它！但是，假如通过指针来实现回调，就必须在很大程度上依赖程序员的正确操作，祈祷他们不会用错了指针。大家知道，Java 在这些问题上往往都非常谨慎，所以最终并未在语言里提供“指针”的概念。

由内部类提供的“封闭”概念则是一种更加完美的方案；和指针相比，它要灵活和安全得多。下面是一个简单的例子：

```
//: c08:Callbacks.java
// Using inner classes for callbacks

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 {
  public void increment() {
    System.out.println("Other operation");
  }
  public 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 {
  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();
  }
} ///:~
```

对于在外部类或内部类中实现一个接口，这个例子也进行了进一步的区分。就编码量来说，Callee1 显然是一种更简单的方案。Callee2 是从 MyIncrement 继承来的，后者已经有一个不同的 increment()方法，它做的事情和 Incrementable 接口原本指望的可没什么关系。将 MyIncrement 继承到 Callee2 后，increment()就不可由 Incrementable 覆盖使用。因此，我们只好用一个内部类，来单独实现它。另外要注意的是，在创建一个内部类时，我们并没有增加或修改外部类的接口。

请注意，在 Callee2 中，除了 getCallbackReference()之外，其他所有东西都是“私有的”。这便使我们到外部世界——interface Incrementable——的连接成为可能！在这里，大家可以看到我们是如何利用接口，将接口同具体实现完全分离开的！

Closure 这个内部类只是实现了 Incrementable，以便提供返回 Callee2 的一个挂钩——但这是一个保险的挂钩。得到了 Incrementable 引用的任何人都只能调用 increment()，并不能进行其他操作（这和指针不同，由于指针不具备这样的保险措施，所以才有“失控”的危险）。

Caller 在它的构造函数中取得一个 Incrementable 引用（尽管对回调引用的捕获可在任何时候进行），然后（有时再晚些时候）利用到“回调”的引用，回过头去调用 Callee 类。

回调的价值当然在于它的灵活性——可在运行时间动态决定调用哪些函数。等到第 13 章，它的优点还会更加凸显，那时我们会在大量地方运用回调，从而实现图形化用户界面（GUI）功能。

8.2.12　内部类和控制框架

运用了内部类的一个更典型的例子见于我在这儿所说的“控制框架”中。

要想理解控制框架，首先必须理解应用程序框架。“应用程序框架”是指一个或者一系列类，它们设计用来解决特定类型的问题。要想使用应用程序框架，我们可从一个或多个类继承，然后覆盖其中的部分方法。我们在覆盖方法中编写的代码用于定制由那些应用程序框架提供的常规方案，以便解决自己的实际问题。“控制框架”则属于应用程序框架的一种特殊类型，根据对事件的响应方式进行设计。在这里，主要用来响应事件的一个系统叫作“由事件驱动的系统”。在应用程序设计语言中，最重要的问题之一便是“图形用户界面”（GUI），它几乎完全是由事件驱动的。正如大家会在第 13 章学习的那样，Java Swing 库便属于一种控制框架，它通过内部类完美地解决了 GUI 的问题。

为理解内部类如何简化控制框架的创建与使用，可认为一个控制框架的工作就是在事件“就绪”以后执行它们。尽管“就绪”的意思很多，但在目前这种情况下，我们却是以计算机时钟为基础。另外，请注意对那些要由控制框架控制的东西来说，框架内并不包含和它们有关的任何特定信息。首先，它是一个特殊的接口，描述了所有控制事件。它可以是一个抽象类，而非一个实际的接口，这是由于默认行为是根据时间控制的。因此，部分实现可能包括：

```
//: c08:controller:Event.java
```

```
// The common methods for any control event.
package c08.controller;

abstract public class Event {
  private long evtTime;
  public Event(long eventTime) {
    evtTime = eventTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= evtTime;
  }
  abstract public void action();
  abstract public String description();
} ///:~
```

希望 Event（事件）运行时，构造函数只需简单地捕获时间。同时 ready()会在到时间运行它时通知我们。当然，ready()也可以在一个派生类中被覆盖，将事件建立在除时间以外的其他东西上。

action()是事件就绪（ready()）后需要调用的方法，而 description()提供了与事件有关的文字信息。

下面这个文件包含了实际的控制框架，用于管理和触发事件。第一个类实际只是一个“助手”类，它的职责是容纳 Event 对象。可用任何适当的集合替换它。而且通过第 9 章的学习，大家会知道有一些现成的集合可简化我们的工作，不需要再编写多余的代码：

```
//: c08:controller:Controller.java
// Along with Event, the generic
// framework for all control systems:
package c08.controller;

// This is just a way to hold Event objects.
class EventSet {
  private Event[] events = new Event[100];
  private int index = 0;
  private int next = 0;
  public void add(Event e) {
    if(index >= events.length)
      return; // (In real life, throw exception)
    events[index++] = e;
  }
  public Event getNext() {
    boolean looped = false;
    int start = next;
    do {
      next = (next + 1) % events.length;
```

```
      // See if it has looped to the beginning:
      if(start == next) looped = true;
      // If it loops past start, the list
      // is empty:
      if((next == (start + 1) % events.length)
         && looped)
        return null;
    } while(events[next] == null);
    return events[next];
  }
  public void removeCurrent() {
    events[next] = null;
  }
}

public class Controller {
  private EventSet es = new EventSet();
  public void addEvent(Event c) { es.add(c); }
  public void run() {
    Event e;
    while((e = es.getNext()) != null) {
      if(e.ready()) {
        e.action();
        System.out.println(e.description());
        es.removeCurrent();
      }
    }
  }
} ///:~
```

在 EventSet（事件集合）中，可容纳 100 个事件（若在这里使用来自第 9 章的一个“真实”容器，就不必担心它的最大尺寸，因为它会根据情况自动改变大小）。index（索引）在这里用于跟踪下一个可用的空间，而 next（下一个）帮助我们寻找列表中的下一个事件，了解自己是否已经循环到头。在对 getNext()的调用中，这一点是至关重要的，因为一旦运行，Event 对象就会从列表中删去（使用 removeCurrent()）。所以 getNext()在列表中向前移动时会遇到“空洞”。

注意 removeCurrent()并不只是设置一些标志，指出对象不再使用。相反，它将引用设为 null。这一点是非常重要的，因为假如垃圾收集器发现一个引用仍在使用，就不会清除对象。若认为自己的引用以后可能象现在这样被“挂起”，那么最好还是将其设为 null，使垃圾收集器能正常地清除它们。

Controller 是进行实际工作的地方。它用一个 EventSet 容纳自己的 Event 对象，而 addEvent()允许我们向这个列表加入新事件。但最重要的方法是 run()。该方法会在 EventSet 中遍历，搜索一个准备就绪（ready()）、可以运行的 Event 对象。对于它发现了 ready()的每一个对象，都会调用 action()方法，打印出 description()，然后将事件从列表中删去。

注意在整个设计中，到目前为止，我们仍然不能准确地知道一个“事件”要做什么。而这正是整个设计的关键——它怎样“将要变化的东西同不变化的东西区分开”？或者用我的话来讲，“变化矩阵”由不同类型 Event 对象的不同行动构成，我们通过创建不同的 Event 子类，从而表达出不同的行动。听起来有些拗口，是吧？多想一下便能明白！

这时正是内部类大显身手的好地方。它们允许我们做两件事情：

(1) 在单独一个类里表达出一个控制框架应用程序所需的全部实现，从而完整地封装与那个实施有关的一切东西。内部类用于表达多种不同类型的 action()，它们用于解决实际问题。除此以外，后面例子还使用了 private 内部类，所以实现会被完全隐藏起来，可供你安全地修改。

(2) 内部类使我们具体的实现能变得更加巧妙，因为能方便地访问外部类的任何成员。若不具备这种能力，代码看起来就可能没那么让人舒服，最后不得不寻求其他解决之道。

现在要请大家思考控制框架的一种具体实施方式，它用来控制温室（Greenhouse）功能[39]。每种具体的行动都是完全不同的：控制灯光、供水以及温度自动调节开关、控制响铃以及重新启动系统等等。但控制框架的设计宗旨是将不同的代码方便地隔离开。利用内部类的概念，在单独一个类中，同一个基类 Event 便可有多个不同的派生版本。针对每种类型的行动，我们都继承了一个新的 Event 内部类，并在 action()内编写相应的控制代码。

作为应用程序框架的一种典型行为，GreenhouseControls 这个类是从 Controller 继承来的：

```
//: 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 c08.controller.*;

public class GreenhouseControls
    extends Controller {
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";
  private class LightOn extends Event {
    public LightOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String description() {
```

[39] 由于某些特殊原因，这对我来说是一个经常需要解决的、非常有趣的问题；原来的例子在《C++ Inside & Out》一书里也出现过，但 Java 提供了一种更令人舒适的解决方案。

```
      return "Light is on";
    }
  }
  private class LightOff extends Event {
    public LightOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String description() {
      return "Light is off";
    }
  }
  private class WaterOn extends Event {
    public WaterOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = true;
    }
    public String description() {
      return "Greenhouse water is on";
    }
  }
  private class WaterOff extends Event {
    public WaterOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = false;
    }
    public String description() {
      return "Greenhouse water is off";
    }
  }
  private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
      super(eventTime);
    }
```

```
    public void action() {
      // Put hardware control code here
      thermostat = "Night";
    }
    public String description() {
      return "Thermostat on night setting";
    }
  }
  private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Day";
    }
    public String description() {
      return "Thermostat on day setting";
    }
  }
  // An example of an action() that inserts a
  // new one of itself into the event list:
  private int rings;
  private class Bell extends Event {
    public Bell(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Ring every 2 seconds, 'rings' times:
      System.out.println("Bing!");
      if(--rings > 0)
        addEvent(new Bell(
          System.currentTimeMillis() + 2000));
    }
    public String description() {
      return "Ring bell";
    }
  }
  private class Restart extends Event {
    public Restart(long eventTime) {
      super(eventTime);
    }
    public void action() {
      long tm = System.currentTimeMillis();
```

```
      // Instead of hard-wiring, you could parse
      // configuration information from a text
      // file here:
      rings = 5;
      addEvent(new ThermostatNight(tm));
      addEvent(new LightOn(tm + 1000));
      addEvent(new LightOff(tm + 2000));
      addEvent(new WaterOn(tm + 3000));
      addEvent(new WaterOff(tm + 8000));
      addEvent(new Bell(tm + 9000));
      addEvent(new ThermostatDay(tm + 10000));
      // Can even add a Restart object!
      addEvent(new Restart(tm + 20000));
    }
    public String description() {
      return "Restarting system";
    }
  }
  public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls();
    long tm = System.currentTimeMillis();
    gc.addEvent(gc.new Restart(tm));
    gc.run();
  }
} ///:~
```

注意 light（灯光）、water（供水）、thermostat（温控）以及 rings（响铃）都隶属于 GreenhouseControls 这个外部类，所以内部类可以毫无阻碍地访问那些字段，毋需限定，也毋需授权。此外，大多数 action()方法都涉及到某种形式的硬件控制，这通常都要求调用一些非 Java 代码来实现。

大多数 Event 类看起来都是相似的，但 Bell（铃）和 Restart（重启）属于特殊情况。Bell 会发出铃声，若尚未振铃足够的次数，它会在事件列表里添加一个新的 Bell 对象，所以以后会再度响铃。请注意内部类看起来为什么总是和“多重继承”差不多——Bell 拥有 Event 的所有方法，而且也拥有外部类 GreenhouseControls 的所有方法。

Restart 负责对系统进行初始化，所以会添加所有必要的事件。当然，一种更灵活的做法是避免进行“硬编码”，而是从一个文件里读入它们（第 11 章的一个练习会要求大家修改这个例子，从而达到这个目标）。由于 Restart()不过是另一个 Event 对象，所以也可以在 Restart.action()里添加一个 Restart 对象，使系统能定时重启。在 main()中，我们需要做的全部事情就是创建一个 GreenhouseControls 对象，并添加一个 Restart 对象，令其工作起来。

这个例子应该使大家对内部类的价值有一个更加深刻的认识——特别是到一个控制框架里运用它们的时候。此外，在第 13 章，大家还会看到如何巧妙地利用内部类描述一个图形用户界面的行为。完成那里的学习后，对内部类的认识将上升到一个前所未有的新高度！

8.3 总　　结

接口和内部类是 Java 的重要特色，它比起其他许多 OOP 语言来都要复杂得多。例如，C++中便没有与此等同的概念。两者联合在一起，便轻松解决了 C++试图用它的多重继承（Multiple Inheritance，简称 MI）解决的同样的问题。但是，由于 C++的 MI 颇难操作，实际用处并不大。相反，Java 的接口和内部类却显得“友好”得多。

尽管本章介绍的各种特性表面上都不难理解，但真正用起来还是有点儿“考人”的。因为对它们的选择实际上是个设计问题，这和“多态”的概念是差不多的。随着时间的推移，你的阅历也会逐渐增多，那时也许能更好地运用它们。就目前来说，只需掌握它的基本语法和用法就行。随着实际经验的增多，对这些语言特性的运用最终会成为你的一种“习惯”。

8.4 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 证明接口中的字段都自动具有 static 和 final 属性。

(2) 在其自己的封装内，创建一个接口，在其中包括三个方法。接着在一个不同的封装内，实现该接口。

(3) 证明接口中的所有方法都自动具有 public 属性。

(4) 在 c07:Sandwich.java 中，创建一个名为 FastFood 的接口（有适当的方法）。再对 Sandwich 进行修改，使其同时也实现了 FastFood。

(5) 创建三个接口，每个都有两个方法。从三个接口继承一个新接口，增加一个新方法。通过实现新接口，并从一个具体类（非抽象类）继承，从而创建一个类。接着写四个方法，每个都采用四个接口之一作为自己的参数。在 main()中，创建该类的一个对象，然后把它传递给每一个方法。

(6) 修改练习(5)，创建一个抽象类，并把它继承到派生类。

(7) 修改 Music5.java，增加一个 Playable 接口。从 Instrument 中删去 play()声明。在派生类中增加 Playable，把它包括到 implements 列表中去。修改 tune()，使其用一个 Playable 作为参数，而不是 Instrument。

(8) 修改第 7 章的练习(6)，将 Rodent 变成一个接口。

(9) 在 Adventure.java 中，增加一个名为 CanClimb 的接口，采用和其他接口相同的形式。

(10) 写一个程序，导入并使用 Month2.java。

(11) 参照 Month2.java 中的例子，对星期 x（星期一到星期日）进行列举。

(12) 创建一个接口，其中至少包括一个方法（在它自己的封装内）。再在另一个独立的封装里创建一个类。增加一个“受保护的”的内部类，令其实现上述接口。在第三个封装内，从那个类继承，然后在一个方法里，返回“受保护的”内部类的一个对象。返回期间，将其向上强制转型到接口。

(13) 创建一个接口，其中至少包括一个方法，然后在方法里定义一个内部类，从而实现那个接口，令方法返回指向那个接口的一个引用。

(14) 重复练习(13)，但改为用一个方法在一个作用域内定义内部类。

(15) 用一个匿名内部类重复练习(13)。

(16) 创建一个私有内部类，令其实现一个公共接口。写一个方法，用它返回指向私有

内部类的一个实例的引用，向上强制转型到接口。证明在你试图向下强制转型的时候，内部类会被完全隐藏起来。

(17) 创建一个类，在其中包括一个非默认构造函数，同时不指定默认构造函数。创建另一个类，在其中包括一个方法，可返回指向第一个类的引用。创建一个对象，让一个匿名内部类从第一个类继承，从而返回该对象。

(18) 创建一个类，其中含有一个私有字段和一个私有方法。再创建一个内部类，其中包括一个方法，可对外部类字段进行修改，并可调用外部类的方法。在第另一个外部类方法中，创建内部类的一个对象，并调用其方法，说明对外部类对象的影响。

(19) 用一个匿名内部类，重复练习(18)。

(20) 创建一个类，其中包括一个静态内部类。在 main()中，创建内部类的一个实例。

(21) 创建一个接口，其中包括一个静态内部类。实现该接口，并创建内部类的一个实例。

(22) 创建一个类，在其中包括一个内部类，后者又包括了另一个内部类。用静态内部类重复这一过程。请注意由编译器生成的.class 文件的名字。

(23) 创建一个类，其中含有一个内部类。在一个单独的类里，生成内部类的一个实例。

(24) 创建一个类，在其中包括一个内部类，后者有一个非默认的构造函数。接着创建另一个类，在其中包括一个从第一个内部类继承来的新内部类。

(25) 改正在 WindError.java 中存在的问题。

(26) 修改 Sequence.java，增加一个名为 getRSelector()的方法，用它对 Selector 作不同的实现，以便从序列的末尾到开头，进行逆向移动。

(27) 创建接口 U，它带有三个方法。创建类 A，通过构建一个匿名内部类，它的一个方法能生成对 U 的一个引用。创建类 B，在其中包括由 U 构成的一个数组。B 总共有三个方法：第一个方法负责在数组里接收和保存对 U 的引用；第二个方法可将数组里的一个引用（由方法的参数决定）设为 null；第三个方法可在数组里移动，并调用 U 中的方法。在 main()中，创建一组 A 对象以及一个 B。在 B 中填入由一系列 A 对象产生的 U 引用。用 B 回调所有 A 对象。最后，从 B 中删除某些 U 引用。

(28) 在 GreenhouseControls.java 中，增加 Event 内部类，负责风扇的开和关。

(29) 证明一个内部类有权访问其外部类的私有元素；再证实一下，反过来行吗？

第 9 章 对象的容纳

假如一个程序只含有数量固定的对象，而且提前知道了它们的存在时间，那么这个程序也未免太简单了！

在常规应用中，我们的程序往往要“复杂”一些。通常，我们需要在程序中创建一些特殊的新对象，而它们创建的标准要等进入程序的运行时间才可获知。若非程序正式运行，否则根本不知道到底需要多少数量的对象，甚至不知道它们的准确类型。为了满足这种常规编程的需要，我们要求能够在任何时候、任何地点创建任意数量的对象。因此，我们不可依赖一个已命名的引用来容纳自己的每一个对象，就象下面这样：

```
MyObject myReference;
```

因为根本不知道最终需要多少个这样的对象。

为了解决这个非常关键的问题，Java 为我们设计了容纳对象（或对象引用）的多种方式。其中，语言内置的方式是“数组”，我们之前已研究过它。在这一章，则准备加深大家对它的认识。此外，Java 的工具（实用程序）库提供了一套完整的“容器类”（亦称作“集合类”，但由于 Java 2 的函数库已经用“集合”这个词来称呼库的一个特殊子集，所以这儿换用更为广义的“容器”一词）。使用容器，我们可在其中容纳自己的对象，甚至能对它们进行操纵。

9.1 数 组

对数组的大多数必要的介绍已在第 4 章的最后一节进行。通过那里的学习，大家已知道自己该如何定义及初始化一个数组。对象的容纳是本章的重点，而数组只是容纳对象的一种方式。但由于还有其他大量方式可容纳数组，所以数组的优势何在呢？

有两方面的因素将数组与其他类型的容器区分开：效率和类型。对于 Java 来说，要想保存和随机访问一系列对象（实际是对象引用），效率最高的方法莫过于数组。数组实际代表一个简单的线性序列，它使得元素的访问速度非常快，但我们却要为这种速度付出代价。这个代价就是：当你创建一个数组对象时，它的大小是固定的，而且不可在那个数组对象的“存在时间”内发生改变。当然，你也可以创建大小固定的一个数组，然后假如用光了存储空间，就再创建一个新数组，将所有引用从旧数组移到新数组。这属于“数组列表”（ArrayList）类的行为，本章稍后还会详细讨论它。然而，由于要为这种能自由改变大小的灵活性付出较高的代价，所以我们认为 ArrayList 实际工作起来的效率并没有数组高。

C++的矢量类（Vector）知道自己容纳的是什么类型的对象，但同 Java 的数组相比，它却有一个明显的缺点：C++矢量类的 operator[]不能进行范围（边界）检查，所以很容易越界[40]。在 Java 中，无论使用的是数组还是容器，都会进行范围检查——若超过边界，就会获得一个 RuntimeException（运行时间违例）错误。正如大家在第 10 章会学到的那样，这类违例指出的是一个由于程序员造成的错误，所以作为客户程序员，不需要在自己的代码中检查它。而另一方面，由于 C++的 vector 不进行范围检查，所以访问速度也比较快——在 Java 中，

[40] 然而，它也许能查询到 vector 有多大，而且 at()方法也确实能进行范围检查。

由于对数组和容器都要进行范围检查，所以会对性能造成一定的影响。

本章还要学习另几种常见的容器类——它们是 List、Set 和 Map——所有这些类都能把对象当作没有具体类型那样对待。换言之，它们将其当作 Object 类型处理（Object 类型是 Java 中所有类的“根”类）。从某种角度看，这种处理方法是非常合理的：我们只需构建一个容器，然后所有 Java 对象都可进入那个容器（原始数据类型除外——可用 Java 的“基类型封装器”类将其作为常数置入容器；或自建一个类，把它们封装到里面，当作可变值进行对待）。这再一次体现出数组相较于普通容器的优越性：创建一个数组时，可让它容纳一种特定的类型。这意味着可进行编译时间的类型检查，防范自己设置了错误的类型，或者错误地提取了一种类型。当然，在编译或运行时间，Java 会自动防止我们将不恰当的消息发给一个对象。所以两种方法相比，哪一种都不比另一种更加危险——只要编译器能及时地指出错误，运行时间能加快速度，而且尽量避免拿“违例”去骚扰用户，目的便达到了！

考虑到执行效率和类型检查，应优先采用数组。然而，当我们试图解决一个更常规的问题时，数组的局限性也可能变得非常明显。因此，在讨论过数组之后，本章剩下的部分将把重点转到 Java 提供的容器类身上。

9.1.1 数组是对象

无论使用的数组属于什么类型，数组标识符实际都是指向真实对象的一个引用。那些对象本身是在内存“堆”里创建的。堆对象既可“隐式”创建（即默认产生），亦可“显式”创建（即明确指定，用一个 new 表达式）。堆对象的一部分（实际是我们能访问的唯一字段或方法）是只读的 length（长度）成员，它告诉我们那个数组对象里最多能容纳多少个元素。对于数组对象，“[]”语法是我们唯一能够采取的另一种访问方法。

下面这个例子展示了对数组进行初始化的不同方式，以及如何将数组引用分配给不同的数组对象。它也揭示出对象数组和原始数据类型数组在使用方法上几乎是完全一致的。唯一的差别在于对象数组容纳的是引用，而原始数据类型数组容纳的是具体的数值。

```
//: c09:ArraySize.java
// Initialization & re-assignment of arrays.

class Weeble {} // A small mythical creature

public class ArraySize {
  public static void main(String[] args) {
    // Arrays of objects:
    Weeble[] a; // Null reference
    Weeble[] b = new Weeble[5]; // Null references
    Weeble[] c = new Weeble[4];
    for(int i = 0; i < c.length; i++)
      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);
  }
} ///:~
```

输出如下：

```
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
a.length = 2
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
```

其中，数组 a 只是初始化成一个 null 引用。此时，编译器会禁止我们对这个引用作任何实际操作，除非已正确地初始化了它。数组 b 被初始化成指向由 Weeble 引用构成的一个数组，但那个数组里实际并未放置任何 Weeble 对象。然而，我们仍然可以查询那个数组的大小，因为 b 指向的是一个合法对象。这也为我们带来了一个难题：不知道那个数组里实际包含了多少个元素，因为 length 只会告诉我们最多可将多少元素置入那个数组。换言之，我们只知道数组对象的大小或容量，不知其实际容纳了多少元素。不过，由于数组对象在创建之初会自动初始化成 null，所以可检查它是否为 null，从而判断数组内一个特定的位置是否正容纳着一个对象。类似地，由原始数据类型构成的数组会自动初始化成零（针对数值类型）、(Char)0（针对字符类型）或者 false（针对布尔类型）。

数组 c 显示出我们首先创建一个数组对象，再将 Weeble 对象赋给那个数组内的所有“位置”。数组 d 揭示了“容器初始化”采用的语法，它可创建数组对象（默认为在内存堆中使用 new 命令，这和数组 c 类似），然后用 Weeble 对象进行初始化，全部工作均在一条语句内完成。

接下来的数组初始化可想象成一种“动态集合初始化”。d 使用的集合初始化必须在 d 定义的同时使用；但采用第二种语法，我们就可在任何地方创建和初始化一个数组对象。举个例子来说，假定 hide()方法用于取得由 Weeble 对象构成的一个数组。那么要想调用它，可采用：

```
hide(d);
```

但对于一个想作为参数传递的数组，也可以动态地创建它：

```
hide(new Weeble[] { new Weeble(), new Weeble() });
```

某些情况下，用这种新形式的语法可简化代码的编写。

来看看下面这个表达式：

```
a = d;
```

它向我们展示了如何取得同一个数组对象连接起来的引用，并将其赋给另一个数组对象——这和我们对其他类型的对象引用采取的操作是一样的。现在，a 和 d 都指向内存堆里面的同一个数组对象。

ArraySize.java 的第二部分向我们揭示出：就工作方式而言，由原始数据类型构成的数组和由对象构成的数组是“大致”相同的，唯一的例外就是前者直接容纳了实际的值，而不

是引用。

原始数据类型容器

容器类只能容纳对象引用。但对一个数组来说，却既可令其直接容纳基类型的数据，亦可容纳指向对象的引用。利用象 Integer、Double 之类的“封装器”或“包裹”（Wrapper）类，可将原始数据类型的值置入一个容器里。不过，这样做显得有些“笨”。而且，比起封装了原始数据类型的一个容器，创建和访问一个数组的效率要高得多！

当然，假如准备一种原始数据类型，同时又想要容器的灵活性（在需要的时候可自动扩展，腾出更多的空间），就不宜使用数组，必须使用由封装的数据构成的一个容器。大家或许认为针对每种原始数据类型，都应有一种特殊类型的 ArrayList。但 Java 并未提供这一特性。不过，某种形式的模板机制或许会在某一天帮助 Java 更好地解决这个问题[41]。

9.1.2 数组的返回

假定我们现在想写一个方法，同时不希望它仅仅返回一样东西，而是想返回一系列东西。此时，象 C 和 C++这样的语言会使问题复杂化，因为我们不能返回一个数组，只能返回指向数组的一个指针。这样就非常麻烦，因为很难控制数组的“存在时间”，所以很容易造成内存“漏洞”的出现。

Java 采用的尽管是类似的方法，但我们能“返回一个数组”。当然，此时返回的实际还是指向数组的一个指针。但在 Java 里，我们永远不必担心那个数组的是否可用——只要需要，它就会自动存在。而且垃圾收集器会在我们完成后自动将其清除。

作为一个例子，请思考如何返回一个字串数组：

```
//: c09:IceCream.java
// Returning arrays from methods.

public class IceCream {
  static String[] flav = {
    "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  static String[] flavorSet(int n) {
    // Force it to be positive & within bounds:
    n = Math.abs(n) % (flav.length + 1);
    String[] results = new String[n];
    boolean[] picked =
      new boolean[flav.length];
    for (int i = 0; i < n; i++) {
      int t;
```

[41] 这正是 C++比 Java 做得好的一个地方，因为 C++通过 template（模板）关键字提供了对“参数化类型”的支持。

```
      do
        t = (int)(Math.random() * flav.length);
      while (picked[t]);
      results[i] = flav[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(flav.length);
      for(int j = 0; j < fl.length; j++)
        System.out.println("\t" + fl[j]);
    }
  }
} ///:~
```

其中，flavorSet()方法创建了一个名为 results 的字串数组。该数组的大小为 n——具体数值取决于我们传递给方法的参数。随后，它从数组 flav 里随机挑选一些“香料”（Flavor），并将它们置入 results 里，并最终返回 results。返回数组与返回其他任何对象没什么区别——最终返回的都是一个引用。至于数组到底是在 flavorSet()里创建的，还是在其他什么地方创建的，这个问题并不重要，因为反正返回的仅仅是一个引用。一旦操作完成，垃圾收集器就会自动关照数组的清除工作。而且只要我们还需要使用数组，它就会乖乖地听候调遣。

另一方面，注意当 flavorSet()随机挑选香料的时候，它需要保证以前出现过的一次随机选择不会再度出现。为达到这个目的，它使用了一个无限 while 循环，不断地作出随机选择，直到发现未在 picked 数组里出现过的一个元素为止（当然，也可以进行字串比较，检查随机选择是否在 results 数组里出现过，但字串比较的效率较低）。若成功，就添加这个元素，并中断循环（break），再查找下一个（i 值会递增）。

main()能显示出 20 套完整的香料，以便我们证实 flavorSet()每次都在用一个随机顺序选择香料。为体会这一点，最简单的办法就是将输出重导向进入一个文件，然后直接看这个文件的内容。

9.1.3 Arrays 类

在 java.util 中，大家可发现一个 Arrays（数组）类，其中容纳着一系列静态方法，可简化我们对数组的操作。总共有四个函数：equals()用于比较两个数组是否相等；fill()可将一个值填入数组；sort()可对数组排序；而 binarySearch()用于在排好序的数组中查找一个元素。所有这些方法都已为全部原始数据类型及对象重载使用。除此以外，还有一个 asList()方法，可用它获取任意数组，然后把数组转变成一个 List 容器——本章后面还会详加说明。

尽管非常有用，但 Arrays 这个类的功能还并不是特别全面。例如，我们完全没理由每次都要亲手写一个 for 循环，用它打印出一个数组里的元素。用 Arrays 类来实现该有多好！而且就象大家会看见的那样，fill()方法每次只能取得一个值，然后把它放到数组里。所以，假如你想用随机生成的数字来填写一个数组，fill()便没招儿了！

幸好，我已为大家解决了这个问题，在 Arrays 类中补充了一些非常实用的工具，它们放在 com.bruceeckel.util 这个封装内，便于大家利用。利用这些新增工具，大家可打印任意类型的数组；可利用一个名叫 generator（生成器）的对象，创建一系列值或对象，然后把它们填入数组（注意自己可对这个 generator 对象进行配置）。

由于我要为每个原始数据类型和对象都编制代码，所以必然会造成大量重复性代码[42]。例如，针对每种类型，都要有一个“generator”接口，因为 next()的返回类型在每种情况下都必须是不同的：

```
//: 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();
} ///:~

//: 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 {
```

[42] 对我的这个库来说，那些习惯使用默认参数和模板的 C++程序员会发现其中大量代码都属于“浪费”；Python 程序员则会发现整个库其实纯属“多余”。

```
  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();
} ///:~
```

在 Array2 中，包含了一系列 print()函数，针对每种类型进行了重载。可直接打印一个数组，可在打印前增加一条消息，亦可只打印数组内一定范围内的元素。print()函数的代码是很容易看懂的：

```
//: 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 printed,
// and to be filled via a user-defined
// "generator" object.
package com.bruceeckel.util;
import java.util.*;

public class Arrays2 {
  private static void
  start(int from, int to, int length) {
    if(from != 0 || to != length)
      System.out.print("["+ from +":"+ to +"] ");
    System.out.print("(");
  }
  private static void end() {
    System.out.println(")");
  }
```

```
  public static void print(Object[] a) {
    print(a, 0, a.length);
  }
  public static void
  print(String msg, Object[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(Object[] a, int from, int to){
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(boolean[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, boolean[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(boolean[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(byte[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, byte[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
```

```
  public static void
  print(byte[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(char[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, char[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(char[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(short[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, short[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(short[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
```

```
    end();
  }
  public static void print(int[] a) {
     print(a, 0, a.length);
  }
  public static void
  print(String msg, int[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(int[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(long[] a) {
    print(a, 0, a.length);
  }
  public static void
  print(String msg, long[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(long[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(float[] a) {
     print(a, 0, a.length);
  }
  public static void
  print(String msg, float[] a) {
    System.out.print(msg + " ");
```

```
    print(a, 0, a.length);
  }
  public static void
  print(float[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(double[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, double[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(double[] a, int from, int to){
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  // 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();
```

```
  }
}
static String ssource =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
  "abcdefghijklmnopqrstuvwxyz";
static char[] src = ssource.toCharArray();
public static class RandCharGenerator
implements CharGenerator {
  public char next() {
    int pos = Math.abs(r.nextInt());
    return src[pos % 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();}
  }
} ///:~
```

为了用一个“generator”填写数组，fill()方法必须先取得指向一个适当的“生成器接口”的引用。该接口有一个 next()方法，可生成正确类型的对象（具体类型由接口的实现方式决定）。fill()只需一直调用 next()，直到指定的范围被完全填写完毕。接下来，就可通过实现适当的接口，来创建任何生成器，然后在生成器中使用 fill()。

随机数据生成器是我们测试的重点，所以这里创建了一系列内部类，以便实现所有原始数据类型生成器接口；另外，用一个 String（字串）生成器来代表“对象”。大家可以看到，RandStringGenerator 用 RandCharGenerator 来填写一个字符数组，后者再转换成一个字串。数组的大小由提供给构造函数的参数（参数）决定。

生成的数字并不大。RandIntGenerator 默认为 10000 的模数，但重载的构造函数允许我们选择一个较小的值。

下面是一个对这个库进行测试的程序，同时也演示了它具体如何使用：

```
//: c09: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]);
    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];
    Arrays2.fill(a1,
      new Arrays2.RandBooleanGenerator());
    Arrays2.print(a1);
    Arrays2.print("a1 = ", a1);
    Arrays2.print(a1, size/3, size/3 + size/3);
    Arrays2.fill(a2,
      new Arrays2.RandByteGenerator());
    Arrays2.print(a2);
    Arrays2.print("a2 = ", a2);
    Arrays2.print(a2, size/3, size/3 + size/3);
    Arrays2.fill(a3,
      new Arrays2.RandCharGenerator());
    Arrays2.print(a3);
    Arrays2.print("a3 = ", a3);
    Arrays2.print(a3, size/3, size/3 + size/3);
    Arrays2.fill(a4,
      new Arrays2.RandShortGenerator());
    Arrays2.print(a4);
    Arrays2.print("a4 = ", a4);
    Arrays2.print(a4, size/3, size/3 + size/3);
    Arrays2.fill(a5,
      new Arrays2.RandIntGenerator());
    Arrays2.print(a5);
    Arrays2.print("a5 = ", a5);
    Arrays2.print(a5, size/3, size/3 + size/3);
    Arrays2.fill(a6,
      new Arrays2.RandLongGenerator());
    Arrays2.print(a6);
    Arrays2.print("a6 = ", a6);
    Arrays2.print(a6, size/3, size/3 + size/3);
    Arrays2.fill(a7,
      new Arrays2.RandFloatGenerator());
    Arrays2.print(a7);
    Arrays2.print("a7 = ", a7);
    Arrays2.print(a7, size/3, size/3 + size/3);
    Arrays2.fill(a8,
      new Arrays2.RandDoubleGenerator());
    Arrays2.print(a8);
    Arrays2.print("a8 = ", a8);
    Arrays2.print(a8, size/3, size/3 + size/3);
    Arrays2.fill(a9,
      new Arrays2.RandStringGenerator(7));
```

```
    Arrays2.print(a9);
    Arrays2.print("a9 = ", a9);
    Arrays2.print(a9, size/3, size/3 + size/3);
  }
} ///:~
```

注意，size 参数有一个默认值，但你也可在命令行中自行指定。

9.1.4 数组的填充

Arrays 这个 Java 标准库也提供了一个 fill()方法，但它的功能很弱——每次只能将一个值复制到一个位置；如果是对象数组，则将引用复制到每一个位置。利用 Arrays2.print()，Arrays.fill()方法的功用可以轻易地演示出来：

```
//: c09:FillingArrays.java
// Using Arrays.fill()
import com.bruceeckel.util.*;
import java.util.*;

public class FillingArrays {
  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);
    Arrays2.print("a1 = ", a1);
    Arrays.fill(a2, (byte)11);
    Arrays2.print("a2 = ", a2);
    Arrays.fill(a3, 'x');
    Arrays2.print("a3 = ", a3);
    Arrays.fill(a4, (short)17);
    Arrays2.print("a4 = ", a4);
    Arrays.fill(a5, 19);
    Arrays2.print("a5 = ", a5);
    Arrays.fill(a6, 23);
```

```
    Arrays2.print("a6 = ", a6);
    Arrays.fill(a7, 29);
    Arrays2.print("a7 = ", a7);
    Arrays.fill(a8, 47);
    Arrays2.print("a8 = ", a8);
    Arrays.fill(a9, "Hello");
    Arrays2.print("a9 = ", a9);
    // Manipulating ranges:
    Arrays.fill(a9, 3, 5, "World");
    Arrays2.print("a9 = ", a9);
  }
} ///:~
```

既可填充整个数组，亦可（见最后两条语句）填充一定范围内的元素。但由于 Arrays.fill() 同时只能提供一个值，所以 Arrays2.fill()产生的结果要有趣得多。

9.1.5 数组的复制

Java 标准库提供了一个静态方法，名为 System.arraycopy()，专门用于数组的复制。它复制数组的速度比自己亲自动手写一个 for 循环来复制快得多。System.arraycopy()已进行了重载，可对所有类型进行控制。下面是它对 int 数组进行操纵时的情况：

```
//: c09:CopyingArrays.java
// Using System.arraycopy()
import com.bruceeckel.util.*;
import java.util.*;

public class CopyingArrays {
  public static void main(String[] args) {
    int[] i = new int[25];
    int[] j = new int[25];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    Arrays2.print("i = ", i);
    Arrays2.print("j = ", j);
    System.arraycopy(i, 0, j, 0, i.length);
    Arrays2.print("j = ", j);
    int[] k = new int[10];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    Arrays2.print("k = ", k);
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    Arrays2.print("i = ", i);
    // Objects:
```

```
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    Arrays2.print("u = ", u);
    Arrays2.print("v = ", v);
    System.arraycopy(v, 0,
      u, u.length/2, v.length);
    Arrays2.print("u = ", u);
  }
} ///:~
```

要为 arraycopy()指定的参数包括：源数组、复制起点在源数组内的偏移量、目的数组、保存起点在目的数组内的偏移量以及打算复制的元素数量。很自然地，任何超出数组边界的行为都会造成一个“违例”。

从这个例子可以看出，无论原始数据类型数组，还是对象数组，我们都可对它们进行复制。但是，假如复制的对象数组，那么真正复制的只是引用——对象本身可不会复制！我们把这种情况叫作“浅复制”（参见附录 A）。

9.1.6　数组的比较

Arrays 提供了一个重载方法 equals()，可让我们对比整个数组是否相等。同样地，它们为所有原始数据类型和对象都进行了重载。要想得出“相等”的结果，两个数组必须含有相同数量的元素，而且每个元素都必须和另一个数组里的对应元素相等——这时便要用 equals()来测试每个元素的相等性（对原始数据类型来说，使用的是原始数据类型的封装器类 equals()；例如，为 int 值使用的是 Integer.equals()）。如下例所示：

```
//: c09:ComparingArrays.java
// Using Arrays.equals()
import java.util.*;

public class ComparingArrays {
  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));
  }
```

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

最开始，a1 和 a2 是完全一致的，所以输出是“true”（真）。但紧接着，一个元素发生了改变，所以第二行输出变成“false”（假）。在最后一种情况中，s1 的所有元素都指向同一个对象，而 s2 包含了五个各自独立的对象，那么两个数组相等吗？此时，由于测试数组是否“相等”根据的是实际内容（用 Object.equals()进行判断），所以结果还是“true”（真）。

9.1.7 数组元素的比较

Java 1.0 和 1.1 的函数库均欠缺的一项特性便是“算法”——就连最简单的排序算法都没有！由于 Java 是如此重要的一种语言，所以对一个原本希望会用到一个功能全面的标准库的人来说，这多少会使自己有点儿“吃惊”。幸好，这种情况已经一去不复返了！Java 2 多少改正了这个问题——至少在排序问题上。

编写通用的排序代码时，要解决的一个问题是排序必须依据对象的实际类型来进行比较。当然，有一个办法是为每种类型都写一个不同的排序方法。不过，以后一旦有新类型出现，原来的代码便没法用了——使你的代码不易“复用”。

正如我们一贯坚持的那样，一项基本的编程宗旨是：“把那些要改变的东西同不变的东西分开”！在这里，保持不变的代码便是通用的排序“算法”，而那些要发生改变的就是每一次对象的“比较方式”。所以，千万不要把用于比较的代码“硬性”编码到大量各不相同的排序例程里，而应使用“回调”（Callback）这一技术。使用“回调”，要经常变化的那部分代码便封装到它自己的类内，而一直保持不变的代码则“回调”发生变化的代码。这样一来，一方面不同的对象可表达出不同的比较方式，另一方面只需向它们传递相同的排序代码。

在 Java 2 中，有两个办法可提供比较功能。第一个办法是用“自然比较方法”，这是通过实现 java.lang.Comparable 接口来实现的。这其实是一个非常简单的接口，其中只有一个方法，名为 compareTo()。该方法要取得另一个对象作为自己的参数，如果参数（对象）小于当前对象，返回一个负数；如果相等，返回 0；如果参数大于当前对象，则返回一个正数。

下面是一个实现了 Comparable 的类，它演示了如何利用由 Java 标准库提供的方法 Arrays.sort()，来进行数组元素的比较：

```
//: c09: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();
  private static int randInt() {
    return Math.abs(r.nextInt()) % 100;
  }
  public static Generator generator() {
    return new Generator() {
      public Object next() {
        return new CompType(randInt(),randInt());
      }
    };
  }
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, generator());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a);
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

定义比较函数时，你需要决定当一个对象同一个比较时，它的具体含义。在这里，比较时只使用了 i 值，j 值就被忽略了。

randInt( )这个静态方法用于生成 0 到 100 之间的一个正数；而 **generator( )**方法可生成一个对象，负责在其中实现 Generator 接口——这是通过创建一个匿名内部类（参见第 8 章）来实现的。由此，我们可构建出 CompType 对象，同时用随机数对它们进行初始化。在 main() 中，用生成器来填写一个由 CompType 对象构成的数组，然后对其进行排序。假如 Comparable 尚未实现，就会在试图调用 sort()的时候，获得一条编译时间出错提示。

接下来，假定有人拿给你一个类，它并未实现 Comparable；或者尽管实现了 Comparable，但你却不喜欢它的工作方式，而是想为类型采用一个不同的比较函数。在这个时候，必须用 Java 2 提供的第二个办法来进行对象的比较——单独创建一个类，实现名为 Comparator 的一个接口。接口提供了两个方法，分别是 compare()和 equals()。不过，除非考虑到一些特殊的性能方面的因素，否则我们用不着实现 equals()，因为每次创建一个类的时候，它都会默认从 Object 继承，而 Object 已经有了一个 equals()。为此，我们只需使用默认的 Object equals()，同时注意遵守由接口制订的规则就可以了。

Collections（集合）类（后文还会详述）包括了一个 Comparator，可将自然的排列顺序反转过来。它可轻松应用于 CompType：

```
//: c09:Reverse.java
// The Collecions.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());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a, Collections.reverseOrder());
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

调用 Collections.reverseOrder()，会生成指向 Comparator 的引用。

再来看看另一个例子，下面这个 Comparator 可对 CompType 对象进行比较，但依据的是 j 值，而非 i 值：

```
//: c09: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());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a, new CompTypeComparator());
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

compare()方法只能返回一个负整数、零或者一个正整数，分别对应于第一个参数小于、等于或者大于第二个。

9.1.8 数组的排序

利用内置的排序方法，不管一个数组是原始数据类型的数组，还是一个对象数组，也不管该数组是自己实现了 Comparable，还是有一个关联的 Comparator，都可对它进行排序。

这其实是填补了 Java 库存在的一处巨大空白——不管你相不相信，在 Java 1.0 和 1.1 中，居然没提供对字串排序的支持！下面这个例子可生成随机字串对象，并对它们进行排序：

```
//: c09: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));
    Arrays2.print("Before sorting: ", sa);
    Arrays.sort(sa);
    Arrays2.print("After sorting: ", sa);
  }
} ///:~
```

在字串排序的输出中，有一件事情大家应该注意到：它采用的是“字典顺序”。也就是说，以大写字母开头的单词会全部排在前面，而小写字母开头的单词全部排在后面（美国的电话簿也通常采用这种方式）。有时，我们希望能将所有单词都组合到一起，而无论它们的大小写形式。此时，可通过定义一个 Comparator 类来做到这一点，藉此覆盖默认的 String Comparable 行为。考虑到代码以后要复用的问题，所以把它添加到 util 封装内：

```
//: 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());
  }
} ///:~
```

比较之前，每个字串都会转换成小写形式。字串内置的 compareTo()方法为我们提供了相应的支持。

下面是用 AlphabeticComparator（字典顺序比较器）进行的一次测试：

```
//: c09: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));
    Arrays2.print("Before sorting: ", sa);
    Arrays.sort(sa, new AlphabeticComparator());
    Arrays2.print("After sorting: ", sa);
  }
} ///:~
```

Java 标准库采用的排序算法可针对你实际排序的类型加以优化——“快速排序”用于原始数据类型；稳定的合并式排序则用于对象。所以，除非你的分析工具指出排序部分已成为整个程序的“瓶颈”，否则根本不必多花时间去操心排序的性能问题。

9.1.9 在排好序的数组中搜索

一个数组排好序后，可用 Arrays.binarySearch()在其中快速查找一个特定的项目。但是，千万不要对一个尚未排好序的数组使用 binarySearch()——那样会造成无法预测的后果。下面这个例子用 RandIntGenerator 来填充一个数组，然后生成最终想要查找的值：

```
//: c09: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);
    Arrays2.print("Sorted array: ", 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，我们生成随机数，当作搜索项使用，直到找到它们中的一个为止。

如果发现搜索项，Arrays.binarySearch()会生成一个大于或等于 0 的值。否则的话，它会生成一个负数，代表元素应该插入的位置（如果你亲手对排好序的数组进行维护的话）。生成的值是：

```
-(insertion point) - 1
```

插入点是大于关键值的第一个元素的索引位置；而假如数组内所有元素都小于指定的关键值，那么插入点就是 a.size()。

假如数组内包含了重复的元素，便无法保证最后搜索到的是哪一个。这种算法并不是真正为支持重复元素而设计的。但是，假如你需要由一个非重复元素构成的、已经排好序的列表，可以使用 TreeSet——本章后面会对它详细介绍。TreeSet 可为我们自动照顾到所有细节。其实除非出现性能瓶颈，否则就不要用人工维护的数组，而是坚持用 TreeSet。

如果用一个 Comparator 完成了对一个对象数组的排序（原始数据类型数组不允许用 Comparator 进行排序），那么在执行 binarySearch()搜索的时候，必须同时包括那个 Comparator——使用函数提供的重载版本。举个例子来说，我们可以修改一下 AlphabeticSorting.java 程序，让它执行一次搜索：

```
//: c09:AlphabeticSearch.java
// Searching with a Comparator.
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSearch {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa,
      new Arrays2.RandStringGenerator(5));
    AlphabeticComparator comp =
      new AlphabeticComparator();
    Arrays.sort(sa, comp);
    int index =
      Arrays.binarySearch(sa, sa[10], comp);
    System.out.println("Index = " + index);
  }
} ///:~
```

其中，Comparator 必须作为第三个参数，传递给重载的 binarySearch()。在上述例子中，由于要搜索的项已经从数组内取出来了，所以保证能够成功。

9.1.10 数组总结

总之，在你想容纳一组对象的时候，第一个、也是最有效的一个选择便是数组。而且假如你想容纳的是一系列原始数据类型，那么数组也是你唯一的选择！在本章剩余的部分，大家会接触到一种更常见的应用场合：写程序时，并不知道实际需要多少个对象，或者希望用一种更高级的方式来保存自己的对象！

Java 提供了“容器类”库，来解决这方面的问题，它们的基类型便是 List、Set 和 Map。利用这些工具，许多令人头疼的问题可以马上迎刃而解！

例如，Set 只能容纳每个值的一个对象；而 Map 本质上是一种“关联数组”，可将任何一个对象同其他任何对象关联到一起——Java 容器类能自动改变它们的大小。因此，和数组不同，我们可在其中放置任意数量的对象；而且在写程序的时候，用不着关心一个容器到底需要多大。

9.2 容 器 入 门

对我来说，“容器”（Container）是原始开发阶段最强劲的工具之一，因为它可显著加强编程力量。Java 1.0 和 1.1 的容器只能说是一块“鸡肋”，令人非常失望。不过，Java 2 在这方面已有了长足进步，它对“容器”进行了通盘的重新设计[43]，把它变成了一个真正强劲的工具！重新设计后的版本显得更紧凑，也更容易理解。它也充实了容器库的功能，提供了象链接列表、队列以及双头队列（即"Double-Ended Queues"，英文缩写为"Deques"，发音作"decks"）。

容器库的设计是相当困难的（其实大多数库的设计都很麻烦）。在 C++中，容器类利用数量众多、但很难学习和掌握的类，来克服这个问题。当然，这一点比 C++之前的任何一种语言都要做得好（以前根本没想到过容器库的问题）。但是，Java 刚问世的时候，这一点并未良好地借鉴过来。在极端的情况下，我甚至看到过一个容器库，它只由一个类构成，这个类就是“container”（容器）——它既是一个线性序列，也是一个关联数组，居然同时扮演两个角色！不过，Java 2 的容器库对此进行了很好的平衡：一方面，它提供了一个成熟的容器库应当具有的全部功能；而另一方面，和 C++容器类和其他类似的容器库相比，对它的学习和运用要容易得多。当然，也得为这种平衡付出一些代价。代价就是：在某些场合下，它显得有点儿“别扭”。不过，和早期在 Java 库中作出的某些“古怪”决定不同，这些“别扭”的地方并不是由于设计者不谨慎造成的。相反，它是设计者综合各方面的因素，均衡复杂性和易用性这两个方面的因素，有意作出的决定。学习 Java 2 容器库的使用时，刚开始也许要有一个适应阶段。但我可以保证，只要你够用心，那么用不了多久的时间，就能熟练掌握并运用这些新工具了！

Java 2 容器库解决的是“如何容纳对象”的问题，并为此设计了两种手段：

(1) 集合（Collection）：一组单独的元素，通常为它们应用了某种规则。在这里，一个 List（列表）必须按特定的顺序容纳元素，而一个 Set（集）不可包含任何重复的元素。注意，尽管“包”（Bag）没有这些限制，但它未在 Java 容器库中实现，因为 List 具有和它差

[43] 此项目的负责人是 Sun 公司的 Joshua Bloch。

不多的功能。

(2) 映射（Map）：一系列“键－值”对。从表面看，它似乎就是一种由键值对构成的“集合”。但假如你真的那样实现，整个设计就会显得很“笨”。所以，我们有必要把它分离出来，当作一个单独的概念看待。不过另一方面，假如将 Map 的某一部分看作“集合”，有时候也还是显得非常方便的。换言之，你可以创建一个“集合”，用它来表达 Map 的那一部分。综上所述，一个 Map 可以返回的东西包括：它的键值构成的一个 Set、由它的值构成的一个集合或者由它的“键－值”对构成的一个 Set。Map 类似于数组，也能方便地扩展成“多维”，用不着再牵涉到其他概念——只需将一个 Map 的值设为 Map 就可以了（后者又可以包含更多的 Map，以此类推）！

首先，让我们来归纳一下容器的常规特性，再探讨细节，最后研究一下为什么有些容器会有这么多不同的版本，以及实际应该如何挑选。

9.2.1　容器的打印

和数组不同，容器不需凭借任何外来力量，便能进行良好的打印。这里有一个例子，它同时也展示了基本的容器类型：

```
//: c09:PrintingContainers.java
// Containers print themselves automatically.
import java.util.*;

public class PrintingContainers {
  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()));
  }
} ///:~
```

就象前面说的那样，Java 容器库共有两个基本类别——集合和映射。它们的区别在于容器内每个位置所容纳的项目数量。其中，“集合”类每个位置只能容纳一个（其实，“集合”这个名字有一定的误导之嫌，因为整个容器库通常都叫作“集合”）。在集合中，包括了 List，用于按指定顺序容纳一组项目；以及 Set，每种类型只允许添加一个项目。ArrayList（数组

列表）是 List 的一种类型，而 HashSet（散列集）又是 Set 的一种类型。要想将项目添加到任何一个集合里，必须采用 add()方法。

“映射”（Map）容纳的只是一系列“键－值”对，就象一个小型数据库。上面的程序采用的是 Map 的一个分枝，名为 HashMap（散列映射）。假定我们现在有一个 Map，它可将美国州名同它们的首府联系（映射）起来，那么为了知道俄亥俄（Ohio）州的首府，只需在其中搜索就可以了——就象我们对一个数组进行索引那样（注意，“映射”也被称作“关联数组”）。

要想在一个 Map 里增添新元素，必须用一个 put()方法。该方法需要取得一个“键”（Key）和一个“值”（Value），作为自己的参数（参数）使用。上面讲的例子只演示了如何添加元素，而没有在加入后实现对它们的搜索。不过，我们不久就要增添这一功能。

重载的 fill()方法可对集合和映射进行填充。仅需看一看它的输出，就会知道默认的打印行为（通过容器的各种 toString()方法提供）已经产生了令人无可挑剔的结果，所以和数组不一样，不必再为它添加额外的打印支持：

```
[dog, dog, cat]
[cat, dog]
{cat=Rags, dog=Spot}
```

一个“集合”打印出来之后，它的两头会被加上方括号，其中每个元素都用一个逗号分隔。不过，“映射”是用花括号封闭的，而且每个“键－值”对都用一个等号连接（键在左，值在右）。

大家同时还能马上看出不同容器的基本行为。其中，List 容纳的对象和它们输入时一模一样，不会进行重新排序或编辑处理；但 Set 每个对象只接收一个，并采用它自己的内部排序方法（通常，我们只需关心一样东西是不是一个 Set 成员，而不应关心它的顺序——否则用 List 好了）。同时 Map 每种类型的项目只接收一个（由键来决定），而且也有它自己的内部排序机制，而且不关心我们输入项目时的顺序。

9.2.2 容器的填充

尽管容器的打印问题已得到妥善解决，但在进行填充时，却遇到了和 java.util.Arrays 相同的问题。

不过，和 Arrays 类似，我们也有一个名为 Collections 的辅助类，它提供了一系列静态工具方法，其中就有一个 fill()。fill()的作用很简单，就是在容器里复制一个对象引用，但它只能用于 List，不能用于 Set 或 Map：

```
//: c09:FillingLists.java
// The Collections.fill() method.
import java.util.*;

public class FillingLists {
  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);
  }
} ///:~
```

但令人遗憾的是，这个方法对我们来说甚至更没用！因为它只能替代那些已经在 List 内的元素，不能新增。

为了能创建出更实用的例子，我在这儿也写了一个补充性的 Collections2 库（同属 com.bruceeckel.util 的一部分，便于大家使用）。其中，新的 fill()方法可利用一个“生成器”来增添元素，并允许用户指定想要添加的元素数量（用 add()进行）。前面定义的 Generator 接口这里同样适用于集合，只不过 Map 要求设计它自己的生成器接口，因为每一次对 next()调用时，都得同时生成“一对”对象才行（一个键，和一个值）。下面是我们的 Pair（键值对）类：

```
//: com:bruceeckel:util:Pair.java
package com.bruceeckel.util;
public class Pair {
  public Object key, value;
  Pair(Object k, Object v) {
    key = k;
    value = v;
  }
} ///:~
```

接下来，是用于生成 Pair 的生成器接口：

```
//: com:bruceeckel:util:MapGenerator.java
package com.bruceeckel.util;
public interface MapGenerator {
  Pair next();
} ///:~
```

有了上面这些东西，就可为容器类开发出一系列真正“实用”的工具：

```
//: 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( )**均采用了一个参数，用于决定要在容器中添加的项目数量。除此以外，Map 还用到了两个生成器，其中包括 RandStringPairGenerator，用于创建数量任意的一系列随机字串对，每个字串的长度由构造函数的参数决定；还包括 StringPairGenerator，它的作用也是产生字串对，但这些字串由一个二维的字串数组指定。StringGenerator 也采用了一个二维字串数组，只不过生成的是一系列单独的字串，而不是字串“对”。静态的 rsp、geohraphy、countries 和 capitals 对象提供了预先构建好的生成器，最后三个使用了世界上的所有国家以及它们的首都。要注意的是，如果想创建的“对”数多于可用的，生成器就会循环回开头；而且如果将这些“对”放到一个 Map 里，重复的“对”就会被忽略。

下面是一个预先定义好的数据集，其中包括了国家名及其首都（之所以要用小字体印刷，是为了节省一定的空间）：

```
//: 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","Dijibouti"},
    {"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"}, {"GUINEA","Conakry"},
    {"GUINEA","-"}, {"BISSAU","Bissau"},
    {"CETE 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","Jerusalem"}, {"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","Ljujiana"}, {"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"},
  };
} ///:~
```

这实际是一个二维的字串数据数组[44]。下面则是一个简单的例子，它对 fill()方法和生成器进行了测试：

```
//: c09:FillTest.java
import com.bruceeckel.util.*;
import java.util.*;

public class FillTest {
  static Generator sg =
    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();
```

[44] 原始数据摘抄于 Internet，然后用一个 Python 程序来处理的（参见 www.Python.org）。

```
    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);
  }
} ///:~
```

利用这些工具，可对各种容器进行方便的测试，只需把数据填到容器里即可。

9.3 容器的缺点：类型未知

使用 Java 容器的一个“缺点”是，一个对象在置入一个容器后，它的类型信息也同时丢掉了。之所以会发生这种情况，是由于当初编写容器时，那个容器的程序员根本不知道用户到底想把什么类型置入容器。假若只让这个容器支持特定的类型，那么会妨碍它成为一个“常规用途”的工具，反而会为更多的用户带来麻烦。所以，为了解决这个问题，容器实际容纳的是对 Object 的引用；而由于 Object 是所有类的根，所以它可支持任意类型！（当然，原始数据类型除外，因为它们根本不是“继承”来的）。这是一个很好的方案，只是不适用下述场合：

(1) 将一个对象引用置入容器时，由于类型信息会被抛弃，所以任何类型的对象都能进入我们的容器——即便特别指示它只能容纳特定类型的对象。举个例子来说，虽然指示它只能容纳猫，但事实上任何人都可以把一条狗抛进来。

(2) 由于类型信息已不复存在，所以容器唯一能肯定的事情就是自己容纳的是指向一个对象的引用。正式使用它之前，必须对其进行强制转型，使其具有正确的类型。

不过令人欣慰的是，Java 并不允许人们滥用置入容器的对象。假如将一条狗扔进一个装猫的容器（试图将容器内的所有东西都当猫看待），那么一旦从猫容器里取出一个狗引用，并试图将其强制转型成猫，就会得到一个运行时间的“违例”。

下面是一个例子，它采用了最基本的容器：ArrayList。对新手来说，可将 ArrayList（数组列表）想象成“能自动扩展的一个数组”。ArrayList 的用法非常简单：创建一个，用 add()放入对象，以后则用 get()取得那些对象（根据索引）——一切都和操作真正的“数组”一样，只是没用到方括号[45]。

ArrayList 另外还有一个名为 size()的方法，用它可以知道已经添加了多少个元素，从而

[45] 在这个地方，运算符重载真正体现出了它的“好处”。

避免不经意地“撑爆”，造成“违例”。

首先，我们要创建 Cat（猫）和 Dog（狗）这两个类：

```
//: c09:Cat.java
public class Cat {
  private int catNumber;
  Cat(int i) { catNumber = i; }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
} ///:~

//: c09:Dog.java
public class Dog {
  private int dogNumber;
  Dog(int i) { dogNumber = i; }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
} ///:~
```

将所有猫、狗都放到容器里，再从中取出来：

```
//: c09:CatsAndDogs.java
// Simple container example.
import java.util.*;

public class CatsAndDogs {
  public static void main(String[] args) {
    ArrayList cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    // Not a problem to add a dog to cats:
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.get(i)).print();
    // Dog is detected only at run-time
  }
} ///:~
```

Cat 和 Dog 这两个类的区别是明显的——除了都是“Object”（对象）之外，两者毫无共通之处（如果不明确指出从什么类继承，就默认为从 Object 继承）。由于 ArrayList 里容纳的都是对象，所以不仅能用 ArrayList 的 add()方法将 Cat 对象置入这个容器，也能将 Dog 对象放到其中——编译和运行时间不会报告任何错误。用 ArrayList 的 get()方法取出自己认为

是 Cat 的对象时，实际得到的是一个对象引用，必须把它强制转型成一个 Cat。因此，整个表达式都必须封到一对括号内，以便在为 Cat 正式调用 print()方法之前，强制完成强制转型。否则的话，一旦到了运行时间，只要你试图将 Dog 对象强制转型成 Cat，就会产生“违例”。

非常麻烦，是吧？但事情还没完呢！除了操作麻烦之外，这种方式还有可能为你的程序带来潜在的 Bug，而且以后很难发现！试想，假如程序的一个部分（或几个部分）将对象插入一个容器，但通过一次“违例”事件，我们只在程序某个单独的部分里，发现肯定有一个错误的对象插入了容器，那么就必须亲自动手，找出到底是在哪里进行了错误的插入。不过，从好的一方面讲，编程时如果从一些标准化的容器类开始，那么仍会为自己带来较大的便利——尽管它有点儿麻烦，也有点儿不足。

9.3.1　错误被悄悄地解决

有些时候，即便不强制转型回原来的类型，程序似乎也能正常工作。在这些情况下，我们说：“错误被悄悄地解决了。”其中，最特别的一种情况是：字串（String）类由于能从编译器那里得到一些额外的帮助，所以我们并不一定非要强制转型不可。

假如编译器原本期望的是字串对象，但得到的又不是，它就会自动调用 toString()方法。该方法定义于 Object 中，并可由任何 Java 类覆盖。这样一来，便能满足编译器的“愿望”，还它一个正确的字串。

因此，碰到这种情况，为了让自己的类打印出来，我们要做的唯一事情就是覆盖 toString()方法，就象下面这个例子展示的那样：

```
//: c09:Mouse.java
// Overriding toString().
public class Mouse {
  private int mouseNumber;
  Mouse(int i) { mouseNumber = i; }
  // Override Object.toString():
  public String toString() {
    return "This is Mouse #" + mouseNumber;
  }
  public int getNumber() {
    return mouseNumber;
  }
} ///:~

//: c09:WorksAnyway.java
// In special cases, things just
// seem to work correctly.
import java.util.*;

class MouseTrap {
  static void caughtYa(Object m) {
    Mouse mouse = (Mouse)m; // Cast from Object
    System.out.println("Mouse: " +
      mouse.getNumber());
```

```
  }
}

public class WorksAnyway {
  public static void main(String[] args) {
    ArrayList 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));
    }
  }
} ///:~
```

从中看出，我们在 Mouse 中对 toString()进行了覆盖。而在 main()的第二个 for 循环中，大家可见到下面这一条语句：

```
System.out.println("Free mouse: " + mice.get(i));
```

在“+”之后，编译器本来预计看到的是一个字串对象。但由于 get()产生的是一个 Object（对象），所以为了满足编译器的“愿望”，编译器会自动调用 toString()。不过不幸的是，只有字串才具有这种“魔术”般的效果，其他类型均不能如法炮制。

为了隐藏强制转型，我们在 MouseTrap 中采用了另一个办法。在其中，caughtYa()方法接收的并不是一个 Mouse，而是一个 Object，然后再把它强制转型为 Mouse。当然，这样做是非常冒失的，因为假如接收的是一个 Object，那么任何东西都可以传递给方法。不过也没关系，如果强制转型不正确——我们传递了错误的类型——那么在运行时间也会得到一个违例。当然，这样做比不上在编译时间便完成检查，但它仍然是一个可行的办法。注意在使用下面这个方法时：

```
MouseTrap.caughtYa(mice.get(i));
```

不必进行强制转型！

9.3.2 让 ArrayList 自动判断类型

接着讨论上面的问题，我们可不想半途而废！那么，到底有没有一个更“健壮”的方案呢？答案是肯定的，那便是用 ArrayList 创建一个新类，使其只接收我们指定的类型，也只生成我们希望的类型。如下所示：

```
//: c09:MouseList.java
// A type-conscious ArrayList.
import java.util.*;
```

```
public class MouseList {
  private ArrayList 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(); }
} ///:~
```

下面是对新容器的一个测试：

```
//: c09:MouseListTest.java
public class MouseListTest {
  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));
  }
} ///:~
```

与前一个例子类似，只是新的 MouseList 类现在拥有了类型为 ArrayList 的一名私有成员，它的方法和 ArrayList 的一样。只不过，它现在并不接收和输出普通的 Object，而是只能针对 Mouse 对象行事！

注意，假如我们换成让 MouseList 从 ArrayList 那里“继承”，那么 add(Mouse)方法就会重载原有的 add(Object)。这样一来，对能够添加的对象类型就依然没什么限制。换言之，MouseList 会变成 ArrayList 的一名“代表”——在移交自己的职权之前，采取一些行动（详见《Thinking in Patterns with Java》电子书，可从 www.BruceEckel.com 下载）。

由于 MouseList 只能接收 Mouse，所以假如利用下述语句：

```
mice.add(new Pigeon());
```

试图向它传递一个“鸽子”（Pigeon），就会在编译时间得到一条出错消息（注意是在“编译时间”！）因此，尽管编程时显得有些麻烦，但它的好处是：只要类型不正确，就会立即得到提示，不必等到“运行时间”！

参数化类型

这类问题可不是孤立存在的——我们许多时候都要在其他类型的基础上创建新类型。因此，在编译时间拥有特定的类型信息是非常有帮助的。这便是“参数化类型”的来由。在 C++中，它由语言通过“模板”提供了直接支持。对 Java 来说，它的未来某个版本或许会正式支持某种形式的“参数化类型”；不过就目前来说，要想达到同样的效果，你只有使用一些由其他人制作的工具，用它们自动创建类似于 MouseList 的类。

9.4 迭 代 器

在任何容器类中，必须通过某种方法在其中置入对象，再用另一种方法取出对象。毕竟，容纳各式各样的对象正是“容器”的首务。在 ArrayList 中，add()便是我们插入对象的方法，而 get()是从中取出对象的一种方法。ArrayList 显得非常灵活——可在任何时候选择任何东西，也能利用不同的索引同时选择多个元素。

不过假如能够站高一点儿，再来思考这个问题，就会发现它的一个缺点：要想真正使用一个容器，必须提前知道它的准确类型！从表面看，这似乎并无什么不妥，但假如你刚开始用的是一个 ArrayList，但随着程序设计的深入，考虑到自己使用容器的特殊方式，又发现用 LinkedList 似乎效率还要高一些，那么该如何是好呢？或者，假定你想写一些常规性代码，它不知道或不关心容器的准确类型是什么，那么在不改写代码的前提下，可让它同时应用于不同类型的容器吗？

在这些情况下，用一个“迭代器”（Iterator）可达到你的目的。所谓的“迭代器”实际是一个对象，它的工作就是在一系列对象中巡视（遍历），并选中序列中的每一个对象——作为客户程序员，他们不必知道、也不必关心这个序列的基础结构是什么！除此以外，迭代器也被认为是对象的一种“缩水”版本——创建它的代价相当低！不过也正是由于有这一点好处，所以你也往往能发现对它实行的一些看似“奇怪”的限制——例如，有些迭代器只允许朝一个方向移动。

Java 提供的 Iterator 正是具有这些限制的一个迭代器的例子。除了下面这些工作之外，不要指望能用它做别的事情：

(1) 用一个名为 iterator()的方法要求一个容器向你传递一个迭代器。这个迭代器会在你首次调用它的 next()方法时，返回序列中的第一个元素。

(2) 用 next()获得序列中的下一个元素。

(3) 用 hasNext()查询序列中是否还有更多的对象。

(4) 用 remove()删除由迭代器返回的最后一个元素。

如此而已！有点儿失望，不是吗？不过，尽管 Java 实现的这个“迭代器”显得有些单调，但仍然可以帮助我们解决许多问题（针对 List，还有一个更复杂、更高级的 ListIterator）。为了展示其效果，让我们来修订一下本章早些时候的 CatsAndDogs.java（猫和狗）一例。在原来的版本中，我们是用 get()方法来选中每一个元素。但在下面的修订版本中，我们改为使用一个迭代器：

```
//: c09:CatsAndDogs2.java
// Simple container with Iterator.
import java.util.*;

public class CatsAndDogs2 {
  public static void main(String[] args) {
    ArrayList 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()).print();
```

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

从中可以看到，最后几行现在用的是一个迭代器在序列中步进，而不是用一个 for 循环。利用迭代器，我们不必关心容器里元素的数量。hasNext()和 next()这两个方法可帮助我们照看到这一点。

再来看看另一个例子。下面，让我们来创建一个常规用途的打印方法：

```
//: c09:HamsterMaze.java
// Using an Iterator.
import java.util.*;

class Hamster {
  private int hamsterNumber;
  Hamster(int i) { hamsterNumber = i; }
  public String toString() {
    return "This is Hamster #" + hamsterNumber;
  }
}

class Printer {
  static void printAll(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

public class HamsterMaze {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 3; i++)
      v.add(new Hamster(i));
    Printer.printAll(v.iterator());
  }
} ///:~
```

请仔细研究一下打印方法 printAll()。注意其中没有与序列类型有关的任何信息。我们拥有的全部东西便是一个“迭代器”，通过它可实现对序列的各种操作：取得下一个对象，知道何时抵达末尾，等等。换言之，我们现在能使用一个对象容器，然后通过它对其中的每个对象执行操作——这可是一个颇有价值的编程技巧。在本书许多地方，都会按这一方法行事！

这个例子甚至具有更大的“通用”性，因为它默认使用了 Object.toString()方法。println()方法已为所有原始数据类型及对象进行了重载；不管在哪种情况下，都会通过调用恰当的 toString()方法，自动供应字串。

另外——尽管没必要——我们还可以更加明确地使用一个强制转型，它具有与调用 toString()等同的效果：

```
System.out.println((String)e.next());
```

但是，我们想做的事情通常并不仅仅是调用 Object 的各种方法，所以会再度面临“类型－强制转型”的问题。对于自己感兴趣的某种特定类型的序列，必须假定自己已获得了一个“迭代器”，再将结果对象强制转型成那种类型（如果有错，运行时便会产生违例）。

无意递归

由于和其他所有类都一样，Java 标准容器也是从 Object 继承来的，所以它们自动包含了一个 toString()方法。这个方法已被覆盖，所以它们能产生一个代表自身（包括它们容纳的对象）的字串。例如，在 ArrayList 中，toString()会遍历 ArrayList 的所有元素，并为每一个元素都调用 toString()。现在，假定我们想打印出一个类的地址。从表面看，似乎只需引用 this 就可以了（特别是 C++程序员，他们非常倾向于这种方式）：

```
//: c09:InfiniteRecursion.java
// Accidental recursion.
import java.util.*;

public class InfiniteRecursion {
  public String toString() {
    return " InfiniteRecursion address: "
      + this + "\n";
  }
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new InfiniteRecursion());
    System.out.println(v);
  }
} ///:~
```

假如只是单纯地创建一个 InfiniteRecursion（无限循环）对象，然后打印它，就会得到无穷无尽的一系列违例。另外，如果将 InfiniteRecursion 对象放到一个 ArrayList 里，并用同样的方式打印出那个 ArrayList，那么也会得到同样的结果。为什么呢？原来一切都是字串的“自动类型转换”在作怪。一旦你执行：

```
"InfiniteRecursion address: " + this
```

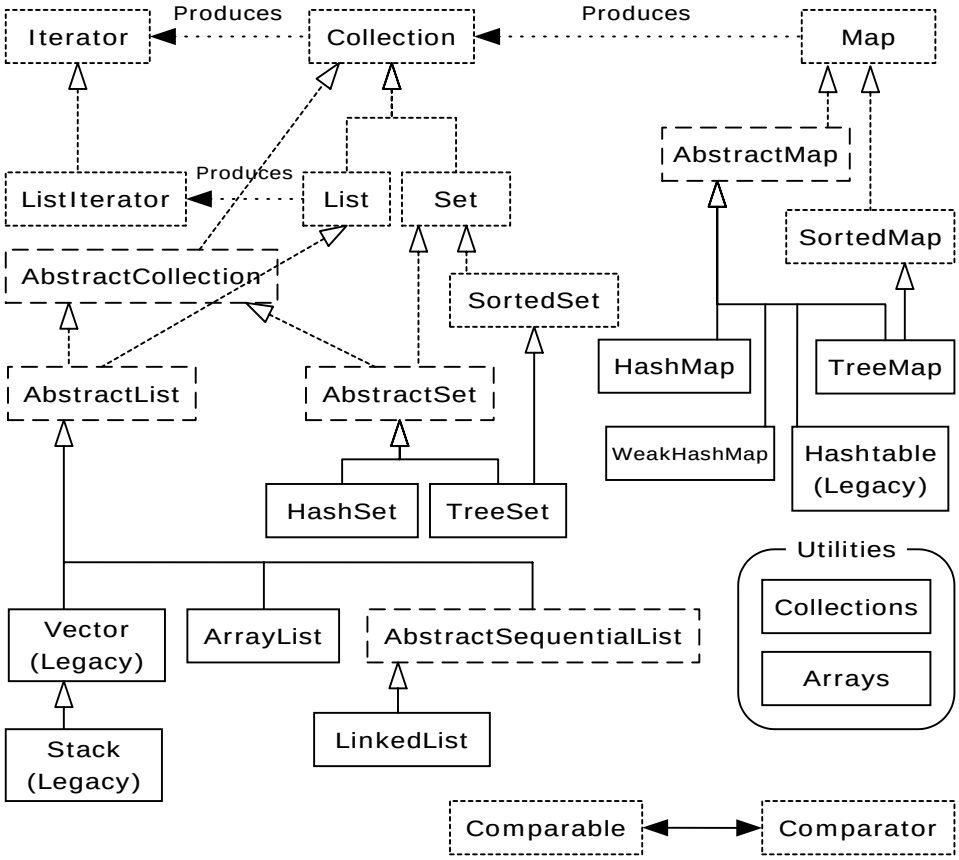
那么，编译器看到的就是一个字串，接着是一个“+”号，再接着又是非字串的东西。因此，它会试着将 this 转换成一个真正的字串。而这种转换又需要用到 toString()，由此便产生了一次递归调用；或者说，你进行了一次“无意递归”。

在这种情况下，如果你真的想打印出对象的地址，那么唯一的办法是调用 Object toString()方法，将对象明确地转换成字串！

因此，这儿不要用 this，而应该用 super.toString()——但有一个前提，那就是你需要直接从 Object 继承；或者没有一个父类覆盖了 toString()方法）。

9.5　容器的分类

“集合”（Collection）和“映射”（Map）可采用不同的方式来实现——具体由你的编程需要决定。在这里，大家有必要先看一下 Java 2 的一幅容器示意图：



刚开始，这幅图可能会让你觉得有点儿迷糊。不过假如能集中一下注意力，便会发现其中实际只包括了三个容器组件：Map、List 和 Set，而且每个组件实际只有两、三种实现方式，而且通常都只有一种特别好的方式（首选版本）。看出这一点后，整个“容器”的世界就会立即变得清晰起来！

点线方框代表“接口”，划线方框代表“抽象类”，而实线方框代表普通类（即具体类，而非抽象类）。虚线箭头指出一个特定的类实现了一个接口（在抽象类的情况下，则是“部分”实现了那个接口）。实线箭头指出一个类可生成箭头指向的那个类的对象。例如，任何集合都能产生一个迭代器，而一个 List 除了能生成一个 ListIterator（列表迭代器）外，还能生成一个普通迭代器，因为 List 正是从集合继承来的！

与对象的容纳有关的接口是集合、List、Set 以及 Map。在理想情况下，我们写的大多数代码都在从事着同这些接口的通信活动；而且只有在创建的时候，才需要指出自己要使用的准确类型是什么。因此，我们可以象下面这样创建一个 List：

```
List x = new LinkedList();
```

当然，也可以决定将 x 变成一个 LinkedList（而不是一个普通的 List），并指定和 x 有关的准确类型信息。使用“接口”真正的好处在于（也是我们用它的目的）：假如想改变自己的实施方式，那么只需在它的创建位置进行修改，就象下面这样：

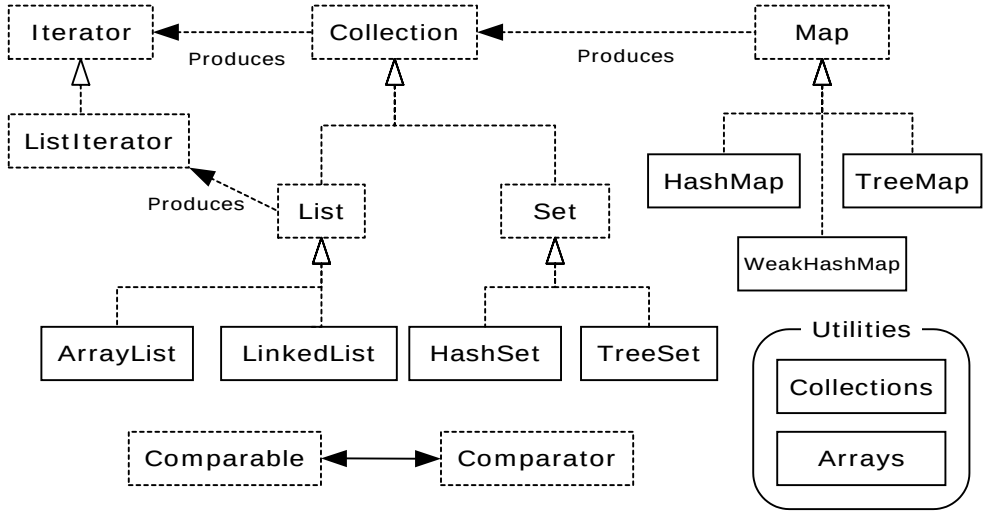
```
List x = new ArrayList();
```

其他地方的代码则用不着变动（注意用“迭代器”也能达到差不多的目的）。

在上述分类结构图中，大家可以看到许多类的名字是以“Abstract”（抽象）开头的——最开始，它们可能为你带来一定程度的迷惑。事实上，它们只不过是一些工具，“部分”实现了一个特定的接口。例如，假定你想制作自己的一个 Set，那么一般不会从 Set 接口开始，并实现它的所有方法。相反，你会从 AbstractSet 继承，然后只需再做少量工作，就可完成自己的新类。不过，在容器库中包括了丰富的功能，几乎可满足我们任何时候的需要。所以就目前来说，我们可以暂时忽略那些以“Abstract”开头的类。

因此，大家在看这幅图的时候，真正关心的只有顶部那些接口以及那些“具体类”（实线框里的）。我们通常的做法是，先生成具体类的一个对象，将其向上强制转型成对应的接口。以后，在代码中只需一直用那个接口就可以了！除此以外，如果要编的是新程序，那些“老版本才有”的元素就不必考虑了。

综上所述，那幅看似复杂无比的示意图其实完全可以简化成下面这个样子：



在新图中，只包括了接口和一些经常要用到的类，当然同时还有一些本章要重点强制的元素。

下面有一个简单的例子，用于在一个集合（这里用一个 ArrayList 表示）里填充字串对象，然后打印出集合里每个元素：

```
//: c09:SimpleCollection.java
// A simple example using Java 2 Collections.
import java.util.*;

public class SimpleCollection {
  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());
  }
} ///:~
```

main()的第一行创建了一个 ArrayList 对象，然后把它向上强制转型成一个集合。由于这个例子只用到了由“集合”提供的方法，所以从集合继承的某个类的所有对象都能正常工作，只是 ArrayList 属于集合的一种特例而已。

add()方法用于在集合里添加一个新元素。但是，根据用户文档，我们知道 add()会“确定这个容器包含指定的元素”。这才使 Set 有了自己的意义，只有在指定元素尚不存在的前提下，它才会将元素加入！对 ArrayList 或者任何形式的 List 来说，add()则无论如何都会加入元素，因为 List 并不关心是否存在重复元素。

所有集合都能通过它们的 iterator()方法来产生一个“迭代器”。在这里，我们创建了一个迭代器，并用它在集合里进行巡视，打印出每一个元素。

9.6 深 入 集 合

下面这张表格总结了用一个“集合”（Collection）能做的所有事情（注意不包括随 Object 一道自动继承来的方法）。显然，既然用集合能做这些事情，那么用一个 Set 或者 List 同样也能做到（注意 List 还增加了一些自己的功能）。而 Map 由于不是从“集合”继承的，所以要单独对待。

boolean add(Object)	保证容器有自己的参数。假如没有添加参数，就返回 false（假）（这是一个“可选”的方法，稍后还会讲述它的详情）
boolean addAll(Collection)	添加参数内的所有元素。如果元素已成功添加，就返回 true（真）（同样“可选”）
void clear()	删除容器内所有元素（同样“可选”）
boolean contains(Object)	若容器包含参数，就返回 true
boolean containsAll(Collection)	若容器包含了参数内的所有元素，就返回 true
boolean isEmpty()	若容器内没有元素，就返回 true
Iterator iterator()	返回一个迭代器，用它遍历容器内的各个元素
boolean remove(Object)	如参数在容器里，就删除那个元素的一个实例。如果进行了一次删除，就返回 true（同样“可选”）
boolean removeAll(Collection)	删除参数里包含的所有元素。如果进行了一次删除，就返回 true（同样“可选”）
boolean retainAll(Collection)	只保留那些包含在一个参数里的元素（亦即集合理论中的一个“交集”）。如果进行了这样的改变，就返回 true（同样“可选”）
int size()	返回容器内的元素数量
Object[] toArray()	返回包含了容器内所有元素的一个数组
Object[] toArray(Object[]a)	返回包含了容器内所有元素的一个数组，数组的类型必须和数组 a 一样，而不应该是一个普通的 Object（当然，必须将数组强制转型成正确的类型）

注意这里没有一个特殊的 get()函数，可帮助我们实现随机的元素选定。这是由于集合也包括了 Set，而 Set 内置了自己的排序机制，所以随机访问是没有意义的。换言之，假如你想对一个集合里的所有元素进行检查，那么必须使用一个“迭代器”；它是我们用来从里面取出东西的唯一手段！

下面这个例子向大家演示了所有方法。同样地，它们适用于从“集合”继承的所有东西——只是 ArrayList 对自己的应用场合有点儿“挑剔”：

```
//: c09:Collection1.java
// Things you can do with all Collections.
import java.util.*;
import com.bruceeckel.util.*;

public class Collection1 {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Collections2.fill(c,
      Collections2.countries, 10);
    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, 10);
    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, 10);
    System.out.println(c);
    c.clear(); // Remove all elements
    System.out.println("after c.clear():");
    System.out.println(c);
  }
} ///:~
```

我们创建 ArrayList 的目的是在其中包括不同的数据集，然后向上强制转型成 Collection 对象，所以很明显，除了 Collection 接口之外，我们在这里用不着其他任何东西。在 main() 中，我们用简单的手段，显示出了由 Collection 提供的所有方法。

在后续的小节里，我们将比较 List、Set 和 Map 的不同实现，同时指出在每种情况下哪一种方案应成为自己的默认选择（带＊号的那个）。大家会发现这里并未包括一些传统的类，如 Vector、Stack 以及 Hashtable 等。因为不管在什么情况下，Java 2 容器内部都有自己首选的类！

9.7 深入 List

基本 List 的使用非常简单——就象大家迄今为止在 ArrayList 身上看到的那样。尽管在大多数时候，我们只需用 add()插入对象，用 get()一次取出一个对象，并用 iterator()来获得

一个迭代器，但事实上，我们还有别的一系列方法可供利用！

另外，List 实际有两种类型，其中包括最基本的 ArrayList，它擅长于对元素的随机访问；以及功能更强大的 LinkedList（用它进行随机访问的效率并不高，但它提供的方法要丰富得多）。

List（接口）	顺序是 List 最重要的特性；它可保证元素按规定的顺序排列。List 为 Collection 添加了大量方法，以便我们在 List 的中部插入和删除元素（只推荐对 LinkedList 这样做）。List 也会生成一个 ListIterator（列表迭代器），利用它可在一个 List 里朝两个方向巡视，同时插入和删除位于 List 中部的元素
ArrayList *	由一个数组实现的 List。可对元素进行速度非常快的随机访问，但用它在 List 中部插入或删除元素的时候，速度却比较慢。ListIterator 通常只能用于在一个 ArrayList 中来回“遍历”，而不要用它插入或删除元素——那应该是 LinkedList 的事情！
LinkedList	提供优化的顺序访问性能，同时可以高效地在 List 中部进行插入和删除操作。但在进行随机访问时，速度却相当慢，此时应换用 ArrayList。另外，它还提供了 addFirst()、addLast()、getFirst()、getLast()、removeFirst()以及 removeLast()等有用的方法（这些方法在任何接口或基类中均未定义），适用于对堆栈、队列以及双头队列进行操作

在下面这个例子中，每个方法都演示了一组不同的行动。这些行动包括：每个列表都能做的事情（basicTest()）、通过一个迭代器遍历（iterMotion()）、用一个迭代器改变某些东西（iterManipulation()）、体验 List 处理的效果（testVisual()）以及只有 LinkedList 才能做的事情等等：

```
//: c09: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;
  }
  static boolean b;
  static Object o;
  static int i;
  static Iterator it;
  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();
  }
} ///:~
```

在 basicTest()和 iterMotiion()中，只是简单地发出调用，以便揭示出正确的语法。而且尽管捕获了返回值，但是并未使用它。在某些情况下，之所以不捕获返回值，正是由于它们并无什么特别的用处。在正式运用这些方法之前，应仔细研究一下它们的联机文档（在 java.sun.com），掌握完整、正确的用法。

9.7.1 用 LinkedList 生成堆栈

“堆栈”（Stack）有时也可以称作一种“后入先出”（Last-in/First-Out，即 LIFO）容器。换言之，我们在堆栈里最后“压入”的东西将是以后第一个“弹出”的。和其他所有 Java 容器一样，我们压入和弹出的都是“对象”，所以必须对自己弹出的东西进行“强制转型”，除非只是想使用 Object 的行为。

LinkedList 提供了用来直接实现堆栈功能的方法，所以亦可考虑只用一个 LinkedList，不用再创建一个堆栈类。不过，堆栈类有时能取得更好的效果：

```
//: c09:StackL.java
// Making a stack from a LinkedList.
import java.util.*;
import com.bruceeckel.util.*;

public class StackL {
  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());
  }
} ///:~
```

假如只是想获得堆栈的行为，那么这儿进行继承便显得颇不恰当，因为它会生成一个类，同时带有剩下的所有 LinkedList 方法（稍后大家便会知道，这实际是 Java 1.0 的库设计人员对“堆栈”犯下的一个巨大错误）。

9.7.2 用 LinkedList 生成队列

“队列”（Queue）是一种“先入先出”（First-In/First-Out，FIFO）容器。也就是说，我们在一端压入东西，再在另一端依次取出。因此，“压入”的顺序和“弹出”的顺序是完全一致的。LinkedList 提供了对队列行为提供支持的方法，所以可在一个 Queue 类中使用它们：

```
//: c09:Queue.java
// Making a queue from a LinkedList.
import java.util.*;

public class Queue {
  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());
  }
} ///:~
```

亦可根据一个 LinkedList 方便地创建出一个“双头队列”（Deque）。它和普通队列大致相似，只是在两端都能增删元素。

9.8 深入 Set

Set 拥有和 Collection 完全一致的接口，所以和同时存在两种不同的 List 不同，这儿并不存在两种不同的 Set，并没有什么“额外”的 Set 功能。相反，你可以这样认为：Set 完全就是一个 Collection，只是行为有所区别！事实上，这正是继承和多态的一种理想应用：用来表示不同的行为！使用 Set 时，每个对象值只能对应一个实例，Set 拒绝容纳更多的实例（如后所示，对象“值”的构成是相当复杂的）。

Set（接口）	添加到 Set 的每个元素都必须是独一无二的；Set 不会添加重复的元素。添加到 Set 里的对象必须定义 equals()，以树立对象的“唯一”性。Set 拥有与 Collection 完全相同的接口。Set 接口并不保证自己会按任何特定的顺序来容纳元素。
HashSet*	假如在一个 Set 中的搜索速度是至关重要的，就应考虑用 HashSet。同时，Object 还必须定义 hashCode()。
TreeSet	排好序的一种 Set，采用树形结构。这样一来，就可从 Set 里提取出一个固定顺序的元素序列。

下面这个例子并没有列出用一个 Set 能做的全部事情！由于接口和 Collection 是完全一致的，前例已向大家展示过了，所以这儿不再重复。相反，这个例子只演示出了 Set 专有的那些行为：

```
//: c09:Set1.java
// Things you can do with Sets.
import java.util.*;
import com.bruceeckel.util.*;

public class Set1 {
  static Collections2.StringGenerator gen =
    Collections2.countries;
  public static void testVisual(Set a) {
    Collections2.fill(a, gen.reset(), 10);
    Collections2.fill(a, gen.reset(), 10);
    Collections2.fill(a, gen.reset(), 10);
    System.out.println(a); // No duplicates!
    // Add another set to this one:
    a.addAll(a);
    a.add("one");
    a.add("one");
    a.add("one");
    System.out.println(a);
    // Look something up:
    System.out.println("a.contains(\"one\"): " +
```

```
      a.contains("one"));
  }
  public static void main(String[] args) {
    System.out.println("HashSet");
    testVisual(new HashSet());
    System.out.println("TreeSet");
    testVisual(new TreeSet());
  }
} ///:~
```

重复的值被有意加入 Set，但在打印时，我们会发现 Set 只接受每个值的一个实例，多余（重复）的值被剔除了。

运行这个程序时，会注意到由 HashSet 维持的顺序与 TreeSet 是不同的。这是由于它们采用了不同的方法来保存元素，以便以后的定位（TreeSet 会将元素排好序；而 HashSet 使用的是一个散列函数，目的是以后更快地搜索目标）。创建自己的类型时，要注意 Set 需通过一种方式来维持一种存储顺序，这便意味着我们必须实现 Comparable 接口，同时定义 compareTo()方法。下面是一个例子：

```
//: c09:Set2.java
// Putting your own type in a Set.
import java.util.*;

class MyType implements Comparable {
  private int i;
  public MyType(int n) { i = n; }
  public boolean equals(Object o) {
    return
      (o instanceof MyType)
      && (i == ((MyType)o).i);
  }
  public int hashCode() { return i; }
  public String toString() { return i + " "; }
  public int compareTo(Object o) {
    int i2 = ((MyType)o).i;
    return (i2 < i ? -1 : (i2 == i ? 0 : 1));
  }
}

public class Set2 {
  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());
  }
} ///:~
```

对 equals()及 hashCode()的定义将在本章后面讲述。在两种情况下，都必须定义一个 equals()。但只有当类需要放入一个 HashSet 的时候，才绝对有必要使用 hashCode()——这种情况是完全有可能的，因为在实现一个 Set 时，HashSet 是你的首要选择。不过，作为一种良好的编程习惯，在你覆盖 equals()的时候，无论如何都应该覆盖 hashCode()。对此，本章后面还会详细讲述。

在 compareTo()中，请注意我并未使用“简单和明显”的 return i-i2 形式。尽管这通常都是一种编程错误，但也并不是绝对的——假如 i 和 i2 均为“无符号”或“无正负号”的 int 值（前提是 Java 有一个“unsigned”关键字，不过现在没有实现），这样做便丝毫没有问题！事实上，它破坏了 Java 的“有符号”int，因为后者并不足够“大”，不足以表示出两个“有符号”int 数值的差异。假如 i 是一个大的正整数，而 j 是一个大的负整数，那么 i-j 就会溢出，并返回一个负数，这当然是不可行的。

9.8.1 SortedSet

如果你有一个 SortedSet，元素便保证能按指定的顺序排列，从而方便我们利用由 SortedSet 接口提供的下面这些方法，对其进行额外的操作：

Comparator comparator()：产生用于这个 Set 的 Comparator（比较器）；对于自然顺序，则产生 null。

Object first()：产生位置最低的元素（第一个）

Object last()：产生位置最高的元素（最后一个）

SortedSet subSet(fromElement, toElement)：产生这个 Set 的一个“景象”（View），其中含有从 fromElement（包括 fromElement）到 toElement（不包括 toElement）的一系列元素（即“元素的一个子集”）。

SortedSet headSet(toElement)：产生这个 Set 的一个“景象”，其中包括从第一个到 toElement 之前的所有元素。

SortedSet tailSet(fromElement)：产生这个 Set 的一个“景象”，其中包括从 fromElement 开始，一直到最后的所有元素。

9.9 深入 Map

ArrayList 允许我们根据一个数字，从一个对象序列中作出选择。换句话说，它是将数字同对象关联到一块儿了。但是，假如我们想根据其他标准，从一个对象序列中作出选择，

那么又该如何是好呢？“堆栈”便是这样的一个例子，它的选择标准是：“最后一个压入堆栈的东西”。所以，为了能够“从一个序列中作出选择”，我们还可采用另一种功能更为强大的方式，即“Map”（映射）——你也可以把它叫作“字典”或者“关联数组”。从概念上说，它和 ArrayList 差不多，只是不再用一个数字来查找对象，而是用“另一个对象”来进行查找！这可是一个至关重要的编程理念！

在 Java 中，我们利用 Map 接口，便可真正地实践这一理念。它的 put(Object key, Object value)方法可以添加一个值（我们希望的东西），并把它同一个键（用来搜索的东西）关联到一起；而 get(Object key)可根据你指定的一个“键”（Key），找出对应的“值”（Value）；另外，亦可利用 containsKey()和 containsValue()，分别测试一个 Map 里是否包含了一个指定的键或值。

在标准 Java 库中，包含了两种不同类型的 Map——HashMap 和 TreeMap。两者均有相同的接口（因为都实现了 Map 嘛），只是在“工作效率”这一点上，两者有着显著的区别。例如，注意一下 get()的工作过程，你就会发现假如在一个 ArrayList 里查找指定的键，速度会慢得令人“伤心”。而这正是 HashMap 可以大展拳脚的地方。它并不是以极慢的速度来搜索一个键，而是利用一个特殊的值，名为“散列码”（Hash Code）。散列码要求先从对象中提取出一些关键信息，然后转变成一个“相对唯一”的 int 值——这个值就是该对象的“散列码”。事实上，所有 Java 对象都能生成一个散列码，而且 hashCode()已成为在“Object”这个根类中实现的一个方法。HashMap 可取得一个对象的 hashCode()，并用它以极快的速度，找出指定的键——结果当然便是显著的性能提升[46]！

Map（接口）	维持“键－值”对应关系（对），以便根据一个键，查找到相应的值
HashMap＊	基于一个散列表实现（用它代替 Hashtable）。针对“键－值”对的插入和检索，这种形式具有最稳定（但不是最好）的性能。可通过构造函数设置散列表的“容量”与“负载比”，从而对性能进行调整
TreeMap	在一个“红－黑”树的基础上实现。查看键或者“键－值”对时，它们会按固定的顺序排列（取决于 Comparable 或 Comparator，稍后即会讲到）。TreeMap 最大的好处就是我们得到的是已排好序的结果。TreeMap 是提供了 subMap()方法的唯一一种 Map，用它可返回树的一部分（子映射）

有的时候，大家还需要知道“散列”的基本原理，所以稍后也会对其进行探讨。

下面这个例子使用了以前定义过的 Collections2.fill()方法以及测试数据集：

```
//: c09:Map1.java
// Things you can do with Maps.
import java.util.*;
import com.bruceeckel.util.*;

public class Map1 {
  static Collections2.StringPairGenerator geo =
```

[46] 假如这种加速手段仍然不能满足你对性能的要求，那么还可以编写自己的 Map，根据自己要用到的特定类型，对其进行专门的“定制”——避免由于到“Object”的强制转型或者从“Object”的强制转型而产生额外延时——从而进一步加快表搜索速度。此外，更进一步的提速手段可参照由 Donald Knuth 编著的“计算机编程艺术”第 3 卷:《排序和搜索:第 2 版》，将满溢的桶列表换成数组——这样做有两方面的好处：可根据磁盘存储的物理特征，进行专门的优化；而且省去了花在单条记录创建与垃圾收集上的大多数时间。

```
    Collections2.geography;
  static Collections2.RandStringPairGenerator
    rsp = Collections2.rsp;
  // Producing a Set of the keys:
  public static void printKeys(Map m) {
    System.out.print("Size = " + m.size() +", ");
    System.out.print("Keys: ");
    System.out.println(m.keySet());
  }
  // Producing a Collection of the values:
  public static void printValues(Map m) {
    System.out.print("Values: ");
    System.out.println(m.values());
  }
  public static void test(Map m) {
    Collections2.fill(m, geo, 25);
    // Map has 'Set' behavior for keys:
    Collections2.fill(m, geo.reset(), 25);
    printKeys(m);
    printValues(m);
    System.out.println(m);
    String key = CountryCapitals.pairs[4][0];
    String value = CountryCapitals.pairs[4][1];
    System.out.println("m.containsKey(\"" + key +
      "\"): " + m.containsKey(key));
    System.out.println("m.get(\"" + key + "\"): "
      + m.get(key));
    System.out.println("m.containsValue(\""
      + value + "\"): " +
      m.containsValue(value));
    Map m2 = new TreeMap();
    Collections2.fill(m2, rsp, 25);
    m.putAll(m2);
    printKeys(m);
    key = m.keySet().iterator().next().toString();
    System.out.println("First key in map: "+key);
    m.remove(key);
    printKeys(m);
    m.clear();
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
    Collections2.fill(m, geo.reset(), 25);
    // Operations on the Set change the Map:
    m.keySet().removeAll(m.keySet());
```

```
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
  }
  public static void main(String[] args) {
    System.out.println("Testing HashMap");
    test(new HashMap());
    System.out.println("Testing TreeMap");
    test(new TreeMap());
  }
} ///:~
```

其中，printKeys()和 printValues()方法并不仅仅是非常有用的两个工具，它们还向大家演示了如何生成一个 Map 的“集合”景象。keySet()方法会根据 Map 中的键，产生一个 Set。类似地，再用 values()产生一个集合，其中包括了 Map 中的所有值（注意键必须是独一无二的，而值可以有重复）。由于这些集合的数据来源是 Map，所以集合中发生的任何改变都会在对应的 Map 中反映出来。

程序剩余的部分则演示了每一种 Map 操作，同时对每类 Map 都进行了测试。

作为运用 HashMap 的一个例子，可考虑用一个程序来检验 Java 的 Math.random()方法的随机性到底如何。在理想情况下，它应该产生一系列完美的随机分布数字。但为了验证这一点，我们需要生成数量众多的随机数字，然后计算落在不同范围内的数字有多少。此时利用散列表，便可极大简化这一工作，因为它能将对象同对象关联起来（这里是将 Math.random()生成的值同那些值出现的次数关联起来）。如下所示：

```
//: c09:Statistics.java
// Simple demonstration of HashMap.
import java.util.*;

class Counter {
  int i = 1;
  public String toString() {
    return Integer.toString(i);
  }
}

class Statistics {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10000; i++) {
      // Produce a number between 0 and 20:
      Integer r =
        new Integer((int)(Math.random() * 20));
      if(hm.containsKey(r))
        ((Counter)hm.get(r)).i++;
      else
```

```
        hm.put(r, new Counter());
    }
    System.out.println(hm);
  }
} ///:~
```

在 main()中，每次产生一个随机数字，它都会封装到一个 Integer 对象里，使引用能随 HashMap 一起使用（不可随一个容器使用原始数据类型，只能使用对象引用）。containKey()方法检查这个键是否已经在容器里（也就是说，那个数字以前发现过吗？）若已在容器里，则 get()方法会产生与那个键对应的值，此时就是一个 Counter（计数器）对象。计数器内的值 i 随后会递增 1，表明这个特定的随机数字又出现了一次。

假如键以前尚未发现过，那么方法 put()仍然会在 HashMap 内置入一个新的“键－值”对。在创建之初，Counter 会自己的变量 i 自动初始化为 1，它标志着该随机数字的第一次出现。

为显示出 HashMap，只需把它简单地打印出来即可。HashMap toString()方法能遍历所有“键－值”对，并为其中的每一对都调用 toString()方法。Integer toString()是事先定义好的，大家可看到为 Counter 使用的 toString()。下面是某一次运行的结果（自己加了一些换行）如下：

```
{19=526, 18=533, 17=460, 16=513, 15=521, 14=495,
 13=512, 12=483, 11=488, 10=487, 9=514, 8=523,
 7=497, 6=487, 5=480, 4=489, 3=509, 2=503, 1=475,
 0=505}
```

大家或许会对 Counter 类是否必要感到疑惑，它看起来似乎根本没有封装器类 Integer 的功能。那么为什么不用 int 或 Integer 呢？事实上，由于所有容器仅能容纳对象引用，所以根本不可能使用一个 int 值。学过容器之后，封装器类对大家来说就可能更有意义了，因为你无法将任何原始数据类型放到一个容器里。然而，利用 Java 的封装器，我们能做的唯一一件事情就是将其初始化成一个特定的值，然后读取那个值。也就是说，一旦封装器对象已经创建，就没办法改变一个值。这使得 Integer 封装器对解决我们的问题来说毫无意义，所以不得不创建一个新类，用它来满足自己的要求。

9.9.1 SortedMap

如果你有一个 SortedMap，那么其中的“键”会保证按固定顺序排列，从而允许我们利用由 SortedMap 提供的下述方法，对其采取一些额外的操作：

Comparator comparator()：产生用于这个 Map 的 Comparator（比较器）；对于自然顺序，则产生 null。

Object firstKey()：产生位置最低的键（第一个）

Object lastKey()：产生位置最高的键（最后一个）

SortedMap subMap(fromKey, toKey)：产生这个 Map 的一个“景象”（View），其中含有从 fromKey（包括 fromKey）到 toKey（不包括 toKey）的一系列键（即“Map 的一个子集”）。

SortedMap headMap(toKey)：产生这个 Map 的一个“景象”，其中包括从第一个到 toKey 之前的所有键。

SortedMap tailMap(fromKey)：产生这个 Map 的一个“景象”，其中包括从 fromKey 起，

一直到最后的所有键。

9.9.2 散列和散列码

在前面的例子里，我们将一个标准库的类（Integer）作为 HashMap 的一个键使用。作为一个键，它确实能很好地工作，因为它已具备了正确运行的所有条件。但在使用 HashMap 的时候，一旦需要创建自己的类，把它们作为键使用，就会遇到一个颇为常见的问题。例如，假设一套天气预报系统将 Groundhog（土拔鼠）和 Prediction（预报）这两个对象对应起来。从表面看，做法似乎很简单——我们创建两个类，然后将 Groundhog 作为键使用，而将 Prediction 作为值使用。就象下面所示：

```
//: c09:SpringDetector.java
// Looks plausible, but doesn't work.
import java.util.*;

class Groundhog {
  int ghNumber;
  Groundhog(int n) { ghNumber = n; }
}

class Prediction {
  boolean shadow = Math.random() > 0.5;
  public String toString() {
    if(shadow)
      return "Six more weeks of Winter!";
    else
      return "Early Spring!";
  }
}

public class SpringDetector {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Groundhog(i), new Prediction());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Looking up prediction for Groundhog #3:");
    Groundhog gh = new Groundhog(3);
    if(hm.containsKey(gh))
      System.out.println((Prediction)hm.get(gh));
    else
      System.out.println("Key not found: " + gh);
  }
} ///:~
```

每个 Groundhog 都指定了一个标识号码，所以为了在 HashMap 中查找一个 Prediction，只需指示它“把与第 3 号 Groundhog 有关的 Prediction 告诉我”。Prediction 类包含了一个布尔值，用 Math.random()进行初始化，同时还有一个 toString()为我们解释结果。在 main()中，用 Groundhog 以及与它们相关的 Prediction 填充一个 HashMap。HashMap 被打印出来，以便我们看到它们确实已被填充。随后，用标识号码为 3 的一个 Groundhog 查找与 Groundhog #3（大家可以看到，它肯定在 Map 中）对应的预报。

看起来似乎非常简单，但这实际上是不可行的。问题在于 Groundhog 是从通用的 Object 根类继承的（若当初未指定基类，则所有类最终都是从 Object 继承的）。事实上，我们是用 Object 的 hashCode()方法来生成每个对象的散列码，而且默认情况下，只使用了它的对象的地址。所以，Groundhog(3)的第一个实例并不会产生与 Groundhog(3)第二个实例相同的散列码，而我们正是用第二个实例来进行检索的。

考虑到这一点，大家或许又会想，是不是只需要正确地覆盖 hashCode()就行了呢？这样依然行不通，除非你再做另一件事情：覆盖也属于 Object 一部分的 equals()。当 HashMap 试图判断我们的键是否“等于”表内的某个键时，就会用到这个方法。同样地，默认的 Object.equals()只是简单地比较对象地址，所以一个 Groundhog(3)并不等于另一个 Groundhog(3)。

综上所述，为了能在 HashMap 中将自己的类作为键使用，必须同时覆盖 hashCode()和 equals()，就象下面展示的那样：

```
//: c09:SpringDetector2.java
// A class that's used as a key in a HashMap
// must override hashCode() and equals().
import java.util.*;

class Groundhog2 {
  int ghNumber;
  Groundhog2(int n) { ghNumber = n; }
  public int hashCode() { return ghNumber; }
  public boolean equals(Object o) {
    return (o instanceof Groundhog2)
      && (ghNumber == ((Groundhog2)o).ghNumber);
  }
}

public class SpringDetector2 {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Groundhog2(i),new Prediction());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Looking up prediction for groundhog #3:");
    Groundhog2 gh = new Groundhog2(3);
```

```
    if(hm.containsKey(gh))
      System.out.println((Prediction)hm.get(gh));
  }
} ///:~
```

注意这段代码使用了来自前一个例子的 Prediction 类，所以 SpringDetector.java 必须首先编译，否则就会在试图编译 SpringDetector2.java 时得到一个编译时间错误。

Groundhog2.hashCode()将土拔鼠号码作为一个标识符返回。在这个例子中，需要由程序员来保证没有两个土拔鼠拥有同样的 ID 号码。如果只是为了返回一个独一无二的标识符，那么并不需要 hashCode()（对于这个问题，到本章后面还会有更好的理解），但 equals()方法必须能严格对比两个对象是否相等。

尽管从表面看，equals()方法只检查了一个参数是不是 Groundhog2 的一个实例（利用 instanceof 关键字，它是我们在第 12 章的主题），但 instanceof 实际上还“默默”地执行了另一次检查——检查对象是不是 null。这是由于假如左边的参数为 null 的话，instanceof 就会产生一个 false（假）。好了，现在假定类型正确，而且也不是 null，就会接着根据实际的 ghNumber，进行正式的比较。在这一次，当你运行程序后，就会看到它终于产生了正确的输出！

如果想自行创建一个类，并在 HashSet 中使用，那么就和在一个 HashMap 中使用一个键那样，必须对同样的问题加以注意。

理解 hashCode()

针对问题的全面、正确解决，上面的例子仅仅是一个开头。它显示出：假如不为自己的键覆盖 hashCode()和 equals()，那么散列数据结构（HashSet 或 HashMap）就不能对你的键进行正确的操作。然而，要想为问题寻找一个真正好的解决方案，你首先需要理解在一个散列数据结构中发生的事情。

首先，请考虑我们采用“散列”的动机：希望利用一个对象，来查找另一个对象。但是，用一个 TreeMap 或者 TreeMap 同样可以达到这个目的。此外，甚至可以实现自己的 Map，来解决这个问题。为此，我们必须提供 Map.entrySet()方法，以便产生一系列 Map.Entry 对象。MPair 将被定义成一种新类型的 Map.Entry。为了能把它放到一个 TreeSet 里，它必须实现 equals()，而且必须是“Comparable”（可比较）的：

```
//: c09:MPair.java
// A Map implemented with ArrayLists.
import java.util.*;

public class MPair
implements Map.Entry, Comparable {
  Object key, value;
  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);
  }
} ///:~
```

注意进行比较时，感兴趣的只是键，所以即便有重复的值，也丝毫没有关系！

下面这个例子利用一对 ArrayList，实现了一个 Map：

```
//: c09:SlowMap.java
// A Map implemented with ArrayLists.
import java.util.*;
import com.bruceeckel.util.*;

public class SlowMap extends AbstractMap {
  private ArrayList
    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 static void main(String[] args) {
    SlowMap m = new SlowMap();
    Collections2.fill(m,
      Collections2.geography, 25);
    System.out.println(m);
  }
} ///:~
```

put()方法的作用很简单，它将键和值放到对应的 ArrayList 里。在 main()中，我们载入了一个 SlowMap，然后打印出来，证明它能正常工作。

由此可以看出，我们其实不难生成一个新类型的 Map。但正象它的名字暗示的那样，“SlowMap”的速度并不快，所以假如还有其他方式，那么也许不会考虑它。其根本问题还是在于对键的搜索：由于这些“键”是无序的，所以只好采用简单的“线性搜索”方式，它是在你找什么东西的时候最慢的一种方法！

而“散列”就不同，它的一切都是为了“速度”服务的。使用散列，当你要找什么东西的时候，便可以极快地完成。由于我们目前的“瓶颈”在于键的搜索速度，所以为了解决这个问题，一个办法是保持键的排序状态，然后利用 Collections.binarySearch()方法来执行搜索（本章结尾的一个练习会带领大家逐步实现它）。

“散列”则更进一步，它认为我们想做的一切事情就是把键保存在“某个地方”，以便往后快速查找。就象大家在本章中看到的那样，用来保存一组元素的最快的结构就是“数组”。因此，我们用它来代表键信息（注意这里说的是“键信息”，而不是“键”本身）。不过，通过本章的学习，大家还知道对一个数组来说，一旦它分配好，便不可再改变大小。这样一来，我们就碰到了一个难题：原本是想能在 Map 里保存任意数量的值，但假如键的数量已被数组的大小固定死了，那又该如何是好呢？

个中巧妙完全在于数组不会实际容纳“键”这一事实！根据键对象，会得出一个数字，它相当于数组的索引。该数字正是我们的“散列码”，由 hashCode()方法产生（用计算机科学的术语来讲，那个方法就是“散列函数”）。该方法是在 Object 中定义的，而且已由我们的类覆盖。为了解决数组大小不能变动的问题，多个键可能产生相同的索引。换言之，其中可能存在“冲突”。正是由于这一点，数组不管多大都没关系，因为反正每个键对象都会到那个数组内的某个地方去。

因此，为了查找一个具体的值，我们首先应该计算出散列码，然后用它对数组进行索引。如果能担保其中不存在“冲突”（假如值的数量固定，那是完全有可能的），就相当于得到了一个“完美散列函数”——但那只能属于一种特别的情况。在其他任何情况下，我们都要利用“外部链接”（External Chaining）来应付这种“冲突”。“外部链接”的全部含义就是：数组不直接指向一个值，而是指向由值构成的一个列表。使用 equals()方法，这些值将采用一种“线性”方式进行搜索。当然，前面也已说过，这样的搜索方式是相当慢的。不过，只要散列函数设计得好，那么每个位置实际上只有少数几个值（大多数时候如此）。因此，我们完全没必要对整个列表进行搜索，只需快速跳到一个位置，然后在几个值间搜索，找到自己需要的那一个就可以了！这样速度会快上许多，而这也正是 HashMap 的检索速度为何如此

之快的原因！

知道了“散列”的原理之后，再来实现一个简单的散列 Map 就简单多了：

```
//: c09: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 final static 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);
  }
} ///:~
```

在一个散列表中，“位置”这个概念显然有点儿表述不清，因此我们通常把它称作“桶”（Buckets）；而用来表示实际“表格”（数据表）的数组就叫作“桶”。为保证平均分布，“桶”的数目通常应该是一个质数。要注意的是，它是一个由 LinkedList 构成的数组，而 LinkedList 对于“冲突”是没有任何免疫力的——每个新项目都会那么简简单单地添加到列表末尾。

put()的返回值是一般是 null；但假如键已经在列表中了，老值就会同那个键关联在一起。返回值一般是 result，它被初始化成 null；但假如在列表中找到了一个键，就会将 result 赋给那个键。

无论 put()还是 get()，发生的第一件事情都是为键调用 hashCode()，而且结果（散列码）被强制成一个正数。随后，使用模数运算符，并根据数组的大小，强迫它与 bucket 数组相符。如果那个位置是 null，表明没有元素散列到那个位置，所以会创建一个新的 LinkedList，以容纳刚才得到的对象。但是，正常的操作应该是在列表中搜索，检查是否有重复的；如果有，老值就会放到 result 里，而新值会替代老值。found 标记用于跟踪记录一个老的“键－值”对是否已经发现过；如果还没有发现，就将新的“键－值”对追加到列表末尾。

在 get()中，大家会看到与 put()大致相同的代码，只不过显得更简单。索引被计算到 bucket 数组里。而且假如存在 LinkedList，就在其中查找是否有相符的项目。

entrySet()必须查找和巡视所有列表，把它们添加到结果 Set 里。一旦创建好这个方法，就可向 Map 中填入值，然后打印出它们，从而完成对 Map 的测试。

HashMap 的性能因素

为了理解本小节要讲述的问题，有几个术语应该首先弄明白：

容量：表中最多允许的“桶”数

初始容量：表最初创建时，其中的“桶”数

大小：表中当前有多少个“桶”

负载比：即“大小 / 容量”。如负载比等于 0，表明是一个空表；0.5 表明是一个半空表；以此类推。对一个“负担”不重的表来说，由于其中只有数量极少的冲突，所以特别适合进行插入和搜索操作（但是，假如用一个迭代器在其中进行巡视，速度反而会降低）。HashMap 和 HashSet 都提供了相应的构造函数，允许我们自行设定负载比。也就是说，一旦达到负载比，容器就会自动增大容量（即最多允许的“桶”数）——大致增加到以前的两倍！同时，原有的对象会重新分布，由一系列新的“桶”来装载——这个过程叫作“重散列”。

对 HashMap，它采用的默认负载比是 0.75（换言之，除非装满了 3/4 的空间，否则不会重新散列）。综述考察时间及空间这两方面的因素，0.75 似乎取得了一个比较好的平衡——如果采用一个更大的负载比，那么尽管会减少数据表对空间的需求，但也会同时增大检索时的代价。而由于我们经常都要进行“检索”（包括使用 get()和 put()），所以对“更快的检索速度”有着比较高的要求。

当然，假定事前知道自己以后会在 HashMap 里保存大量条目，那么最好一开始就设置一个比较大的“容量”，从而减少由于以后频繁自动“重散列”而带来的开销。

9.9.3　覆盖 hashCode()

现在，大家已理解了 HashMap 背后的原理。接下来，让我们考虑一下写一个 hashCode()时，可能会牵涉到哪些问题。

首先应该记住的是，我们无权控制那个对“桶”数组进行索引的实际值的创建。它具体是由一个特定 HashMap 对象的“容量”来决定的，而那个“容量”又要依据容器目前的“负载”来决定——而这正是设计“负载比”的原因。一旦超出这一比值，容器便需自动“扩容”。由我们的 hashCode()产生的值需作进一步的处理，以便创建出“桶”索引（在 SimpleHashMap 中，其实就是用桶数组的大小，求出这个值的模数）。

hashCode()最有魅力的地方在于，无论在什么时候调用 hashCode()，每次调用它的时候，它都会为一个特定的对象生成固定的值！假定我们有一个对象，在用 put()把它“放”到一个 HashMap 的时候，生成了一个 hashCode()值；然后又在使用 get()取出的时候得到了另一个 hashCode()值，那么实际上并不能取得对象。所以，假如你的 hashCode()要依赖于对象中不断发生变化的数据，那么必须警告自己的用户——假如数据发生改变，那么由于生成的是不同的 hashCode()，所以最后产生的“键”也是不同的！

除此以外，有某些情况下，我们并不希望根据“独一无二”的对象信息来生成一个 hashCode()值——特别要指出的是，this 的值会产生一个错误的 hashCode()，因为这样一来，尽管随后也能生成一个键，但它和当初通过 put()放入原始“键－值”对的那一个键相比，两者是并不相同的。我们的 SpringDetector.java 便存在这样的问题，因为 hashCode()的默认实现代码采用的是对象地址。因此，我们在这里的希望是：对象里用来真正标定出那个对象的信息应该是“有意义”的。

String 类里便有这样的一个例子。“字串”的一个特点在于，假如程序中有几个字串对象，每个都包含了一致的字符序列，那么那些字串对象都会映射到相同的内存区域（具体机制将在本书附录 A 讲述）。换句话说，对于 new String("hello")两个单独的实例来说，它们产生的 hashCode()值应该是一模一样的。用下面这个程序可以证明：

```
//: c09:StringHashCode.java
public class StringHashCode {
```

```
  public static void main(String[] args) {
    System.out.println("Hello".hashCode());
    System.out.println("Hello".hashCode());
  }
} ///:~
```

要让它运行起来，String 的 hashCode()必须建立在 String 的内容的基础上。

所以为了让 hashCode()真正发挥效用，它的速度必须非常快，而且必须“有意义”。换句话说，它必须真正根据对象的内容而产生一个值。请记住这个值并不一定非要“独一无二”——我们真正关心的应该是速度，而不是“唯一性”。不过，在 hashCode()到 equals()之间，对象的身份辨认问题必须得到彻底解决。

由于在“桶”索引产生之前，hashCode()还会得到进一步处理，所以值的范围（值域）并不重要；它只需生成一个 int 值就可以了。

这儿还有另一个因素：一个好的 hashCode()应该造成值的平均分布。假如这些值东一块西一块地分布不均，那么 HashMap 或 HashSet 在某些区域搜索起来就会比较“费劲”，最终造成整体速度的下降，得不偿失。

下面也有一个例子可供证明：

```
//: c09:CountedString.java
// Creating a good hashCode().
import java.util.*;

public class CountedString {
  private String s;
  private int id = 0;
  private static ArrayList created =
    new ArrayList();
  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() + "\n";
  }
  public int hashCode() {
    return s.hashCode() * id;
  }
```

```
  public boolean equals(Object o) {
    return (o instanceof CountedString)
      && s.equals(((CountedString)o).s)
      && id == ((CountedString)o).id;
  }
  public static void main(String[] args) {
    HashMap m = new HashMap();
    CountedString[] cs = new CountedString[10];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CountedString("hi");
      m.put(cs[i], new Integer(i));
    }
    System.out.println(m);
    for(int i = 0; i < cs.length; i++) {
      System.out.print("Looking up " + cs[i]);
      System.out.println(m.get(cs[i]));
    }
  }
} ///:~
```

CountedString 包括了 String 和 id 这两个字段，后者指出有多少个 CountedString 对象包含的是一模一样的 String。真正的计数是在构造函数里进行的，它会对 static ArrayList 进行“遍历”，所有 String 都保存在这个 ArrayList 中。

hashCode()和 equals()都同时根据两个字段来产生结果。假如它们只根据 String 或者 id，那么不同的值就可能产生重复的匹配。

请注意 hashCode()其实是非常简单的，它等于“String 的 hashCode()值乘以 id”。对 hashCode()值来说，越小当然越好（而且更快）！

在 main()中，我们创建了一系列 CountedString 对象，用相同的 String 证明：即使字串有重复，但它们也会产生不相同的、不重复的值，因为它们的计数 id 是不同的！而且，我们打印出了 HashMap 的内容，让大家看清楚它内部的存放方式（没有明显顺序），然后单独对每个键进行搜索，确认搜索机制能正常工作。

9.10 引用的容纳

在 java.lang.ref 库中，设计者提供了一系列类，目的是为“垃圾收集”提供更大的灵活性。假如你要设计一些大型对象，而且它们很容易耗尽内存资源，就绝对有必要考虑这些类。在这个库中，包含了自抽象类 Reference（引用）继承来的三个类：SoftReference、WeakReference 以及 PhantomReference。假如你的“问题对象”只能通过这些“引用”对象之一才能“访问”到，那么每个对象都能为垃圾收集器提供某种程度的“迂回”。

平常，如果我们说一个对象可被“访问到”，那么真正的意思是：在程序的某个地方，可以发现这个对象。而“发现”又有多方面的含义，比方说，堆栈里可能有一个标准引用，它一下子便能指向目标对象；但另外也可能有一个引用，它先指向对象甲，再通过对象甲指向对象乙……可能存在着许多这样的“中间链接”。只要一个对象还能被“访问到”，垃圾收集器便没法子把它“回收”掉，因为这说明你的程序仍在用它；而假如一个对象再也访问不

到了，由于你的程序已经没办法用它了，所以就能把它当作垃圾安全地“回收”掉。

那么，假如我们既想“访问”到一个对象，同时又想让垃圾收集器在必要的时候“回收”掉那个对象，又该怎么办呢？这时便要用到 Reference（引用）对象，用它保持到那个对象的一个“引用”。这样一来，我们在能继续使用对象的同时，也能在内存吃紧的情况下，让那个对象释放出自己占据的空间。

要做到这一点，需要将 Reference 对象当作你与标准引用之间的一个“中间人”使用，而且要做到绝对不能有到那个对象的标准引用（那些没有封装到 Reference 对象里的引用）。假如垃圾收集器发现一个对象可通过标准引用“访问到”，便不会主动释放那个对象。

对于 SoftReference、WeakReference 和 PhantomReference 这三个类来说，它们按照顺序是越来越“弱”的，各自对应一个不同的“可访问”级别。其中，SoftReference 用于实现对内存要求较苛刻的高速缓存；WeakReference 用于实现“标准映射”——对象实例可在一个程序内的多处地方同时使用，从而节省空间，但却不能防止它们的键（或值）被回收掉；PhantomReference 引用是最“弱不禁风”的一个，便于我们按照一种比 Java“收尾”机制更灵活的方式，事先安排好清除行动。

对 SoftReference 和 WeakReference 来说，我们可“决定”是否把它们放在一个 ReferenceQueue（即“引用队列”，用于事先安排清除行动的设备）里；但对 PhantomReference 来说，它却只能在一个 ReferenceQueue 中构建。下面是一个简单的演示：

```
//: c09:References.java
// Demonstrates Reference objects
import java.lang.ref.*;

class VeryBig {
  static final int SZ = 10000;
  double[] d = new double[SZ];
  String ident;
  public VeryBig(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizing " + ident);
  }
}

public class References {
  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();
    }
  }
} ///:~
```

运行这个程序后（建议你最好用一个“more”工具，对输出进行管道式处理，以便分页观看输出结果，比如命令行参数“|more”；其中的“|”就是一个管道符），大家可以发现对象已被进行了垃圾收集——尽管仍可通过 Reference 对象来访问它们（如果想获得实际的对象引用，需使用 get()）。另外，大家还会看到 ReferenceQueue 无论如何都会产生包含了一个 null 对象的 Reference。要想真正利用它，我们可从自己感兴趣的那个 Reference 类继承，然后为新类型的 Reference 添加更多有用的方法。

9.10.1 WeakHashMap

容器库提供了一个特殊的 Map 类，可用来容纳 WeakReference，这个 Map 就是 WeakHashMap。利用这个类，“标准映射”的创建工作可以变得更加容易。通过这样的映射，我们可有效节省存储空间，因为只需为每个特定的值产生一个实例！一旦程序要用到那个值，便会对映射关系中存在的对象进行检索，然后直接用那个现成的（而不是再从头创建一个）。当然，映射可在自己初始化的时候便将所有值都生成好，但在实际应用中，我们往往是根据实际的使用需要，让那些值“现场”生成。

由于设计 WeakHashMap 的目的是尽可能地节省空间，所以它也允许垃圾收集器自动清除键和值。因此，对于那些想放到 WeakHashMap 里的键和值来说，我们不需要对它们采取任何特别的操作；通过映射，它们会自动封装到 WeakReference 里。一旦某个键不再使用，对它的清除工作便会开始。如下例所示：

```
//: c09:CanonicalMapping.java
// Demonstrates WeakHashMap.
import java.util.*;
import java.lang.ref.*;

class Key {
  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 {
  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 whm = 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
      whm.put(k, v);
    }
    System.gc();
  }
} ///:~
```

其中，就象本章前面讲到的那样，Key 类必须同时实现了一个 hashCode()和一个 equals()，因为它会在一个散列数据结构中作为一个“键”使用。

运行程序时，大家会发现垃圾收集器每隔两个键便会跳过一个，这是由于在 keys 数组中，同时还放置了对那个键一个“标准引用”，所以不可被当作“垃圾”收去。

9.11 再论迭代器

现在，我们终于可以开始体验“迭代器”（Iterator）的真正威力——无论一个序列的基础结构是什么，都可在这个序列中顺利地遍历（巡视）！在后面的例子里，PrintData 会用一个迭代器在一个序列中移动，并为每个对象都调用 toString()方法。此时会创建两个不同类型的容器：一个 ArrayList 和一个 HashMap——并分别用 Mouse 和 Hamster 对象来填充它们（本章早些时候已定义好了这些类）。由于迭代器已将基础容器结构成功地隐藏起来，所以 PrintData 不知道、也不用关心迭代器到底是从哪种容器来的：

```
//: c09:Iterators2.java
// Revisiting Iterators.
import java.util.*;

class PrintData {
  static void print(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

class Iterators2 {
```

```
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 5; i++)
      v.add(new Mouse(i));
    HashMap m = new HashMap();
    for(int i = 0; i < 5; i++)
      m.put(new Integer(i), new Hamster(i));
    System.out.println("ArrayList");
    PrintData.print(v.iterator());
    System.out.println("HashMap");
    PrintData.print(m.entrySet().iterator());
  }
} ///:~
```

针对 HashMap，entrySet()方法会产生由 Map.entry 对象构成的一个 Set，其中包含了用于每个条目（Entry）的键和值。因此，键和值都会打印出来。

对于 PrintData.print()，请大家注意它利用了这样的一个事实：这些容器中的所有对象均属于 Object 类，所以 System.out.println()会自动发出对 toString()的调用。但在实际应用中，我们的“迭代器”往往需要在一个“特定”类型的容器中巡视。举个例子来说，在使用 draw()方法时，我们可能要假定容器中的所有东西都是一个“Shape”（几何形状）。为此，针对从 Iterator.next()返回的 Object，必须对其进行向下强制转型，以便正确地生成一个“Shape”。

9.12 如何选择实现

迄今为止，大家应该能理解我们实际上只有三种容器组件可用：Map、List 和 Set，而且每个接口只有两、三种实现。如果要使用由一个特定的接口提供的功能，那么到底该选用哪种实现呢？

为了理解这个问题，必须认识到每种不同的实现都有自己的特点、优点和缺点。比如通过本章最早的那张示意图，可以看到 Hashtable、Vector 和 Stack 的“特点”便是它们均属于“老版本才有”的类。之所以仍然保留，是为了与那些老代码保持“向后兼容”。但另一方面，如果你想写新程序（Java 2 程序），最好就不要再考虑它们！

其他容器的差异通常都可归纳为它们具体是由什么提供“支持”的。换言之，你得考虑在物理意义上用来实现一个接口的数据结构是什么。例如，ArrayList 才 LinkedList 都实现了 List 接口，所以无论选用哪一个，程序都会产生相同的结果。不过，为 ArrayList 提供“支持”的肯定是一个数组；而 LinkedList 是采用与“双重链接列表”一样的标准方式来实现的——因此对于每个单独的对象来说，它们除了包含数据之外，还包含了指向列表内上一个及下一个元素的引用。正是由于这个原因，如果想在一个列表中部进行大量插入及删除操作，那么 LinkedList 无疑是最恰当的选择（LinkedList 还有一些额外的功能，专门由 AbstractSequentialList 提供）。若非如此，就情愿选择 ArrayList，它的速度通常会快一些。

这里还有另一个例子：Set 既可作为一个 TreeSet 实现，亦可作为 HashSet 实现。为 TreeSet 提供支持的是一个 TreeMap，设计用于产生连贯排序的数据集。然而，一旦需要在自己的 Set 中容纳大量元素，TreeSet 的性能就会大打折扣（因为不断的排序会消耗大量时间）。写一个需要 Set 的程序时，最开始无论如何都应该选择 HashSet；只有在你真正需要一个不断排序的数据集时，才应考虑转换成 TreeSet。

9.12.1 挑选不同的 List

为体会各种 List 实现间的差异，最简便的方法就是进行一次性能测验。下述代码的作用是建立一个内部基类，将其作为一个“测试床”使用。然后为每一次测验都创建一个匿名内部类。每个这样的内部类都由一个 test()方法调用。利用这种方法，可以方便添加和删除测试项目。

```
//: c09:ListPerformance.java
// Demonstrates performance differences in Lists.
import java.util.*;
import com.bruceeckel.util.*;

public class ListPerformance {
  private abstract static class Tester {
    String name;
    int size; // Test quantity
    Tester(String name, int size) {
      this.name = name;
      this.size = size;
    }
    abstract void test(List a, int reps);
  }
  private static Tester[] tests = {
    new Tester("get", 300) {
      void test(List a, int reps) {
        for(int i = 0; i < reps; i++) {
          for(int j = 0; j < a.size(); j++)
            a.get(j);
        }
      }
    },
    new Tester("iteration", 300) {
      void test(List a, int reps) {
        for(int i = 0; i < reps; i++) {
          Iterator it = a.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
    new Tester("insert", 5000) {
      void test(List a, int reps) {
        int half = a.size()/2;
        String s = "test";
```

```
      ListIterator it = a.listIterator(half);
      for(int i = 0; i < size * 10; i++)
        it.add(s);
    }
  },
  new Tester("remove", 5000) {
    void test(List a, int reps) {
      ListIterator it = a.listIterator(3);
      while(it.hasNext()) {
        it.next();
        it.remove();
      }
    }
  },
};
public static void test(List a, int reps) {
  // A trick to print out the class name:
  System.out.println("Testing " +
    a.getClass().getName());
  for(int i = 0; i < tests.length; i++) {
    Collections2.fill(a,
      Collections2.countries.reset(),
      tests[i].size);
    System.out.print(tests[i].name);
    long t1 = System.currentTimeMillis();
    tests[i].test(a, reps);
    long t2 = System.currentTimeMillis();
    System.out.println(": " + (t2 - t1));
  }
}
public static void testArray(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[tests[i].size];
    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, reps);
    long t2 = System.currentTimeMillis();
    System.out.println(": " + (t2 - t1));
  }
```

```
  }
  public static void main(String[] args) {
    int reps = 50000;
    // Or, choose the number of repetitions
    // via the command line:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    System.out.println(reps + " repetitions");
    testArray(reps);
    test(new ArrayList(), reps);
    test(new LinkedList(), reps);
    test(new Vector(), reps);
  }
} ///:~
```

内部类 Tester 是一个抽象类，用于为特定的测试提供一个基类。它包含了一个要在测试开始时打印的字串、一个用于计算测试次数或元素数量的 size 参数、用于初始化字段的一个构造函数以及一个抽象方法 test()。test()做的是最实际的测试工作。各种类型的测试都集中到一个地方，即 tests 数组里。我们用从 Tester 继承的不同匿名内部类来初始化该数组。为了添加或删除一个测试项目，只需在数组里简单地添加或移去一个内部类定义即可，其他所有工作都是自动进行的。

为了对比数组和容器的访问速度（主要是数组与 ArrayList 的比较），我们为数组准备了一次特殊的测试——方法是用 Arrays.asList()将数组作为一个 List 封装起来。注意在这种情况下，只能头两个测试才会执行，因为我们没法子在一个数组里增删元素。

首先用元素填充传递给 test()的 List，然后为 tests 数组中的测试计时。注意每次测试用的机器不同，结果当然也会不同。这个程序的宗旨只是揭示出不同容器类型的“相对”性能比较。下面是其中一次运行得到的结果：

类 型	获 取	迭 代	插 入	删 除
数组	1430	3850	未进行	未进行
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

和我们预期的一样，在进行随机访问和迭代时，数组的速度比任何容器都要快。进行随机访问时（get()），ArrayList 显得较为“轻松”，而 LinkedList 却显得非常“费劲”（令人奇怪的是，LinkedList 中进行“迭代”操作的速度居然快于 ArrayList）。但另一方面，假如要在列表中部进行大量插入和删除操作，用 LinkedList 却显得比用 ArrayList 划算得多（特别是删除操作，LinkedList 所花的时间只有极短的 60!）。Vector 普遍没有 ArrayList 快，所以应尽量避免不用；它在库里之所以还占有“一席之地”，完全是考虑到“向后兼容”的目的（它在这个程序中还能工作，完全是由于我们把它改成了 Java 2 的 List）。最好的方法或许是先挑选一个 ArrayList 作为自己的默认方案。以后若发现由于大量的插入和删除造成了性能的降低，再考虑换成 LinkedList 不迟。另外理所当然地，假如要处理的是一组数量固定的

元素，那么肯定应该用数组！

9.12.2 挑选不同的 Set

可在 TreeSet 和 HashSet 间作出选择，具体应该由 Set 的大小决定（如果要通过一个 Set 获得一个顺序列表，请务必用 TreeSet）。下面这个测试程序将有助于大家理解自己的选择：

```
//: c09:SetPerformance.java
import java.util.*;
import com.bruceeckel.util.*;

public class SetPerformance {
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Set s, int size, int reps);
  }
  private static Tester[] tests = {
    new Tester("add") {
      void test(Set s, int size, int reps) {
        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, int reps) {
        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, int reps) {
        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, int reps) {
    System.out.println("Testing " +
      s.getClass().getName() + " 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, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // Or, choose the number of repetitions
    // via the command line:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    // Small:
    test(new TreeSet(), 10, reps);
    test(new HashSet(), 10, reps);
    // Medium:
    test(new TreeSet(), 100, reps);
    test(new HashSet(), 100, reps);
    // Large:
    test(new TreeSet(), 1000, reps);
    test(new HashSet(), 1000, reps);
  }
} ///:~
```

下表总结了某一次运行的结果（由于计算机和 JVM 的不同，结果当然有所差异；你应该亲自执行一次测试）：

类 型	测试大小	添 加	包 含	迭 代
TreeSet	10	138.0	115.0	187.0
	100	189.5	151.1	206.5
	1000	150.6	177.4	40.04
HashSet	10	55.0	82.0	192.0
	100	45.6	90.0	202.2
	1000	36.14	106.5	39.39

不管什么操作，HashSet 的性能通常都要优于 TreeSet（只是在特殊的增添和检索操作中，

才会发生少许例外）。那么，为什么还要保留 TreeSet 呢？唯一的理由就是它能保持元素总处在排好序的状态。因此，只有在你需要一个排好序的 Set 时，才应考虑使用 TreeSet。

9.12.3 挑选不同的 Map

选择不同的 Map 实现时，注意 Map 的大小对于性能的影响是最大的，下面这个测试程序清楚地阐明了这一点：

```
//: c09:MapPerformance.java
// Demonstrates performance differences in Maps.
import java.util.*;
import com.bruceeckel.util.*;

public class MapPerformance {
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Map m, int size, int reps);
  }
  private static Tester[] tests = {
    new Tester("put") {
      void test(Map m, int size, int reps) {
        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, int reps) {
        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, int reps) {
        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, int reps) {
    System.out.println("Testing " +
      m.getClass().getName() + " 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, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // Or, choose the number of repetitions
    // via the command line:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    // Small:
    test(new TreeMap(), 10, reps);
    test(new HashMap(), 10, reps);
    test(new Hashtable(), 10, reps);
    // Medium:
    test(new TreeMap(), 100, reps);
    test(new HashMap(), 100, reps);
    test(new Hashtable(), 100, reps);
    // Large:
    test(new TreeMap(), 1000, reps);
    test(new HashMap(), 1000, reps);
    test(new Hashtable(), 1000, reps);
  }
} ///:~
```

由于 Map 的大小最重要，所以计时测试会用时间除以大小，以得到令人信服的结果。下面给出一次测试的结果（在你的机器上可能不同）：

类 型	测试大小	置 入	取 出	迭 代
TreeMap	10	143.0	110.0	186.0
	100	201.1	188.4	280.1

类 型	测试大小	置 入	取 出	迭 代
	1000	222.8	205.2	40.7
HashMap	10	66.0	83.0	197.0
	100	80.7	135.7	278.5
	1000	48.2	105.7	41.4
Hashtable	10	61.0	93.0	302.0
	100	90.6	143.3	329.0
	1000	54.1	110.95	47.3

正象期望的那样，Hashtable 的性能大致与 HashMap 相当（HashMap 的速度通常会快一点儿，它其实就是为了用来代替 Hashtable 而设计的）。TreeMap 的速度通常要比 HashMap 慢，之所以还要用到它，是由于我们可以不把它当作一个 Map 使用，而是利用它创建一个排好序的列表。对 TreeMap 来说，它无论如何都是排好序的，所以不必专门为它排序。一旦为 TreeMap 填好内容，就可调用 keySet()，获得那些键的一个“Set 景象”；再调用 toArray()，产生由那些键构成的一个数组。随后，可利用 Arrays.binarySearch()这个静态方法（稍后还会详细介绍），在排好序的数组中快速检索需要的对象。当然，只有当一个 HashMap 的行为由于某种原因，变得令人无法接受的前提下，才应该考虑这样做，因为 HashMap 本来的设计宗旨便是让你快速、方便地找东西。另外，只需再创建一个对象，就可在一个 TreeMap 的基础上，方便地创建出一个 HashMap。总之，假如你要在自己的程序中使用 Map，那么首先考虑的就应该是 HashMap——只有在需要一个不断自动排序的 Map 时，才应考虑换成 TreeMap！

9.13 List 的排序和搜索

用来对 List 进行排序和搜索的工具拥有和对象数组排序工具一样的名字和签名，只是换成了由 Collections 提供的静态方法，而不是由 Arrays 提供的。下面是一个例子，在 ArraySearching.java 的基础上修改而来：

```
//: c09: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));
  }
} ///:~
```

这些方法的用法和在 Arrays 中的完全一致，不过我们不再用一个数组，而是用一个 List。就象数组的搜索和排序，假如用一个 Comparator 排序，那么必须用相同的 Comparator 进行 binarySearch()。

该程序也演示了由 Collections 提供的 shuffle()方法，它的作用是随机性地打乱一个 List 的顺序。

9.14 工 具 简 介

Collections 类还含有另一些实用的工具：

enumeration(Collection)	为参数产生旧式风格的 Enumeration（枚举）
max(Collection) min(Collection)	使用 Collection 中对象的自然比较方法，产生参数中的最大或最小元素
max(Collection,Comparator) min(Collection,Comparator)	利用 Comparator，产生 Collection 中的最大或最小元素
reverse()	逆向排列所有元素
copy(List dest, List src)	将元素从 src 复制到 dest
fill(List list, Object o)	将 list 中的所有元素替换成 o
nCopies(int n, Object o)	返回长度为 n 的一个不可变列表，它的所有引用均指向 o

注意 min()和 max()只适用于 Collection（集合）对象，不可用于 List（列表），所以你不必担心自己是否应该对一个集合进行排序（但就象我们早先指出的那样，在执行一次 binarySearch()搜索之前，必须先将一个 List 或者数组排好序）。

9.14.1 使集合或 Map 不可修改

通常，创建 Collection 或 Map 的一个“只读”版本显得更有利一些。Collections 类允许我们达到这个目标，方法是将原始容器传递到一个方法里，然后命令它传回一个只读版本。这个方法共有四种变化形式，分别用于 Collection（如果不想更特别地对待一个 Collection）、List、Set 以及 Map。下面这个例子演示了为它们分别构建只读版本的正确方法：

```
//: c09:ReadOnly.java
// Using the Collections.unmodifiable methods.
import java.util.*;
import com.bruceeckel.util.*;

public class ReadOnly {
  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 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 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 OK
    //! m.put("Ralph", "Howdy!");
  }
} ///:~
```

对于每种情况，在将其正式变为只读以前，都必须用有有效的数据填充容器。一旦载入成功，最佳的做法就是用“不可修改”调用产生的引用替换掉原来的引用。这样一来，把它变成“不可修改”之后，就可有效地避免由于操作不慎，而错误地更改了其中的内容。另一方面，该工具也允许我们在一个类中将那些“可以修改”的容器保持为 private 状态，并可从一个方法调用中返回指向那个容器的一个只读引用。这样一来，尽管我们可在类的内部修改它，但其他任何人都只能读。

为特定类型调用“不可修改”的方法不会造成编译时间的类型检查，但只要完成了变化，以后一旦有个方法试图修改特定容器的内容，就会产生一个 UnsupportedOperationException（不支持的操作）违例。

9.14.2 使一个集合或 Map 同步

“synchronized”（已同步）关键字是“多线程”机制一个非常重要的部分。我们到第 14 章才会对这一机制作深入探讨。在这儿，大家只需注意到 Collections 类提供了对整个容器进行自动同步的一种途径。它的语法与“不可修改”的方法是类似的：

```
//: c09: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());
  }
} ///:~
```

在这种情况下，我们通过适当的“已同步”方法，立即传递新容器；这样一来，便不会由于不慎，而暴露出那些未同步的版本。

“立即失败”

Java 容器也提供了防止多个进程同时修改一个容器内容的手段。假如你正在一个容器里迭代，但此时有另一些进程介入，也在那个容器里插入、删除或者修改一个对象，便有可能发生冲突——我们可能已传递了那个对象；可能正要对它进行处理；容器可能在调用了 size() 后发生了收缩——有许多类似这样的危险存在！针对这个问题，新的容器库集成了一套解决机制，名为“立即失败”（Fail Fast）。它能侦测到容器里发生的、并非由自己的进程造成的改变。一旦它侦测到还有其他人试图修改容器，就会立即产生一个 ConcurrentModificationException（并发修改违例）。这正是“立即失败”一词的来历——它

并不用更复杂的算法在“以后”侦测问题，而是“立即”产生违例。

要见识它的效果并不难——只需创建一个迭代器，然后在迭代器指向的那个集合里聚合添加一点儿东西就可以了。如下例所示：

```
//: c09:FailFast.java
// Demonstrates the "fail fast" behavior.
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();
  }
} ///:~
```

当然会产生违例！因为在从容器获得迭代器之后，“居然”又在容器里放了点儿什么东西。这意味着同一个程序的两个部分可能正在修改同一个容器，从而造成了一个非常危险的、不稳定的状态。因此，这个违例会“果断”地站出来，要你马上修改代码！就目前来说，为了解决上例的问题，需要在所有元素都添加进容器之后，再去获得迭代器。

注意在用 get()访问一个 List 的元素时，并不会进行这样的监视，当然也不会有什么“违例”产生。

9.15 不支持的操作

使用 Arrays.asList()方法，我们可将一个数组转变成 List：

```
//: c09:Unsupported.java
// Sometimes methods defined in the
// Collection interfaces don't work!
import java.util.*;

public class Unsupported {
  private static String[] s = {
    "one", "two", "three", "four", "five",
    "six", "seven", "eight", "nine", "ten",
  };
  static List a = Arrays.asList(s);
  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(" + s[0] + ") = " +
      a.contains(s[0]));
    System.out.println(
      "a.containsAll(a2) = " +
      a.containsAll(a2));
    System.out.println("a.isEmpty() = " +
      a.isEmpty());
    System.out.println(
      "a.indexOf(" + s[5] + ") = " +
      a.indexOf(s[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(s[0]); // Unsupported
    a.removeAll(a2); // Unsupported
  }
} ///:~
```

从中可以看出，实际上只实现了 Collection 和 List 的一部分接口。其他那些方法（接口）会导致一种颇不受欢迎的情况——UnsupportedOperationException（不支持的操作违例）。在下一章里，我们会讲述违例的详细情况，但在这里有必要进行一下简单说明。这里的关键在于 Collection 接口，以及 Java 容器库的另一些接口，它们都包含了“可选的”方法。在实现那个接口的具体类中，那些方法也许并没得到支持。而假如调用一个未获支持的方法，当然会导致一个 UnsupportedOperationException，它指出这里出现了一个编程错误。

大家或许会觉得奇怪，不是说“接口”和基类最大的“卖点”就是它们许诺这些方法能产生一些有意义的行为吗？上述违例破坏了那个许诺——它调用的一部分方法不仅不能产生有意义的行为，而且还会中止程序的运行。在这个时候，类型的所谓安全保证似乎变得一钱不值！

但是，情况并没有想象的那么坏。对 Collection、List、Set 或者 Map 来说，编译器仍然限制我们只能调用那个接口中的方法，所以它和 Smalltalk 还是存在一些区别的（在 Smalltalk 中，可为任何对象调用任何方法，而且只有在运行程序时才知道这些调用是否可行）。除此

以外，用Collection作为参数的大多数方法也只能从那个Collection中读取数据——Collection的所有“read”（读）方法都不是可选的。

这样一来，系统就可避免在设计时出现接口的冲突。而在其他容器库的设计方案中，最终常常得到数量过多的接口，要用它们描述基本方案的每一种变化形式，所以学习和掌握显得非常困难。有些时候，甚至难于捕捉接口中的所有特殊情况，因为人们可能设计出任何新接口。但 Java 的“不支持的操作”方法却达到了新容器库的一个重要设计目标：易于学习和使用。但是，为了使这一方法真正有效，却需满足下述条件：

(1) UnsupportedOperationException 必须属于一种“非常”事件。也就是说，对于大多数类来说，所有操作都应是可行的。只有在一些特殊情况下，一、两个操作才可能未获支持。新容器库满足了这一条件，因为平常经常使用的类——ArrayList、LinkedList、HashList 和 HashMap，以及其他具体实现——都提供了对所有操作的支持。但是，假如你想新建一个 Collection，但又不想为容器接口中的每个方法都提供有意义的定义，而且仍然让它与原来的库配合，那么这种设计方法也确实提供了一个“后门”可供利用。

(2) 若一个操作未获支持，那么 UnsupportedOperationException（未支持的操作违例）极有可能在实现期间出现，而不是在产品交付给客户以后才会出现。它毕竟指出的是一个编程错误——不正确地使用了一个类。这一点不能十分确定，通过也可以看出这种方案的“试验”特征——只有经过多次试验，才能找出最理想的工作方式。

在上面的例子中，Arrays.toList()产生了一个 List（列表），为该列表提供支持的是一个固定长度的数组。因此，唯一应该支持的就是那些不改变数组长度的操作。而另一方面，假如你要固执地用一个新接口来表达这种“另类”行为（可能叫作“FixedSizeList”——固定长度列表），那么事情只有被弄糟的可能，你会面临一个无比复杂的局面；而且以后开始使用库的时候，很快就会发现自己不知从何处下手。

对那些采用了 Collection、List、Set 或者 Map 作为参数使用的方法，在它们的配套文档中，应向用户指出哪些可选的方法是必须实现的。举个例子来说，排序要求实现 set()和 Iterator.set()方法，但不需要 add()和 remove()。

9.16 Java 1.0/1.1 容器

前面学了这么多新东西，但很不幸，以往写的大量代码用的都是 Java 1.0/1.1 的容器，甚至有些新代码也偶尔在使用那些类。所以，尽管在写新程序的时候，大家应尽量避免使用老容器，但仍需做到对它们心中有数。不过令人高兴的是，老容器的数量并不多，所以和它们有关的介绍也不会有很多（即便换作从前，我也会尽量避免过多强调一些并不十分出众的设计决定）。

9.16.1 Vector 和 Enumeration

在 Java 1.0/1.1 中，能自行扩展的唯一一种序列便是 Vector（矢量），所以它得到了广泛运用。不过，它的缺点也有不少，这里恕不一一说明（详见本书的第一版，配套光盘和 www.BruceEckel.com 均有此书电子版）。从根本上讲，你可把它想象成一个 ArrayList，只是采用了冗长的、难记的方法名。在 Java 2 容器库中，Vector 已被进行了改编，使其能变作 Collection 和 List 使用。这正是在下面的例子中，Collections2.fill()方法能正常使用的原因。不过，也正是由于这一点，才导致许多人得到了错误的结论，他们认为 Java 2 已将 Vector 改造得更加好用——实际情况是，之所以还允许 Vector 呆在那儿，仅仅是为了支持 Java 2 之前的那些老代码！

“迭代器”（Iterator）的 Java 1.0/1.1 版本发明了一个新词儿来称呼自己：Enumeration；

在本书第一版中，我们把它译作“枚举”。不管怎么说，现在人们的认识已经统一，它的正式名称还是“迭代器”，可不是什么发明创造的“枚举器”、“列举器”等等。事实上，老版本的 Enumeration 接口要比现在的 Iterator 接口小，只有两个方法，而且用的是比较冗长的方法名。这两个方法包括：boolean hasMoreElements()，在一次“枚举”还有更多元素的前提下，它会返回 true；以及 ObjectnextElement()，如果还有更多的元素，它就返回这一次“枚举”中的下一个元素（否则会产生一个违例）。

Enumeration 仅仅是一个接口，而不是一种实现方式，就连一些新库有时也在用老的 Enumeration——尽管并不鼓励这样做，但通常也无伤大雅。即便你在自己的代码中一直都坚持用 Iterator，但在使用别人的库时，也得时刻准备好有人扔给你一个 Enumeration。

另外，使用 Collections.enumeration()这个方法，我们可为任何 Collection 都产生一个 Enumeration。如下例所示：

```
//: c09:Enumerations.java
// Java 1.0/1.1 Vector and Enumeration.
import java.util.*;
import com.bruceeckel.util.*;

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 只有一个 addElement()方法，但 fill()用了 add()方法，因为 Vector 已被变成一个 List。为了产生一个 Enumeration，需要调用 elements()，然后就可用它执行一次向前“迭代”。

最后一行创建了一个 ArrayList，它用 enumeration()将 ArrayList Iterator 改编成一个 Enumeration。然后，如果有一些老代码希望用到 Enumeration，那么仍然可以使用新容器。

9.16.2 Hashtable

就象大家在本章前面的“性能测试”中看到的那样，基本的 Hashtable（散列表）同新的 HashMap 非常类似——甚至包括它们的方法名！不过在你新设计的程序中，没有任何理由需要弃 HashMap 不用，而换用老的（已经过时的）Hashtable。

9.16.3 Stack

“堆栈”（Stack）的概念早先已在讲述 LinkedList 的时候介绍过了。对于 Java 1.0/1.1 的 Stack 来说，它最让人觉得奇怪的一项设计是：并不是用一个 Vector 作为基本构建单元使用，

而是从 Vector 那里继承得到一个 Stack。这样一来，除具有 Vector 的特征和行为之外，还具有一些额外的 Stack 行为。不过，时过境迁，我们现在也很难揣测设计者当初是怎么想的。到底是有目的而为的一个思路呢？还是纯属一项试验性的设计？

这儿简单地演示了 Stack，我们在一个堆栈里“压”入来自某个 String 数组的每一行：

```
//: c09:Stacks.java
// Demonstration of Stack Class.
import java.util.*;

public class Stacks {
  static String[] months = {
    "January", "February", "March", "April",
    "May", "June", "July", "August", "September",
    "October", "November", "December" };
  public static void main(String[] args) {
    Stack stk = new Stack();
    for(int i = 0; i < months.length; i++)
      stk.push(months[i] + " ");
    System.out.println("stk = " + stk);
    // Treating a stack as a Vector:
    stk.addElement("The last line");
    System.out.println(
      "element 5 = " + stk.elementAt(5));
    System.out.println("popping elements:");
    while(!stk.empty())
      System.out.println(stk.pop());
  }
} ///:~
```

months 数组中的每一行都通过 push()方法插入 Stack，以后则用 pop()从堆栈的顶部开始依次取出。为了能说明问题，同时还对 Stack 对象执行 Vector 操作。之所以能这样做，是由于继承的关系，Stack 同时也是一个 Vector！这样一来，可对 Vector 采取的操作也可对一个 Stack 执行，比如 elementAt()。

就象前面说的那样，如果想获得堆栈的行为，请换用 LinkedList。

9.16.4 BitSet

如果想高效率地保存大量“开－关”形式的信息（标志），就可考虑使用 BitSet（位集）。不过，它只有从“省空间”的角度看才有意义；如果你想获得高效率的访问，那么很遗憾地告诉你，它的速度要比使用一些固有类型的数组慢一些。

此外，BitSet 的最小长度是一个长整数（Long）的长度：64 位。这意味着假如你想准备保存比这更小的数据，比如 8 位数据，那么 BitSet 就显得浪费了。此时更好的做法是创建自己的类，或者干脆创建一个数组，用它来保存自己的开关标志。

对一个标准的容器来说，随我们加入越来越多的元素，容器也会自我膨胀，BitSet 也不例外。下面这个例子展示了 BitSet 的运用：

```
//: c09:Bits.java
// Demonstration of BitSet.
import java.util.*;

public class Bits {
  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);
  }
} ///:~
```

随机数字生成器用于创建一个随机的 byte、short 和 int 值。每个值都会转换成 BitSet 内相应的位范式。此时一切都很正常，因为 BitSet 是 64 位的，所以它们都不会造成最终尺寸的增大。随后，我们创建了一个 512 位的 BitSet。构造函数分配的存储空间是位数的两倍。但是，你仍然可以设置 1024 或者更高的位。

9.17 总　　结

下面复习一下由标准 Java 库提供的容器：

(1) 数组（Array）包含了对象的数字化索引。它容纳的是一种已知类型的对象，所以在查找一个对象时，不必对结果进行强制转型处理。数组可以是多维的，而且能够容纳原始数据类型。但是，一旦把它创建好以后，大小便不能改变了。

(2) 集合（Collection）容纳的是各自独立的元素；而映射（Map）容纳了互相对应的“键一值”对。

(3) 列表（List）和数组类似，也包含了对象的数字化索引——完全可以将数组和 List 都想象成排好序的容器。在你添加更多的元素时，List 会自动改变自己的大小。但是，在 List 里只能容纳“对象引用”，不能将原始数据类型放到其中。另外，从容器里取出一个对象引用时，无论如何都需要对结果进行强制转型。

(4) 数组列表（ArrayList）使频繁的随机访问变得更加轻松；而链接列表（LinkedList）在你需要于一个列表的中部进行大量插入和删除操作时，可提供最优的性能。

(5) 队列、双头队列（Deque）和堆栈的行为通过 LinkedList 来实现。

(6) 映射（Map）不仅能将数字和数字联系到一起，也能将对象与对象对应到一块儿。HashMap 强调的是快速访问，而 TreeMap 将键总保持在排好序的状态，所以速度比不上 HashMap。

(7) 对 Set 来说，它只能接受一种对象类型。HashSet 具有最快的检索速度，而 TreeSet 用于将元素保持在排好序的状态。

(8) 在新设计的程序中，Vector、Hashtable 和 Stack 这几种“过时”的类已没有使用的必要。

在我们正式用 Java 写程序的时候，每天都要同“容器”这种工具打上许多交道。善加利用，可使自己的程序更简单、功能更强、更有效率。

9.18 练 习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 创建一个由 double 值构成的数组，用 RandDoubleGenerator 来填充（fill()）它。

(2) 创建一个名为 Gerbil（沙鼠）的新类，在构造函数中初始化一个 int gerbilNumber（类似本章早些时候的 Mouse 例子）。为它设计一个名为 hop()的方法，它应该能打印当前的 gerbilNumber 是哪个，以及它要跳到哪个编号。再创建一个 ArrayList，在 List 里加入一系列 Gerbil 对象。接下来，用 get()方法在 List 中移动，并为每个 Gerbil 都调用 hop()。

(3) 修改练习(2)，用一个 Iterator（迭代器）在 List 中移动，仍然要调用 hop()。

(4) 取出练习(2)的 Gerbil 类，把它改放到一个 Map 里，针对放到映射表里的每个 Gerbil（值），都以字串的形式，为其匹配上 Gerbil 的名字。为 keySet()生成一个迭代器，用它在 Map 里巡视，找出与每个键对应的 Gerbil。打印出键，然后让沙鼠“跳跃”（hop()）。

(5) 创建一个 List（ArrayList 和 LinkedList 都请试试），用 Collections2.countries 来填充它。对列表进行排序，打印出它，然后不断地对列表应用 Collections2.shuffle()，每次都把结果打印下来，体验列表每一次的顺序都不同。

(6) 证明在 MouseList 里，除了一个 Mouse 之外，其他无论什么都不能加入。

(7) 修改 MouseList.java，将其变为从 ArrayList 继承，而不是采用合成方式。请观察并说明这种做法存在的问题。

(8) 修正 CatsAndDogs.java，创建一个 Cats 容器（用 ArrayList），令其只接收和获取 Cat 对象。

(9) 创建一个容器，在其中封装一个 String 数组，要求它只能添加和获取字串，从而保证以后使用时，不至于出现没有强制转型的问题。如内部数组的大小不够，无法为下一次的添加操作提供支持，你的容器应该能自动更改自己的大小。在 main()中，请对比你的容器和一个装有字串的 ArrayList 的性能。

(10) 重复练习(9)，这一次改为使用 int 容器，将它的性能同装有 Integer（整数）对象的一个 ArrayList 进行对比。注意在对比性能时，让容器里每一个对象的值都递增。

(11) 利用由 com.bruceeckel.util 提供的工具，创建一个由原始数据类型构成的数组，再创建一个字串数组。然后，用恰当的生成器填充每一个数组，同时用恰当的 print()方法，打印出每一个数组。

(12) 创建一个生成器，它能根据你喜爱的电影检索到演员名字。而且一旦查遍了所有名字，就会自动返回开头。请用由 com.bruceeckel.util 提供的工具来填充一个数组、一个 ArrayList、一个 LinkedList 以及两种类型的 Set，然后打印出每一个容器。

(13) 创建一个包含了两个字串对象的类，将这个类设为 Comparable（可比较的），以便只对第一个字串进行比较。使用 geography 生成器，用这个类的对象来填充一个数组一个 ArrayList。请证实排序工作正确无误地进行。接下来，让一个 Comparator（比较器）只对第二个字串进行比较，并证实排序正确进行。另外，利用自己的 Comparator，执行一次“二进制搜索”（binarySearch()）。

(14) 修改练习(13)，按“字典顺序”排序。

(15) 用 Arrays2.RandStringGenerator 填充一个 TreeSet，但是注意用“字典顺序”进行排

序。打印出这个 TreeSet，核实排序是否正确。

(16) 同时创建一个 ArrayList 和 LinkedList，均用 Collections2.capitals 生成器进行填充。用一个标准 Iterator（迭代器）打印出每个列表，然后用一个 ListIterator（列表迭代器）将一个列表插入另一个——注意每隔一个位置插入一个。接下来，从第一个列表的末尾开始插入，并逐渐向后推移。

(17) 写一个方法，要求用一个迭代器在一个集合中“步进”，并打印出容器中每个对象的 hashCode()。为各种不同类型的集合都填充对象，然后为每个容器都应用自己的方法。

(18) 改正 InfiniteRecursion.java（无限循环）中存在的问题。

(19) 创建一个类，生成由该类对象构成的、已初始化好的一个数组。将数组中的内容填充到一个 List 里。用 subList()创建该 List 的一个子集，再用 removeAll()从 List 中删除那个子集。

(20) 修改第 7 章的练习(6)，用一个 ArrayList 来容纳 Rodent（啮齿动物）；并用一个迭代器在 Rodent 序列中移动。记住 ArrayList 容纳的只能是对象，所以在你访问单个 Rodent 时，必须进行强制转型。

(21) 参照 Queue.java 例子，创建一个 Deque（双头队列）类，并对其进行测试。

(22) 在 Statistics.java 中使用一个 TreeMap。添加相应的代码，测试 HashMap 和 TreeMap 的性能差异。

(23) 生成一个 Map 和一个 Set，在其中包括以“A”起头的所有国家。

(24) 利用 Collections2.countries，用相同的数据多次填写一个 Set，并证实 Set 最后只包括一个实例。对两种类型的 Set 都进行这样的测试。

(25) 以 Statistics.java 为基础创建一个程序，重复运行测试，检查结果中是否有一个数字的出现频率比其他高。

(26) 用一个由 Counter 对象构成的 HashSet 改写 Statistics.java（必须修改 Counter，使其能在 HashSet 中使用）。哪种方法更好？

(27) 修改练习(13)的那个类，使其能配合 HashSet 使用，并可作为 HashMap 中的一个键使用。

(28) 借鉴 SlowMap.java，创建一个 SlowSet。

(29) 将 Map1.java 中的测试应用到 SlowMap，证实这样做是否可行。修改 SlowMap 里任何不能正确工作的地方。

(30) 为 SlowMap 实现 Map 接口的剩余部分。

(31) 修改 MapPerformance.java，在其中包括 SlowMap 的测试。

(32) 修改 SlowMap，不是用两个 ArrayList，而是只在其中容纳由 MPair 对象构成的一个 ArrayList。请查实修改过的版本能正常工作。利用 MapPerformance.java，对新 Map 的速度进行测试。接下来，修改 put()方法，以便在每个“键一值”对（MPair）输入后，都能执行一次 sort()（排序）。然后修改 get()方法，通过 Collections.binarySearch()查找到键。请对比新老两个版本的性能。

(33) 在 CountedString 里添加一个 char 字段，也在构造函数中进行初始化。然后修改 hashCode()和 equals()方法，把这个 char 的值包括到其中。

(34) 修改 SimpleHashMap，使其能报告冲突。将同样的数据重复添加两遍，测试是否能正常工作。

(35) 修改 SimpleHashMap，使其能在冲突发生的时候，报告需要进行的“探查”次数。也就是说，在迭代器上要进行多少次 next()操作，才能在 LinkedList 里找到有问题的那一条？

(36) 为 SimpleHashMap 实现 clear()和 remove()方法。

(37) 为 SimpleHashMap 实现剩余的 Map 接口。

(38) 在 SimpleHashMap 中添加一个 private rehash()方法，用于“重新散列”。一旦负载比超过 0.75，便调用此方法。重散列期间，请先将原来的“桶”数增倍，再计算出比这个数字大的第一个质数，最终确定 HashMap 的新“桶”数。

(39) 参照 SimpleHashMap.java 中的例子，创建并测试一个 SimpleHashSet。

(40) 修改 SimpleHashMap，用 ArrayList 替换 LinkedList。修改 MapPerformance.java，对比两种实现的性能。

(41) 参照 JDK 的 HTML 文档（JDK 可从 java.sun.com 下载），注意其中对 HashMap 类的说明。创建一个 HashMap，在其中填充一些元素，并判断负载比。测试这个 Map 的搜索速度。然后，我们要试着提高一下这个速度，方法是创建一个新 HashMap，但为其指定更大的初始空量，将老 Map 复制到新 Map 里，再次执行搜索速度测试。

(42) 在第 8 章，找到那个 GreenhouseControls.java（温室控制）例子，它应该由三个文件构成。在 Controller.java 中，EventSet 只是一个容器。请修改代码，把 EventSet 换成一个 LinkedSet。不过要注意的是，并不能简单地用 LinkedList 替换 EventSet 了事，你还需要用一个 Iterator（迭代器）在事件集中“巡视”（遍历）。

(43)（选做题）自己写一个 HashMap 类，要求为一个特定的键类型加以定制（这儿假定为 String 类型）。不要从 Map 继承它。相反，请直接复制方法，使 put()和 get()方法都只能接收 String 对象作为自己的键使用，而不是接收默认的 Object。牵涉到键的所有东西都不能采用通用类型，而是必须为 String，以避免由于上下强制转型而带来的多余开销。你的目标是设计出一套速度尽可能最快的方案。请改一下 MapPerformance.java，用它对比这个方案与 HashMap 方案的性能差异。

(44)（选做题）在 Java 源码库里找到 List 的源码。复制这些代码，自行创建它一个特别版本，名为 intList，专门用来容纳 int 值。请思考假如要制作一个特别版本的 List，让它适用于所有原始数据类型，那么自己需要做哪些工作？接下来，思考假如要制作一个 LinkedList 类，让它适用于所有原始数据类型，那么需要做哪些工作？事实上，假如在 Java 里实现了“参数化类型”，那么所有这些工作都可以自动完成，不必我们操心（它还有其他许多优点）！

第 10 章 违例差错控制

Java 非常基本的一个想法就是：“形式错误的代码不会真正运行起来”。

捕获错误最理想的当然是在编译时间，最好是在试图运行程序以前。然而，并非所有错误都能在编译时间侦测到。有些问题只有在运行时间才会显露出来。为了解决那些问题，需要通过一系列标准手段，向错误的接收者传递一些适当的信息，使其知道该如何正确处理遇到的问题。

在 C 和其他早期语言中，可通过几种手段来达到这个目的，而且它们通常都是作为一种“约定”建立起来的，而不是固化成程序设计语言的一部分。在那些语言中，我们通常需要返回一个特殊的值或设置一个标志（位），接收者需要检查这些值或标志，判断具体发生了什么事情。然而，随着时间的流逝，终于发现假若采用这种做法，那些经常使用某个现成库的程序员会渐渐滋生麻痹情绪。他们往往会这样想：“唔，错误也许会在其他人的代码里出现吧，但绝不是在我的代码中！”后果便是他们一般不检查是否出现了错误（有时出错条件也确实令人迷惑，让人不知道该如何检查[47]）。另一方面，若每次调用一个方法时都进行全面、细致的错误检查，那么代码的可读性也可能大幅降低。有过这一系列体验的程序员完全有理由相信：假如真的按这种方式老老实实地控制错误，那么甭想真正创建一个大型的、健壮的、易于维护的系统！

要解决这个问题，唯一出路便是排除错误控制机制中所有“不确定”的因素，强调格式的统一。如果大家都能按手续办事，事情不就简单了吗？在程序设计的世界里，我们完全有能力做到这一点！事实上，这个思路并不是今天才问世的，它已有很长的历史。早在 60 年代，便在操作系统里采用了“违例控制”手段；我们甚至可以追溯到 BASIC 语言的“on error goto”语句。不过，C++的违例控制建立在 Ada 的基础上，而 Java 又主要建立在 C++的基础上（尽管它看起来更象 Object Pascal）。

“违例”（Exception）这个词表达的是一种“例外”情况，亦即正常情况之外的一种“异常”。问题发生时，我们也许不知道具体该如何解决，但可以肯定的是，此时已不能再不顾一切地做下去。因此，必须坚决地停下来，并要“由某人在某个地方指出该怎么办”。但要想真正解决问题，当时可能并没有足够多的信息。因此，我们需要将其移交给更级的“负责人”，令其作出正确决定（就象一个“分级指挥系统”，不是吗？）

违例机制的另一项好处就是能够简化错误控制代码。我们再也不用检查一个特定的错误，然后在程序的多处地方都对其进行控制。此外，也不需要在方法调用时检查错误（因为“违例”保证会有人能“捉”住它）。我们只需在一个地方，便可处理所有问题，这个地方便是“违例控制模块”或者“违例控制器”（Exception Handler）。这样便能有效减少代码量，并将那些用于实际工作的代码与专门改正错误的代码分开。一般情况下，采用了“违例”之后，同老式的“错误控制”相比，读取、写入以及调试代码都会变得更有条理。

由于违例控制是由 Java 编译器强行实施的，所以毋需深入学习违例控制，便可正确使用本书编写的大量例子。本章向大家介绍了用于正确控制违例所需的代码，以及在某个方法出现问题的时候，该如何生成（掷出）自己的违例。

---

[47] C 程序员研究一下 printf()的返回值，便可对此有所体会。

10.1　基本违例

所谓“违例条件”，是指当出现什么问题的时候，就应中止方法或作用域的继续。显然，应当将“违例条件”同那些普通的问题严格地区分开。在普通问题的情况下，我们当时便已拥有足够多的信息，可在某种程度上解决问题。而在“违例条件”的情况下，却根本无法再继续下去，因为当时没有提供解决问题所需的足够多的信息。此时，我们能做的唯一事情便是离开“当前环境”，将那个问题丢到一个“级别更高的环境”里。这便是我们“掷”出一个违例时，要发生的事情！

一个简单的例子是“除法”。如可能被零除，就有必要进行检查，确保程序不会冒进，在那种情况下坚持执行除法。但被零除也有有意和无意两种情况，需要我们区别对待。在一个特定的方法里，就当时要解决的特定问题来说，假如可以预知分母会变成一个零，那么你也许知道自己该怎么办，可以自行解决问题。但假如在事先根本没料到的情况下，分母突然变成了一个零，你便没办法自行解决，所以必须产生一个违例，不能不顾一切地执行下去。

产生一个违例后，会发生几件事情。首先，按照与创建 Java 对象一样的方法创建违例对象：在内存“堆”里，用 new 来创建。随后，停止当前的执行路径（记住不可沿这条路继续走下去），然后从当前的环境中取得违例对象的引用。此时，违例控制机制会接管一切，它会找到一个恰当的地方，让程序从这个地方继续执行下去。这个地方便是“违例控制器”，它的职责便是从问题中恢复，使程序要么再尝试一遍，要么简单地继续。

作为产生违例的一个简单例子，大家可思考一个名为 t 的对象引用。有些时候，程序可能传递一个尚未初始化的引用。所以在用那个对象引用调用一个方法之前，最好进行一番检查。可将与错误有关的信息发送到一个更大的场景中，方法是创建一个特殊的对象，用它代表我们的信息，并将其“掷”出我们当前的场景之外。这便叫作“产生一个违例”或者“掷出一个违例”。下面是它的大概形式：

```
if(t == null)
  throw new NullPointerException();
```

这样便“掷”出了一个违例。在当前场景中，它使我们能放弃进一步解决该问题的企图。该问题会被转移到更恰当的“某个地方”加以解决——准确地说，转移到不久就会显露出来的“那个地方”。

10.1.1　违例参数

和 Java 的其他任何对象一样，需要用 new 在内存堆里创建违例。new 会分配存储空间，并调用一个构造函数。在所有标准违例中，都有两个构造函数：第一个是默认构造函数，第二个则需采用一个字串参数，使我们能在违例里置入相关信息：

```
if(t == null)
  throw new NullPointerException("t = null");
```

稍后，字串可用各种方法提取出来，就象稍后会展示的那样。

在这儿，关键字 throw 会象变戏法一样做出一系列不可思议的事情。通常，我们首先要用 new 创建一个对象，用它代表“出错条件”。结果引用则传递给 throw。在这里，对象实际上是从方法“返回”的——尽管设计那个方法时，通常并没有要求返回那个对象。其实，你完全可以把违例控制想象成一种“另类”的返回机制——只是千万不要沿这个思路走得太远，否则会遇到麻烦的。“掷”出一个违例后，亦可从原来的作用域中退出。但是会先返回一个值，再退出方法或作用域。

不过，与普通“方法返回”的相似之处到此便全部结束了，因为我们返回的地方与从普

通方法返回的地方是迥然有异的（我们会回到一个适当的违例控制器，它距违例“掷”出的地方可能相当遥远——在调用堆栈中要低上许多级）。

此外，我们可根据需要，选择掷出任何一种类型的“可掷”（Throwable）对象。典型情况下，我们要为每种不同类型的错误都“掷”出一类不同的违例。为了具体地标识出一个错误，相关的信息不仅要在违例对象中表示，也要用你选择的违例对象的类型来表示。只有这样，一个更大场景中的“负责人”才能知道该如何对待我们的特定违例（通常，真正有用的信息只有违例对象的类型，而违例对象中保存的信息是毫无意义的）。

10.2 违例的捕获

若某个方法产生一个违例，必须保证该违例能被捕获，并获得正确处理。对于 Java 的违例控制机制来说，它最大的一项好处就是允许我们在一个地方将精力集中在要解决的问题上，然后在另一个地方应付来自那个代码内部的错误。

为理解违例是如何捕获的，首先必须掌握“警戒区”（Guarded Region）的概念。它代表一个特殊的代码区域，有可能产生某些违例——这个区域的后面则跟上用于控制那些违例的代码。

10.2.1 try 块

假如你在一个方法内部，同时又“掷”出了一个违例（或由该方法内部调用的另一个方法“掷”出了违例），那么在此过程中，那个方法就会退出。假如不想由于“掷”出违例而退出方法，也可在那个方法内设置一个特殊的代码块，用它来捕捉违例。这就是“try 块”的概念，因为我们要在这个地方“尝试”（try）各种方法调用，看看它们是否会出现违例。try 块属于一种普通的作用域，用一个 try 关键字开头：

```
try {
  // 可能产生违例的代码
}
```

另一方面，假如想用一种不支持“违例控制”的程序语言全面检查错误，就必须用设置及错误检测代码将每个方法调用都包围起来——即使你多次调用的都是相同的方法，每一次都要进行这样的“包围”。而在使用了违例控制技术后，就可将所有东西都放到一个 try 块里，在同一地点捕获所有违例。这样可极大简化我们的代码，并使其更易辨读——代码本身要做的事情和错误检查这两大功能被划分得清清楚楚！

10.2.2 违例控制器

当然，“掷”出来的违例必须有一个地方可去，这个地方便是违例控制器。而且针对想捕获的每种违例类型，都必须有一个相应的违例控制器。违例控制器紧接在 try 块后面，而且用 catch（捕获）这个关键字加以标记。如下所示：

```
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 从句——即违例控制器——都类似一个小型方法，它需要采用一个（而且只有一个）特定类型的参数。可在控制器内部使用标识符（id1、id2 等等），就象一个普通的方法参数那样。当然，我们有时也根本不用标识符，因为违例的“类型”已提供了足够多的信息，可供我们有效地处理违例。不过，即使不用，标识符也必须就位，不能省去。

控制器模块必须紧接在 try 块后面。若“掷”出一个违例，违例控制机制会搜寻参数与违例类型相符的第一个控制器。随后，它会进入那个 catch 从句，并认为违例已得到控制。一旦 catch 从句结束，对控制器的搜索也会停止。只有相符的 catch 从句才会得到执行；它与 switch 语句不同，后者在每个 case（条件）后都需要一个 break（中断）命令，防止误执行其他不相干的语句。

在 try 块内部，请注意多个不同的方法调用可能会产生同一个违例，但我们只需要一个控制器。

中断与恢复

在违例控制理论中，共有两种基本模型。在“中断”模型中（Java 和 C++提供了对这一模型的支持），我们假定一个错误非常关键，没办法再返回发生违例的地方。无论谁只要“掷”出了这样的一个违例，就表明没法子补救错误，而且也不希望再回去。

另一种模型叫作“恢复”。它意味着违例控制器有责任来纠正当前的状况。纠正之后，再重新试一遍，并假定第二次会成功执行（恢复），不会再有违例。如果使用恢复，便意味着在违例得到控制以后，我们仍然想继续执行。在这种情况下，我们的违例更象一个方法调用——我们用它在 Java 中设置各种各样特殊的环境，产生类似于“恢复”的行为（换言之，此时并非简单地“掷”出一个违例了事，而是调用一个解决问题的方法）。另外，也可将自己的 try 块置入一个 while 循环里，用它不断进入 try 块，直到结果令人满意为止。

不过从历史角度看，即便操作系统本身提供了对可恢复违例的支持，但程序员对此通常并不“感冒”，他们最后往往还是选择了类似于“中断”的代码，直接了当地跳过“恢复”。为什么呢？尽管“恢复”表面上非常不错，但在实际应用中却显得困难重重。其中决定性的原因可能是：我们的控制模块必须随时留意是否产生了违例，以及是否包含了由产生位置专用的代码。这便使代码很难编写和维护——大型系统尤其如此，因为违例可能在多个位置产生。

10.3　创建自己的违例

并不一定非要使用现成的 Java 违例。这是非常重要的一个问题，因为经常都需要创建自己的违例，以指出自己的库可能生成的某个特殊错误——但在设计 Java 违例机制时，这些错误却是没法子预知的。

创建自己的违例类时，必须从一个现成的违例类型继承——它在含义上最好与你的新违例近似（不过，要想真正做到这一点，却非常难）。创建新违例类型最简单的做法是让编译器为你创建一个默认构造函数。我们一来，就几乎没什么代码需要我们亲自写的：

```
//: c10:SimpleExceptionDemo.java
// Inheriting your own exceptions.
class SimpleException extends Exception {}

public class SimpleExceptionDemo {
  public void f() throws SimpleException {
    System.out.println(
      "Throwing 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!");
    }
  }
} ///:~
```

编译器创建默认构造函数时，会自动调用基类的默认构造函数（这个过程是隐藏的）。当然，在这种情况下，我们得不到一个 SimpleException(String)构造函数，但在实际应用中，它却通常没太大的用处。就象大家会看到的那样，对一个违例来说，它最重要的便是“类名”，所以大多数情况下，象上面那样显示的一个违例已经足够了！

在这儿，我们将结果写入 System.err，从而在控制台“标准错误”流中打印出来。注意，在你发送错误信息的时候，最好将目的地设为 System.err，而不要发给 System.out，后者还有被重定向的可能。将结果发到 System.err 后，由于它不会随 System.out 一道被重新定向，所以用户能够更容易地注意到这些信息。

要想创建一个违例类，让它的一个构造函数也能取得一个 String 参数，具体做法也十分简单：

```
//: c10:FullConstructors.java
// Inheriting your own exceptions.

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}

public class FullConstructors {
```

```
  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(System.err);
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

补充的代码非常少——创建了两个对 MyException 的创建方式进行定义的构造函数。在第二个构造函数里，我们利用 super 关键字，明确调用了一个带 String 参数的基类构造函数。

堆栈跟踪信息被发给 System.err，使其在 System.out 被重定向的情况下，仍然有可能在用户面前清楚地显示出来。

程序输出如下：

```
Throwing MyException from f()
MyException
       at FullConstructors.f(FullConstructors.java:16)
       at FullConstructors.main(FullConstructors.java:24)
Throwing MyException from g()
MyException: Originated in g()
       at FullConstructors.g(FullConstructors.java:20)
       at FullConstructors.main(FullConstructors.java:29)
```

可以看到，在自 f()“掷”出的 MyException 违例中，缺少更详细的消息。

接下来，让我们继续自行创建违例的工作。下面，让我们来添加一些附加的构造函数及成员：

```
//: c10:ExtraFeatures.java
```

```
// Further embellishment of exception classes.

class MyException2 extends Exception {
  public MyException2() {}
  public MyException2(String msg) {
    super(msg);
  }
  public MyException2(String msg, int x) {
    super(msg);
    i = x;
  }
  public int val() { return i; }
  private int i;
}

public class ExtraFeatures {
  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(System.err);
    }
    try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace(System.err);
    }
    try {
```

```
      h();
    } catch(MyException2 e) {
      e.printStackTrace(System.err);
      System.err.println("e.val() = " + e.val());
    }
  }
} ///:~
```

这儿添加了一个数据成员 i；另外还添加了一个方法，用于读取 i 值；最后还有一个新的构造函数，用于对 i 进行设置。输出如下：

```
Throwing MyException2 from f()
MyException2
       at ExtraFeatures.f(ExtraFeatures.java:22)
       at ExtraFeatures.main(ExtraFeatures.java:34)
Throwing MyException2 from g()
MyException2: Originated in g()
       at ExtraFeatures.g(ExtraFeatures.java:26)
       at ExtraFeatures.main(ExtraFeatures.java:39)
Throwing MyException2 from h()
MyException2: Originated in h()
       at ExtraFeatures.h(ExtraFeatures.java:30)
       at ExtraFeatures.main(ExtraFeatures.java:44)
e.val() = 47
```

由于违例不过是另一种形式的“对象”，所以你完全可以自由发挥下去，进一步增强违例类的能力。但要注意，等你的这个封装交付到客户程序员的手中之后，所有这些“发挥”都可能变成空影儿。这是由于他们可能只是简单地查找要“掷”出的违例，除此以外便不再做任何事情（这也是大多数 Java 库违例的标准用法）。

10.4　违 例 规 范

在 Java 中，对于那些即将调用自己方法的客户程序员，我们有必要把该方法可能产生的违例提前告诉他们。这是一种文明的做法，只有这样才能使客户程序员准确知道要编写什么代码来捕获所有潜在的违例。当然，若你同时提供了源码，客户程序员甚至能全盘检查你的代码，找出相应的 throw 语句。当然了，我们发行的库大多都是不配带源码的。为了更有效地解决这个问题，Java 提供了一种特殊的语法格式（并强迫程序员采用），以便你“有礼貌”地告诉客户程序员一个方法会“掷”出什么样的违例，令其方便地加以控制。这便是我们在这儿要讲述的“违例规范”，它属于“方法声明”的一部分，位于参数（参数）列表的后面。

违例规范采用了一个新关键字：throws；后面要跟上所有可能的违例类型。因此，我们的方法定义看起来应该象下面这个样子：

```
void f() throws TooBig, TooSmall, DivZero { //...
```

但假如使用的是下述形式：

```
void f() { // ...
```

它的意思就是：别担心，这个方法不会产生违例（类型为 RuntimeException 的违例除外，“运行时间违例”可能在任何地方产生——稍后还会详细讲述）。

在违例规范的强制约束下，我们根本没有“哄骗”它的可能——如果方法造成了一个违例，但又没对其进行控制，编译器就会侦测到这个情况，并警告我们要么自己解决这个违例，要么根据违例规范，指出它有可能在你的方法中产生。通过至始至终对违例规范的贯彻，Java 便在编译时间有效保证了对违例的控制（一个违例无论如何都会被“捉”住[48]）

不过，有一个地方确实是允许我们“哄骗”的：你可以请求“掷”出一个并没有真正发生的违例。编译器理解并“同情”我们的请求，并强迫这个方法的用户当作真的发生了那个违例进行处理。在实际应用中，可将其作为那个违例的一个“占位符”使用。这样一来，以后就可方便地产生“真正”的违例，毋需修改原来的代码。另外，在创建抽象基类和接口的时候，这个窍门也相当管用，因为它们的派生类或者具体实现代码有可能要求真的产生违例。

10.4.1 捕获所有违例

我们可创建一个控制器，令其捕获所有类型的违例。具体做法是捕获名为“Exception”的一个基类违例类型（当然还有其他类型的基础违例，不过 Exception 是基础中的基础，它几乎适用于所有的编程活动）：

```
catch(Exception e) {
  System.err.println("Caught an exception");
}
```

这段代码能捕捉住所有违例，所以实际使用时，最好还是把它放在控制器列表的末尾，防止跟随其后的其他任何专用违例控制器失效（违例都被它“捉”去了，我还能干什么？）

由于 Exception 类是程序员常用的所有违例类的基础，所以通过它“捉”住一个违例之后，实际上并不能知道多少“特定”的违例信息。不过，我们可以调用来自它的基类型 Throwable 的方法：

```
String getMessage( )
String getLocalizedMessage( )
```

获得详细消息，或者由这个特定场合专用的一条消息。

```
String toString()
```

返回对 Throwable 的一段简要说明，其中包括详细消息（如果有的话）。

```
void printStackTrace()
void printStackTrace(PrintStream)
void printStackTrace(PrintWriter)
```

打印出 Throwable 和 Throwable 的调用堆栈路径。调用堆栈显示出将我们带到违例发生地点的方法调用的顺序。它的第一个版本会打印到标准错误流里；第二个和第三个则会打印

[48] 这是在 C++违例控制基础上一个显著的进步，后者除非到了运行时间，否则是“捉”不住那些不符合违例规范的错误的。也正是由于这个原因，C++的违例控制机制其实并无多大实际用处。

到我们选择一个数据流里（到了第 11 章，大家就会理解为什么有两种类型的“数据流”了）。

`Throwable fillInStackTrace()`

记录这个 Throwable 对象中与堆栈帧当前状态有关的信息。假如一个程序会重新产生错误或违例，便相当有用（马上还要详述）。

除此以外，我们还可从 Throwable 的基类 Object（它是所有对象的基类型）那里获得另外一些方法。对于违例控制来说，可能有用的一个方法是 getClass()，它的作用是返回一个对象，用它表示这个对象的类。然后，就可用 getName()或 toString()反查出这个 Class 对象的名字。还可对 Class 对象进行另一些更复杂的操作，尽管它们在违例控制中并不是特别必要的。本章稍后还会详细讲述 Class 对象的问题。

这里有一个例子，它展示了各个基本 Exception 方法的运用：

```
//: c10:ExceptionMethods.java
// Demonstrating the Exception Methods.

public class ExceptionMethods {
  public static void main(String[] args) {
    try {
      throw new Exception("Here's my Exception");
    } catch(Exception e) {
      System.err.println("Caught Exception");
      System.err.println(
        "e.getMessage(): " + e.getMessage());
      System.err.println(
        "e.getLocalizedMessage(): " +
         e.getLocalizedMessage());
      System.err.println("e.toString(): " + e);
      System.err.println("e.printStackTrace():");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

输出如下：

```
Caught Exception
e.getMessage(): Here's my Exception
e.getLocalizedMessage(): Here's my Exception
e.toString(): java.lang.Exception:
   Here's my Exception
e.printStackTrace():
java.lang.Exception: Here's my Exception
 at ExceptionMethods.main(ExceptionMethods.java:7)
java.lang.Exception:
```

```
   Here's my Exception
 at ExceptionMethods.main(ExceptionMethods.java:7)
```

可以看到，这些方法提供的信息量呈递增趋势——每类信息都是前一类信息的一个“超集”。

10.4.2 重新“掷”出违例

某些时候，我们又想重新“掷”出刚才已被“捉住”过的违例——特别是在用 Exception 捕获所有可能的违例的时候。由于我们已拥有了对当前违例的引用，所以只需简单地重新“掷”出那个引用即可。下面是一个例子：

```
catch(Exception e) {
  System.err.println("An exception was thrown");
  throw e;
}
```

重新“掷”出一个违例势必导致违例进入更高一级环境的违例控制器中。不过，同一个 try 块内任何更高级的 catch 从句仍会被忽略。此外，与违例对象有关的所有东西都会得到保留，所以用于捕获特定违例类型的高一级控制器可从那个对象里提取出所有信息。

若只是简单地重新“掷”出当前违例，我们用 printStackTrace()打印出来的与那个违例有关的信息仍会对应于违例的起源地，而非对应于重新“掷”出它的那个位置。如果想安放新的堆栈跟踪信息，需要调用 fillInStackTrace()，它会返回一个新的违例对象，该对象是通过将当前的堆栈信息打乱后放到老违例对象里而创建的。如下所示：

```
//: c10:Rethrowing.java
// Demonstrating fillInStackTrace()

public class Rethrowing {
  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(System.err);
      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(System.err);
    }
  }
} ///:~
```

注意，重要代码行的编号已作为注释标记出来。假如第 17 行本身不是注释（就象这里显示的那样），那么输出如下：

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
       at Rethrowing.f(Rethrowing.java:8)
       at Rethrowing.g(Rethrowing.java:12)
       at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
       at Rethrowing.f(Rethrowing.java:8)
       at Rethrowing.g(Rethrowing.java:12)
       at Rethrowing.main(Rethrowing.java:24)
```

也就是说，违例堆栈跟踪无论如何都会记住它的真正起点，无论自己被重复“掷”了好几次。

但假如将第 17 行变成注释，同时将第 18 行变成真正的代码（撤消注释），我们就换用了 fillInStackTrace()。这一次的结果如下：

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
       at Rethrowing.f(Rethrowing.java:8)
       at Rethrowing.g(Rethrowing.java:12)
       at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
       at Rethrowing.g(Rethrowing.java:18)
       at Rethrowing.main(Rethrowing.java:24)
```

由于 fillInStackTrace()的功劳，第 18 行成为违例的新起点。

针对 g()和 main()，Throwable 类必须在违例规范中出现，因为 fillInStackTrace()会生成一个 Throwable 对象的引用。由于 Throwable 是 Exception 的一个基类，所以它有可能获得一个能够“掷”出的对象（具有 Throwable 属性），而非获得一个普通 Exception（违例）。这样一来，在 main()中用于 Exception 的引用就可能丢失自己的目标。为保证所有东西均井然有序，编译器会强制为 Throwable 使用一个违例规范。举个例子来说，下述程序产生的违例便不会在 main()中被“捉”到：

```
//: c10:ThrowOut.java
public class ThrowOut {
  public static void
  main(String[] args) throws Throwable {
    try {
      throw new Throwable();
    } catch(Exception e) {
      System.err.println("Caught in main()");
    }
  }
} ///:~
```

也有可能从一个已经捕获到的违例重新“掷”出一个不同的违例。但假如这样做，就会得到与使用 fillInStackTrace()类似的效果：与违例起源地有关的信息会全部丢失，我们留下的只是与新的 throw 有关的信息。如下所示：

```
//: c10:RethrowNew.java
// Rethrow a different object
// from the one that was caught.

class OneException extends Exception {
  public OneException(String s) { super(s); }
}

class TwoException extends Exception {
  public TwoException(String s) { super(s); }
}

public class RethrowNew {
  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(System.err);
      throw new TwoException("from main()");
    }
  }
} ///:~
```

输出如下：

```
originating the exception in f()
Caught in main, e.printStackTrace()
OneException: thrown from f()
       at RethrowNew.f(RethrowNew.java:17)
       at RethrowNew.main(RethrowNew.java:22)
Exception in thread "main" TwoException: from main()
       at RethrowNew.main(RethrowNew.java:27)
```

最后一个违例只知道自己来自 main()，而非来自 f()。

永远都不要关心如何清除前一个违例；或者干脆这样说，不要关心任何违例的清除。它们本质上都是用 new 创建的、建立在“内存堆”基础上的对象，垃圾收集器会毫不客气地帮你清除它们！

10.5　标准 Java 违例

Java 包含了一个名为 Throwable 的类，它对可以作为违例“掷”出的所有东西进行了描述。Throwable 对象有两种常规类型（一般地，假如说甲是乙的某种类型，意思就是说甲是从乙“继承”的）。其中，Error 类型代表编译时间和系统错误，我们一般不必特意去捕获它们（特殊情况除外）。Exception 则可是从任何标准 Java 库的类方法、你自己设计的方法以及运行时任何异常中“掷”出来的基类型。因此，Java 程序员应将全副精心放在 Exception 上。

为了对违例的一个总体上的认识，最好的办法是先阅读一下标准的 Java 用户文档（HTML 格式），你可从 java.sun.com 下载。为了对各种违例都有一个大致的印象，这个准备工作是相当有必要的。不过，也千万不要依赖其中的讲解。因为大家看久了便会知道，文档并没有把不同违例的区别准确描述出来。除名字不一样外，似乎所有违例看起来都差不多！另外，Java 中的违例数量也在呈不断递增趋势。如果你正在使用由其他厂商开发的一个新库，那么它也极有可能设计了自己的违例。因此，在学习违例的时候，你需要抓住最本质的东西：违例是什么？用它能做什么？千万别被它“牵着鼻子走”。

所有违例的共通之处在于，违例的名字应该代表着会发生的问题，而且这些名字经过有意设计，仅从名称便应知道它的含义。并不是所有违例都定义于 java.lang 中；有的被拿去支持其他库去了，比如 util、net 和 io 这几个库。看看那些违例的完整类名，或者查一查它

们是从哪里继承的，便能知道一切！举个例子来说，所有 I/O 违例都肯定是从 java.io.IOException 继承的。

10.5.1 RuntimeException 的特殊情况

下面是本章的第一个例子：

```
if(t == null)
  throw new NullPointerException();
```

看起来，似乎要在传递给一个方法的每个引用中都检查 null（因为不知道调用者是否向你传递了一个有效的引用），这无疑是相当可怕的。但幸运的是，我们根本不必这样做——它属于 Java 进行的标准运行时间检查的一部分。若对一个空引用发出了调用，Java 会自动产生一个 NullPointerException 违例。所以上述代码在任何情况下都是多余的。

这个类别里含有一系列违例类型，全都是由 Java 自动生成，毋需我们亲自动手把它们包含到自己的违例规范里。最方便的是，通过将它们置入单独一个名为 RuntimeException 的基类下面，它们会全部组合到一起。这其实是一个很好的“继承”的例子——它建立了一系列具有某种共通性的类型，都具有某些共通的特征与行为。此外，我们没必要专门写一个违例规范，指出一个方法可能会“掷”出一个 RuntimeException，因为已经假定可能出现那种情况。由于它们用于指出编程中的错误，所以几乎永远不必专门捕获一个“运行时间违例”——RuntimeException——它在默认情况下会自动得到处理。若必须检查 RuntimeException，我们的代码就会变得相当繁复。不过，尽管通常用不着捕获 RuntimeExceptions，但在我们自己的封装里，也许仍然要选择“掷”出一部分 RuntimeException。

如果不捕获这些违例，又会出现什么情况呢？由于编译器并不强制违例规范捕获它们，所以假如不捕获的话，一个 RuntimeException 似乎就会过滤掉我们到达 main()方法的所有途径。为体会此时发生的事情，请试试下面这个例子：

```
//: c10:NeverCaught.java
// Ignoring RuntimeExceptions.

public class NeverCaught {
  static void f() {
    throw new RuntimeException("From f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
  }
} ///:~
```

大家已经看到，一个 RuntimeException（或者从它继承的任何东西）属于一种特殊情况，因为编译器不要求为这些类型指定违例规范。

输出如下：

```
Exception in thread "main"
java.lang.RuntimeException: From f()
       at NeverCaught.f(NeverCaught.java:9)
       at NeverCaught.g(NeverCaught.java:12)
       at NeverCaught.main(NeverCaught.java:15)
```

所以答案就是：假若一个 RuntimeException 获得到达 main()的所有途径，同时不被捕获，那么当程序退出时，会为那个违例调用 printStackTrace()。

注意在你写程序的时候，只能忽略 RuntimeException——其他东西是没法子忽略的，编译器会严格地进行检查。其实，这也并不是由于编译器要故意“放 RuntimeException 一马”，而是由于它把 RuntimeException 看成了一个“编程错误”。这样的错误有以下几方面的含义：

(1) 一个我们没办法捕获的错误（例如，接收由客户程序员传给你的方法的一个 null 引用）。

(2) 你作为一名程序员，在自己的代码中应该检查过的错误（就好比说，在你已经注意到数组大小的情况下，仍然产生了 ArrayIndexOutOfBoundsException——数组索引越界违例）

显然，在上述情况下，产生违例绝对会带来很大的好处，因为它有利于程序的调试！

因此，对于 Java 的违例，我们发现了它一个非常有趣的地方——不可把它简单地想象成单一用途的工具。的确，设计它们是为了控制那些讨厌的运行时间错误——那是由代码控制范围之外的其他力量造成的。然而，它也特别有助于调试某些特殊类型的“编程错误”（也就是那些 Bugs），而那些错误是编译器侦测不到的。

10.6　用 finally 清除

通常，无论 try 块内是否产生了一个违例，我们都想执行一些特定的代码，采取某些特定的操作。但要注意的是，“内存回收”则不在我们想采取的操作范围内，因为它是由垃圾收集器自动完成的。要达到这个目的，可在 try 块最末尾，使用一个 finally 从句[49]。所以，完整的违例控制小节看起来就象下面这个样子：

```
try {
  // 警戒区：
  // 可能产生 A,B 或 C 违例的危险行动
} catch(A a1) {
  // 出现 A 时用的控制器
} catch(B b1) {
  // 出现 B 时用的控制器
} catch(C c1) {
  // 出现 C 时用的控制器
} finally {
```

[49] C++违例控制未提供 finally 从句，因为它依赖构造函数来达到这种清除效果。

```
  // 每次都想采取的操作
}
```

为证实 finally 每次都会运行，请试验下面这个程序：

```
//: c10:FinallyWorks.java
// The finally clause is always executed.

class ThreeException extends Exception {}

public class FinallyWorks {
  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"
      }
    }
  }
} ///:~
```

通过该程序，我们亦可知道如何应付 Java 违例（类似 C++的违例）不允许我们恢复至违例产生地方的这一事实（前面已经讲过）。若将自己的 try 块置入一个循环内，就可建立一个条件，只有满足这个条件才能继续程序。亦可添加一个 static（静态）计数器或另一些设备，允许循环在彻底放弃之前尝试数种不同的方法。这样一来，我们的程序就会变得更加“健壮”，在各式各样的错误面前更能“免疫”。

输出如下：

```
ThreeException
In finally clause
No exception
In finally clause
```

显然，无论是否“掷”出一个违例，finally 从句都会执行。

10.6.1　用 finally 做什么

在没有“垃圾收集”以及“自动调用破坏器”机制的一种语言中[50]，finally 显得特别重要，因为程序员可用它担保内存得以正确释放（回收）——无论在 try 块内部发生了什么事情。但由于 Java 提供了垃圾收集机制，所以内存的回收几乎绝对不会成为问题。另外，Java 里也没有“破坏器”可供调用。既然如此，在 Java 里什么时候才轮到使用 finally 呢？

答案在于，假如你除了想把内存恢复成原始状态之外，还想设置另一些东西，finally 就是必需的！例如，我们有时需要打开一个文件或者建立一个网络连接，或者在屏幕上画一些东西，甚至设置外部世界的一个开关，等等。如下例所示：

```
//: c10:OnOffSwitch.java
// Why use finally?

class Switch {
  boolean state = false;
  boolean read() { return state; }
  void on() { state = true; }
  void off() { state = false; }
}
class OnOffException1 extends Exception {}
class OnOffException2 extends Exception {}

public class OnOffSwitch {
  static Switch sw = new Switch();
  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();
    }
  }
} ///:~
```

[50] “破坏器”（Destructor）是“构造函数”（Constructor）的反义词。它代表一个特殊的函数，一旦某个对象失去用处，就肯定会调用它。我们肯定知道在哪里以及何时调用破坏器。C++提供了破坏器的自动调用机制，但 Delphi 公司的 Object Pascal 版本 1 及 2 却不具备这一能力（在这种语言中，破坏器的含义与用法都发生了变化）。

在这里，我们的目标是保证 main()完成时开关处于关闭状态，所以将 sw.off()置于 try 块以及每个违例控制器的末尾。但产生的一个违例有可能不是在这里捕获的，这样便会错过 sw.off()。不过，利用 finally，来自一个 try 块的清除代码只需放在一个地方，我们就可以高枕无忧了。不必象上面那样把 sw.off()放得到处都是：

```
//: c10: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();
    }
  }
} ///:~
```

在这儿，sw.off()只存在于一个地方。无论发生什么事情，都肯定会运行它。

即使违例不在当前的 catch 从句集里捕获，finally 都会在违例控制机制转到一个更高级别的控制器之前得以执行。如下所示：

```
//: c10:AlwaysFinally.java
// Finally is always executed.

class FourException extends Exception {}

public class AlwaysFinally {
  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");
    }
  }
} ///:~
```

从该程序的输出，我们看到具体发生了什么事情：

```
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
```

若调用了 break 和 continue 语句，finally 语句也会得以执行。请注意，通过与标签式 break 以及标签式 continue 的配合，finally 便排除了在 Java 里使用 goto 跳转语句的必要！

10.6.2　缺点：丢失的违例

一般情况下，Java 的违例实现都显得十分出色，也非常好用。但不幸的是，它依然存在一个缺点。尽管违例已郑重指出程序里存在的一处危险，而且绝不该把它忽略，但一个违例仍有可能就那么不明不白地“丢失”了。在采用 finally 从句的一种特殊配置下，便有可能发生这种情况：

```
//: c10:LostMessage.java
// How an exception can be lost.

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 {
  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();
    }
  }
} ///:~
```

输出如下：

```
Exception in thread "main" A trivial exception
    at LostMessage.dispose(LostMessage.java:21)
    at LostMessage.main(LostMessage.java:29)
```

可以看到，这里并没有 VeryImportantException（非常重要的违例）存在的迹象，它在 finally 从句里被 HoHumException 这个违例替换掉了！这显然是一项相当严重的缺陷，因为这意味着一个违例可能完全丢失。而且就象上例演示的那样，这种丢失显得非常“自然”，很难被人查出蛛丝马迹。而与此相反，在 C++里，假如第一个违例还没有得到控制，便产生了第二个违例，就会被认为是一个极端严格的程序错误。希望 Java 以后的版本能纠正这个问题（不过另一方面，对于会产生违例的任何方法——比如上例的 dispose()——它们通常都应该放到一个 try-catch 从句内）。

10.7 违例的限制

覆盖一个方法时，只能产生已在方法的基类版本中定义的违例。这是一个有用的限制，因为它意味着与基类协同工作的代码也会自动应用于从基类派生的任何对象（当然，这属于基本的 OOP 概念），其中自然包括违例。

下面这个例子向大家演示了（在编译时间）施加在违例身上的限制：

```
//: c10: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 {
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  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:
  StormyInning() throws RainedOut,
    BaseballException {}
  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 base version does:
  public void event() {}
  // Overridden methods can throw
  // inherited exceptions:
  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 error");
    }
    // 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 中，可以看到无论构造函数还是 event()方法都指出自己会“掷”出一个违例，但它们实际上没有那样做。这是合法的，因为它允许我们强迫用户捕获可能在覆盖过的 event()版本里添加的任何违例。同样的道理也适用于 abstract 方法，就象在 atBat()里展示的那样。

“interface Storm”非常有趣，因为它包含了在 Incoming 中定义的一个方法——event()，以及不是在其中定义的一个方法。这两个方法都会“掷”出一个新的违例类型：RainedOut。当执行到“StormyInning extends”和“implementsStorm”的时候，可以看到 Storm 中的 event()方法不能改变 Inning 中的 event()的违例接口。同样地，这种设计是十分合理的；否则的话，

当我们操作基类时，便根本无法知道自己捕获的是否正确的东西。当然，假如 interface 中定义的一个方法不在基类里，比如 rainHard()，它产生违例时就没什么问题。

对违例的限制并不适用于构造函数。在 StormyInning 中，我们可看到一个构造函数能够“掷”出它希望的任何东西，无论基类构造函数“掷”出什么。然而，由于必须坚持按某种方式调用基类构造函数（在这里，会自动调用默认构造函数），所以派生类构造函数必须在自己的违例规范中声明所有基类构造函数违例。

StormyInning.walk()不会编译的原因是它“掷”出了一个违例，而 Inning.walk()却不会“掷”出。若允许这种情况发生，就可让自己的代码调用 Inning.walk()，而且它不必控制任何违例。但在以后替换从 Inning 派生的一个类的对象时，违例就会“掷”出，造成代码执行的中断。通过强迫派生类方法遵守基类方法的违例规范，对象的替换可保持连贯性。

覆盖过的 event()方法向我们显示出一个方法的派生类版本可以不产生任何违例——即便基类版本要产生违例。同样地，这样做是必要的，因为它不会中断那些已假定基类版本会产生违例的代码。差不多的道理亦适用于 atBat()，它会“掷”出 PopFoul——从 Foul 派生出来的一个违例，而 Foul 违例是由 atBat()的基类版本产生的。这样一来，假如有人在自己的代码里操作 Inning，同时调用了 gatBat()，就必须捕获 Foul 违例。由于 PopFoul 是从 Foul 派生的，所以违例控制器也会捕获 PopFoul。

最后一个有趣的地方在 main()内部。在这个地方，假如我们明确操作一个 StormyInning 对象，编译器就会强迫我们只捕获特定于那个类的违例。但假如我们向上强制转型到基类型，编译器就会强迫我们捕获针对基类的违例。通过所有这些限制，违例控制代码的“健壮”程度获得了大幅度改善[51]。

我们必须认识到这一点：尽管违例规范是由编译器在继承期间强行遵守的，但违例规范并不属于方法类型的一部分，后者仅包括了方法名以及参数类型。因此，我们不可在违例规范的基础上覆盖方法。除此以外，尽管违例规范存在于一个方法的基类版本中，但并不表示它必须在方法的派生类版本中存在。这与方法的“继承”颇有不同（进行继承时，基类中的方法也必须在派生类中存在）。换言之，用于一个特定方法的“违例规范接口”可能在继承和覆盖时变得更“窄”，而不会变得更“宽”——这与继承时的类接口规则是正好相反的。

10.8　构 造 函 数

为违例编写代码时，我们经常要解决的一个问题是：“一旦产生违例，会正确地进行清除吗？”大多数时候都会非常安全，但在构造函数中却是一个大问题。构造函数将对象置于一个安全的起始状态，但它可能执行一些操作——如打开一个文件。除非用户完成对象的使用，并调用一个特殊的清除方法，否则那些操作不会得到正确的清除。若从一个构造函数内部“掷”出一个违例，这些清除行为也可能不会正确地发生。所有这些都意味着在编写构造函数时，我们必须特别加以留意。

刚学了 finally 之后，大家也许会认为它是一种“最理想”的方案。但事情并没有那么简单，因为 finally 每一次都会执行清除代码——即便我们希望在清除方法运行之前，不要执行那些清除代码。正是考虑到这个问题，所以假如真的要在 finally 里进行清除，那么必须在构造函数正常结束时设置某种形式的标志——只要设置了标志，就不再执行 finally 块里的任何东西。显然，由于这种做法并不完美（需要你将一个地方的代码同另一个地方的结

[51] ISO C++施加了类似的限制，要求派生方法违例与基类方法掷出的违例相同，或者从后者派生。这是 C++能在编译时间检查违例规范的一种特殊情况。

合起来），所以除非特别必要，否则一般不要轻易尝试在 finally 中进行这种形式的清除。

在下面这个例子里，我们创建了一个名为 InputFile 的类。它的作用是打开一个文件，然后每次读取它的一行内容（转换为一个字串）。它利用了由 Java 标准 IO 库提供的 FileReader 以及 BufferedReader 类（将于第 11 章讨论）。这两个类都非常简单，大家现在可以毫无困难地掌握它们的基本用法：

```
//: c10:Cleanup.java
// Paying attention to exceptions
// in constructors.
import java.io.*;

class InputFile {
  private BufferedReader in;
  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!!!
    }
  }
  String getLine() {
    String s;
    try {
      s = in.readLine();
    } catch(IOException e) {
      System.err.println(
        "readLine() unsuccessful");
```

```
      s = "failed";
    }
    return s;
  }
  void cleanup() {
    try {
      in.close();
    } catch(IOException e2) {
      System.err.println(
        "in.close() unsuccessful");
    }
  }
}

public class Cleanup {
  public static void main(String[] args) {
    try {
      InputFile in =
        new InputFile("Cleanup.java");
      String s;
      int i = 1;
      while((s = in.getLine()) != null)
        System.out.println(""+ i++ + ": " + s);
      in.cleanup();
    } catch(Exception e) {
      System.err.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

用于 InputFile 的构造函数采用了一个 String（字串）参数，它代表我们想打开的那个文件的名字。在一个 try 块内部，它用该文件名创建了一个 FileReader。对 FileReader 来说，除非转移并用它创建一个能够实际与之“交谈”的 BufferedReader，否则便没什么用处。注意 InputFile 的一个好处就是它同时合并了这两种行动。

若 FileReader 构造函数不成功，就会产生一个 FileNotFoundException（文件未找到违例）。必须单独捕获这个违例——这属于我们不想关闭文件的一种特殊情况，因为文件尚未成功打开。其他任何捕获从句（catch）都必须关闭文件，因为文件已在进入那些捕获从句时打开（当然，如果多个方法都能产生一个 FileNotFoundException 违例，就需要稍微用一点儿技巧。此时，我们可将不同的情况分隔到数个 try 块内）。close()方法会掷出一个尝试过的违例。即使它在另一个 catch 从句的代码块内，该违例也会得以捕获——对 Java 编译器来说，那个 catch 从句不过是另一对花括号而已。执行完本地操作后，违例会被重新“掷”出。这样做是必要的，因为这个构造函数的执行已经失败，我们不希望调用方法来假设对象已正确创建

以及有效。

在这个例子中，没有采用以前用过的“标志”技术，finally 从句显然不是关闭文件的适当地方，因为那样一来，每次构造函数结束的时候，都有可能关闭它。由于我们希望文件在 InputFile 对象处于活动状态时一直保持打开状态，所以那样做并不恰当。

getLine()方法会返回一个字串，其中包含了文件中下一行的内容。它调用了 readLine()，后者可能产生一个违例，但那个违例会被捕获，使 getLine()不会再产生任何违例。对违例来说，一项特别的设计问题是决定在这一级完全控制一个违例，还是进行部分控制，并传递相同（或不同）的违例，或者只是简单地传递它。在适当的时候，简单地传递可极大简化我们的编码工作。getLine()方法会变成：

```
String getLine() throws IOException {
  return in.readLine();
}
```

但是当然，调用者现在需要对可能产生的任何 IOException 进行控制。

用户使用完毕 InputFile 对象后，必须调用 cleanup()方法，以便释放由 BufferedReader 和（或）FileReader 占用的系统资源（如文件句柄）[52]。除非 InputFile 对象使用完毕，而且到了需要弃之不用的时候，否则不应进行清除。大家可能想把这样的机制置入一个 finalize()方法内，但正如第 4 章指出的那样，并非总能保证 finalize()获得正确的调用（即便确定它会调用，也不知何时开始）。这属于 Java 的一项缺陷——除内存清除之外的所有清除都不会自动进行，所以必须知会客户程序员，告诉他们有责任用 finalize()保证清除工作的正确进行。

在 Cleanup.java 中，我们创建了一个 InputFile，用它打开用于创建程序的相同的源文件。同时一次读取该文件的一行内容，而且添加相应的行号。所有违例都会在 main()中被捕获——尽管我们可选择更大的可靠性。

这个例子也向大家揭示出为何要在本书的这个地方引入违例的概念——如果不用违例的话，基本 I/O（输入／输出）便没办法进行，而基本 I/O 正是我们马上就要讲到的主题！违例与 Java 编程有着牢不可分的联系，特别是由于编译器会强制它们，所以只有正确掌握了违例，才能用 Java 做更多的事情！

10.9 违 例 匹 配

“掷”出一个违例后，违例控制系统会按当初编写的顺序搜索“最近”的控制器。一旦找到相符的控制器，就认为违例已获得控制，不必再进行更多的搜索。

在违例和它的控制器之间，并不需要非常精确的匹配。一个派生类对象可与它的基类的一个控制器相配，如下例所示：

```
//: c10:Human.java
// Catching exception hierarchies.

class Annoyance extends Exception {}
```

[52] 在 C++里，“破坏器”可帮我们控制这一局面。

```
class Sneeze extends Annoyance {}

public class Human {
  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");
    }
  }
} ///:~
```

通过搜索，Sneeze 违例会被符合条件的第一个 catch 从句捕获——当然，它也是我们安排的第一个从句。然而，假如我们删除第一个 catch 从句，只剩下：

```
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    }
```

那么也能正常工作，因为它捕获的是 Sneeze 的基类。换言之，catch(Annoyance e)能捕获一个 Annoyance 以及从它派生出去的任何类。这一点非常重要，因为以后需要为一个方法添加更多派生的违例时，客户程序员那一边的代码便用不着任何修改——只要客户端捕获的是基类的违例！

假如将基类捕获从句放在最开头，试图“屏蔽”派生类违例，就象下面这样：

```
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    } catch(Sneeze s) {
      System.err.println("Caught Sneeze");
    }
```

那么编译器会产生一条出错消息，因为它发现永远不可能抵达 Sneeze 那一条捕获从句。

10.9.1　违例准则

我们用违例是为了做下面这些事情：

(1) 修正问题，并再次调用造成违例的方法。

(2) 平息事态，并在不重试方法的情况下继续运行。

(3) 绕过方法原本的企图，产生“另类”结果。

(4) 在当前环境中尽可能解决问题，并将相同的违例重新丢给一个更高级的环境。

(5) 在当前环境中尽可能解决问题，并将一个不同的违例丢给一个更高级的环境。

(6) 中止程序。

(7) 简化编码。如果违例使事态变得更为复杂，那必然会令人烦恼，不如不用。

(8) 使自己的库和程序变得更安全。这既是一种“短期投资”（便于调试），也是一种“长期投资”（使你的程序更“健壮”）。

10.10 总　　结

通过先进的错误纠正与恢复机制，我们可有效地增强代码的健壮程度。对我们写的每个程序来说，错误恢复都属于一个基本的、优先的考虑目标。它在 Java 中显得尤为重要，因为该语言的一个设计宗旨就是创建不同的程序组件，让其他人（客户程序员）使用。要想构建一套健壮的系统，其中的每个组件当然都必须非常健壮。

在 Java 里，违例控制的目的是使用尽可能精简的代码创建大型、可靠的应用程序，同时排除程序里那些不能控制的错误。

违例的概念很难掌握。但只有很好地运用它，才可使自己的项目立即获得显著的收益。Java 强迫遵守违例所有方面的问题，所以无论库设计者还是客户程序员，都能够连续一致地使用它。

10.11 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 用 main()创建一个类，令其在一个 try 块内产生 Exception 类的一个对象。在构造函数里，为 Exception 指定一个 String 参数。在 catch 从句内捕获违例，并打印出 String 参数。添加一个 finally 从句，用它打印一条消息，证明程序真的到过那里。

(2) 用 extends 关键字创建自己的违例类。为这个类写一个构造函数，令其采用一个 String 参数，并和 String 引用一道把它保存到对象内。写一个方法，用它打印出保存的 String。创建一个 try-catch 从句，练习这个新违例的运用。

(3) 写一个类，为其设计一个方法，以便产生练习(2)那个类型的一个违例。试着在没有违例规范的前提下编译它，观察编译器会报告什么。接着添加适当的违例规范。在一个 try-catch 从句中尝试自己的类以及它的违例。

(4) 定义一个对象引用，把它初始化成 null。试着通过该引用来调用一个方法。接着，将代码封装到一个 try-catch 从句里，以捕获违例。

(5) 创建一个类，它有两个方法，分别是 f()和 g()。在 g()中，产生刚才定义的新类型的一个违例。在 f()中，调用 g()并捕获它的违例。同时在 catch 从句中，产生一个不同的违例（即你定义的第二种类型）。在 main()中对代码进行测试。

(6) 创建新类型违例。写一个类，让它的一个方法产生上述所有三种类型的违例。在 main()中，调用该方法，但只用一个 catch 从句来捕获全部三种类型的违例。

(7) 编写代码，生成并捕获一个 ArrayIndexOutOfBoundsException（数组索引越界违例）。

(8) 自行模拟“中断－恢复”行为，具体做法是在一个 while 循环中不断尝试，直到不再有违例产生为止。

(9) 创建违例的一个三级结构。再创建基类 A，它的一个方法可在这个结构的最底级产生一个违例；从 A 继承得到 B，对上述方法进行覆盖，使其在该结构的第二级产生一个违例；重复这个过程，从 B 继承得到 C。在 main()中，请创建一个 C，再将其向上强制转型成 A，随后调用方法。

(10) 证明派生类构造函数不能捕获从它的基类构造函数产生的违例。

(11) 证明假如在一个 try 块内产生一个 RuntimeException，那么 OnOffSwitch.java 有可能执行失败。

(12) 证实假如在一个 try 块内产生一个 RuntimeException，那么 WithFinally.java 不会失败。

(13) 修改练习(6)，在其中添加一个 finally 从句。证实这个 finally 从句无论如何都会执行——即使产生了一个 NullPointerException（空指针违例）。

(14) 自己设计一个例子，用一个标志来控制是否调用清除代码，就象在本章“构造函数”那一节的第二自然段说的那样。

(15) 修改 StormyInning.java，在其中增添一个 UmpireArgument 违例类型，并设计相应的方法，以便产生该违例。请对修改过的违例分级结构进行测试。

(16) 删除 Human.java 的第一个 catch 从句，证实代码仍能正确编译并运行。

(17) 为 LostMessage.java 添加第二级的“违例丢失”，使 HoHumException 自己都被第三个违例所替代了。

(18) 在第 5 章，找到名为 Assert.java 的两个程序。对它们进行修改，令其产生自己类型的违例，而不是把出错消息打印到 System.err 里。该违例应该是一个对 RuntimeException 进行了扩展的内部类。

(19) 为 c08:GreenhouseControls.java 添加一个适当的违例集。

第 11 章 Java I/O 系统

对语言的设计者来说，要想弄出一套好的输入 / 输出（I/O）系统，可不是一件容易的事情！

如果你不信，大可考察一下目前各种不同的设计方案，便可很容易地总结出其中存在的挑战！其中最大的挑战似乎是如何照顾到所有可能的因素。不仅要与不同的 I/O 起点及终点通信（文件、控制台、网络连接等等），还要能用各种不同的方式进行通信（顺序、随机、缓冲式、二进制、字符式、按行、按字等等）。

Java 库的设计者通过创建大量类来攻克这一难关。事实上，Java 的 I/O 系统采用了如此多的类，以至于刚开始会产生不知从何处着手的感觉（不过，大家以后就会知道，JavaI/O 设计实际是防止了类的过度膨胀）。抛开 Java 1.0 不谈，在它之后，I/O 库的设计也发生了根本性的变化。特别地，由于新补充了面向 char 的、以 Unicode 为基础的 I/O 类，所以原来那个纯粹面向 byte 的库简直就跟换了一张脸似的。不过也正是由于这个原因，我们最开始不得不学习大量类的运用，否则便不能充分理解 Java 的 I/O 机制。除此以外，I/O 库的“进化史”也非常重要——尽管许多人的第一反映可能是：“不要再拿历史来骚拢我吧，直接从用法开始，如何？”嗯嗯……不过，假如你对历史不了解，很快就会对某些类感到迷惑，因为不知道什么时候该用它，什么时候不该用它！

本章将帮助大家理解标准 Java 库内的各种 I/O 类，并学习如何使用它们。

11.1 File 类

正式学习那些用来在“数据流”中实际读写数据的类之前，首先要掌握标准 Java 库提供的一个工具，它用于解决文件目录的问题。

“File”类的名字具有一定的欺骗性——通常会认为它对付的是一个“文件”，但实情并非如此。它既代表一个特定文件的名字，也代表目录内一系列文件的名字。若代表一个文件集，便可用 list()方法查询这个集，返回的是一个字串数组。之所以要返回一个数组，而非返回某个更“灵活”的容器类，是由于其中的元素数量是固定的。而且假如想得到一个不同的目录列表，只需创建一个不同的 File 对象即可。事实上，对这个类来说，“FilePath”（文件路径）似乎是个更好的名字。本节将向大家展示如何使用该类的一个例子，其中也讲到了如何使用对应的 FilenameFilter（文件名过滤器）接口。

11.1.1 目录列表器

现在，假定我们想观看一个目录列表。File 对象可采用两种方式列表。若在不指定参数（参数）的情况下调用 list()，得到的就是 File 对象包含的一个完整列表。然而，假如还想对这个列表进行某些限制，就需要使用一个“目录过滤器”——它其实是一个特殊的类，指出应将哪些 File 对象选中并显示出来。

下面是该例使用的代码。注意利用 java.utils.Array.sort() 和第 9 章定义的 AlphabeticComparator，可以毫不费力地完成结果的排序（字母顺序）。

```
//: c11:DirList.java
```

```
// Displays directory listing.
import java.io.*;
import java.util.*;
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 {
  String afn;
  DirFilter(String afn) { this.afn = afn; }
  public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    return f.indexOf(afn) != -1;
  }
} ///:~
```

DirFilter 类“实现”了 FilenameFilter 接口。请看看这个接口有多简单：

```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

它指出这种类型的所有对象都提供了一个名为 accept()的方法。之所以要创建这样的一个类，背后的全部原因就是把 accept()方法提供给 list()方法，使 list()能够“回调”accept()，从而判断应将哪些文件名包括到列表中。因此，通常将这种技术称为“回调”，有时也称为“算子”（也就是说，DirFilter 是一个算子，因为它唯一的作用就是容纳一个方法）。另外，也有人把这种技术叫作“命令范式”（Command Pattern）。由于 list()采用一个 FilenameFilter 对象作为自己的参数使用，所以我们能向它传递实现了 FilenameFilter 的任何类的一个对象，从而（甚至在运行时间）决定 list()方法的行为方式。“回调”的目的正是在代码的行为上提供更大的灵活性。

通过 DirFilter，我们看出尽管一个接口里只包括了一套方法，但并不是说我们便只能写那些方法（不过在一个接口里，至少要为所有方法都提供相应的定义）。就目前来说，我们还创建了 DirFilter 构造函数。

accept()方法必须接纳一个 File 对象，用它表示在其中找到了一个特定文件的目录；同时接纳一个 String 对象，其中包含了那个文件的名字。当然，你可决定使用或忽略这两个参数之一，但有时至少要使用文件名。记住 list()方法会为目录对象中的每个文件名都调用 accept()，核实哪个应包含在内——具体由 accept()返回的“布尔”结果决定。

为了保证我们操作的元素仅仅是文件名，其中没有包括任何路径信息，必须采用 String 对象，并在它的外部创建一个 File 对象。然后调用 getName()，它的作用是去除所有路径信息（采用与具体运行平台无关的方式）。随后，accept()用 String 类的 indexOf()方法检查文件名内部是否存在搜索字串"afn"。若在字串内找到 afn，那么返回值就是 afn 的起点索引；但假如没有找到，返回值就是-1。注意这只是一个简单的字串搜索例子，并不支持我们常用的“通配符”方案，比如“fo?.b?r*”——后面这种方案更难实现。

list()方法返回的是一个数组。可查询这个数组的长度，然后在其中遍历，选定数组元素。和 C/C++相比，由于数组能方便地传进和传出一个方法，所以无疑是一个巨大的进步。

匿名内部类

这个例子如果能用一个匿名内部类（已在第 8 章讲述）来改写一遍就显得更理想了。首先，让我们来创建一个 filter()方法，令其返回指向 FilenameFilter 的一个引用：

```
//: c11:DirList2.java
// Uses anonymous inner classes.
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class DirList2 {
  public static FilenameFilter
  filter(final String afn) {
    // Creation of anonymous inner class:
    return new FilenameFilter() {
      String fn = afn;
      public boolean accept(File dir, String n) {
        // Strip path information:
        String f = new File(n).getName();
        return f.indexOf(fn) != -1;
      }
    }; // 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 filter()的参数必须是 final。这一点是匿名内部类所要求的，否则便无法使用其作用域之外的一个对象。

之所以换用匿名内部类之后更“理想”，是由于 FilenameFilter 类现在已同 DirList2 紧密结合在一起。不过，我们还可采取进一步的操作，将匿名内部类定义成 list()的一个参数，使其显得更加精简。如下所示：

```
//: c11:DirList3.java
// Building the anonymous inner class "in-place."
import java.io.*;
import java.util.*;
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() {
        public boolean
        accept(File dir, String n) {
          String f = new File(n).getName();
          return f.indexOf(args[0]) != -1;
        }
      });
    Arrays.sort(list,
      new AlphabeticComparator());
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
  }
} ///:~
```

main()现在的参数是 final，因为匿名内部类是直接来使用 args[0]的。

本例展示了如何利用匿名内部类来快速创建一个精简的类，从而简化对一些复杂问题的

处理。由于 Java 中的所有东西都与类有关，所以这无疑是一种相当有用的编码技术。它的一个好处是将特定的问题隔离在一个地方统一解决。但另一方面，这样生成的代码不是十分容易阅读，所以使用时也必须慎重。

11.1.2 检查与创建目录

File 类并不仅仅是对当前文件或目录的一个表示。亦可用一个 File 对象新建一个目录，甚至创建一个完整的目录路径——如果它尚不存在的话。亦可用它了解文件的属性（长度、上一次修改日期、读 / 写属性等）；检查一个 File 对象到底代表一个文件还是一个目录；以及删除一个文件等等。下列程序完整展示了如何运用由 File 类提供的其他方法：

```
//: c11:MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
import java.io.*;

public class MakeDirectories {
  private final static String usage =
    "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\n";
  private static void usage() {
    System.err.println(usage);
    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;
    }
    for( ; count < args.length; count++) {
      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);
    }
  }
} ///:~
```

在 fileData()中，我们应用了各种文件查询方法来显示与文件或目录路径有关的信息。

main()应用的第一个方法是 renameTo()，利用它可重命名（或移动）一个文件至一个全新的路径（该路径由参数决定），它属于另一个 File 对象。这也适用于任何长度的目录。

试验上述程序，大家可发现自己能制作任意复杂程度的一个目录路径，因为 mkdirs()会帮我们完成所有工作。

11.2 输入和输出

在 I/O 库中，“数据流”（Stream）是一个重要的抽象概念，它将任何数据起点（源）和终点（接收者）都表达成一个对象，而这个对象有能力生成或接收数据片断。正是由于有了对数据流这种“抽象”，所以对于数据在物理性的 I/O 设备中到底发生了什么事情，便不是我们需要关心的了！

可将 Java 库的 I/O 类分割为输入与输出两个部分，这一点在用 Web 浏览器阅读联机 Java 类文档时便可知道。通过继承，从 InputStream（输入流）或 Reader 类派生出去的所有类都拥有名为 read()的基本方法，用于读取单个字节或者字节数组。类似地，从 OutputStream 或 Writer 派生出去的所有类都拥有基本方法 write()，用于写入单个字节或者字节数组。然而，我们通常不会用到这些方法；它们之所以存在，是因为更复杂的类可以利用它们，以便提供一个更有用的接口。因此，我们很少用单个类创建自己的流对象。一般情况下，我们都是将多个对象重叠在一起，提供自己期望的功能。而我们之所以觉得 Java 的流库（Stream Library）异常复杂，正是由于为了创建单独一个结果流，却需要创建这么多个对象的缘故。

在此，颇有必要按功能对类进行分类。在 Java 1.0 中，库的设计者的思路是：与输入有关的所有类都从 InputStream 继承，而与输出有关的所有类都从 OutputStream 继承。

11.2.1 InputStream 的类型

InputStream（输入流）的作用是表示出从各种不同的起点（数据源）产生输入的类。这些能产生输入的数据源包括：

(1) 字节数组

(2) 字串对象

(3) 文件

(4) “管道”（Pipe），它的工作原理与现实生活中的管道类似：一些东西放进一端，再从另一端出来。

(5) 一系列其他的流，以便将其统一收集到单独一个流内。

(6) 其他数据源，如 Internet 连接等（将在本书后面的部分讲述）。

所有这些都对应地拥有 InputStream 的一个子类。除此以外，FilterInputStream 也属于 InputStream 的一种类型，用它可为各种“装饰”类提供一个基类，以便将属性或者有用的接口同输入流联系到一起。这将在以后讨论。

表 **11.1** **InputStream** 的类型

类	功 能	构造函数参数 如 何 使 用
ByteArrayInputStream	允许内存中的一个缓冲区作为 InputStream 使用	从中提取字节的缓冲区
		作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口
StringBufferInputStream	将一个 String 转换成 InputStream	一个 String（字串）。基础实现实际采用一个 StringBuffer（字串缓冲）
		作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口

FileInputStream	用于从文件读取信息	代表文件名的一个 String，或者一个 File 或 FileDescriptor 对象
		作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口
PipedInputString	产生要写到对应 PipedOutputStream 里的数据。实现了“管道化”的概念	PipedOutputStream
		作为多线程处理时一种数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口
SequenceInputStream	将两个或更多的 InputStream 对象转换成单个 InputStream 使用	两个 InputStream 对象，或者对 InputStream 对象容器的一个 Enumeration
		作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口
FilterInputStream	作为装饰类一个接口使用的抽象类，为其他 InputStream 类提供了有用的功能。详见表 11.3	参考表 11.3
		参考表 11.3

11.2.2 OutputStream 的类型

这一类别包括的类决定了我们的输出该往何处去，是去一个“字节数组”（但不能去一个字串，尽管可用字节数组创建一个）呢？去一个“文件”呢？还是去一个“管道”。

除此以外，FilterOutputStream 为“装饰”类提供了一个基类，它将属性或者有用的接口同输出流连接起来。这将在以后讨论。

表 11.2 OutputStream 的类型

类	功 能	构造函数参数 如何使用
ByteArrayOutputStream	在内存中创建一个缓冲区。我们发送给流的所有数据都会置入这个缓冲区。	缓冲区初始大小（可选） 用于指出数据的目的地。若将其同 FilterOutputStream 对象连接到一起，可提供一个有用的接口
FileOutputStream	将信息发给一个文件	代表文件名的一个字串，或者一个 File/FileFileDescriptor 对象 用于指出数据的目的地。若将其同 FilterOutputStream 对象连接到一起，可提供一个有用的接口
PipedOutputStream	我们写给它的任何信息都会自动变成对应的一个 PipedInputStream 的输入，从而实现了“管道化”的概念	PipedInputStream 为多线程处理指出自己的数据目的地。将其同 FilterOutputStream 对象连接到一起，可提供一个有用的接口
FilterOutputStream	作为装饰类一个接口使用的抽象类；为其他 OutputStream 类提供了有用的功能。详见表 11.4	参考表 11.4 参考表 11.4

11.3 增添属性和有用的接口

利用层次化对象，为单独的对象动态及透明地添加职责的做法就叫作“装饰器”（Decorator）范式——“范式”[53]属于《Thinking in Patterns with Java》一书的主题，该书电子版可自 www.BruceEckel.com 下载。按照装饰器范式的规定，在你的初始化对象之外包裹着的所有对象都有着相同的接口。这样一来，装饰器的“透明”特性就有了一个最基本的应用——将相同的消息发给一个对象，无论它是否已被“装饰”。这正是在 Java I/O 库里存在“过滤器”（Filter）类的原因：抽象的“过滤器”类是所有装饰器的基类（装饰器必须拥有与它装饰的那个对象相同的接口，但装饰器亦可对接口作出扩展，这种情况见诸于几个特殊的“过滤器”类中）。

进行简单的子类处理时，由于要满足各种各样可能的组合，所以最后往往会得到数量众多的子类——正是由于数量太多，所以这样的子类处理往往是不切实际的。在这个时候，“装饰器”的重要性便凸显出来。由于 Java I/O 库要求许多不同的特性组合方案，所以我们才会考虑选用装饰器范式。不过，装饰器范式也有一个缺点。也就是说，在我们写一个程序的时候，装饰器尽管为我们提供了大得多的灵活性（因为可以方便地混合与匹配属性），但它们也使自己的代码变得更加复杂。Java I/O 库刚开始用时之所以觉得十分不“爽”，正是由于我们必须创建如此多的类——包括“核心 I/O”和所有的“装饰器”——最终却只能得到一个自己真正需要的 I/O 对象！

FilterInputStream 和 FilterOutputStream（这两个名字不十分直观）提供了相应的装饰器接口，用于控制一个特定的输入流（InputStream）或者输出流（OutputStream）。FilterInputStream 和 FilterOutputStream 分别是从 InputStream 和 OutputStream 这两个 I/O 库基类中派生出来的抽象类。要想使用装饰器，这便是非常重要的一个要求（否则便不能为打算装饰的所有对象都提供通用的接口）。

11.3.1 通过 FilterInputStream 从 InputStream 里读入数据

根据要做的事情，我们可将 FilterInputStream 里的各个类分为两大截然相反的类型。其中，DataInputStream 类让我们读取不同的基类型数据以及 String 对象（所有方法均以“read”开头，比如 readByte()、readFloat()等等）。与 DataOutputStream 配套使用，我们就可通过一个“流”，将基类型的数据从一个地方搬到另一个地方。这些“地方”是由表 11.1 总结的那些类决定的。

在 FilterInputStream 里，剩下的其他类用于修改一个 InputStream 的内部行为方式，其中包括是否进行缓冲；是否跟踪自己读入的数据行（以便查询或设置行号）；以及能否将一个字符向后推移等等。对后两种类来说，它们看起来颇似提供了对构建一个编译器的支持（换言之，用它们可支持一个 Java 编译器的构建），所以在常规编程中，一般都都不要用它们。

在实际应用中，我们几乎肯定要对自己的输入进行缓冲处理——无论要连接的那个 I/O 设备本身是否支持缓冲！因此，I/O 库的设计在这里似乎有一点儿不对头——为什么要将“缓冲的输入”当作特殊情况对待呢？应该将“未缓冲的输入”当作特殊情况对待才是呀！

表 11.3 FilterInputStream 的类型

[53] 参见《Design Patterns》，Erich Gamma 著，Addision-Wesley 出版社于 1995 年出版。

类	功　能	构造函数参数 如 何 使 用
DataInputStream	与 DataOutputStream 联合使用，使自己能从一个流中读取原始数据类型（int、char、long 等等），而且做到了与具体的运行平台无关（从而保证了程序的移植能力）	InputStream 包含了一个完整的接口，以便读取原始数据类型
BufferedInputStream	避免每次想要更多的数据时都进行实际的物理性读取。这相当于你告诉它“先用着缓冲”	InputStream（可选缓冲区大小） 本身并不提供一个接口，只是在使用缓冲时的一个要求而已。要求再连接一个接口对象
LineNumberInputStream	用于跟踪输入流中的行号；可调用 getLineNumber() 以及 setLineNumber-(int)，分别获取或设置行号	InputStream 只是增加了对数据行编号的能力，所以可能需要连接一个接口对象
PushbackInputStream	让你拥有一个字节的后推缓冲，以便将刚才读入的最后一个字符向后推	InputStream 通常在由一个编译器使用的扫描器中使用。也许是由于 Java 编译器的要求，所以才在这里提供。但你自己通常不必用它

11.3.2　通过 FilterOutputStream 向 OutputStream 里写入数据

与 DataInputStream 对应的是 DataOutputStream，后者对各个原始数据类型以及 String 对象进行格式化，并将其置入一个数据“流”中，以便任何机器上的 DataInputStream 都能正常地读取它们。所有方法都以“wirte”开头，例如 writeByte()、writeFloat()等等。

PrintStream 的初衷是采用一种便于阅读的格式，打印出所有原始数据类型和 String 对象。这一点是和 DataOutputStream 不同的，后者的目的是将数据元素放到一个“流”里，以便 DataInputStream 通过一种“与平台无关”的方式，对其进行重新组织。

在 PrintStream 中，两个重要的方法是 print()和 println()。它们已进行了覆盖处理，可打印出所有数据类型。print()和 println()之间的差异是后者在操作完毕后，还会自动添加一个新行。

而 PrintStream 是一种易于出问题的技术，因为它会捕捉住所有 IOException（I/O 违例）——必须用 checkError()明确地测试是否出现错误；如果真的有错误发生，它会返回 true。另外，在你的程序需要推出国际多语言版本时，PrintStream 也显得毛病多多，而且它不能采用一种“与平台无关”的方式，正确地控制换行（不过，所有这些问题已在 PrintWriter 中得到了解决）。

BufferedOutputStream 属于一种“修改器”，用于指示数据流使用缓冲技术，使自己不必每次都向一个“流”里物理性地写入数据。进行文件处理以及控制台 I/O 时，通常都应该用它。

表 **11.4　FilterOutputStream** 的类型

类	功　能	构造函数参数 如 何 使 用

DataOutputStream	与 DataInputStream 配合使用，以便采用“与平台无关”的形式，将原始数据类型（int、char、long 等等）写入一个数据流	OutputStream 包含了完整接口，以便我们写入原始数据类型
PrintStream	用于产生格式化输出。注意 DataOutputStream 控制的是数据的“存储”，而 PrintStream 控制的是数据的“显示”	OutputStream（可选一个布尔参数，指出每当有一个新行时，都对缓冲区进行刷新） 对于自己的 OutputStream 对象，应该用“final”将其封闭在内。可能经常都要用到它
BufferedOutputStream	用它避免每次发出一点儿数据的时候都要进行物理性的写入，要求它“先用缓冲区”。可随时调用 flush()，对缓冲区进行刷新，从而完成物理性的写入	OutputStream（可选缓冲区大小） 本身并不提供一个接口，只是在使用缓冲时的一种要求。需要再和一个接口对象连接到一起

11.4 Reader 和 Writer

在基本 I/O 流库的基础上，Java 1.1 进行了一些重大修改（不过，Java 2 并未进行根本性的变动）。在你首次看到 Reader 和 Writer 类时候，也许下意识的印象就是（就象我当初一样）：它们是用来替代 InputStream 和 OutputStream 类的！但实情并非如此。尽管不再建议使用原始数据流库的某些功能（如坚持使用，会从编译器收到一条警告消息），但在进行“面向字节”的 I/O 操作时，InputStream 和 OutputStream 类仍然提供了大量颇有价值的功能。只有在进行面向 Unicode 的、以字符为基础的 I/O 操作时，才应考虑 Reader 和 Writer 这两个新类。除此以外：

(1) Java 1.1 又为 InputStream 和 OutputStream 添加了许多新类，所以很明显，Sun 公司的意图是“补充”、而不是“替代”它们。

(2) 在许多情况下，“字节”（以 InputStream 和 OutputStream 为代表和“字符”（以 Reader 和 Writer 为代表）这两个体系中的类需要混合使用，才能取得预期的效果。当然，为了达到这个目的，我们还需要用到一些“桥”类或者“中间”类。比如 InputStreamReader 可将一个 InputStream 转换成 Reader，而 OutputStreamWriter 可将一个 OutputStream 转换成 Writer。

之所以要推出 Reader 和 Writer，最重要的原因就是提供“国际化支持”，让你的程序能方便地推出“多国语言版本”。老式 I/O 流只支持 8 位字节流，不能很好地控制 16 位 Unicode 字符。由于 Unicode 主要面向的是国际化支持（Java 固有的 char 是 16 位的 Unicode），所以设计者添加了 Reader 和 Writer，以便在所有 I/O 操作中提供对 Unicode 的支持。除此之外，新库也对速度进行了优化，可比旧库更快地运行。

与本书其他地方一样，我会试着提供为这些类提供一个概述，但假定你会去查询联机文档，自己搞定所有细节，比如方法的详尽列表等。

11.4.1 数据的起源和接收

对原始的那些 Java I/O 流类来说，它们几乎每个都有一个对应的 Read 和 Writer 类，以提供内建的 Unicode 支持。然而，在某些特殊场合下，面向字节的 InputStream 和 OutputStream 才应该是你最正确的选择。其中一个非常重要的例子便是 java.util.zip，这个库绝对是“面向字节”的，而非“面向字符”的。因此，你的正确做法应该是尽可能地“尝试”使用 Reader 和 Writer，而不要马上作决定。这样便能很轻易地发现那些必须使用老库（“面向字节”）的

特殊场合，因为否则你的代码便不能正确编译。

下表分别针对新旧两大体系（面向字节和面向字符），总结了信息源与接收地之间的对应关系（亦即在物理意义上，数据从什么地方来的，又要到哪里去）：

起源&接收：Java 1.0 类	对应的 Java 1.1 类
	Reader
InputStream	converter：
	InputStreamReader
	Writer
OutputStream	converter：
	OutputStreamWriter
FileInjputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
（no corresponding class）	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

我们发现即便不完全一致，但旧库组件中的接口与新接口通常也是类似的。

11.4.2　修改流的行为

前面已经说过，对 InputStream 和 OutputStream 这两种“流”来说，利用由 FilterInputStream 和 FilterOutputStream 提供的“装饰器”（Decorator）子类，它们可以满足各种不同的特殊需要。Reader 和 Writer 类也沿用了这一思路——但却并不是全盘照搬。

在下面这张表格中，对应关系显得要比上一张表“粗略”。之所以会出现这个差别，是由类的组织造成的：尽管 BufferedOutputStream 是 FilterOutputStream 的一个子类，但 BufferedWriter 却并不是 FilterWriter 的子类（对后者来说，尽管它是一个抽象类，但却并没有自己的子类。所以 BufferedWriter 的作用看起来更象是为了“占位”，大家不必为此操过多的心）。然而，两个类的接口是非常相似的。

过滤器：Java 1.0 类	对应的 Java 1.1 类
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter（没有子类的抽象类）
BufferedInputStream	BufferedReader（也有 readLine()）
BufferedOutputStream	BufferedWriter
DataInputStream	使用 DataInputStream（你要用 readLine()的时候除外，那时你应该用一个 BufferedReader）
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer（用一个构造函数，换用一个 Reader 作为参数）
PushBackInputStream	PushBackReader

有一条规律是显然的：若想使用 readLine()，就不要再用一个 DataInputStream 来实现（否则会在编译时得到一条出错消息），而应使用一个 BufferedReader。但除这种情况以外，DataInputStream 仍是 I/O 库的“首选”成员。

为了使向 PrintWriter 的过渡变得更加自然，它提供了能取任何 OutputStream 对象（以及 Writer 对象）作为参数的构造函数。然而，PrintWriter 提供的格式化支持没有 PrintStream 那么多；只是接口几乎都是相同的。

PrintWriter 构造函数也提供了一个选项，让你执行自动化的缓冲区刷新（倒空）。如在构造函数里设置了相应的标记，那么每次 println()之后，都会进行这样的刷新。

11.4.3 未改变的类

从 Java 1.0 过渡到 Java 1.1 后，有些类一丁点儿也没发生改变，所以 1.1 里并无新类可供对应。这些未改变的类包括：

在 Java 1.1 里没有对应类的 Java 1.0 类
DataOutputStream
File
RandomAccessFile
SequenceInputStream

特别未作任何改动的是 DataOutputStream。因此，要想用一种便于移植（与平台无关）的格式来存储和获取数据，你需要采用 InputStream 和 OutputStream 体系。

11.5 完全独立的 RandomAccessFile

如预先知道文件里包含的记录的长度，便可用 RandomAccessFile 对其进行访问。这样一来，我们便可通过 seek()方法从一条记录移至另一条。各记录的长度并不一定非要相同；只是必须提前知道它们到底有多大，以及保存在文件的什么地方。

最开始，大家也许有点儿难以相信 RandomAccessFile 居然不属于 InputStream 或者 OutputStream 体系的一部分。但经过深入考察，就会发现除了恰巧实现了 DataInput 以及 DataOutput 接口（DataInputStream 和 DataOutputStream 也实现了它们两个）之外，它和那些体系并无其他任何关系。它甚至根本没使用当前由 InputStream 或 OutputStream 提供的任何功能。换句话说，它是一个完全独立的类！设计者从头设计了它，并为其赋予了自己的一系列方法（大多是固有或专用方法）。之所以要这样做，是因为 RandomAccessFile 拥有与其他 I/O 类型完全不同的行为——我们可在一个文件里向前或向后移动。不管在哪种情况下，它都是独立运作的，作为 Object 的一个“直接继承人”使用。

从根本上说，RandomAccessFile 工作起来类似于 DataInputStream 和 DataOutputStream 的联合使用，同时用 getFilePointer()方法了解当前位于文件的什么地方；用 seek()移至文件内一个新位置；而用 length()判断文件的最大长度。此外，构造函数还要求使用另一个参数（与 C 的 fopen()完全一样），指出是只允许随机读取（"r"）呢，还是同时允许读写（"rw"）。注意这里没有提供对“只写文件”的支持。这可能暗示着假如 RandomAccessFile 是从 DataInputStream 继承来的，那么也能很好地工作。

搜索方法（seek()）只能在 RandomAccessFile 中使用，而且只能用于操作文件。反过来，

BufferedInputStream 则允许我们标注出一个位置（mark()），并将它的值保存在一个内部变量中；而且 reset()允许我们移至那个位置。不过，由于各种各样的因素，我们在使用这些功能时实际会受到很大的限制，不如不用！

11.6　I/O 流的典型应用

尽管各种 I/O 流类可通过多种不同的方式合并到一块儿，但实际上只有几种方式才会经常用到。下面这个例子可以说是你的“基本参考”，它演示了如何创建和使用典型的 I/O 配置。以后若有不明之处，也许经常都要回过头来参考一下这个例子。要注意的是，每种配置都用一个注释形式的编号及标题开头，对应于后文的各个专题小节，便于大家参考对照。

```
//: c11:IOStreamDemo.java
// Typical I/O stream configurations.
import java.io.*;

public class IOStreamDemo {
  // 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.writeChars("That was pi\n");
    out2.writeBytes("That was pi\n");
    out2.close();
    DataInputStream in5 =
      new DataInputStream(
        new BufferedInputStream(
          new FileInputStream("Data.txt")));
    BufferedReader in5br =
      new BufferedReader(
        new InputStreamReader(in5));
    // Must use DataInputStream for data:
    System.out.println(in5.readDouble());
```

```
      // Can now use the "proper" readLine():
      System.out.println(in5br.readLine());
      // But the line comes out funny.
      // The one created with writeBytes is OK:
      System.out.println(in5br.readLine());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }

    // 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();
  }
} ///:~
```

下面是对程序中各编号小节的详细说明：

11.6.1　输入流

在下文中，第 1 到第 4 部分演示了输入流的创建与使用。注意第 4 部分还展示了一个输出流的简单应用。

1. 缓冲的输入文件

为了打开一个文件进行字符输入，上述程序使用了一个 FileInputReader，并将一个字串或 File 对象指定为文件名。考虑到速度，最好能对文件进行缓冲处理，以便将结果引用拿给一个 BufferedReader 的构造函数。由于 BufferedReader 同时也提供了 readLine()方法，所以它就是我们最终从中读取数据的对象和接口。抵达文件末尾后，readLine()会返回 null，从而中断并退出 while 循环。

“String s2”用于汇集整理出完整的文件内容（包括必须添加的新行，因为 readLine()去除了那些行）。随后，在本程序的后面部分使用了 s2。最后，我们调用 close()，用它关闭文件。从技术角度说，close()会在 finalize()运行时自动调用，而且在程序退出的时候，也应该会自动调用它（不管是否进行垃圾收集）。不过，理想与现实总是存在一定差距的。由于其中的各种不确定因素，我们并不能依赖它的“自动”调用。相反，唯一安全的做法就是为文件明确调用 close(。

在程序中，1b 小节向大家演示了如何对 System.in 进行封装，从而读取控制台输入。System.in 本身是一个 DataInputStream，但 BufferedReader 需要的又是一个 Reader 参数，所以这里要求 InputStreamReader 的介入，由它担当起“中间人”的角色，完成双方之间的转换。

2. 从内存输入

这一部分采用已经包含了完整文件内容的 String s2，并用它创建一个 StringReader。随后，用 read()每次读取一个字符，并把它发至控制台。注意 read()将下一个字节当作一个 int 值返回，所以必须把它强制转型为一个 char，以便正确打印。

3. 格式化内存输入

为了读取“格式化好的”数据，我们需要用一个 DataInputStream，注意它是一个“面向字节”（而不是“面向字符”）的 I/O 类。因此，我们必须使用所有 InputStream 类，而不是用 Reader 类。当然，用 InputStream 可以将任何东西（比如一个文件）当作字节读入。要想将 String 转换成一个字节数组（对 ByteArrayInputStream 来说，“字节数组”才是恰当的选择），需要用到由 String 提供的 getBytes()方法。完成了这些工作后，我们便拥有了一个适当的 InputStream，可把它放心传给 DataInputStream 了。

如果采取每次一个字节的形式，用 readByte()从一个 DataInputStream 里读取字符，那么每个字节值都会是一个合法结果，所以不可用返回值探测输入的结束。相反，可以用 available()方法来调查当前还有多少个字符“可用”。这里有一个例子，展示了如何每次从文件里读回一个字节：

```
//: c11:TestEOF.java
// Testing for the 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());
  }
```

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

注意取决于当前是从什么媒体读入的，avaiable()的工作方式也有所区别。它表面的意思是“可以不受阻塞读取的字节数量”。对一个文件来说，它意味着整个文件。但对一个不同种类的数据流来说，它却可能有不同的含义。因此在使用时应考虑周全。

为了在这样的情况下侦测输入的结束，也可以采用捕获一个违例的办法。然而，如果真的用违例来控制数据流，却有点儿大材小用的感觉。

4. 文件输出

这个例子也展示了如何将数据写入一个文件。首先，我们创建一个 FileWriter，建立同文件的连接。通常几乎肯定要对输出进行缓冲处理，方法是把它封装到一个 BufferedWriter 里（自己试验一下假如删除这个封装，那么会对性能造成什么影响——缓冲的目的就是大幅提升 I/O 操作的性能）。随后，考虑到格式化的目的，我们把它转变成一个 PrintWriter。用这种方式创建的数据文件将具有良好的可读性，和普通文本文件没什么两样。

随着数据行向文件的写入，行号也会自动添加。注意这里并未使用 LineNumberInputStream，因为它是一个设计得非常糟糕的类，一般都要杜绝用它！就象这儿展示的那样，即使亲自动手对自己的行号进行跟踪，也不见得是一件多么困难的事情！

输入流的内容被取空后，readLine()会返回 null。此时，我们会为 out1 明确地调用 close()，因为假如不为所有输出文件都调用一下 close()，缓冲区就有可能没完全刷新，造成数据的不完整。

11.6.2　输出流

两类主要的输出流是按它们写入数据的方式划分的：一种是按人的习惯写入，采用人易于阅读的格式；另一种是便于以后由 DataInputStream 重新取回而写入的。RandomAccessFile 是独立使用的一个类，尽管它的数据格式仍然兼容于 DataInputStream 和 DataOutputStream。

1. 保存与恢复数据

PrintWriterm 能格式化数据，使其能按我们的习惯阅读。但为了输出数据，以便由另一个数据流取得，则需用一个 DataOutputStream 写入数据，再用一个 DataInputStream 恢复（获取）数据。当然，这些数据流可以是任何东西，但这里我们采用的是一个文件，并针对读和写都进行了缓冲处理。DataOutputStream 和 DataInputStream 是“面向字节”的，所以要求用到 InputStream 和 OutputStream。

如果用一个 DataOutputStream 来写入数据，那么 Java 可以担保你以后能用一个 DataInputStream 准确地取回数据——无论数据是用什么操作系统平台进行读写的。显然，这一特性对我们来说非常有价值，因为和平台有关的具体问题，已经有人事先帮我们操好心了！在你跨平台使用 Java 程序的时候，便不用再关心那方面的问题[54]。

注意字串是同时用 writeChars()和 writeBytes()写入的。运行程序时，你会发现 writeChars()输出的是 16 位 Unicode 字符。用 readLine()读取数据行时，会发现每个字符间都有一个空格，那个多出来的字符是由 Unicode 插入的（为支持国际语言，Unicode 必须采用双字节）。在 DataInputStream 中，由于没有一个专门的“readChars”方法，所以必须用 readChar()，每次

[54] XML 是解决数据跨平台移动问题的另一个方案，而且并不要求在所有平台上都安装 Java。不过，Java 工具存在着那种支持 XML。

一个地删除那些多余的字符。所以对 ASCII 来说，更方便的做法是将字符作为字节写入，在后面跟随一个新行；然后再用 readLine()将字节当作普通的 ASCII 行读回。

writeDouble()将 double（双精度）数字保存到数据流中，并用对应的 readDouble()取回它（读写其他类型的数据时，也有相应的方法可用）。但为了保证任何读方法能够正常工作，必须知道数据项在流中的准确位置，因为既有可能将保存的 double 数据作为一个简单的字节序列读入，也有可能作为 char 或其他格式读入。所以你要么需要为文件中的数据采用固定的格式，要么将一些额外的信息保存到文件中，以便根据它们判断数据的正确存放位置。

2. 读写随机访问文件

正如早先指出的那样，RandomAccessFile 是非常“独立特行”的一个类，它和 I/O 体系的其他部分几乎是完全分开的——尽管它也实现了 DataInput 和 DataOutput 接口。所以，不可把它与 InputStream 及 OutputStream 子类的任何部分关联起来。尽管也许能将一个 ByteArrayInputStream 当作一个随机访问元素对待，但只能用 RandomAccessFile 打开一个文件。必须假定 RandomAccessFile 已得到了正确的缓冲，因为我们不能亲自动手。

可以自行选择的是第二个构造函数参数：可决定以“只读”（r）方式或“读写”（rw）方式打开一个 RandomAccessFile 文件。

使用 RandomAccessFile 的时候，它类似于组合使用 DataInputStream 和 DataOutputStream（因为它实现了等价的接口）。除此以外，大家还可看到程序中使用了 seek()，以便在文件中到处移动，对某个值作出修改。

11.6.3 是一个错误吗？

仔细研究一下程序中的第 5 节，便会发现数据是在文字之前写入的。之所以要那么做，是由于 Java 1.1 存在的一个问题（同样的问题在 Java 2 中也存在）。在我来看，这似乎是一个 Bug，但等我向 JavaSoft 的人报告之后，他们却告诉我这是“正常”的（同样的问题在 Java 1.0 中却不会出现，这正是我有所怀疑的原因）。下面这段代码清晰地揭示了这个问题：

```
//: c11:IOProblem.java
// Java 1.1 and higher I/O Problem.
import java.io.*;

public class IOProblem {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    DataOutputStream out =
      new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream("Data.txt")));
    out.writeDouble(3.14159);
    out.writeBytes("That was the value of pi\n");
    out.writeBytes("This is pi/2:\n");
    out.writeDouble(3.14159/2);
    out.close();
```

```
    DataInputStream in =
      new DataInputStream(
        new BufferedInputStream(
          new FileInputStream("Data.txt")));
    BufferedReader inbr =
      new BufferedReader(
        new InputStreamReader(in));
    // The doubles written BEFORE the line of text
    // read back correctly:
    System.out.println(in.readDouble());
    // Read the lines of text:
    System.out.println(inbr.readLine());
    System.out.println(inbr.readLine());
    // Trying to read the doubles after the line
    // produces an end-of-file exception:
    System.out.println(in.readDouble());
  }
} ///:~
```

可以看出，在一次 writeBytes()调用之后再写入的任何东西都没法子收回了！这使我想起了一个老戏中的对白：“道克，我这样做你会受伤的!”“那就别做吧!”

11.6.4　管道化数据流

PipedInputStream、PipedOutputStream、PipedReader 和 PipedWriter 在本章只是进行了简单说明。不过，这并不是说它们的用处不大。相反，等你开始学习“多线程”的时候，才能发现它们更大的价值。在不同的“线程”之间，我们必须利用管道化的数据流来进行通信。在第 14 章的一个例子里，对此有清晰的说明。

11.7　标准 I/O

追溯历史，“标准 I/O”其实是来自 Unix 的一个概念（Windows 和其他许多操作系统都用自己的方式或多或少地演绎了这一概念），它是指由程序使用的某种信息流。所有程序的输入都来自“标准输入”，它的输出进入“标准输出”，而所有错误消息都发给“标准错误”。标准 I/O 的宗旨是将不同的程序方便地“链接”到一起。一个程序的标准输出可以成为另一个程序的标准输入。这是一个潜力无限的工具!

11.7.1　从标准输入中读取

根据前述的标准 I/O 模型，Java 提供了相应的 System.in，System.out 以及 System.err。贯这一整本书，大家都会看到如何用 System.out 进行标准输出，它已预封装成一个 PrintStream 对象。System.err 类似于一个 PrintStream，但 System.in 是一个原始的 InputStream，未作任何封装处理。这意味着尽管能直接使用 System.out 和 System.err，但必须事先封装 System.in，否则不能从中读取数据。

典型情况下，我们希望用 readLine()每次读取一行输入信息，所以需要将 System.in 封装到一个 BufferedReader 中。要做到这一点，必须用 InputStreamReader 将 System.in 转换成一

个 Reader。下面是一个简单的例子，可以在屏幕上回应我们键入的每一行内容：

```
//: c11:Echo.java
// How to read from standard input.
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()).length() != 0)
      System.out.println(s);
    // An empty line terminates the program
  }
} ///:~
```

之所以要使用违例规范，是由于 readLine()可能“掷”出一个 IOException。注意同其他大多数流一样，System.in 通常也应该进行缓冲处理。

11.7.2 将 System.out 变成 PrintWriter

System.out 是一个 PrintStream，后者又是一个 OutputStream。PrintWriter 有一个构造函数，可取得 OutputStream 作为自己的参数（参数）。因此，如果你希望能用那个构造函数将 System.out 转换成一个 PrintWriter，那么：

```
//: c11:ChangeSystemOut.java
// Turn System.out into a PrintWriter.
import java.io.*;

public class ChangeSystemOut {
  public static void main(String[] args) {
    PrintWriter out =
      new PrintWriter(System.out, true);
    out.println("Hello, world");
  }
} ///:~
```

这里很重要的一点就是使用 PrintWriter 构造函数的双参数版本，并将第二个参数设为 true，从而允许缓冲区进行自动刷新；否则的话，便难以看到输出。

11.7.3 标准 I/O 的重定向

Java 的 System 类允许我们重新定向标准输入、输出以及错误消息的 I/O 流，方法是使

用下述三个静态方法之一：

```
setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)
```

如果突然要在屏幕上生成大量输出，而且滚动的速度快于人们的阅读速度，输出的重定向就显得特别有用[55]。在一个命令行程序中，如果想重复测试一个特定的用户输入序列，输入的重定向也显得特别有价值。下面这个简单的例子展示了上述那些方法的应用：

```
//: c11:Redirecting.java
// Demonstrates standard I/O redirection.
import java.io.*;

class Redirecting {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    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!
  }
} ///:~
```

该程序将标准输入同一个文件连接起来，然后将标准输出和标准错误重定向到另一个文

[55] 第 13 章提供了该问题的一个更佳方案：用一个 GUI 程序来滚动显示文本区域!

件。

I/O 重定向只能处理字节流，不能对字符流进行操作。因此，这里只能使用 InputStream 和 OutputStream，不能用 Reader 和 Writer。

11.8 压　　缩

Java I/O 库提供了一系列类，可采用压缩格式来读写数据流。这些类包围在原有 I/O 类的外面，以提供压缩功能。

注意这些类并不是从 Reader 和 Writer 类派生出来的，而是属于 InputStream 和 OutputStream 体系的一部分。这是由于对压缩库来说，它们操作的是字节，而非字符。不过，我们有时也不得不混合使用两种类型的数据流（记住可以用 InputStreamReader 和 OutputStreamWriter 在不同类型之间方便地转换）。

压　缩　类	功　　能
CheckedInputStream	GetCheckSum()为任何 InputStream 产生校验和（不仅是解压）
CheckedOutputStream	GetCheckSum()为任何 OutputStream 产生校验和（不仅是解压）
DeflaterOutputStream	压缩类的基类
ZipOutputStream	一个 DeflaterOutputStream，将数据压缩成 Zip 文件格式
GZIPOutputStream	一个 DeflaterOutputStream，将数据压缩成 GZIP 文件格式
InflaterInputStream	解压类的基类
ZipInputStream	一个 DeflaterInputStream，解压用 Zip 文件格式保存的数据
GZIPInputStream	一个 DeflaterInputStream，解压用 GZIP 文件格式保存的数据

尽管目前有多种压缩算法，但可能只有 Zip 和 GZIP 才是最常用的。所以利用大量能读写这两种格式的工具，我们可以方便地处理自己的压缩数据。

11.8.1 用 GZIP 进行简单压缩

GZIP 接口非常简单，所以假如只有单个数据流需要压缩（而不是由一系列不同的数据构成的一个容器），那么它就可能是你最恰当的选择。下面是对一个文件进行压缩的例子：

```
//: c11:GZIPcompress.java
// Uses GZIP compression to compress a file
// whose name is passed on the command line.
import java.io.*;
import java.util.zip.*;

public class GZIPcompress {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    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);
  }
} ///:~
```

压缩类的用法非常直观——只需将输出流封装到一个 GZIPOutputStream 或者 ZipOutputStream 内，并将输入流封装到 GZIPInputStream 或者 ZipInputStream 内即可。其他所有操作都属于标准的 I/O 读写。注意在这个例子中，我们混合使用了“面向字符”的流以及“面向字节”的流——in 使用 Reader 类，而 GZIPOutputStream 的构造函数只能接收一个 OutputStream 对象，不能接收 Writer 对象。文件打开后，GZIPInputStream 会被转换成一个 Reader。

11.8.2 用 Zip 进行多文件保存

用来提供 Zip 支持的库显得全面得多，利用它可以方便地保存多个文件。其中甚至有一个单独类，可以简化对一个 Zip 文件的读取。这个库采用的是标准 Zip 格式，所以和当前能从网上下载的所有工具很好地配合。下面这个例子采取了与前例相同的形式，但能根据我们需要，支持任意数量的命令行参数。除此之外，它展示了如何用 Checksum 类来计算和校验文件的“校验和”（Checksum）。可选用两种类型的 Checksum：Adler32（速度较快）和 CRC32（较慢，但准确一点）。

```
//: c11:ZipCompress.java
// Uses Zip compression to compress any
// number of files given on the command line.
import java.io.*;
import java.util.*;
import java.util.zip.*;
```

```
public class ZipCompress {
  // 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 out =
      new ZipOutputStream(
        new BufferedOutputStream(csum));
    out.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]));
      out.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(
        new BufferedInputStream(csumi));
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
```

```
      System.out.println("Reading file " + ze);
      int x;
      while((x = in2.read()) != -1)
        System.out.write(x);
    }
    System.out.println("Checksum: " +
      csumi.getChecksum().getValue());
    in2.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
    }
  }
} ///:~
```

对于要加入压缩档的每一个文件，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全面的接口，利用它可以获取和设置 Zip 文件内那个特定的 Entry（入口）上能够接受的所有数据：名字、压缩后和压缩前的长度、日期、CRC 校验和、额外字段的数据、注释、压缩方法以及它是否为一个目录入口等等。然而，虽然 Zip 格式本身提供了设置密码的方法，但 Java 的 Zip 库没有提供这方面的支持。而且尽管 CheckedInputStream 和 CheckedOutputStream 同时提供了对 Adler32 和 CRC32 校验和的支持，但是 ZipEntry 只支持 CRC 的接口。这虽然属于基层 Zip 格式的限制，但却限制了我们使用速度更快的 Adler32。

为解压文件，ZipInputStream 提供了一个 getNextEntry()方法，能在有的前提下返回下一个 ZipEntry。作为一个更简洁的方法，可以用 ZipFile 对象读取文件。该对象有一个 entries()方法，可以为 ZipEntry 返回一个 Enumeration。

为了读取校验和，必须多少拥有对关联的 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，也可以只占有指向 Checksum 对象的一个引用。

Zip 流中一个令人困惑的方法是 setComment()。正如前面展示的那样，我们可在写一个文件时设置注释内容，但却没有办法取出 ZipInputStream 内的注释。看起来，似乎只能通过 ZipEntry 逐个入口地提供对注释的完全支持。

当然，GZIP 或 Zip 库并不仅仅限于操作文件——你完全可以压缩任何东西，包括要通过网络发送的数据等等。

11.8.3 Java 归档工具（JAR）

Zip 格式亦在 Java 的 JAR（Java ARchive）文件格式中得到了采用。这种文件格式的作用是将一系列文件合并到单个压缩文件里，就象 Zip 那样。然而，同 Java 中其他任何东西一样，JAR 文件是跨平台的，所以不必关心涉及具体平台的问题。除了可以包括声音和图像

文件以外，也可以在其中包括类文件。

涉及 Internet 应用时，JAR 文件显得特别有用。在 JAR 文件之前，Web 浏览器必须重复多次请求 Web 服务器，以便下载完构成一个“小程序”（Applet）的所有文件。除此以外，下回来的每个文件都是未压缩的。但在将所有这些文件合并到一个 JAR 文件里以后，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，所以可在更短的时间里取回全部数据。另外，JAR 文件里的每个入口（条目）都可以加上数字化签名（详情参考 Java 用户文档）。

一个 JAR 文件由一系列采用 Zip 压缩格式的文件构成，同时还有一张“详情表”，对所有这些文件进行了描述（可创建自己的详情表文件；否则，jar 程序会为我们代劳）。在联机用户文档中，可以找到与 JAR 详情表更多的资料（“详情表”的英语说法是“Manifest”——译注）。

jar 实用程序已与 Sun 的 JDK 配套提供，可以按我们的选择自动压缩文件。请在命令行调用它：

```
jar [选项] 目的地 [详情表] 输入文件
```

其中，“选项”用一系列字母表示（不必加上连字号或其他任何指示符）。如果你是 Unix/Linux 用户，便会注意到它和 tar 的选项非常相似。如下所示：

c	创建一个新的或空的压缩档
t	列出目录表
x	解压所有文件
x file	解压指定文件
f	指出“我准备向你提供文件名”。若省略此参数，jar 会假定它的输入来自标准输入；
m	指出第一个参数将是用户自建的详情表文件的名字
v	产生详细输出，对 jar 做的工作进行巨细无遗的描述
0	只保存文件；不压缩文件（用于创建一个 JAR 文件，以便我们将其置入自己的类路径中）
M	不自动生成详情表文件

对准备放进 JAR 压缩档的文件来说，如果它们还包括了一个子目录，那个子目录也会自动添加，其中包括它自己的所有子目录……等等。路径信息也会得到保留。

下面是调用 jar 的一些典型方法：

```
jar cf myJarFile.jar *.class
```

创建一个名为 myJarFile.jar 的 JAR 文件，其中包含了当前目录中的所有类文件，同时还有自动产生的详情表文件。

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

与前例类似，但添加了一个名为 myManifestFile.mf 的用户自建详情表文件。

```
jar tf myJarFile.jar
```

生成 myJarFile.jar 内所有文件的一个目录表。

```
jar tvf myJarFile.jar
```

添加“verbose”（详尽）标志，提供与 myJarFile.jar 中的文件有关的、更详细的资料。

```
jar cvf myApp.jar audio classes image
```

假定 audio、classes 和 image 是子目录，这便可将所有子目录合并到文件 myApp.jar 文

件中。其中也包括了“verbose”标志，可在 jar 程序工作时反馈更详尽的信息。

如果用 O 选项创建了一个 JAR 文件，那个文件就可放到我们的类路径（CLASSPATH）中：

```
CLASSPATH="lib1.jar;lib2.jar;"
```

这样一来，Java 就能在 lib1.jar 和 lib2.jar 中搜索目标类文件。

jar 工具的功能没有 zip 工具那么丰富。例如，不能够添加或更新一个现成 JAR 文件中的文件，只能从头开始新建一个 JAR 文件。此外，不能将文件移入一个 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 文件可在其他任何平台上由 jar 工具毫无阻碍地读出（这个问题有时会困扰 zip 工具）。

正如大家在第 13 章会看到的那样，我们也用 JAR 为 JavaBeans 打包。

11.9 对象序列化

利用 Java 的“对象序列化”（Object Serialization）功能，我们可取得实现了 Serializable 接口的任何对象，把它转变成一个字节序列，并可在以后完全恢复回原来的样子（对象）。整个过程亦可通过网络进行。这意味着序列化机制能自动补偿操作系统间的差异。换句话说，可以先在 Windows 机器上创建一个对象，将其序列化，然后通过网络发给一台 Unix 机器，然后在那里准确无误地重新“装配”回原来的样子。不必关心数据在不同机器上如何表示，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“有限的持久性”（Lightweight Persistence）。请记住“持久性”的含义是：对象的生存时间并不取决于一个程序是否正在执行——它存在或“生存”于程序的每一次调用之间。通过序列化一个对象，将其写入磁盘，以后在程序重新调用时重新恢复那个对象，就能圆满实现一种“持久”效果。之所以称其为“有限”，是因为不能用某种“persistent”（持久）关键字简单地定义一个对象，并让系统自动照看其他所有细节问题（尽管这将来可能成为现实）。相反，必须在自己的程序中明确地序列化和组装对象。

语言里增加了对象序列化的概念后，可提供对两种主要特性的支持。Java 的“远程方法调用”（Remote Method Invocation，RMI）使本来存在于其他机器的对象可以表现出好象就在本地机器上的行为。将消息发给远程对象时，需要通过对象序列化来传输参数和返回值。RMI 将在第 15 章作具体讨论。

对象的序列化也是 JavaBeans 必需的，JavaBeans 的详情将在第 13 章讲述。使用一个 Bean 时（对于“Bean”，你暂时可以想象它是一个“咖啡豆”，或者说是某种形式的程序组件），它的状态信息通常要在设计时间配置好。这种状态信息必须保存下来，以便程序启动后恢复；具体工作便由“对象序列化”完成。

对象的序列化处理非常简单——只要对象实现了 Serializable 接口（该接口仅是一个标记，不提供方法）。自从 Java 有了“对象序列化”的概念之后，许多标准库类也发生了改变，以便能够序列化——其中包括用于原始数据类型的全部封装器、所有容器类以及其他许多东西。现在，甚至连 Class 对象也可以序列化（第 12 章讲述了具体实现）。

为了序列化一个对象，首先要创建某些 OutputStream 对象，然后将其封装到 ObjectOutputStream 对象内。此时，只需调用 writeObject()即可完成对象的序列化，并将其发送给 OutputStream。相反的过程是将一个 InputStream 封装到 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强制转型 Object 的引用，所以必须向下强制转型，以便能直接设置。

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；接着又能对每个对象内包含的引用进行追踪……以此类推。我们有时将这种情况称为“对象之网”，每个对象都可与之建立连接。另外，它还包含了对象的引用数组以及成员对象。若必须自行操纵一套对象序列化机制，那么在代码里追踪所有这些链接时可能会显得非常麻烦。在另一方面，由于 Java 对象的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“对象之网”。下面这个例子对序列化机制进行了测试。它建立了由许多链接对象构成的一个“Worm”（蠕虫），每个对象都与 Worm 中的下一段链接；同时还有一个引用数组，每个引用都指向另一个类——Data——的一个对象。

```
//: c11:Worm.java
// Demonstrates object serialization.
import java.io.*;

class Data implements Serializable {
  private int i;
  Data(int x) { i = x; }
  public String toString() {
    return Integer.toString(i);
  }
}

public class Worm implements Serializable {
  // Generate a random int value:
  private static int r() {
    return (int)(Math.random() * 10);
  }
  private Data[] d = {
    new Data(r()), new Data(r()), new Data(r())
  };
  private Worm next;
  private char c;
  // Value of i == number of segments
  Worm(int i, char x) {
    System.out.println(" Worm constructor: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
  }
  Worm() {
    System.out.println("Default constructor");
  }
  public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++)
```

```
      s += d[i].toString();
    s += ")";
    if(next != null)
      s += next.toString();
    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");
    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");
    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字初始化的（这样便不用怀疑编译器保留了某种“元信息”）。每个 Worm 段都用一个 Char 标记。这个 Char 是在重复生成链接的 Worm 列表时自动产生的。创建一个 Worm 时，需告诉构造函数希望它有多长。为产生下一个引用（next），它总是用减去 1 的长度来调用 Worm 构造函数。最后一个 next 引用则保

持为 null（空），表示已抵达 Worm 的尾部。

上面所有操作的目的都是为了加大问题的复杂程度，使序列化“好象”没那么容易。然而，序列化过程实际却是非常简单的。一旦从另外某个流里创建了 ObjectOutputStream，writeObject()就会序列化对象。注意也可以为一个 String 调用 writeObject()。亦可使用与 DataOutputStream 相同的方法写入所有原始数据类型（它们有相同的接口）。

有两个独立的 try 块看起来是类似的。第一个读写一个文件；但为示区别，另一个读写的是一个 ByteArray（字节数组）。可通过对任何 DataInputStream 或者 DataOutputStream（就象大家在第 15 章会看到的那样，其中甚至包括“网络”）的序列化来读写一个对象。下面列出一次运行的结果：

```
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w2 = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w3 = :a(262):b(100):c(396):d(480):e(316):f(398)
```

可以看出，装配回原状的对象确实包含了原来那个对象里包含的所有链接。

注意在对一个 Serializable（可序列化）对象进行重新装配的过程中，不会调用任何构造函数（甚至包括默认构造函数）。整个对象都是通过从 InputStream 中取得数据恢复的。

另外，要注意对象序列化是“面向字节”的，所以你必须得采用 InputStream 和 OutputStream 体系。

11.9.1 寻找类

现在，大家也许会产生这样一个疑问：要使一个对象从序列化状态中恢复，我们都需要掌握哪些信息呢？举个例子来说，假定我们序列化了一个对象，并通过网络将其作为文件传给另一台机器。那么，在只能利用文件目录的前提下，另一台机器上的程序可不可以完成对象的重新构建呢？

解答这个问题最好的办法就是做一个实验。下面这个文件放在本章对应的子目录下：

```
//: c11: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：

```
//: c11:FreezeAlien.java
// Create a serialized output file.
```

```
import java.io.*;

public class FreezeAlien {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    ObjectOutput out =
      new ObjectOutputStream(
        new FileOutputStream("X.file"));
    Alien zorcon = new Alien();
    out.writeObject(zorcon);
  }
} ///:~
```

该程序并不会捕获和控制违例，而是将违例简单、直接地传递到 main()外部，这样便能在命令行报告它们。

程序编译并运行后，将结果产生的 X.file 复制到名为 xfiles 的一个子目录，代码如下：

```
//: c11:xfiles:ThawAlien.java
// Try to recover a serialized file without the
// class of object that's stored in that file.
import java.io.*;

public class ThawAlien {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("X.file"));
    Object mystery = in.readObject();
    System.out.println(mystery.getClass());
  }
} ///:~
```

该程序能打开文件，并成功读取 mystery 对象。然而，一旦尝试查找与对象有关的任何资料——这要求 Alien 的 Class 对象——Java 虚拟机（JVM）便会发现自己找不到 Alien.class（除非它恰好在类路径内，本例并非如此）。这样就会得到一个名叫 ClassNotFoundException 的违例（同样地，在 Alien 的存在可以被证明之前，它的所有证据都消失了）。

恢复了一个序列化的对象后，如果还想对其做更多的事情，必须保证 JVM 能在本地类路径或 Internet 的其他什么地方找到相应的.class 文件。

11.9.2　序列化的控制

正如大家看到的那样，默认的序列化机制并不难操纵。然而，假若有特殊要求又该怎么办呢？我们可能有特殊的安全问题，不希望对象的某一部分序列化；或者某一个子对象完全

不必序列化，因为对象恢复以后，那一部分需要重新创建。

此时，通过实现 Externalizable 接口，用它代替 Serializable 接口，便可控制序列化的具体过程。这个 Externalizable 接口扩展了 Serializable，并增添了两个方法：writeExternal()和 readExternal()。在序列化和重新装配的过程中，会自动调用这两个方法，以便我们执行一些特殊操作。

下面这个例子展示了 Externalizable 接口方法的简单应用。注意 Blip1 和 Blip2 几乎完全一致，除了极微小的差别（自己研究一下代码，看看是否能发现）：

```
//: c11:Blips.java
// Simple use of Externalizable & a pitfall.
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 {
  // 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();
  }
} ///:~
```

输出如下：

```
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
```

未恢复 Blip2 对象的原因是那样做会导致一个违例。你找出了 Blip1 和 Blip2 之间的区别吗？Blip1 的构造函数是“公共的”（public），Blip2 的构造函数则不然，这样便会在恢复时造成违例。试试将 Blip2 的构造函数属性变成“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 默认构造函数。这与恢复一个 Serializable（可序列化）对象不同。在后者的情况下，对象完全以它保存下来的二进制位作为基础进行恢复，不存在构造函数调用的问题。而对一个 Externalizable 对象来说，所有普通的默认构建行为都会首先发生（包括在字段定义时的初始化），然后才会调用 readExternal()。必须注意这一事实——特别注意所有默认的构建行为都会进行——否则很难在自己的 Externalizable 对象中产生正确

的行为。

下面这个例子揭示出为了完整保存并恢复一个 Externalizable 对象，你必须采取的一整套操作：

```
//: c11:Blip3.java
// Reconstructing an externalizable object.
import java.io.*;
import java.util.*;

class Blip3 implements Externalizable {
  int i;
  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);
  }
} ///:~
```

其中，字段 s 和 i 只在第二个构造函数中初始化，而不是在默认构造函数中进行。这意味着假如不在 readExternal()中初始化 s 和 i，它们就会成为 null（因为对象创建的第一步时已将对象存储空间清零）。如果把跟在“You must do this”字样后面的两行代码变成注释，再运行程序，就会发现当对象恢复以后，s 是 null，而 i 是零。

若从一个 Externalizable 对象继承，通常需要调用 writeExternal()和 readExternal()的基类版本，以便正确地保存和恢复基类组件。

所以为了让一切正常运作起来，除了必须在 writeExternal()方法执行期间写入对象的重要数据之外（没有默认的行为可用来为一个 Externalizable 对象写入任何成员对象）的，而且还必须在 readExternal()方法中恢复那些数据。初次操作时可能会有些不习惯，因为 Externalizable 对象的默认构建行为使其看起来似乎在自动进行某种存储与恢复操作。但实情并非如此。

1. transient（临时）关键字

控制序列化过程时，可能有一个特定的子对象不愿让 Java 的序列化机制自动保存与恢复。一般地，若某个子对象包含了不想序列化的敏感信息（如密码），就会面临这种情况。即使那种信息在对象中具有“private”（私有）属性，但只要经过序列化处理，人们就可通过读取一个文件，或拦截网络传输而得到它。

为防止对象的敏感部分被序列化，一个办法是将自己的类实现为 Externalizable，就象前面展示的那样。这样一来，没有任何东西可以自动序列化，只能在 writeExternal()内明确序列化那些需要的部分。

然而，若操作的是一个 Serializable 对象，所有序列化操作都会自动进行。为解决这个问题，可以用 transient（临时）逐个字段关闭序列化，它的意思是“不用麻烦你（指自动机制）保存或恢复它了——我会自己处理的”。

例如，假设我们有一个 Login 对象，它包含了与一个特定的登录会话有关的信息。校验登录的合法性时，一般都想将数据保存下来，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 字段标记为“transient”。下面是具体的代码：

```
//: c11:Logon.java
// Demonstrates the "transient" keyword.
```

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

class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  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 IOException, ClassNotFoundException {
    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();
    // Delay:
    int seconds = 5;
    long t = System.currentTimeMillis()
          + seconds * 1000;
    while(System.currentTimeMillis() < t)
      ;
    // 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);
  }
} ///:~
```

可以看到，其中的 date 和 username 字段保持原始状态（未设成 transient），所以会自动序列化。然而，password 被设为 transient，所以不会自动保存到磁盘；另外，自动序列化机制也不会作恢复它的尝试。输出如下：

```
logon a = logon info:
  username: Hulk
  date: Sun Mar 23 18:25:53 PST 1997
  password: myLittlePony
Recovering object at Sun Mar 23 18:25:59 PST 1997
logon a = logon info:
  username: Hulk
  date: Sun Mar 23 18:25:53 PST 1997
  password: (n/a)
```

一旦对象恢复成原来的样子，password 字段就会变成 null。注意必须用 toString()检查 password 是否为 null，因为若用重载的“+”运算符来装配一个 String 对象，而且那个运算符遇到一个 null 引用，就会造成一个名为 NullPointerException 的违例（新版 Java 可能会提供避免这个问题的代码）。

另外，也可看到 date 字段被保存到磁盘，并从磁盘恢复，没有重新生成。

由于 Externalizable 对象默认情况下不保存它的任何字段，所以 transient 关键字只能伴随 Serializable 使用。

2. Externalizable 的替代方法

若不是特别在意要实现 Externalizable 接口，还有另一种方法可供选用。我们可以实现 Serializable 接口，并添加（注意是“添加”，而非“覆盖”或者“实现”）名为 writeObject()和 readObject()的方法。一旦对象被序列化或者重新装配，就会分别调用那两个方法。也就是说，只要提供了这两个方法，就会优先使用它们，而不考虑默认的序列化机制。

这些方法必须含有下列签名（在你的程序中，必须一模一样）：

```
private void
  writeObject(ObjectOutputStream stream)
    throws IOException;

private void
  readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException
```

从设计的角度出发，情况似乎变得有些扑朔迷离。首先，大家可能认为这些方法不属于基类或者 Serializable 接口的一部分，它们应该在自己的接口中得到定义。但请注意，它们被定义成“private”，这意味着它们只能由这个类内的其他成员调用。然而，我们实际并不从这个类的其他成员中调用它们，而是由 ObjectOutputStream 和 ObjectInputStream 的 writeObject()及 readObject()方法来调用我们对象的 writeObject()和 readObject()方法（注意我在这里非常小心地避免使用相同的方法名——因为怕混淆）。大家可能觉得奇怪，

ObjectOutputStream 和 ObjectInputStream 为什么有权访问我们的类的 private 方法呢？只能认为这是序列化机制玩的一个把戏！

在任何情况下，接口中的定义的任何东西都会自动具有 public 属性，所以假若 writeObject()和 readObject()必须为 private，那么它们不能成为接口的一部分。但由于我们准确地加上了签名，所以最终的效果实际与实现一个接口是相同的。

看来在调用 ObjectOutputStream.writeObject()的时候，似乎会对传递给它的 Serializable 对象进行检查（无疑是通过“反射”机制检查的），看它是否实现了自己的 writeObject()。若答案是肯定的，便会跳过常规的序列化过程，并调用 writeObject()。readObject()也会遇到同样的情况。

还存在另一个问题。在我们的 writeObject()内部，可以调用 defaultWriteObject()，从而采取默认的 writeObject()行动。类似地，在 readObject()内部，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：

```
//: c11:SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods.
import java.io.*;

public class SerialCtl implements Serializable {
  String a;
  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);
  }
} ///:~
```

在这个例子中，一个 String 保持原始状态，其他设为 transient（临时），以便证明非临时字段会被 defaultWriteObject()方法自动保存，而 transient 字段必须在程序中明确保存和恢复。字段是在构造函数内部初始化的，而不是在定义的时候，这证明了它们不会在重新装配的时候被某些自动化机制初始化。

若准备通过默认机制写入对象的非 transient 部分，那么必须调用 defaultWriteObject()，令其作为 writeObject()中的第一个操作；并调用 defaultReadObject()，令其作为 readObject()的第一个操作。这些都是不常见的调用方法。举个例子来说，当我们为一个 ObjectOutputStream 调用 defaultWriteObject()的时候，而且没有为其传递参数，就需要采取这种操作，使其知道对象的引用以及如何写入所有非 transient 的部分。这种做法非常不便。

transient 对象的存储与恢复采用了我们更熟悉的代码。现在考虑一下会发生一些什么事情。在 main()中会创建一个 SerialCtl 对象，随后会序列化到一个 ObjectOutputStream 里（注意这种情况下使用的是一个缓冲区，而非文件——与 ObjectOutputStream 完全一致）。正式的序列化操作是在下面这行代码里发生的：

```
o.writeObject(sc);
```

其中，writeObject()方法必须核查 sc，判断它是否有自己的 writeObject()方法（不是检查它的接口——根本就没有接口；也不是检查类类型，而是利用“反射”机制实际地搜寻方法）。若答案是肯定的，就使用那个方法。类似的情况也会在 readObject()上发生。或许这是解决问题唯一实际的方法，但确实有点儿古怪。

3. 版本问题

有时候可能想改变一个可序列化的类的版本（比如原始类的对象可能保存在数据库中）。尽管这种做法得到了支持，但一般只应在非常特殊的情况下才用它。此外，它要求操作者对背后的原理有一个比较深的认识，而我们在这里还不想达到这种深度。JDK HTML 文档对这一主题进行了非常全面的论述（请从 java.sun.com 下载）。

另外注意在 JDK HTML 文档中，许多注释最开头都有下面这段话：

```
Warning: Serialized objects of this class will not be compatible
with future Swing releases. The current serialization support
```

```
is appropriate for short term storage or RMI between
applications. …
```

它的意思是：

> **警告：**该类的序列化对象并不兼容未来发布的 Swing 版本。当前的序列化支持只适用于应用程序之间的短期存储或 RMI 功能。

之所以会这样，是由于版本机制过于简单，以至于它不能可靠地应用于所有场合——特别是在使用 JavaBeans 的时候。它只适用于设计时的程序订正，因此才会有上述警告。

11.9.3 利用“持久性”

一个比较诱人的想法是用序列化技术保存程序的一些状态信息，从而将程序方便地恢复到以前的状态。但在具体实现以前，有些问题是必须解决的。如果两个对象都有指向第三个对象的引用，该如何对这两个对象序列化呢？如果从两个对象的序列化状态中恢复它们，那么对第三个对象的引用只会出现在一个对象身上吗？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重新装配它们，又会得到什么结果呢？

下例对上述问题进行了很好的说明：

```
//: c11:MyWorld.java
import java.io.*;
import java.util.*;

class House implements Serializable {}

class Animal implements Serializable {
  String name;
  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();
    ArrayList  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()));
    ArrayList animals1 =
      (ArrayList)in1.readObject();
    ArrayList animals2 =
      (ArrayList)in1.readObject();
    ArrayList animals3 =
      (ArrayList)in2.readObject();
    System.out.println("animals1: " + animals1);
    System.out.println("animals2: " + animals2);
    System.out.println("animals3: " + animals3);
  }
} ///:~
```

这儿一个有趣的地方在于，我们也许能针对一个字节数组应用对象的序列化，从而实现对任何 Serializable（可序列化）对象的一个“全面复制”（全面复制意味着复制的是整个对象网，而不仅是基本对象和它的引用）。复制问题将在本书的附录 A 进行探讨。

Animal 对象包含了类型为 House 的字段。在 main()中，会创建这些 Animal 的一个 ArrayList，并对其序列化两次——先送入一个流，再送到另一个流。这些数据重新装配并打印出来后，可看到下面这样的结果（对象在每次运行时都会处在不同的内存位置，所以每次

运行的结果有区别）：

```
animals: [Bosco the dog[Animal@1cc76c], House@1cc769
, Ralph the hamster[Animal@1cc76d], House@1cc769
, Fronk the cat[Animal@1cc76e], House@1cc769
]
animals1: [Bosco the dog[Animal@1cca0c], House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals2: [Bosco the dog[Animal@1cca0c], House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals3: [Bosco the dog[Animal@1cca52], House@1cca5c
, Ralph the hamster[Animal@1cca5d], House@1cca5c
, Fronk the cat[Animal@1cca61], House@1cca5c
]
```

当然，我们希望装配好的对象有与原来不同的地址。但注意在 animals1 和 animals2 中出现了相同的地址，其中包括共享的、对 House 对象的引用。在另一方面，当 animals3 恢复以后，系统没有办法知道另一个流内的对象是第一个流内对象的化身，所以会产生一个完全不同的“对象之网”。

只要将所有东西都序列化到单独一个数据流里，就能恢复获得与以前写入时完全一样的对象网，不会由于不慎而造成对象的重复。当然，在写第一个和最后一个对象的时间之间，可改变对象的状态，但那必须由我们明确采取操作——序列化时，对象会采用它们当时的任何状态（包括它们与其他对象的连接关系）写入。

若想保存系统状态，最安全的做法是作为一种“微观”操作进行序列化。如果序列化了某些东西，再去做其他一些工作，再来序列化更多的东西，以此类推，那么最终将无法安全地保存系统状态。相反，应将构成系统状态的所有对象都置入单个容器内，并在一次操作里完成那个容器的写入。这样一来，同样只需一次方法调用，即可成功恢复之。

下面这个例子是一套假想的计算机辅助设计（CAD）系统，对这一方法进行了很好的演示。此外，它还为我们引入了 static 字段的问题——如留意联机文档，就会发现 Class 是“Serializable”（可序列化）的，所以只需简单地序列化 Class 对象，就能实现 static 字段的保存。这无论如何都是一种明智的做法。

```
//: c11:CADState.java
// Saving and restoring the state of a
// pretend CAD system.
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;
  abstract public void setColor(int newColor);
  abstract public 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 {
    ArrayList 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 = (ArrayList)in.readObject();
      Line.deserializeStaticState(in);
      shapes = (ArrayList)in.readObject();
    }
    // Display the shapes:
    System.out.println(shapes);
  }
} ///:~
```

Shape（几何形状）类“实现了可序列化”（implements Serializable），所以从 Shape 继承的任何东西也都会自动“可序列化”。每个 Shape 都包含了数据，而且每个派生的 Shape 类都包含了一个特殊的 static 字段，用于决定所有那些类型的 Shape 的颜色（如将一个 static 字段置入基类，结果只会产生一个字段，因为 static 字段未在派生类中复制）。可对基类中的方法进行覆盖处理，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，它都会创建一个不同的 Shape（Shape 数据采用随机值）。

Circle（圆）和 Square（矩形）属于对 Shape 的直接扩展；唯一的差别是 Circle 在定义时会初始化颜色，而 Square 在构造函数中初始化。Line（直线）的问题将留到以后讨论。

在 main()中，一个 ArrayList 用于容纳 Class 对象，另一个用于容纳形状。若不提供相应的命令行参数，就会创建 shapeTypes ArrayList，并添加 Class 对象，再创建 shapes ArrayList，然后添加 Shape 对象。接下来，所有 static color 值都会设成 GREEN，而且所有东西都会序

列化到 CADState.out 这个文件中。

若提供了一个命令行参数（比如 CADState.out），便会打开那个文件，并用它恢复程序的状态。无论在哪种情况下，Shape 的 ArrayList 最终都会打印出来。下面列出它某一次运行的结果：

```
>java CADState
[class Circle color[3] xPos[-51] yPos[-99] dim[38]
, class Square color[3] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
, class Circle color[3] xPos[-70] yPos[1] dim[16]
, class Square color[3] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[3] xPos[-75] yPos[-43] dim[22]
, class Square color[3] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[3] xPos[17] yPos[90] dim[-76]
]

>java CADState CADState.out
[class Circle color[1] xPos[-51] yPos[-99] dim[38]
, class Square color[0] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
, class Circle color[1] xPos[-70] yPos[1] dim[16]
, class Square color[0] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[1] xPos[-75] yPos[-43] dim[22]
, class Square color[0] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[1] xPos[17] yPos[90] dim[-76]
]
```

从中可以看出，xPos、yPos 以及 dim 的值都已成功地保存和恢复了。但在获取 static 信息时却出现了问题。所有“3”都已进入，但没有正常地出来。Circle 有一个 1 值（定义为 RED），而 Square 有一个 0 值（记住，它们是在构造函数里初始化的）。看上去似乎 static 根本没得到初始化！实情正是如此——尽管 Class 类是“可序列化的”，但却不能象我们希望的那样工作。所以假如想序列化 static 值，必须亲自动手。

这正是 Line 中的 serializeStaticState()和 deserializeStaticState()两个 static 方法的用途。可以看到，这两个方法都是作为存储和恢复进程的一部分明确调用的（注意写入序列化文件和从中读回的顺序不能改变）。所以为了使 CADState.java 正确运行起来，必须采用下述三种方法之一：

(1) 为几何形状添加一个 serializeStaticState()和 deserializeStaticState()。

(2) 删除 Vector shapeTypes 以及与之有关的所有代码

(3) 在几何形状内添加对新序列化和撤消序列化静态方法的调用

要注意的另一个问题是安全，因为序列化处理也会将 private 数据保存下来。若有需要

保密的字段，应将其标记成 transient。但在这之后，必须设计一种安全的信息保存方法。这样一来，一旦需要恢复，就可以重设那些 private 变量。

11.10　记号式输入

“记号式”（Tokenizing）是指将一个字符序列分割成一系列“记号”（Token）。这些“记号”实际是一些文字，中间可用你选择的任何东西加以分隔。例如，你的记号可以是一些句子，中间用空格和标点符号分隔。在标准 Java 库中，有两个类可供我们进行记号式输入，它们分别是 StreamTokenizer 和 StringTokenizer。

11.10.1　StreamTokenizer

尽管 StreamTokenizer 并不是从 InputStream 或 OutputStream 派生的，但它只能用来操作 InputStream 对象，所以理所当然地被划分到这个库的 I/O 部分。

下面是一个简单的程序，用来计算单词在一个文本文件中重复出现的次数：

```
//: c11:WordCount.java
// Counts words from a file, outputs
// results in sorted form.
import java.io.*;
import java.util.*;

class Counter {
  private int i = 1;
  int read() { return i; }
  void increment() { i++; }
}

public class WordCount {
  private FileReader file;
  private StreamTokenizer st;
  // A TreeMap keeps keys in sorted order:
  private TreeMap counts = new TreeMap();
  WordCount(String filename)
    throws FileNotFoundException {
    try {
      file = new FileReader(filename);
      st = new StreamTokenizer(
        new BufferedReader(file));
      st.ordinaryChar('.');
      st.ordinaryChar('-');
    } catch(FileNotFoundException e) {
      System.err.println(
        "Could not open " + filename);
      throw e;
```

```
    }
  }
  void cleanup() {
    try {
      file.close();
    } catch(IOException e) {
      System.err.println(
        "file.close() unsuccessful");
    }
  }
  void countWords() {
    try {
      while(st.nextToken() !=
        StreamTokenizer.TT_EOF) {
        String s;
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = new String("EOL");
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = st.sval; // Already a String
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
        }
        if(counts.containsKey(s))
          ((Counter)counts.get(s)).increment();
        else
          counts.put(s, new Counter());
      }
    } catch(IOException e) {
      System.err.println(
        "st.nextToken() unsuccessful");
    }
  }
  Collection values() {
    return counts.values();
  }
  Set keySet() { return counts.keySet(); }
  Counter getCounter(String s) {
    return (Counter)counts.get(s);
```

```
  }
  public static void main(String[] args)
  throws FileNotFoundException {
    WordCount wc =
      new WordCount(args[0]);
    wc.countWords();
    Iterator keys = wc.keySet().iterator();
    while(keys.hasNext()) {
      String key = (String)keys.next();
      System.out.println(key + ": "
              + wc.getCounter(key).read());
    }
    wc.cleanup();
  }
} ///:~
```

要想让单词排好序显示出来，最简单的做法就是将数据保存到一个 TreeMap 里。后者会将自己的“键”自动保持在排序状态（参见第 9 章）。用 keySet()取得一系列键之后，它们也会采用排序顺序。

为了打开文件，我们使用了一个FileReader。而为了将文件转换成单词，我们将FileReader封装到一个 BufferedReader 内部，从而创建了一个 StreamTokenizer。在 StreamTokenizer 中，有一个默认的分隔符列表，我们可用一系列方法加入更多的分隔符。在这里，我们用 ordinaryChar()指出“该字符没有特别重要的意义”，所以解析器不会把它当作自己创建的任何单词的一部分。例如，st.ordinaryChar('.')表示小数点（句点）不会成为解析出来的单词的一部分。在 JDK HTML 文档中（请从 java.sun.com 下载），大家可以找到更多的信息。

在 countWords()中，每次从数据流中取出一个记号，而 ttype 信息的作用是判断该对每个记号采取什么操作——因为记号可能代表一个行尾、一个数字、一个字串或者一个字符。

找到一个记号后，会查询 TreeMap counts，核实其中是否已经以“键”的形式包含了那个记号。若答案是肯定的，对应的 Counter（计数器）对象就会增值，指出已找到该单词的另一个实例。若答案为否，就新建一个 Counter——由于 Counter 构造函数会将它的值初始化为 1，所以也相当于“该单词出现了一次”。

WordCount 并不属于 TreeMap 的一种类型，所以它不会继承。它执行的是一种特定类型的操作，所以尽管 keys()和 values()方法都必须重新揭示出来，但仍不表示就应该继承，因为大量 TreeMap 方法在这里都是毫无用处的，继承过来纯属浪费。除此以外，对另一些方法来说（比如 getCounter()——用于获得一个特定字串的计数器；又如 sortedKeys()——用于产生一个 Iterator），它们联合在一起，最终都改变了 WordCount 接口的形式。

在 main()内，我们用 WordCount 打开和计算文件中的单词数量——总共只用了两行代码。随后，我们提取出用于一个排好序的键列表的“迭代器”（Iterator），用它取出每个键以及对应的 Count（计数）。注意必须调用 cleanup()，否则文件不能正常关闭。

11.10.2 StringTokenizer

尽管并不必属于 I/O 库的一部分，但 StringTokenizer 提供了与 StreamTokenizer 极其相似的功能，所以在这里一并讲述。

StringTokenizer 的作用是每次返回字串内的一个记号。这些记号是一些由制表位、空格

以及换行分隔的连续字符。因此，字串“Where is my cat?”的记号分别是“Where”、“is”、“my”和“cat?”。与 StreamTokenizer 类似，我们可以指示 StringTokenizer 按我们的愿望对输入进行分割。但对于 StringTokenizer，却需要向构造函数传递另一个参数，即我们想使用的分隔字串。通常，如果想进行更复杂的操作，应使用 StreamTokenizer。

可用 nextToken()向 StringTokenizer 对象请求字串内的下一个记号。该方法要么返回一个记号，要么返回一个空字串（表示没有记号剩下）。

作为一个例子，下述程序将执行一个有限的句法分析，查询其中的关键短语，了解句子暗示的是快乐还是悲伤的含义。

```
//: c11:AnalyzeSentence.java
// Look for particular sequences in sentences.
import java.util.*;

public class AnalyzeSentence {
  public static void main(String[] args) {
    analyze("I am happy about this");
    analyze("I am not happy about this");
    analyze("I am not! I am happy");
    analyze("I am sad about this");
    analyze("I am not sad about this");
    analyze("I am not! I am sad");
    analyze("Are you happy about this?");
    analyze("Are you sad about this?");
    analyze("It's you! I am happy");
    analyze("It's you! I am sad");
  }
  static StringTokenizer st;
  static void analyze(String s) {
    prt("\nnew sentence >> " + s);
    boolean sad = false;
    st = new StringTokenizer(s);
    while (st.hasMoreTokens()) {
      String token = next();
      // Look until you find one of the
      // two starting tokens:
      if(!token.equals("I") &&
         !token.equals("Are"))
        continue; // Top of while loop
      if(token.equals("I")) {
        String tk2 = next();
        if(!tk2.equals("am")) // Must be after I
          break; // Out of while loop
        else {
          String tk3 = next();
```

```
        if(tk3.equals("sad")) {
          sad = true;
          break; // Out of while loop
        }
        if (tk3.equals("not")) {
          String tk4 = next();
          if(tk4.equals("sad"))
            break; // Leave sad false
          if(tk4.equals("happy")) {
            sad = true;
            break;
          }
        }
      }
    }
    if(token.equals("Are")) {
      String tk2 = next();
      if(!tk2.equals("you"))
        break; // Must be after Are
      String tk3 = next();
      if(tk3.equals("sad"))
        sad = true;
      break; // Out of while loop
    }
  }
  if(sad) prt("Sad detected");
 }
 static String next() {
  if(st.hasMoreTokens()) {
    String s = st.nextToken();
    prt(s);
    return s;
  }
  else
    return "";
 }
 static void prt(String s) {
  System.out.println(s);
 }
} ///:~
```

对于准备分析的每个字串，我们进入一个 while 循环，并将记号从那个字串中取出。请注意第一个 if 语句，假如记号既不是“I”，也不是“Are”，就会执行 continue（返回循环起点，再一次开始）。这意味着除非发现一个“I”或者“Are”，才会真正得到记号。大家可能

想用==代替 equals()方法，但那样做会出现不正常的表现，因为==比较的是引用值，而 equals()比较的是实际内容。

analyze()方法剩余的逻辑是搜索“I am sad”（我很忧伤）、“I am not happy”（我不快乐）或者“Are you sad?”（你悲伤吗？）这样的句式。若没有 break 语句，这方面的代码甚至可能更加散乱。大家应注意对一个典型的解析器来说，通常都有这些记号的一个表格，并能在读取新记号的时候用一小段代码在表格内移动。

无论如何，只应将 StringTokenizer 看作 StreamTokenizer 一种简单而且特殊的简化形式。然而，如果有一个字串需要进行记号处理，而且 StringTokenizer 的功能实在有限，那么应该做的全部事情就是用 StringBufferInputStream 将其转换到一个数据流里，再用它创建一个功能更为强大的 StreamTokenizer。

11.10.3 检查大小写样式

在这一节里，我们打算来看看 Java I/O 的一个综合运用实例。在这个例子中，我们也运用了前述的“记号”技术。请注意，这其实是一个非常有用的项目，你马上就可把它投入自己的实用——它的作用是执行样式检查，保证你的大小写符合 Java 的正式规范（详情见 java.sun.com/docs/codeconv/index.html。它会打开当前目录下的所有.java 文件，从中提取出所有类名和标识符。如发现与 Java 样式相悖的情况，就立即向你指出。

为使程序正常工作，首先必须构建一个类名仓库，用它容纳标准 Java 库内的所有类名。要达到这个目的，你需要进入标准 Java 库的所有源码子目录，并在每个子目录下运行 ClassScanner。以参数（参数）的形式，提供仓库文件的名字（每次都用相同的路径和名字），同时加上-a 命令行选项，指出类名应添加到仓库里。

要用该程序检查你的代码，请为它提供要使用的目标仓库的路径和名字。随后，它会检查当前目录下的所有类和标识符。如发现与 Java 的典型大小写样式不相符的情况，就立即向你指出。

要注意的是，这个程序远未至完美境界；它有些时候会发出“误报”，等你仔细检查代码之后，却发现根本没什么要修改的。这当然让人有点儿恼火，但要想找出与规范不符的所有地方，比起你自己从头到尾检查代码，它依然要省事许多！

```
//: c11:ClassScanner.java
// Scans all files in directory for classes
// and identifiers, to check capitalization.
// Assumes properly compiling code listings.
// Doesn't do everything right, but is a
// useful aid.
import java.io.*;
import java.util.*;

class MultiStringMap extends HashMap {
  public void add(String key, String value) {
    if(!containsKey(key))
      put(key, new ArrayList());
    ((ArrayList)get(key)).add(value);
  }
  public ArrayList getArrayList(String key) {
```

```
    if(!containsKey(key)) {
      System.err.println(
        "ERROR: can't find key: " + key);
      System.exit(1);
    }
    return (ArrayList)get(key);
  }
  public void printValues(PrintStream p) {
    Iterator k = keySet().iterator();
    while(k.hasNext()) {
      String oneKey = (String)k.next();
      ArrayList val = getArrayList(oneKey);
      for(int i = 0; i < val.size(); i++)
        p.println((String)val.get(i));
    }
  }
}

public class ClassScanner {
  private File path;
  private String[] fileList;
  private Properties classes = new Properties();
  private MultiStringMap
    classMap = new MultiStringMap(),
    identMap = new MultiStringMap();
  private StreamTokenizer in;
  public ClassScanner() throws IOException {
    path = new File(".");
    fileList = path.list(new JavaFilter());
    for(int i = 0; i < fileList.length; i++) {
      System.out.println(fileList[i]);
      try {
        scanListing(fileList[i]);
      } catch(FileNotFoundException e) {
        System.err.println("Could not open " +
          fileList[i]);
      }
    }
  }
  void scanListing(String fname)
  throws IOException {
    in = new StreamTokenizer(
        new BufferedReader(
          new FileReader(fname)));
```

```
    // Doesn't seem to work:
    // in.slashStarComments(true);
    // in.slashSlashComments(true);
    in.ordinaryChar('/');
    in.ordinaryChar('.');
    in.wordChars('_', '_');
    in.eolIsSignificant(true);
    while(in.nextToken() !=
         StreamTokenizer.TT_EOF) {
      if(in.ttype == '/')
        eatComments();
      else if(in.ttype ==
             StreamTokenizer.TT_WORD) {
        if(in.sval.equals("class") ||
           in.sval.equals("interface")) {
          // Get class name:
            while(in.nextToken() !=
                 StreamTokenizer.TT_EOF
                 && in.ttype !=
                 StreamTokenizer.TT_WORD)
              ;
            classes.put(in.sval, in.sval);
            classMap.add(fname, in.sval);
        }
        if(in.sval.equals("import") ||
           in.sval.equals("package"))
          discardLine();
        else // It's an identifier or keyword
          identMap.add(fname, in.sval);
      }
    }
  }
  void discardLine() throws IOException {
    while(in.nextToken() !=
         StreamTokenizer.TT_EOF
         && in.ttype !=
         StreamTokenizer.TT_EOL)
      ; // Throw away tokens to end of line
  }
  // StreamTokenizer's comment removal seemed
  // to be broken. This extracts them:
  void eatComments() throws IOException {
    if(in.nextToken() !=
       StreamTokenizer.TT_EOF) {
```

```
    if(in.ttype == '/')
      discardLine();
    else if(in.ttype != '*')
      in.pushBack();
    else
      while(true) {
        if(in.nextToken() ==
          StreamTokenizer.TT_EOF)
          break;
        if(in.ttype == '*')
          if(in.nextToken() !=
            StreamTokenizer.TT_EOF
            && in.ttype == '/')
            break;
      }
  }
}
public String[] classNames() {
  String[] result = new String[classes.size()];
  Iterator e = classes.keySet().iterator();
  int i = 0;
  while(e.hasNext())
    result[i++] = (String)e.next();
  return result;
}
public void checkClassNames() {
  Iterator files = classMap.keySet().iterator();
  while(files.hasNext()) {
    String file = (String)files.next();
    ArrayList cls = classMap.getArrayList(file);
    for(int i = 0; i < cls.size(); i++) {
      String className = (String)cls.get(i);
      if(Character.isLowerCase(
          className.charAt(0)))
        System.out.println(
          "class capitalization error, file: "
          + file + ", class: "
          + className);
    }
  }
}
public void checkIdentNames() {
  Iterator files = identMap.keySet().iterator();
  ArrayList reportSet = new ArrayList();
```

```
    while(files.hasNext()) {
      String file = (String)files.next();
      ArrayList ids = identMap.getArrayList(file);
      for(int i = 0; i < ids.size(); i++) {
        String id = (String)ids.get(i);
        if(!classes.contains(id)) {
          // Ignore identifiers of length 3 or
          // longer that are all uppercase
          // (probably static final values):
          if(id.length() >= 3 &&
             id.equals(
               id.toUpperCase()))
            continue;
          // Check to see if first char is upper:
          if(Character.isUpperCase(id.charAt(0))){
            if(reportSet.indexOf(file + id)
                == -1){ // Not reported yet
              reportSet.add(file + id);
              System.out.println(
                "Ident capitalization error in:"
                + file + ", ident: " + id);
            }
          }
        }
      }
    }
  static final String usage =
    "Usage: \n" +
    "ClassScanner classnames -a\n" +
    "\tAdds all the class names in this \n" +
    "\tdirectory to the repository file \n" +
    "\tcalled 'classnames'\n" +
    "ClassScanner classnames\n" +
    "\tChecks all the java files in this \n" +
    "\tdirectory for capitalization errors, \n" +
    "\tusing the repository file 'classnames'";
  private static void usage() {
    System.err.println(usage);
    System.exit(1);
  }
  public static void main(String[] args)
  throws IOException {
    if(args.length < 1 || args.length > 2)
```

```
      usage();
    ClassScanner c = new ClassScanner();
    File old = new File(args[0]);
    if(old.exists()) {
      try {
        // Try to open an existing
        // properties file:
        InputStream oldlist =
          new BufferedInputStream(
            new FileInputStream(old));
        c.classes.load(oldlist);
        oldlist.close();
      } catch(IOException e) {
        System.err.println("Could not open "
          + old + " for reading");
        System.exit(1);
      }
    }
    if(args.length == 1) {
      c.checkClassNames();
      c.checkIdentNames();
    }
    // Write the class names to a repository:
    if(args.length == 2) {
      if(!args[1].equals("-a"))
        usage();
      try {
        BufferedOutputStream out =
          new BufferedOutputStream(
            new FileOutputStream(args[0]));
        c.classes.store(out,
          "Classes found by ClassScanner.java");
        out.close();
      } catch(IOException e) {
        System.err.println(
          "Could not write " + args[0]);
        System.exit(1);
      }
    }
  }
}

class JavaFilter implements FilenameFilter {
  public boolean accept(File dir, String name) {
```

```
    // Strip path information:
    String f = new File(name).getName();
    return f.trim().endsWith(".java");
  }
} ///:~
```

MultiStringMap 这个类是一个特殊的工具，允许我们将一组字串映射（对应）至每个键条目。它使用了一个 HashMap（这次是继承来的），其中的键作为单个字串映射至 ArrayList 值。add()方法的作用很简单，它会检查 HashMap 中是否已经有一个键。如果还没有，就在其中添加一个。getArrayList()方法为一个特定的键产生一个 ArrayList；printValues()则主要用于程序调试——一个 ArrayList 接一个 ArrayList，打印出所有的值。

为使问题简化，标准 Java 库的类名全部放到一个 Properties 对象里（来自标准 Java 库）。要记住的是，Properties 对象其实是一个 HashMap，其中只容纳了用于键和值条目的 String 对象。不过，只需一个方法调用，便可把它保存到磁盘，再从磁盘上恢复。因此，作为那些类名的一个“仓库”，它无疑是最佳之选。实际上，我们只需要一个名字列表，而 HashMap 无论它的键还是值都不能接受 null 值。所以，我们将相同的对象同时用于键和值。

针对为一个特定目录下的文件而发现的类和标识符，我们使用了两个 MultiStringMap——classMap 和 identMap。另外，程序启动时，会将标准类名仓库载入名为 classes 的一个 Properties 对象；而且在本地目录中发现一个新类名时，该类名除了会添加到 classMap 之外，还会添加到 classes 中。这样一来，我们就可用 classMap“遍历”本地目录下的所有类，而 classes 可用于检查当前“记号”是否为一个类名（它标志着一个对象或方法定义的开始，所以接着提取下一个记号——直到碰到一个分号为止——再将它们全部放进 identMap 中）。

对 ClassScanner 来说，它的默认构造函数会创建一个由文件名构成的列表。这是通过 FilenameFilter 的 JavaFilter 实现来做到的（见文件末尾）。随后，针对每个文件名，它都会调用 scanListing()方法。

在 scanListing()方法内，我们打开源码文件，并将其转变成一个 StreamTokenizer。按 JDK 用户文档的提示，假如将“true”值传递给 slashStarComments()和 slashSlashComments()，那么“应该”能将那些注释内容剥除掉。但实际情况并非如此。相反，那些行会被当作注释标记出来，但注释内容本身却要由另一个方法提取。要做到这一点，“/”这个字符必须作为一个原始的字符捕获下来，而不能让 StreamTokenizer 把它当作一条注释的构成部分看待。此时，我们要用 ordinaryChar()来指示 StreamTokenizer 采取正确的操作。点号（“.”）同样如此，因为我们想让方法调用对不同的标识符进行区别。然而，对下划线（“_”）来说，它最初被 StreamTokenizer 当作一个独立的字符看待，但实际上应该作为标识符的一部分使用，因为它会在“TT_EOF”这样的 static final 值中出现（这属于本程序的一个特殊情况）。wordChars()方法需要取得我们想添加的一系列字符，在一个“记号”中，那些字符将被统一解析成一个完整的单词。最后，在解析单行注释或打算丢弃一个行的时候，我们需要知道“行末”（EOL）到底在哪里。所以，通过调用 eolIsSignificant(true)，那个 EOL 就会显示出来，而不是被 StreamTokenizer 硬生生地“吸收”掉了。

scanListing()方法剩余的部分会读取记号，并对其作出恰当反应，直至文件末尾——nextToken()返回一个为“streamTokenizer.TT_EOF”的 final static 值时，便知道已到达文件末尾了！

假如提取到的一个记号是“/”，那么由于它有可能是一条注释，所以会调用 eatComments()，进行分析判断。当然，可能有其他大量特殊情况都会出现“/”，但在这里，我们唯一关心的就是它到底是不是一个单词。

假如取回来的单词是“class”或“interface”，意味着下一个记号表示的是一个类或接口名，所以把它同时放到 classes 和 classMap 里。假如单词是“import”或“package”，那便不用再关心该行剩下的内容了。其他任何东西都肯定是一个标识符（这是我们感兴趣的），或者是一个关键字（对这个我们没兴趣，但它们无论如何都是全部小写的，所以不会影响程序结果）。它们都被添加到 identMap 里。

discardLine()是一个简单的方法，用于搜寻行尾。注意每次取得一个新记号时，都必须检查是否到了文件末尾。

在主解析循环内，只要碰到了一个正斜杠（“/”），就会调用 eatComments()方法。不过，这并不一定肯定发现了一条注释。因此，还必须接着提取下一个记号，看它是否同样为“/”（如果是，就将整行丢弃）或者是不是一个星号（“*”）。但假如既不是/，也不是*，那么刚才取出的那个记号就应该送还给主解析循环！幸运的是，pushBack()方法允许我们将当前记号“送还”到输入流里，所以在主解析循环调用 nextToken()的时候，得到的实际就是刚才还给它的那一个。

为方便起见，classNames()方法会产生一个数组，其中包含了 classes 容器中的所有名字。该方法并不在程序中使用，只是方便我们进行调试。

接下来的两个方法是我们进行实际检查的地方。在 checkClassNames()中，我们将类名从 classMap 中提取出来（记住，classMap 只包含了该目录下的名字，由于是按文件名组织的，所以文件名可能随错误的类名一道打印出来）。要达到这个目的，需要取得每一个对应的 ArrayList，然后挨个进行检查，看它们的第一个字符是否为小写。如果是，就打印出恰当的错误提示消息。

在 checkIdentNames()中，我们采取了类似的做法：从 identMap 中提取出每个标识符名字。假如名字不在 classes 列表中，就假定它是一个标识符或者关键字。这时还会检查一种特殊的情况：假如标识符的长度是 3 或者以上，而且所有字符都是大写的，那么就忽略该标识符，因为它可能是一个 static final 值（如 TT_EOF）。当然，这并非一种特别完美的算法，但它起码让你注意到所有全大写的标识符在这里都是不适合的。

该方法并不是报告出以一个大写字符开头的所有标识符，而是跟踪那些已在一个名为 reportSet()的 ArrayList 中报告的。这样便可将 ArrayList 当作一个“Set”对待，从而判断一个项目是否已经存在于这个 Set 中。项目是通过将文件名和标识符连接到一起而生成的。假如元素不在 Set 里，便增添之，然后作出报告。

列表剩下的部分便是 main()，它会处理命令行参数，判断我们是打算根据标准 Java 库构建一个类名仓库呢，还是打算对已写的代码进行有效性检查。不过不管在哪种情况下，都会产生一个 ClassScanner 对象。

无论是构建还是使用一个仓库，都必须试着先将现有的仓库打开。通过生成 File 对象，并测试是否存在，便可决定是否需要打开文件，并载入（load()）位于 ClassScanner 内的、名为“classes”的一个 Properties 列表（来自仓库的类会添加到——而不是覆盖到——由 ClassScanner 构造函数发现的类上）。假如只提供了一个命令行参数，意味着你想对类名和标识符名字执行一次检查。但假如同时提供了两个参数（第二个是“-a”），便表明你想构建一个类名仓库。在后一种情况下，会打开一个输出文件，并用 Properties.save()方法将列表写到一个文件里，同时写入的还有一个字串，用于提供标题文件信息。

11.11　总　　结

Java I/O 流库能满足我们的许多基本要求：可通过控制台、文件、内存块甚至 Internet（参见第 15 章）进行读写。通过继承，可创建新的输入和输出对象类型。而且大家都知道，

假如一个方法原本期望接收的是一个 String，但却向其传递了一个对象，那么会自动调用 toString()方法（这是由 Java 提供的、一个限制版本的“自动类型转换”）。因此，通过对 toString() 的重新定义，甚至可对一个流能够接收的对象种类进行少许扩展。

不过就 I/O 流库的联机文档和设计来说，仍有些问题没有解决。比如当我们打开一个文件以便输出时，完全可以指定一旦有人试图覆盖该文件就“掷”出一个违例——有的编程系统允许我们自行指定想打开一个输出文件，但唯一的前提是它尚不存在。但在 Java 中，似乎必须用一个 File 对象来判断某个文件是否存在，因为假如将其作为 FileOutputStream 或者 FileWriter 打开，那么肯定会被覆盖。

I/O 流库易使我们混淆一些概念。它确实能做许多事情，而且也可以在不同的平台间移植。但假如假如事先没有吃透“装饰器”范式的概念，那么所有的设计都多少带有一点盲目性质。所以不管学它还是教它，都要特别花一番功夫才行。而且它并不完整——没有提供对输出格式化的支持，而其他几乎所有语言的 I/O 模块都提供了这方面的支持。

然而，一旦掌握了装饰器范式，并开始在一些迫切需要它所具有的灵活性的场合下使用这个库，就会马上认识到这种设计的好处。到那个时候，为此多花功夫写的代码行应该不至于使你感觉太生气！

11.12 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(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 里。按相反顺序打印出 LinkedList 中的所有行。

(2) 修改练习 1，以命令行参数的形式，提供要读取的那个文件的名字。

(3) 修改练习 2，同时打开一个文本文件，以便将文字写入其中。随行号一起（不要试图使用“LineNumber”类），将 ArrayList 中的行写到文件里。

(4) 修改练习 2，强迫 ArrayList 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，以又一个命令行参数的形式，指定要在文件里搜索的单词。打印出包含了指定单词的所有行。

(6) 修改 DirList.java，使 FilenameFilter 实际性地打开每一个文件，并根据是否有任何命令行参数存在于那个文件中，来接收文件。

(7) 创建一个名为 SortedDirList 的类，让它的构造函数取得文件路径信息，并根据那个路径中的文件，构建出一个排好序的目录列表。请创建两个重载的 list()方法，它们应该能根据参数，决定是产生完整的列表，还是产生列表的一个子集。同时添加一个 add()方法，令其取得一个文件名，并计算出那个文件的长度。

(8) 修改 WordCount.java，使其换用字母排序（利用由第 9 章提供的工具）。

(9) 修改 WordCount.java，使其使用包含了一个字串和一个计数值的类，以保存每个不同的单词。同时使用由这些对象构成的一个 Set，以维持单词列表。

(10) 修改 IOStreamDemo.java，使其利用 LineNumberInputStream 来跟踪行的计数。要注意的是，以程序化的手段进行这样的跟踪要容易得多。

(11) 以 IOStreamDemo.java 的第 4 节为基础写一个程序，用它比较通过缓冲式和非缓冲式 I/O，向文件内写入数据时的性能差异。

(12) 修改 IOStreamDemo.java 的第 5 节，删除行内由于第一次对 in5br.readLine()的调用而产生的空格。请用 while 循环和 readChar()完成。

(13) 根据正文说明，修正 CADState.java 程序。

(14)在 Blips.java 中，复制文件并把它重命名为 BlipCheck.java。然后将 Blip2 类重命名为 BlipCheck（同时将其标记为 public，并从 Blips 类中删除公共作用域）。删除文件中的//!记号，并执行程序。接下来，将 BlipCheck 的默认构造函数变成注释行。运行它，并解释为什么仍然能够工作。注意在编译之后，必须用“java Blips”命令来运行它，因为 main()方法仍在 Blips 类里。

(15) 在 Blip3.java 中，将“You must do this:”字样后的两行变成注释，然后运行程序。解释结果为什么会与执行了那两行代码之后不同。

(16) 在第 8 章中间部分找到那个 GreenhouseControls.java 例子，它应该由三个文件构成。在 GreenhouseControls.java 中，Restart()内部类有一个硬编码的事件集。请修改这个程序，使其能从一个文本文件里动态读取事件以及它们的对应时间（另外可选做：用一个设计范式 Factory 方法来构建事件——详见《Thinking inPatterns with Java》电子版，可从 www.BurceEckel.com 下载）。

第 12 章 运行时间类型鉴定

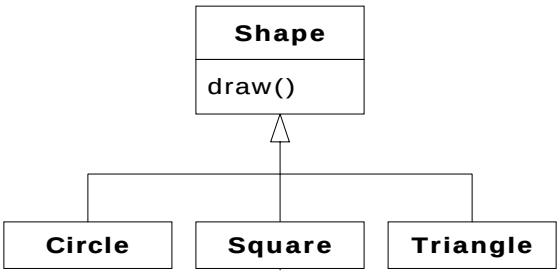
运行时间类型鉴定（RTTI）的概念乍看似乎颇为简单——在你只有对基类型的一个引用时，用它来判断一个对象的正确类型是什么。

然而，对 RTTI 的需要暴露出了面向对象设计许多有趣（而且经常令人困惑）的问题，并把程序的构造问题正式摆上了台面。

本章将讨论如何利用 Java 在运行时间查找与对象和类有关的信息。这主要采取两种形式：一种是“传统”RTTI，它假定我们已在编译和运行时间拥有所有类型；另一种是“反射”机制，利用它可在运行时间独立查找类信息。首先讨论“传统”RTTI，再来讨论反射问题。

12.1 对 RTTI 的需要

请考虑下面这个熟悉的类结构例子，它利用了多态。常规类型是基类 Shape，而专门派生出来的类型是 Circle、Square 和 Triangle：



这是一个典型的类结构示意图，基类位于顶部，派生类向下延展。面向对象编程的基本目标是用“大块”的代码来控制对基类型（这里是 Shape）的引用，所以假如以后决定添加一个新类（比如从 Shape 派生一个 Rhomboid），从而对程序进行扩展，那么根本不会影响到原来的“大块”代码。在这个例子中，Shape 接口中的动态绑定方法是 draw()，所以客户程序员要做的是通过一个标准 Shape 引用调用 draw()。draw()在所有派生类里都会被覆盖。而且由于它是一个动态绑定方法，所以即使通过一个普通的 Shape 引用调用它，也有表现出正确的行为。这正是“多态”的功劳！

所以，我们一般创建一个特定的对象（Circle、Square 或者 Triangle），把它向上强制转型成一个 Shape（从而忽略对象的特殊类型），以后便在程序的剩余部分使用匿名的 Shape 引用。

作为对多态和向上强制转型的一个简要回顾，可以象下面这样为上述例子编码：

```
//: c12:Shapes.java
import java.util.*;

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 {
  public static void main(String[] args) {
    ArrayList s = new ArrayList();
    s.add(new Circle());
    s.add(new Square());
    s.add(new Triangle());
    Iterator e = s.iterator();
    while(e.hasNext())
      ((Shape)e.next()).draw();
  }
} ///:~
```

基类包含了一个 draw()方法，它通过向 System.out.println()传递 this，从而间接地调用 toString()方法，为类打印出一个标识符。假如那个函数看到的是一个对象，便会自动调用 toString()方法，产生它的“字串”表示形式。

每个派生类都覆盖了 toString()方法（从 Object 中），所以 draw()在每种情况下打印出来的东西都有所不同。在 main()中，我们创建了特定类型的 Shape，然后把它们添加到一个 ArrayList 里。这个时候便会发生向上强制转型，因为 ArrayList 里只能容纳“对象”。由于 Java 的一切（原始数据类型除外）都是对象，所以 ArrayList 里也能容纳 Shape 对象。但在向上强制转型到 Object 的过程中，它也同时丢失了任何特殊的信息，其中包括“对象是 Shape”这一事实。在 ArrayList 的眼中，它们仅仅是“对象”而已，并不具有什么更特殊的特征！

用 next()从 ArrayList 里取出一个元素的时候，局面就变得稍微有点儿复杂了。由于 ArrayList 只能容纳对象，next()自然会产生一个对象引用。但我们都知道，它实际是一个 Shape 引用，而且我们想将 Shape 消息发给那个对象。因此，有必要使用传统的“(Shape)”强制转型来对 Shape 进行一次强制转型。这是 RTTI 最基本的一种形式，因为在 Java 中，运行时间会检查所有强制转型的正确性——而 RTTI 的含义正好就是：在运行时间，将对象的类型鉴定出来！

在目前这种情况下，RTTI 强制转型只实现了一部分：Object 强制转型成 Shape，而不是

干脆强制转型成 Circle、Square 或者 Triangle。那是因为我们目前能肯定的唯一事实就是 ArrayList 里充斥着各种 Shape，而不知道它们的具体类别。在编译时间，我们肯定的依据是我们自己制订的规则；而在运行时间，却是通过强制转型来肯定这一点。

现在的局面会交由“多态”进行控制，通过判断引用到底指向一个 Circle，指向一个 Square，还是指向一个 Triangle，从而决定该具体为 Shape 调用哪个方法。而且在一般情况下，这正是我们所希望的结果——希望自己的代码块尽可能少地掌握与对象具体类型有关的事情，而是把注意力放在对一个“对象家族”（这里就是 Shape 家族）的常规表示上。只有这样，我们的代码才更易编写、理解以及修改。所以说，“多态”是面向对象程序设计的一个常规目标！

然而，有时也会碰到一些特殊的程序设计问题，只有在知道常规引用的确切类型后，才能更容易地解决这类问题。这个时候又该怎么办呢？举个例子来说，我们有时候想允许用户用紫色突出显示某种类型的全部形状。这时便要用到 RTTI 技术，用它查询某个 Shape 引用所引用的准确类型是什么。

12.1.1 Class 对象

为理解 RTTI 在 Java 里如何工作，首先必须了解类型信息在运行时间是如何表示的。这时要用到一个名为“Class 对象”的特殊形式的对象，其中包含了与类有关的信息（有时也把它叫作“元类”）。事实上，我们要用 Class 对象创建属于某个类的全部“常规”或“普通”对象。

对应作为程序一部分的每个类，它们都有一个 Class 对象。换言之，每次写一个新类时，同时也会创建一个 Class 对象（更恰当地说，是保存在一个完全同名的.class 文件中）。在运行时间，一旦我们想生成那个类的一个对象，用于执行程序的 Java 虚拟机（JVM）首先就会检查那个类型的 Class 对象是否已经载入。若尚未载入，JVM 就会查找同名的.class 文件，并将其载入。所以 Java 程序启动时并不是完全载入的，这一点与许多传统语言都不同。

一旦那个类型的 Class 对象进入内存，就可用它创建那一类型的所有对象。

如果这种说法多少让你产生了一点儿迷惑，或者你并没有真正理解它，那么下面这个示范程序或许能提供进一步的帮助：

```
//: c12:SweetShop.java
// Examination of the way the class loader works.

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 {
  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) {
      e.printStackTrace(System.err);
    }
    System.out.println(
      "After Class.forName(\"Gum\")");
    new Cookie();
    System.out.println("After creating Cookie");
  }
} ///:~
```

对每个类来说（Candy、Gum 和 Cookie），它们都有一个 static 从句，用于在类首次载入时执行。相应的信息会打印出来，告诉我们载入是什么时候进行的。在 main()中，对象的创建代码位于打印语句之间，以便侦测载入时间。

特别有趣的一行是：

```
Class.forName("Gum");
```

该方法是 Class（所有 Class 对象都从属于它）的一个 static 成员。而 Class 对象和其他任何对象都是类似的，所以能够获取和控制它的一个引用（装载模块就是干这事儿的）。为获得 Class 的一个引用，一个办法是使用 forName()。它的作用是取得包含了目标类“文字名”的一个 String（注意拼写和大小写）。最后返回的是一个 Class 引用。

该程序在某个 JVM 中的输出如下：

```
inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie
```

可以看到，每个 Class 只有在它需要的时候才会载入，而 static 初始化工作是在类载入时执行的。

1. 类文字

在 Java 中，可采用第二种方式来产生对 Class 对象的引用——使用“类文字”（Class Literal）。对上述程序来说，看起来就象下面这样：

```
Gum.class;
```

这样做不仅更加简单，而且更安全，因为它会在编译时得到检查。由于它使我们不必进行方法调用，所以效率也更高。

“类文字”不仅可以应用于普通类，同时还可应用于接口、数组以及原始数据类型。除此以外，针对每种原始数据类型的封装器类，它还存在一个名为 TYPE 的标准字段。TYPE 字段的作用是为相关的原始数据类型产生 Class 对象的一个引用，如下所示：

… 等价于 …	
boolean.class	Boolean.TYPE
char.class	Character.TYPE
byte.class	Byte.TYPE
short.class	Short.TYPE
int.class	Integer.TYPE
long.class	Long.TYPE
float.class	Float.TYPE
double.class	Double.TYPE
void.class	Void.TYPE

我推荐你尽可能地使用“.class”版本，因为它和标准类更显一致。

12.1.2 强制转型前的检查

迄今为止，我们已知的 RTTI 形式包括：

(1) 传统强制转型，如“(Shape)”，它用 RTTI 确保强制转型的正确性，并在遇到一个失败的强制转型后产生一个 ClassCastException 违例。

(2) 代表对象类型的 Class 对象。可查询 Class 对象，获取有用的运行时间资料。

在 C++中，传统的“(Shape)”强制转型并不执行 RTTI。它只是简单地告诉编译器将对象当作新类型处理。而 Java 要执行类型检查，这通常叫作“类型安全”的向下强制转型。之所以叫“向下强制转型”，是由于类分级结构图的历史排布方式造成的。若将一个 Circle（圆）强制转型成一个 Shape（几何形状），就叫做向上强制转型，因为圆只是几何形状的一个子集。反之，若将 Shape 强制转型至 Circle，就叫做向下强制转型。然而，尽管我们能肯定 Circle 也是一个 Shape（所以编译器才能自动向上强制转型），但却不能肯定一个 Shape 就是一个 Circle。因此，编译器不允许自动向下强制转型，除非明确指定一次这样的强制转型。

RTTI 在 Java 中还存在着第三种形式。关键字 instanceof 告诉我们对象是不是一个特定类型的实例（Instance 即“实例”）。它会返回一个布尔值，以便我们采用问题的形式使用，就象下面这样：

```
if(x instanceof Dog)
  ((Dog)x).bark();
```

将 x 强制转型成一个 Dog 前，上面的 if 语句会检查对象 x 是否从属于 Dog 类。进行强制转型前，如果没有其他信息可以告诉自己对象的类型，那么 instanceof 的使用是非常重要的——否则会得到一个 ClassCastException 违例。

我们通常的做法是查找某种类型（比如要变成紫色的三角形），但利用 instanceof，却可以很轻松地标记出所有对象。现在，假定我们有一个 Pet（宠物）家族：

```
//: c12:Pets.java
class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

class Counter { int i; } ///:~
```

其中，Counter 类用于跟踪具体某种类型的 Pet 的数量。可将其想象成一个能够修改的整数。

利用 instanceof，所有宠物的数量都可以方便地计算出来：

```
//: c12:PetCount.java
// Using instanceof.
import java.util.*;

public class PetCount {
  static String[] typenames = {
    "Pet", "Dog", "Pug", "Cat",
    "Rodent", "Gerbil", "Hamster",
  };
  // Exceptions thrown out to console:
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    try {
      Class[] petTypes = {
        Class.forName("Dog"),
        Class.forName("Pug"),
        Class.forName("Cat"),
        Class.forName("Rodent"),
        Class.forName("Gerbil"),
        Class.forName("Hamster"),
      };
      for(int i = 0; i < 15; i++)
```

```
        pets.add(
          petTypes[
            (int)(Math.random()*petTypes.length)]
            .newInstance());
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw e;
    } catch(ClassNotFoundException e) {
      System.err.println("Cannot find class");
      throw e;
    }
    HashMap h = new HashMap();
    for(int i = 0; i < typenames.length; i++)
      h.put(typenames[i], new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.get(i);
      if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
      if(o instanceof Dog)
        ((Counter)h.get("Dog")).i++;
      if(o instanceof Pug)
        ((Counter)h.get("Pug")).i++;
      if(o instanceof Cat)
        ((Counter)h.get("Cat")).i++;
      if(o instanceof Rodent)
        ((Counter)h.get("Rodent")).i++;
      if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
      if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    for(int i = 0; i < typenames.length; i++)
      System.out.println(
        typenames[i] + " quantity: " +
        ((Counter)h.get(typenames[i])).i);
  }
} ///:~
```

在 Java 中，对 instanceof 作了一个限制——你只能将其与一个已命名的类型比较，不可

与 Class 对象对比。在上述例子中，大家可能觉得将所有那些 instanceof 表达式都写出来是件很麻烦的事情。实际情况正是这样。但在 Java 中，没办法让这一工作自动进行——不能创建由 Class 对象构成的一个 ArrayList，再将其与之比较（不过请准备好，马上就有一个替代办法要介绍给大家）。不过，这一限制和许多人想象的不同，它实际上并没有那么“厉害”，因为大家最终会意识到，假如编写了数量过多的 instanceof 表达式，那么整个设计都可能出现问题。

当然，这个例子只是一个构想——最好在每个类型里添加一个 static 数据成员，然后在构造函数中令其增值，以便跟踪计数。编写程序时，大家可能想象自己拥有类的源码控制权，能够自由改动它。但由于实际情况并非总是这样，所以 RTTI 显得特别方便。

1. 使用类文字

PetCount.java 示例可用“类文字”重写一遍。得到的结果显得更明确易懂：

```
//: c12:PetCount2.java
// Using class literals.
import java.util.*;

public class PetCount2 {
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    Class[] petTypes = {
      // Class literals:
      Pet.class,
      Dog.class,
      Pug.class,
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.add(
          petTypes[rnd].newInstance());
      }
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
```

```
      throw e;
    }
    HashMap h = new HashMap();
    for(int i = 0; i < petTypes.length; i++)
      h.put(petTypes[i].toString(),
        new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.get(i);
      if(o instanceof Pet)
        ((Counter)h.get("class Pet")).i++;
      if(o instanceof Dog)
        ((Counter)h.get("class Dog")).i++;
      if(o instanceof Pug)
        ((Counter)h.get("class Pug")).i++;
      if(o instanceof Cat)
        ((Counter)h.get("class Cat")).i++;
      if(o instanceof Rodent)
        ((Counter)h.get("class Rodent")).i++;
      if(o instanceof Gerbil)
        ((Counter)h.get("class Gerbil")).i++;
      if(o instanceof Hamster)
        ((Counter)h.get("class Hamster")).i++;
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    Iterator keys = h.keySet().iterator();
    while(keys.hasNext()) {
      String nm = (String)keys.next();
      Counter cnt = (Counter)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
    }
  }
} ///:~
```

在这里，typenames（类型名）数组已被删除，改为从 Class 对象里获取类型名称。注意系统可以自行区分类和接口。

也可以看到，petTypes 的创建不需要用一个 try 块包围起来，因为它会在编译时得到检查，不会象 Class.forName()那样“掷”出任何违例。

Pet 对象以动态方式创建好之后，可以看到随机数字已得到了限制——肯定在 1 和 petTypes.length 之间，而且不包括零。那是由于零代表的是 Pet.class，而一个标准的 Pet 对象事先已假定不会包括到其中。不过，由于 Pet.class 是 petTypes 的一部分，所以所有 Pet（宠物）都会算入计数中。

2. 动态的 instanceof

利用 Class isInstance 方法，我们可以动态调用 instanceof 运算符。因此，所有那些烦人的 instanceof 语句都可从 PetCount 例子中删去了（这就是前面提到的“替代办法”）。如下所示：

```
//: c12:PetCount3.java
// Using isInstance().
import java.util.*;

public class PetCount3 {
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    Class[] petTypes = {
      Pet.class,
      Dog.class,
      Pug.class,
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.add(
          petTypes[rnd].newInstance());
      }
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw e;
    }
    HashMap h = new HashMap();
    for(int i = 0; i < petTypes.length; i++)
      h.put(petTypes[i].toString(),
        new Counter());
    for(int i = 0; i < pets.size(); i++) {
```

```
      Object o = pets.get(i);
      // Using isInstance to eliminate individual
      // instanceof expressions:
      for (int j = 0; j < petTypes.length; ++j)
        if (petTypes[j].isInstance(o)) {
          String key = petTypes[j].toString();
          ((Counter)h.get(key)).i++;
        }
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    Iterator keys = h.keySet().iterator();
    while(keys.hasNext()) {
      String nm = (String)keys.next();
      Counter cnt = (Counter)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
    }
  }
} ///:~
```

可以看到，isInstance()方法消除了我们使用 instanceof 表达式的必要。此外，这也意味着以后一旦需要添加新型宠物，只需简单地改变 petTypes 数组即可；毋需改动程序剩余的部分（但假如用的是 instanceof 表达式，却必须那样做）。

3. instanceof 和 Class

查询类型信息时，instanceof 的两种形式（即 instanceof 或 isInstance()，两者产生的结果相同）与 Class 对象的直接比较方式之间，存在着一处重要的差异。下例对此进行了演示：

```
//: c12:FamilyVsExactType.java
// The difference between instanceof and class

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
  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());
  }
} ///:~
```

其中，test()方法用于执行类型检查，它的参数（参数）同时采用了 instanceof 的两种形式。随后，它取得 Class 引用，并用“==”和 equals()来测试 Class 对象的相等性。下面是输出：

```
Testing x of type class 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 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
```

这下令人放心了，instanceof 和 isInstance()产生了完全一致的结果；equals()和“==”也一样。但测试本身却得出了不同的结论。按照“类型”的概念，instanceof 的意思是问：“你是这个类，还是从这个类派生出去的另一个类？”而另一方面，假如用“==”来比较实际的 Class 对象，就丝毫不牵涉到继承的问题——不管是不是准确的类型，结果都是一样的。

12.2 RTTI 语法

Java 用 Class 对象实现自己的 RTTI 功能——即便我们要做的只是象强制转型那样的一些工作。Class 类也提供了其他大量方式，以方便我们使用 RTTI。

首先，必须获得指向适当 Class 对象的的一个引用。就象前例演示的那样，一个办法是用一个字串以及 Class.forName()方法。这是非常方便的，因为不需要那种类型的一个对象来获取 Class 引用。不过，对于自己感兴趣的类型，如果已有了它的一个对象，那么为了取得 Class 引用，可调用属于 Object 根类一部分的一个方法：getClass()。它的作用是返回一个特定的 Class 引用，用来表示对象的实际类型。Class 提供了几个有趣且较为有用的方法，从下例可以看出：

```
//: c12:ToyTest.java
// Testing class Class.

interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}
class Toy {
  // Comment out the following default
  // constructor to see
  // NoSuchMethodError from (*1*)
  Toy() {}
  Toy(int i) {}
}

class FancyToy extends Toy
    implements HasBatteries,
      Waterproof, ShootsThings {
  FancyToy() { super(1); }
}

public class ToyTest {
  public static void main(String[] args)
  throws Exception {
    Class c = null;
    try {
      c = Class.forName("FancyToy");
```

```
    } catch(ClassNotFoundException e) {
      System.err.println("Can't find FancyToy");
      throw e;
    }
    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.err.println("Cannot instantiate");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw e;
    }
    printInfo(o.getClass());
  }
  static void printInfo(Class cc) {
    System.out.println(
      "Class name: " + cc.getName() +
      " is interface? [" +
      cc.isInterface() + "]");
  }
} ///:~
```

从中可以看出，FancyToy 这个类相当复杂，因为它从 Toy 中继承，并实现了 HasBatteries、Waterproof 以及 ShootsThings 的接口。在 main()中创建了一个 Class 引用，并用位于相应 try 块内的 forName()初始化成 FancyToy。

Class.getInterfaces 方法会返回 Class 对象的一个数组，用于表示包含在 Class 对象内的接口。

若有一个 Class 对象，也可以用 getSuperclass()查询该对象的直接基类是什么。当然，这种做会返回一个 Class 引用，可用它作进一步的查询。这意味着在运行时间，完全有机会调查到一个对象的完整类层次结构。

若从表面看，Class 的 newInstance()方法似乎是克隆（clone()）一个对象的另一种手段。但两者是有区别的。利用 newInstance()，我们可在没有现成对象供“克隆”的情况下新建一个对象。就象上面的程序演示的那样，当时没有 Toy 对象，只有 cy——即 y 的 Class 对象的一个引用。利用它可以实现“虚拟构造函数”。换言之，我们的意思是：“尽管我不知道你的准确类型是什么，但请你无论如何都正确地创建自己。”在上例中，cy 只是一个 Class 引用，编译时并不知道进一步的类型信息。一旦新建了一个实例后，可以得到 Object 引用。但那

个引用指向一个 Toy 对象。当然，如果要将除 Object 能够接收的其他任何消息发出去，首先必须进行一些调查研究，并进行一些强制转型工作。除此以外，用 newInstance()创建的类必须有一个默认构造函数。在下一节中，大家会学到如何利用 Java 的“反射 API”，通过任意构造函数来动态创建类的对象。

程序中的最后一个方法是 printInfo()，它取得一个 Class 引用，通过 getName()获得它的名字，并用 isInterface()调查它是不是一个接口。

该程序的输出如下：

```
Class name: FancyToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
Class name: Toy is interface? [false]
```

所以，利用 Class 对象，我们几乎能将一个对象的祖宗十八代都调查出来。

12.3 反射：运行时间类信息

如果不知道一个对象的确切类型，RTTI 会帮我们调查。但却有一个限制：类型必须是在编译时间已知的，否则就没法子用 RTTI 调查它，进而无法开展下一步工作。换言之，编译器必须明确知道 RTTI 要处理的所有类。

从表面看，这似乎并不是一个很大的限制，但假若得到的是一个不在自己程序空间内的对象的引用，这时又会怎样呢？事实上，有的时候，对象的类即使在编译时间也不可由我们的程序使用。例如，假设我们从磁盘或者网络获得一系列字节，而且被告知那些字节代表一个类。由于编译器在编译代码时并不知道那个类的情况，所以怎样才能顺利地使用这个类呢？

在传统的程序设计环境中，出现这种情况的概率或许很小。但当我们转移到一个规模更大的编程世界中，却必须对这个问题加以高度重视。第一个要注意的是基于组件的程序设计。在这种环境下，我们用“快速应用开发”（RAD）模型来构建程序项目。RAD 一般是在应用程序构建工具中内建的。这是编制程序的一种可视途径（在屏幕上以“窗体”的形式出现）。可将代表不同组件的图标拖曳到窗体中。随后，通过设定这些组件的属性或者值，进行正确的配置。设计期间的配置要求任何组件都是可以“例示”的（即可以自由获得它们的实例）。这些组件也要揭示出自己的一部分内容，允许程序员读取和设置各种值。此外，用于控制 GUI 事件的组件必须揭示出与相应的方法有关的信息，以便 RAD 环境帮助程序员用自己的代码覆盖这些由事件驱动的方法。“反射”提供了一种特殊的机制，可以侦测可用的方法，并产生方法名。通过 JavaBeans（第 13 章将详细介绍），Java 为这种基于组件的程序设计提供了一个基础结构。

在运行时间查询类信息的另一个原动力是通过网络创建与执行位于远程系统上的对象。这就叫作“远程方法调用”（RMI），它允许 Java 程序使用由多台机器发布或分布的对象。这种对象的分布可能是由多方面的原因引起的：可能要做一件计算密集型的工作，想对它进行分割，让处于空闲状态的其他机器分担部分工作，从而加快处理进度。某些情况下，可能需要将用于控制特定类型任务（比如多层客户 / 服务器架构中的“运作规则”）的代码放置在一台特殊的机器上，使这台机器成为对那些行动进行描述的一个通用储藏所。而且可以方便地修改这个场所，使其对系统内的所有方面产生影响（这是一种特别有用的设计思路，因

为机器是独立存在的，所以能轻易修改软件！）。分布式计算也能更充分地发挥某些专用硬件的作用，它们特别擅长执行一些特定的任务——例如矩阵逆转——但对常规编程来说却显得太夸张或者太昂贵了。

Class 类（本章前面已有详细论述）支持“反射”（Reflection）的概念。而且还有一个专门的库，名为 java.lang.reflect，其中包含了 Field、Method 以及 Constructor 类（每个都实现了 Member 接口）。这些类的对象都是 JVM 在运行时间创建的，用于代表未知类里对应的成员。这样便可用构造函数创建新对象，用 get()和 set()方法读取和修改与 Field 对象关联的字段，并用 invoke()方法调用与 Method 对象关联的方法。此外，我们还可调用 getFields()、getMethods()和 getConstructors()方法，它们分别返回用于表示字段、方法以及构造函数的对象数组（在联机文档中，还可找到与 Class 类有关的更多资料）。因此，匿名对象的类信息可在运行时间被完整的揭露出来，而在编译时间不需要知道任何东西。

大家要认识的很重要的一点是“反射”并没有什么神奇的地方。通过“反射”同一个未知类型的对象打交道时，JVM 只是简单地检查那个对象，并调查它从属于哪个特定的类（就象以前的 RTTI 那样）。但在这之后，在我们能做其他任何进一步的事情之前，Class 对象必须载入。因此，用于那种特定类型的.class 文件必须能由 JVM 调用（要么在本地机器内，要么可通过网络取得）。所以 RTTI 和“反射”之间唯一的区别就是：对 RTTI 来说，编译器会在编译时间打开和检查.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；但对“反射”来说，.class 文件在编译期间是不可使用的，只能由运行时间环境打开和检查。

12.3.1　一个类方法提取器

很少需要直接使用反射工具；之所以在语言中提供它们，仅仅是为了支持其他 Java 特性，比如对象序列化（第 11 章介绍）、JavaBeans（第 13 章）和 RMI（第 15 章）。但是，我们许多时候仍然需要动态提取与一个类有关的资料。其中特别有用的工具便是一个“类方法提取器”。正如前面指出的那样，如果你阅览类定义源码或者联机文档，那么只能看到在那个类定义中被定义或覆盖的方法，而基类那里还有大量资料。要想拿到它们，既麻烦、又花时间[56]。幸运的是，有“反射”可以帮助我们。你可用它写一个简单的工具，令其自动展示整个接口。就象下面这样：

```
//: c12:ShowMethods.java
// Using reflection to show all the methods of
// a class, even if the methods are defined in
// the base class.
import java.lang.reflect.*;

public class ShowMethods {
  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'";
```

[56] 这个问题过去尤其突出。不过，Sun 现已显著改进了他们的 HTML 版 Java 文档，所以要想看到基类方法，已比过去容易多了。

```
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(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++)
          System.out.println(m[i]);
        for (int i = 0; i < ctor.length; i++)
          System.out.println(ctor[i]);
      } else {
        for (int i = 0; i < m.length; i++)
          if(m[i].toString()
             .indexOf(args[1])!= -1)
            System.out.println(m[i]);
        for (int i = 0; i < ctor.length; i++)
          if(ctor[i].toString()
             .indexOf(args[1])!= -1)
          System.out.println(ctor[i]);
      }
    } catch(ClassNotFoundException e) {
      System.err.println("No such class: " + e);
    }
  }
} ///:~
```

Class 方法 getMethods()和 getConstructors()可以分别返回 Method 和 Constructor 的一个数组。每个类都提供了进一步的方法，可解析出它们所代表的方法的名字、参数以及返回值。但也可以象这样一样只使用 toString()，生成一个含有完整方法签名的字串。代码剩余的部分只是用于提取命令行信息，判断特定的签名是否与我们的目标字串相符（使用 indexOf()），并打印出结果。

这里便用到了“反射”技术，因为由 Class.forName()产生的结果不能在编译时间获知，所以所有方法签名信息都会在运行时间提取。若研究一下联机文档中关于“反射”（Reflection）的那部分文字，就会发现它已提供了足够多的支持，可对一个编译期完全未知的对象进行实际的设置以及发出方法调用。同样地，这也属于几乎完全不用我们操心的一个步骤——Java 自己会利用这种支持，所以程序设计环境能够控制 Java Beans——但它无论如何都是非常有趣的。

一个有趣的试验是运行：

```
java ShowMethods ShowMethods
```

这样可产生一个列表，其中包括一个 public 默认构造函数，尽管我们在代码中看见并没

有定义一个构造函数。我们看到的是由编译器自动合成的那一个构造函数。如果随之将 ShowMethods 设为一个非 public 类（即换成“友好”类），合成的默认构造函数便不会在输出结果中出现。合成的默认构造函数会自动获得与类一样的访问权限。

ShowMethods 的输出仍然有些“不爽”。例如，下面是通过调用“java ShowMethods java.lang.String”得到的一部分输出：

```
public boolean
  java.lang.String.startsWith(java.lang.String,int)
public boolean
  java.lang.String.startsWith(java.lang.String)
public boolean
  java.lang.String.endsWith(java.lang.String)
```

若能去掉象 java.lang 这样的限定词，结果显然会更令人满意。有鉴于此，可引入上一章介绍的 StreamTokenizer 类，来解决这个问题：

```
//: com:bruceeckel:util:StripQualifiers.java
package com.bruceeckel.util;
import java.io.*;

public class StripQualifiers {
  private StreamTokenizer st;
  public StripQualifiers(String qualified) {
    st = new StreamTokenizer(
      new StringReader(qualified));
    st.ordinaryChar(' '); // Keep the spaces
  }
  public String getNext() {
    String s = null;
    try {
      int token = st.nextToken();
      if(token != StreamTokenizer.TT_EOF) {
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = null;
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = new String(st.sval);
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
```

```
        }
      }
    } catch(IOException e) {
      System.err.println("Error fetching token");
    }
    return s;
  }
  public static String strip(String qualified) {
    StripQualifiers sq =
      new StripQualifiers(qualified);
    String s = "", si;
    while((si = sq.getNext()) != null) {
      int lastDot = si.lastIndexOf('.');
      if(lastDot != -1)
        si = si.substring(lastDot + 1);
      s += si;
    }
    return s;
  }
} ///:~
```

为简化代码复用，该类被放在了 com.bruceeckel.util 中。如大家所见，它利用了 StreamTokenizer 和字串处理来完成自己的工作。

新版程序利用上述类，对输出进行清理：

```
//: c12:ShowMethodsClean.java
// ShowMethods with the qualifiers stripped
// to make the results easier to read.
import java.lang.reflect.*;
import com.bruceeckel.util.*;

public class ShowMethodsClean {
  static final String usage =
    "usage: \n" +
    "ShowMethodsClean qualified.class.name\n" +
    "To show all methods in class or: \n" +
    "ShowMethodsClean qualif.class.name word\n" +
    "To search for methods involving 'word'";
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    try {
```

```
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
      Constructor[] ctor = c.getConstructors();
      // Convert to an array of cleaned Strings:
      String[] n =
        new String[m.length + ctor.length];
      for(int i = 0; i < m.length; i++) {
        String s = m[i].toString();
        n[i] = StripQualifiers.strip(s);
      }
      for(int i = 0; i < ctor.length; i++) {
        String s = ctor[i].toString();
        n[i + m.length] =
          StripQualifiers.strip(s);
      }
      if(args.length == 1)
        for (int i = 0; i < n.length; i++)
          System.out.println(n[i]);
      else
        for (int i = 0; i < n.length; i++)
          if(n[i].indexOf(args[1])!= -1)
            System.out.println(n[i]);
    } catch(ClassNotFoundException e) {
      System.err.println("No such class: " + e);
    }
  }
} ///:~
```

ShowMethodsClean 类非常接近前一个 ShowMethods，只是它取得了 Method 和 Constructor 数组，并将它们转换成单个 String 数组。随后，每个这样的 String 对象都在 StripQualifiers.Strip()里“过”一遍，删除所有方法限定词。

假如记不得一个类里是否有一个特定的方法，但又不想在联机文档里逐步检查类结构，或者不知道那个类是否能对某个对象（如 Color 对象）做某件事情，该工具便可帮你节省大量时间。

第 13 章提供了这个程序的一个 GUI 版本（专门为 Swing 组件提取相关信息），可在自己写程序的时候运行它，以便快速查找需要的资料。

12.4　总　　结

利用 RTTI 可根据一个匿名的基类引用调查出类型信息。但正是由于这个原因，新手们极易误用它，因为有些时候多态方法便足够了。对那些以前习惯程序化编程的人来说，极易将他们的程序组织成一系列 switch 语句。他们可能用 RTTI 做到这一点，从而在代码开发和维护中损失多态技术的重要价值。Java 的要求是让我们尽可能地采用“多态”，只有在极个别的情况下才使用 RTTI。

但为了利用多态，要求我们拥有对基类定义的控制权，因为有些时候在程序范围之内，可能发现基类并未包括我们想要的方法。若基类来自一个库，或者由别的什么东西控制着，RTTI 便是一种很好的解决方案：可继承一个新类型，然后添加自己的额外方法。在代码的其他地方，可以侦测自己的特定类型，并调用那个特殊的方法。这样做不会破坏多态以及程序的扩展能力，因为新类型的添加不要求查找程序中的 switch 语句。但在需要新特性的主体中添加新代码时，就必须用 RTTI 侦测自己特定的类型。

从某个特定类的利益的角度出发，在基类里加入一个特性后，可能意味着从那个基类派生的其他所有类都必须获得一些无意义的“鸡肋”。这使得接口变得含义模糊。若有人从那个基类继承，且必须覆盖抽象方法，这一现象便会使他们陷入困扰。比如现在用一个类结构来表示乐器（Instrument）。假定我们想清洁管弦乐队中所有乐器的通气音栓（Spit Valve）——假如它们有的话！此时的一个办法是在基类 Instrument 中置入一个 ClearSpitValve()方法。但这样做会造成一个误区，因为它暗示着打击乐器和电子乐器也有音栓（显然，它们没有这种东西）。针对这种情况，RTTI 提供了一个更合理的解决方案，可将方法置入特定的类中（此时是 Wind，即“管乐器”）——这样做是可行的。但事实上一种更合理的方案是将 prepareInstrument()置入基类中。初学者刚开始时往往看不到这一点，一般会认定自己必须使用 RTTI。

最后，RTTI 有时能解决效率问题。如果你的程序正确运用了“多态”，但其中的一个对象在执行效率上很有问题，便可用 RTTI 找出那个类型，然后写一段适当的代码，改进其效率。不过这里也要警告大家——最好不要过早地考虑程序效率的问题！“效率”是一个非常有诱惑力的陷阱，但请你以极大的毅力，克制自己不要过早地涉足它！最好的做法是先让你的程序正常运行起来，再来判断它的效率是否“够格”。效率的改进要摆在整个项目的最后阶段进行——为此，你通常需要事先拟定一个全盘性的计划。

12.5 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 在 Shapes.java 中添加 Rhomboid（长斜方形）。创建一个 Rhomboid，把它向上强制转型成一个 Shape，再向下强制转型回 Rhomboid。试着向下强制转型一个 Circle，看看会有什么情况发生。

(2) 修改练习(1)，使其在执行向下强制转型前用 instanceof 检查类型。 (3) 修改 Shapes.java，使其能“突出”（设置一个标志）特定类型的所有所有形状。每个派生的 Shape 的 toString()方法应该指出那个 Shape 是否已经“突出”了。 (4) 修改 SweetShop.java，使每种类型的对象创建都由一个命令行参数控制。也就是说，假如你的命令行是“java SweetShop Candy”，那么只有 Candy 对象才会创建。请注意如何通过命令行参数控制某类具体 Class 对象的载入。

(5) 在 PetCount3.java 中，添加一种新类型的 Pet。在 main()中，检查它是否已经创建和计数。

(6) 写一个方法，令其取得一个对象，并重复打印出该对象层次结构中的所有类。

(7) 修改练习(6)，令其同时用 Class.getDeclaredFields()显示出与类内那些字段有关的信息。

(8) 在 ToyTest.java 中，将 Toy 的默认构造函数变成注释内容，对结果进行解释。

(9) 将一种新类型的接口集成到 ToyTest.java 中，证实它会被正确侦测并显示出来。

(10) 创建一个新类型的容器，令其用一个 private ArrayList 容纳对象。请捕获你在其中放置的第一个对象类型，然后从此以后，只允许用户插入那种类型的对象。

(11) 写一个程序，判断一个 char 数组到底是一个原始数据类型，还是一个真正的对象。

(12) 参照本章最后的“总结”，实现 clearSpitValve()。

(13) 实现本章描述的 rotate(Shape)方法，使其能检查要旋转的是否为一个 Circle（圆）。如果是，则不执行旋转操作。

(14) 修改练习(6)，使其利用反射机制，而非 RTTI。

(15) 修改练习(7)，使其利用反射机制，而非 RTTI。

(16) 在 ToyTest.java 中，利用反射机制，通过非默认构造函数来创建一个 Toy 对象。

(17) 在 Java 的 HTML 用户文档中（可从 java.sun.com 下载），请查找 java.lang.Class 的接口。写一个程序，采用命令行参数的形式，取得一个类的名字。然后，用 Class 提供的方法来获取与那个类有关的所有信息。请先用一个标准库的类来测试你的程序，再用你自己创建的一个类来测试。

第 13 章 创建窗口和小程序

请记住一个基本的设计准则：“将简单的事情变得容易，使困难的事情成为可能。”[57]

在 Java 1.0 中，图形用户接口（GUI）库最初的设计目标是让程序员构建一个通用的 GUI，使其在所有平台上都能正常显示。但令人遗憾的是，这个目标并未达到。事实上，Java 1.0 版的“抽象窗口工具包”（AWT）产生的是在各种系统上看起来都同样欠佳的图形用户界面。除此之外，它还限制我们只能使用四种字体，并且不能访问由特定操作系统提供的高级 GUI 元素。同时，Jave 1.0 版的 AWT 编程模型也不是面向对象的，极不成熟。对此，我的一位学生作了以下解释（在 Java 最早的开发阶段，他就在 Sun 公司上班）：最初的 AWT 只用了一个月的时间，便完成了从概念、设计，一直到实现的过程。显然，你无法指望这个版本能够非常“成熟”。它已经违背了上述的设计准则：让我们感到了莫大的意外！同时，最初的 AWT 也“现身说法”，向我们证实了“设计”的重要性。

不过，这种情况在 Java 1.1 版的 AWT 事件模型中得到了很好的改进。后者换用了更为清晰的、面向对象的方法。同时，还首次加入了 JavaBeans（咖啡豆）——一种组件式编程模型，使我们能更方便地创建可视编程环境。而版本发展到 Java 2 后，已完全脱离了老的 Java 1.0 AWT 的不利影响。它用“Java 基类”（Java FoundationClasses，JFC）从根本上替代了一切老旧的东西，而它的 GUI 部分则正式更名为“Swing”。Swing 代表一系列易于使用、易于理解的 JavaBeans。你可以利用“拖放”操作（也可采用编程方式），用那些 JavaBeans 创建令自己满意的 GUI。软件业的“第三版”规则看来似乎同样适用于程序语言（通常，一个软件的第三个版本在各方面都会趋于成熟）！

本章的全部内容均围绕最新的 Java 2 Swing 库展开。在此，我也可以非常把握地认为，Swing 就是 Java GUI 库的终极目标！如果出于某种原因，你需要使用“老”AWT（因为要支持老代码，或者由于浏览器的限制等等），那么也可以参考本书的第一版（www.BruceEckel.com 和本书的配套光盘上都有）。

在本章刚开始的部分，大家会看到用 Swing 分别创建小程序和普通应用程序时，两者存在的一系列区别。另外，你还要学习如何创建兼具小程序和应用程序两种身份的程序，使其既能在浏览器中运行，也能通过命令行运行。本书的几乎所有 GUI 例子都可同时作为小程序和应用程序执行。

不过，也要注意本章并没有打算变成探讨全部 Swing 组件的一本“彻底剖析”；另外，即便讲到了一个类，也并一定会讲到它的所有方法。在这儿，我的目的是使你的学习变得尽可能简单。Swing 库本身具有极广的包容面，而本章的宗旨仅仅是让大家“入门”。只要掌握了原理，并熟悉了一些基本概念，那么某个具体的 Swing 组件还不是“手到擒来”？如果你想用它做更多的事情，可自行参考它的联机文档，便可知道它是否能为你提供所需要的工具。

我在这里假定你已从 java.sun.com 下载并安装了免费的、HTML 格式的 Java 库用户文档，而且知道怎样浏览那份文档中的 javax.swing 类，并会查阅 Swing 库的完整细节和方法。由于 Swing 设计宗旨是“简化至上”，所以它通常都能提供解决你的问题所需的足够资料。另外，市面上有许多书籍都是专门讲 Swing 的。如果你需要“更上一层楼”，或者想自己修

[57] 对此还有另一个说法，叫作“最不吃惊准则”，它的意思是：“尽量避免用户感到意外。”

改默认的 Swing 行为，那么也不妨找几本来看看。

通过对 Swing 的学习，大家会逐步理解到下面这几个要点：

(1) 和其他语言及开发环境提供的类似模型相比，Swing 这个编程模型要优秀得多！JavaBeans（在本章末尾介绍）是 Swing 库的基础。

(2) “GUI 构造函数”（即可视编程环境）已整合到 Java 开发环境中。用图形化工具将组件放到自己的窗体中时，JavaBeans 和 Swing 允许 GUI 构造函数为你自动编写对应的代码。这样不仅能加快 GUI 构建期间的开发速度，而且也便于你不断进行试验，最终找出符合自己要求的设计方案。

(3) Swing“简化至上”的宗旨以及良好的内部设计，意味着即便你真的用一个 GUI 构造函数来编程，而不是采用手工编程，结果得到的代码仍然是易于理解的——这便解决了过去 GUI 构造函数存在的一个大问题，它们极易产生根本没法子辨读的代码。

Swing 包括了在一个流行用户界面中能够看到的所有组件——从最简单的按钮开始，一直到内含的图片、树和表格等等。这是相当大的一个库！假如你想用它完成更复杂的任务，那么也可以从它那里得到帮助——只是要自己多动一下手。换句话说，对于简单的事情，你根本不必写太多的代码；但假如要做一些复杂的事情，那么你的代码也需要成比例地复杂。这意味着尽管 Swing 的“起点”非常低，但在你需要的时候，完全可以从它身上发掘出巨大的潜力！

Swing 最让人感到喜欢的一点是它非常“标准”。也就是说，一旦你理解了这个库的常规理论，就可四处运用。这主要应该归功于标准的命名规范。事实上，在我写本章的示范程序时，一些陌生的方法名往往可以“一猜就中”；即便没猜中，也不会差得太远。只有一个经过良好设计的库，才具有这样的“特色”。除此以外，通常可在组件里插入其他组件，而且能够正常工作。

考虑到速度，所有组件都设计得非常精简；而考虑到移植，Swing 完全是用 Java 写成的。

键盘漫游功能是自动赋予的——不必用鼠标，即可运行一个 Swing 应用程序——而且毋需任何专门的编程。滚动功能也是自动的——只需将你的组件封装到一个 JScrollPane 里，再把它添加到窗体。另外，象“工具提示”这样的高级特性只需一行代码即可使用。

Swing 还支持一种相当时髦的特性，名为“可插式外观与感觉”（PluggableLook and Feel）——它意味着用户界面的外观可彻底改换，以适合在不同平台和操作系统上工作的那些用户的需要。另外，甚至有可能（当然较难）由你自己来发明一套全新的“外观与感觉”。

13.1　基本小程序

Java 的一个设计目标就是让你创建“小程序”（Applet），即一种在 Web 浏览器内运行的小型程序。由于必须保证安全，所以小程序能做的事情被进行了极大的限制。不过，它仍然是一个非常强大的工具，特别是在进行客户端编程的时候——这是“全民触网”时代的一个重要问题。

13.1.1　小程序的局限

在小程序内的编程受到了极大的限制，所以通常说它在一个“暗箱”里——也就是说，总有“人”（Java 运行时间保安系统）在时时刻刻监视着你。

然而，你也完全可以脱离暗箱操作的局限，去写一个标准的应用程序，而不是非要写一个依赖浏览器才能运行的“小程序”。这样一来，和其他任何应用程序一样，就可以在程序中访问由操作系统提供的其他特性。迄今为止，本书已向你展示了大量“标准应用程序”，

不过它们都是“基于控制台”或者“基于命令行”的，没牵涉到任何图形组件。在 Swing 的帮助下，我们就可进一步完善那些程序，为其赋予一个图形化的“外壳”（图形化用户界面）。

“小程序”的基本设计宗旨是对浏览器中的 Web 页进行功能性扩展。基于这一前提，你便可正确回答出“到底能用一个小程序做什么？”这一问题。作为网上的一名“冲浪者”，你永远不知道 Web 页来自一个友好的地方，还是来自一个怀有敌意的地方。因此，你最大的希望就是浏览器中运行的代码永远都是安全的。因此，对“小程序”来说，它最大的限制应该是：

(1) 一个小程序不能接触本地磁盘。这意味着根本不允许它在本地（本机）磁盘上读写，因为我们不希望一个小程序在未经我们许可的前提下，擅自通过 Internet 读取和传输隐私信息。当然，写肯定是被禁止的，因为那除了会为病毒畅开大门之外，在极端的情况下，一个心怀恶意的人甚至可以把你的整个硬盘格式化掉。Java 为小程序提供了“数字签名”机制。在你允许一个“受信任的小程序”（由一个受到信任的来源签名）访问你的机器后，许多牵涉到小程序的限制都会被“解禁”。

(2) 小程序打开所花的时间较长。这是由于你每次都得从网上拉回一整套程序。甚至为了获得其中每一个不同的类，都得向服务器发出一次下载申请。当然，你的浏览器也许能将小程序缓存下来，但这根本就是一件没法子绝对保证的事情！考虑到这个原因，你设计的小程序无论如何都应该“打包”到一个 JAR（Java ARchive）文件里，令其合并所有小程序组件（包括其他.class 文件以及相关的图形和声音等等），变成一个统一的、压缩好的文件。这样只需和服务器打一次交道，便可把运行一个小程序所需的全部内容都下载回来。注意针对 JAR 文件里的单独每个条目，都可应用“数字签名”机制。

13.1.2 小程序的优点

如果能容忍那些限制，那么小程序的一些优点也是非常突出的，尤其是在我们构建客户机 / 服务器应用或者其它网络应用时：

(1) 没有安装问题。小程序是真正“与平台无关”的（包括象播放声音文件的功能），所以既不需要在自己的代码中针对不同的平台进行修改，也不需要强求用户在使用前先“安装”一道——这和标准应用程序有着显著的区别！事实上，用户每次载入 Web 页的时候，只要其中含有小程序，那么小程序会“静悄悄”地下载回来，并自动把自己安装好。但在传统的客户机 / 服务器系统中，在需要安装一个新版本的客户机软件时，却通常意味着一场“恶梦”的开始。

(2) 不必担心错误的代码会造成对别人系统的破坏。这是由于在核心 Java 语言及小程序结构中，已集成了强有力的安全保障措施。这个优点，加上前述的第 1 个优点，便使得 Java 成为“内部网”（Intranet）客户机 / 服务器应用的最佳选择。所谓“内部网”，是指仅在一个公司内部或者一个有限区域运行的网络。在这样的网络中，用户环境（Web 浏览器和插件）可以事先指定，而且可以非常轻松地加以控制。

由于小程序是自动同 HTML 语言集成的，所以我们有一个内建的、与平台无关的联机文档系统（HTML 格式）来提供对小程序的支持。这是非常有趣的一种状况，因为我们以前习惯的都是拥有“程序的文档部分”，而不是象这一次，拥有“文档的程序部分”。

13.1.3 应用程序框架

库通常按它们的功能进行分组。但对有些库来说，例如那些能独立使用的库，则不在此范围之内。标准 Java 库的 String 和 ArrayList 便属于这样的“独行侠”例子。但除此之外，其他库都是作为一种基本的构建单元来设计的，它们是创建其他类的基础。其中一个特殊类

别的库便是“应用程序框架”（Application Framework），它的目的是帮助你构建应用程序——通过它提供一个或者一系列类，令其产生某些基本（通用）行为，满足某一类全部应用程序的“共同需要”。然后，在有了这种具有“共通性”的行为之后，就可根据自己的要求，针对不同的场合来“定制”这种行为，满足自己的“特殊需要”。此时，你可以从应用程序类继承，并覆盖（改写）那些需要变动的、打算提供特殊服务的方法。应用程序框架的默认控制机制会在恰当的时间，调用你的那些已被覆盖的方法。事实上，我们在本书早些时候已经多次提出这样一条设计准则：“把变与不变的东西分开”——应用程序框架正是一个绝佳的范例！因为它的目的就是在它覆盖的方法中，将一个程序最“特殊”的那些部分分离出去[58]。

小程序就是利用“应用程序框架”来建立的。我们先从 JApplet 类继承，然后覆盖恰当的方法。有几个方法可供我们控制一个小程序在 Web 页中的创建及执行：

方　法	作　　用
init()	自动调用，首次执行小程序的初始化（其中包括组件布局）。无论如何都要覆盖该方法。
start()	每次 Web 浏览器把注意力转到小程序身上，并允许它启动自己的普通操作时（特别是那些以前用 stop()中止过的操作），都会调用一遍 start()。在执行完 init()后，也会紧接着自动执行 start()
stop()	每次 Web 浏览器把注意力从小程序身上移开，以便它停止某些“代价高昂”操作时（比如在资源资源不足的时候），便会调用 stop()。另外，在执行 destory()之前，也会先自动执行一次 stop()
destroy()	小程序不再需要之后，把它从页内卸载时调用，从而实现资源的最终回收

有了下面这些信息，便可着手一个简单小程序的创建：

```
//: c13:Applet1.java
// Very simple applet.
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
} ///:~
```

注意对小程序来说，main()并不是必需的。它们全部都封装到应用程序框架中；如果你有任何启动代码，请把它们置于 init()中。

在这个程序中，我们采取的唯一实际行动就是在小程序上放置一个文本标签——这是通过 JLabel 类来完成的（老的 AWT 占用了“Label”以及其他类似的组件名。因此，对于新的 Swing 组件，你总能看到它最开头添加的一个“J”字母）。这个类的构造函数会取得一个 String 参数，并用它的内容来创建标签。在上述程序中，该标签会放置到窗体中。

[58] 这也是“模板方法”设计范式的一个经典例子。

在此，init()方法的职责是用 add()方法将所有组件都放到窗体上。有的人或许会认为自己能单独调用 add()，但那实际只是老式 AWT 的做法。在新的 Swing 中，必须将所有组件都添加到一个窗体的“内容区”（Content Pane），所以作为 add()操作的一部分，你必须调用 getContentPane()。

13.1.4 在 Web 浏览器中运行小程序

要想运行这个程序，必须把它放到一个 Web 页（网页）里，然后用支持 Java 的一个 Web 浏览器来观看它。将小程序放到网页中时，必须在 HTML 源码中设置一个特殊的标记[59]，指示那个页如何载入和运行小程序。

以前，这种操作是非常简单的。在那时，Java 语言本身还非常“单纯”，每家制作 Web 浏览器的公司都站在同一条起跑线上，而且在他们的 Web 浏览器中集成了一模一样的 Java 支持。因此，通常只需在网页中设置一条非常简单的 HTML 代码，就可以载入并运行一个小程序。象下面这样：

```
<applet code=Applet1 width=100 height=50>
</applet>
```

不幸的是，后来爆发了“浏览器大战”，我们（程序员和最终用户）输掉了这场战争。过了不多久，JavaSoft 便意识到我们不能再指望浏览器能提供正确的 Java 支持，所以唯一的办法便是提供某种形式的“加载组件”，令其通过浏览器本身的扩展机制，为浏览器添加附加的功能。利用扩展机制（只要没有破坏所有第三方扩展的“非份之想”，浏览器厂商便会继续支持自己的扩展机制。而为了保持竞争优势，浏览器厂商也势必——起码是暂时的——不能变成一个“孤家寡人”），JavaSoft 便保证了没有任何一家敌对厂商的 Web 浏览器能够屏蔽掉“正宗”的 Java 支持！

对 Internet Explorer 来说，它采用的扩展机制是“ActiveX 控件”；而对 Netscape 来说，则采用“插件”的形式加以扩展。在你的 HTML 代码中，必须用特殊的标记来同时支持它们两个。下面是 Applet1 插入 HTML 页后最简单的一种形式[60]：

```
//:! c13:Applet1.html
<html><head><title>Applet1</title></head><hr>
<OBJECT
  classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  width="100" height="50" align="baseline"
codebase="http://java.sun.com/products/plugin/1.2.2/jinstall
-1_2_2-win.cab#Version=1,2,2,0">
<PARAM NAME="code" VALUE="Applet1.class">
<PARAM NAME="codebase" VALUE=".">
<PARAM NAME="type"
VALUE="application/x-java-applet;version=1.2.2">
<COMMENT>
  <EMBED type=
    "application/x-java-applet;version=1.2.2"
```

[59] 这里假定读者已掌握了基本的 HTML 知识。要学懂 HTML 其实并不难，许多书籍和参考资料都可以帮助你。

[60] 这个页——特别是它的“clsid”部分——似乎能在 JDK 1.2.2 和 JDK1.3 rc-1 上很好地工作。不过，未来某个时候，你或许仍然要对标记作出一些改动。详情见 java.sun.com。

```
    width="200" height="200" align="baseline"
    code="Applet1.class" codebase="."
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-
install.html">
  <NOEMBED>
</COMMENT>
   No Java 2 support for APPLET!!
  </NOEMBED>
</EMBED>
</OBJECT>
<hr></body></html>
///:~
```

其中有些行实在是太长了，以至于不得不在书中进行换行处理，以便正确地印刷出来。不过，本书的所有源码（配套光盘和 www.BruceEclek.com 上都有）在实际运行的时候，你都不必担心换行的问题。

code 值指定了.class 文件的名字，我们的小程序就放在那个文件中。width 和 height 指定了小程序的初始大小（以像素为单位）。在小程序标记内，你还可指定其他信息：一个保存着其他.class 文件的 Internet 地址（codebase）；对齐方式（align）；一个特殊的标识符，便于小程序相互间通信（name）；一个小程序参数，提供小程序可以取得的信息。参数采用下列形式：

```
<param name="identifier" value = "information">
```

另外根据你的需要，还有其他许多信息可在这里提供。

在本书的源码包里，为本书的每个小程序都制作了一个 HTML 页，所以大家能找到许多这样的小程序标记。在 java.sun.com，可以找到在网页里设置小程序时所需的一切以及最新的细节。

13.1.5　小程序观察器的使用

Sun 的 JDK（可从 java.sun.com 免费下载）提供了一个名为“小程序观察器”（Appletviewer）的工具，可从 HTML 文件中解析出所有<applet>标记，然后直接运行小程序，不用显示出周围的的 HTML 文本。由于该工具会忽略除 APPLET 标记之外的一切东西，所以可以采用注释形式，将那些标记放在 Java 源码中：

```
// <applet code=MyApplet width=200 height=100>
// </applet>
```

这样一来，我们就可直接运行“**appletviewer MyApplet.java**”，毋需专门创建一个 HTML 文件来执行测试。例如，可在前述的 Applet1.java 中加入下列注释形式的 HTML 标记：

```
//: c13: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!"));
  }
} ///:~
```

接下来，用下述命令调用这个小程序即可：

```
appletviewer Applet1b.java
```

在本书中，我们将采用这种形式来简化小程序的测试。不久，大家还会学习到另一种编程方法，可直接从命令行执行小程序，就连 Appletviewer 也不必使用！

13.1.6 测试小程序

可在不必建立网络连接的前提下进行一次简单的测试，方法是启动我们的 Web 浏览器，然后打开包含了小程序标记的 HTML 文件。HTML 文件载入后，浏览器会发现其中的小程序标记，然后查找由 code 值指定的.class 文件。当然，它会先在 CLASSPATH（类路径）中寻找，如果在 CLASSPATH 下找不到类文件，就在 Web 浏览器状态栏中给出一条出错提示，指出自己找不到.class 文件。

如果想在自己的网站上实验，情况就显得稍微有点儿复杂。首先，你必须有一个自己的 Web 站点。对大多数人来说，这意味着需要向本地或远地的一家 ISP（Internet 服务供应商）申请网站寄放空间。不过，由于小程序仅仅是一个或者一系列文件，所以 ISP 并不需要为 Java 提供什么“特别支持”。另外，你必须通过某种途径将 HTML 文件和.class 文件从自己的机器上传到 ISP 服务器上的正确目录。这一般需要用到一个 FTP（文件传输协议）上传工具软件——这类软件目前有很多，有的是共享软件，有的则是完全免费的。因此，我们要做的全部事情似乎就是用 FTP 工具将文件上传至 ISP 机器，然后用自己的浏览器连接网站和 HTML 文件；假如小程序正确装载和执行，就表明大功告成。但真是这样吗？

千万不要被表象所迷惑！假如客户机上的浏览器找不到服务器上的.class 文件，它就会试图在你的本机的 CLASSPATH 中寻找（注意是在“你的机器上”）。因此，对其他用户来说，小程序可能根本没办法从服务器上正确装载。你自己测试的时候，看起来似乎是成功的（因为浏览器实际是在你的本机上找到了.class 文件），但其他人用他们的浏览器做同样的测试时，却根本找不到它。因此，在你测试的时候，务必先删除自己机器上的相关.class 文件（或.jar 文件），最终证实它们已正确地放在服务器上了。

我自己就曾遇到过这样的一个问题。当时是将小程序置入一个 package（封装）中。上载了 HTML 文件和小程序后，由于封装名的问题，小程序的服务器路径似乎陷入了混乱。但是，我的浏览器在本地类路径（CLASSPATH）中找到了它。这样一来，我就成了唯一能成功装载那个小程序的人。后来我花了一些时间，才发现原来是 package 语句有误。一般地，应该将 package 语句置于小程序的外部。

13.2 从命令行运行小程序

有些时候，我们希望一个视窗化的程序脱离自己的“本份”，不要老“寄生”在一个网页里，而是做一些别的事情。我们有时也希望它做一些“标准”应用程序的事情，但同时依然保留 Java 为它赋予的强大移植能力。在本书以前的章节中，我们编写的都是“基于命令行”的应用程序，但在某些操作系统中（比如 Macintosh），却没有象这样的一个命令行。因

此，综合方方面面的原因，我们希望能用 Java 构建一个视窗化的、非“小程序”的一个程序。这显然是一个合理的要求，也颇有必要。

Swing 库允许我们制作一个这样的应用程序，它可以保持由基础操作系统赋予的“外观与感觉”。如果你想构建一个视窗化应用程序，那么只有在拿到了最新版本的 Java 和相关工具之后，才建议你着手这样做。只有这样，你的程序才能真正赢得用户的心[61]。如果出于某种原因，你被迫使用老版本的 Java，那么在试图构建一个重要的视窗化应用程序之前，请务必仔细斟酌。

通常，我们希望创建这样一个特殊的类，它既能作为一个窗口调用，也能作为一个小程序调用。对小程序进行测试时，这样做可带来极大的方便，因为从命令行直接运行程序要快得多，不必启动浏览器或者 Appletviewer，再等待它们慢吞吞地装载、运行。

为了创建一个能从控制台命令行运行的小程序，只需为小程序添加一个 main()，在一个 JFrame[62]内构建小程序的一个实例即可。作为一个简单的例子，这里可以修改一下前述的 Applet1b.java，让它兼具“应用程序”和“小程序”这两种身份：

```
//: c13:Applet1c.java
// An application and an applet.
// <applet code=Applet1c width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

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:
    Console.setupClosing(frame);
    frame.getContentPane().add(applet);
    frame.setSize(100,50);
    applet.init();
    applet.start();
    frame.setVisible(true);
  }
} ///:~
```

main()是添加到小程序中的唯一元素，小程序剩余的部分丝毫未动。这样一来，我们就

[61] 要我来选择的话，那么在学过 Swing 后，就根本不愿再把时间浪费在那些老旧的东西上。

[62] 就象早先提到的那样，“Frame”这个词被 AWT 占用了，所以 Swing 换成“JFrame”。

创建了小程序，并把它添加到了 JFrame 中，使其能显示出来。

对下面这一行代码来说：

```
Console.setupClosing(frame);
```

它会指示窗口正确关闭。Console（控制台）来自于 com.bruceeckel.swing，稍后还会对它详加解释。

在 main()中，请注意我们对小程序进行了明确的初始化和启动。这是由于在这种情况下，浏览器便没法子帮我们自动完成了。当然，这里也并没有完全模仿浏览器的行为，后者起码还会自动调用 stop()和 destory()。不过，在大多数情况下，我们这种“有限的模仿”都是可以接受的。如果真的出现问题，也可以自己进行那样的调用[63]。

请注意最后一行：

```
frame.setVisible(true);
```

没有这一行代码，屏幕上便不会任何东西显示出来。

13.2.1 一个显示框架

尽管一个程序具备“小程序”和“应用程序”的双重身份确实能得到不错的效果，但也切不可滥用。假如不假思索地到处使用，我们的代码就会变得非常复杂，有浪费纸张骗稿费的嫌疑。因此，在本书剩余的 Swing 例子中，我们会换用下面这个显示框架：

```
//: 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 setupClosing(JFrame frame) {
    // The JDK 1.2 Solution as an
    // anonymous inner class:
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
```

63 随着本章学习的深入，大家还会真的需要亲手这么做。具体做法是：首先，将引用 JApplet 变成类的一个静态成员（而不是 main()中的一个本地变量），然后在调用 System.exit()之前，在 WindowAdapter.windowClosing()内部调用 applet.stop()和 applet.destroy()。

```
      }
    });
    // The improved solution in JDK 1.3:
    // frame.setDefaultCloseOperation(
    //    EXIT_ON_CLOSE);
  }
  public static void
  run(JFrame frame, int width, int height) {
    setupClosing(frame);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
  public static void
  run(JApplet applet, int width, int height) {
    JFrame frame = new JFrame(title(applet));
    setupClosing(frame);
    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));
    setupClosing(frame);
    frame.getContentPane().add(panel);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
} ///:~
```

由于这个工具平常对你也非常有用，所以我把它放到了 com.bruceeckel.swing 库中。Console 类完全由静态（static）方法构成。第一个用于从任意对象中提取类名（通过 RTTI），并用于删除“class”字样——这个单词通常是由 getClass()添加的。为此，我们用 String 方法 indexOf()来判断其中是否存在“class”字样；同时用 substring()来产生新的、已剔除了“class”及尾部多余空格的新字串（子字串）。新名字将用于标记由 run()方法显示的窗口。

setupClosing()用于将一部分代码隐藏起来。当那个 JFrame 关闭的时候，这部分代码会造成 JFrame 退出一个程序。它的默认行为是“什么都不做”，所以假如你不调用 setupClosing()，或者为你的 JFrame 编写等价的代码，那么应用程序不会关闭。之所以要把这部分代码隐藏起来，而不是直接放到后续的 run()方法中，部分原因是由于在我们希望做比 run()支持的更复杂的一些事情时，它便允许我们直接使用方法。不过，它也将一个变化性的因素隔离出来了：Java 2 有两个办法都可造成特定类型的窗口的关闭。在 JDK 1.2 中，解决的办法是创建一个新的 WindowAdapter 类，并实现 windowClosing()方法——就象上面

展示的那样（本章稍后还会讲述其完整含义）。但是，在 JDK 1.3 的开发过程中，库设计人员发现只要创建的不是一个小程序，那么通常都需要关闭窗口。因此，他们为 JFrame 和 JDialog 添加了 setDefaultCloseOperation()，用于设定默认操作。从写程序的角度出发，这个新方法应该是你的最佳选择。不过，在本书写作的时候，JDK 1.3 在 Linux 和其他一些平台上尚未真正实现，所以考虑到跨平台兼容的问题，所有改变均隔离在 setupClosing()内部。

run()方法已进行了覆盖，可随 JApplet、JPanel 和 JFrame 使用。不过要注意的是，只有当你用 init()初始化了一个 JApplet，并调用了 start()之后，才能那样做。

现在，只需创建一个 main()，在其中包含象下面这样的一行代码，便可从控制台运行任何小程序：

```
Console.run(new MyClass(), 500, 300);
```

其中，最后两个参数对应于显示的宽度和高度。在这里，我们对 Applet1c.java 进行了修改，令其使用 Console：

```
//: c13: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);
  }
} ///:~
```

这样一来，我们既免去了重复编码的辛苦，也为示范程序的运行提供了最大的灵活性。

13.2.2 使用 Windows 资源管理器

如果你使用的是 Microsoft Windows 平台，可配置一下操作系统自带的“Windows 资源管理器”（Windows Explorer）工具，从而简化一个命令行 Java 程序的运行。注意这里说的是 Windows 的那个文件浏览器，而不是大家所熟悉的 Internet Explorer。最后要达到的效果应该是：双击一个.class 文件，它就能自动运行！为此，你需要采取几个步骤。

首先，从 www.Perl.org 这个网站下载并安装 Perl 程序语言包。按照网页上的提示，可以找到详细的操作步骤，并可下载采用不同语言的帮助文档。

接下来，自己创建下述脚本（注意去掉第一行和最后一行；这段脚本用不着你亲自录入，本书配套的源码包里有现成的）：

```
//:! c13:RunJava.bat
@rem = '--*-Perl-*--
```

```
@echo off
perl -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl
@rem ';
#!perl
$file = $ARGV[0];
$file =~ s/(.*)\..*/\1/;
$file =~ s/(.*\\)*(.*)/$+/;
`java $file`;
__END__
:endofperl
///:~
```

现在，请打开“Windows 资源管理器”，依次选择“查看”、“文件夹选项”，再点击“文件类型”选项卡标签。按下其中的“新类型”按钮。在“类型说明”框里，输入“Java 类文件”字样；在“相关扩展名”框里，输入“class”；在“操作”区域，请点击下方的“新建”按钮。在随后打开的另一个对话框里，将“操作”填写成“Open”，在“用于执行操作的应用程序”框里，则请输入下面这一行：

```
"c:\aaa\Perl\RunJava.bat" "%L"
```

当然，RunJava.bat 文件的路径应以你自己的实际情况为准，注意务必保证系统能正确地找到这个批处理文件。

完成上述安装后，一旦需要执行某个 Java 程序，只需用鼠标双击对应的.class 文件即可（但其中一定要含有一个 main()）。

13.3 制 作 按 钮

制作一个按钮非常简单：只需要调用 JButton 构造函数，并指定想在按钮上出现的标签就行了。以后，大家还可以做一些更有趣的事情，比如在按钮上放置一个图标等等。

通常，我们需要在自己的类内为按钮创建一个字段，以后将来能直接引用它。

JButton 是一种特殊的组件，它自己的小窗口会在每一次屏幕更新的时候得以自动重画。也就是说，你不必明确地描绘一个按钮或者其他任何种类的控件——只需要把它们放在窗体内，以后的涂涂画画便不用再操心了！把一个按钮放到窗体内部时，需要在 init()内进行操作：

```
//: c13: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 {
```

```
  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);
  }
} ///:~
```

这里出现了一些新鲜的东西：在任何元素放入“内容区”（Content Pane）之前，需要先赋予一个类型为 FlowLayout 的新“布局管理器”。只有通过布局管理器，我们的“内容区”才知道应该将一个控件放在窗体的什么位置。对小程序来说，它一般的行为是使用 BorderLayout，但在这儿却是不现实的，因为它在默认情况下，会用新加入的控件完全覆盖以前的每个控件（本章后面对窗体的布局进行详细讨论时，大家就会实际认识到这一点）。另一方面，FlowLayout 可使所有控件均匀地分布在窗体上——按照从左到右、从上到下的顺序。

13.4 捕 获 事 件

大家会注意到，假如编译和运行上面的小程序，按下按钮后根本不会发生任何事情。这时需要我们亲自动手写一些代码，决定要发生什么事情。对于“由事件驱动”的编程（GUI 的大部分都由此构成）来说，它涉及到的最基本的操作就是将一个事件同响应它的那部分代码联系到一起！

在 Swing 中，为了达到这个目的，需要将接口（图形组件）同实现（在一个组件上发生一个事件时，你想运行的代码）清晰地区分开。每个 Swing 组件都可报告可能在它身上发生的所有事件，也可单独报告每一种事件。因此，举个例子来说，假如你并不关心鼠标是否移经你的按钮，就不要专门去注册对那个事件的响应。在事件驱动编程的世界中，这无疑是一种非常直观、非常出色的控制方案。一旦你掌握了基本概念，所有 Swing 组件在你面前都只是“小菜一碟”——即使以前以前从未见过某个组件。事实，这一模型扩展到了可划为 JavaBean 一类的任何东西（JavaBean 将在本章的后面部分详述）。

最开始，我们只需将重点放在打算使用的组件的主事件身上。在一个 JButton 的情况下，我们“感兴趣的事件”并不是鼠标从它上面移过，而是它被实际地按下。为了表明你对一个按钮被按下之后要发生的事情的兴趣（即“注册”这一事件），你需要调用 JButton 的 addActionListener()方法。该方法要求取得一个对象参数，而且那个对象必须实现了 ActionListener 接口。在 ActionListener 接口中，只包含了一个名为 actionPerformed()的方法。因此，为了将你的代码同 JButton 联系起来，我们要做的全部事情就是在一个类里实现 ActionListener 接口，并通过 addActionListener()，为 JButton 注册那个类的一个对象。最后达到的效果就是：按钮被按下之后，方法会立即得到调用（通常把这称为一次“回调”）。

但那个按钮后，应该得到什么结果呢？我们希望屏幕上有所变化，所以又引入了一个新

的 Swing 组件：JTextField。“TextField”是“文本字段”的意思，你可在这个地方键入文字，或者就目前来说，是由程序修改其中显示的文字。尽管有许多种方法都可以创建一个 JTextField，但最简单的做法还是告诉构造函数你想要一个多宽的文本字段。将 JTextField 放入窗体后，就可接着用 setText()方法来修改它的内容（JTextField 提供了大量方法，详情见 JDK 的 HTML 用户文档，可从 java.sun.com 下载）。结果代码如下：

```
//: c13: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 {
  JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
  JTextField txt = new JTextField(10);
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e){
      String name =
        ((JButton)e.getSource()).getText();
      txt.setText(name);
    }
  }
  BL al = new BL();
  public void init() {
    b1.addActionListener(al);
    b2.addActionListener(al);
    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);
  }
} ///:~
```

创建一个 JTextField，并把它放到窗体中——其间涉及的操作和 JButtons 以及其他任何 Swing 组件都是一样的。对上面的程序来说，唯一的差别在于要创建前述的 ActionListener

类 BL。传给 actionPerformed()的参数采用 ActionEvent 类型，其中包括与事件有关的所有信息，同时也指出了事件是从哪里来的。在这种情况下，我希望对按下的按钮进行说明，所以 getSource()产生始发那个事件的对象。而且我假定由 JButton.getText()返回按钮上的文字，而且它放置在 JTextField 中，以证明按钮被按下的时候，代码得到了实际的调用。

在 init()中，addActionListener()用于为两个按钮都注册 BL 对象。

通常，假如能将 ActionListener 编码成一个匿名内部类，那么事情还要容易得多——特别是只希望使用每个 Listener 类的一个实例的时候。然后，可以修改一下 Button2.java，使其能正常使用匿名内部类：

```
//: c13: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 {
  JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
  JTextField txt = new JTextField(10);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String name =
        ((JButton)e.getSource()).getText();
      txt.setText(name);
    }
  };
  public void init() {
    b1.addActionListener(al);
    b2.addActionListener(al);
    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);
  }
} ///:~
```

对于本书的例子来说，“匿名内部类”将是我们的首选方法（只要有可能）。

13.5　文　本　区

JTextArea（文本区）和 JTextField 非常相似，只是前者支持多行显示，而且支持更丰富的文字处理功能。其中，特别有用的一个方法是 append()——“追加”；用它可把你的程序输出方便地放到 JTextArea 里，这就使 Swing 程序同以往的命令行程序相比，具有了相当大的优势！现在，你可以“向后”滚动文字；而在传统的、基于命令行的程序中，输出只能一行接一行地“向前”进行。例如，下面这个程序可以用第 9 章那个 geography 生成器的输出，来填写一个 JTextArea：

```
//: c13:TextArea.java
// Using the JTextArea control.
// <applet code=TextArea width=475 height=425>
// </applet>
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 JApplet {
  JButton
    b = new JButton("Add Data"),
    c = new JButton("Clear Data");
  JTextArea t = new JTextArea(20, 40);
  Map m = new HashMap();
  public void init() {
    // Use up all the data:
    Collections2.fill(m,
      Collections2.geography,
      CountryCapitals.pairs.length);
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        for(Iterator it= m.entrySet().iterator();
            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);
  }
} ///:~
```

在 init()中，我们用所有国家及其首都的名字来填充 Map。注意针对两个按钮，都会创建并添加 ActionListener，同时不会定义任何中间变量——这是由于在程序中，永远都不再需要引用那个 Listener。“Add Data”按钮可格式化和追加所有数据；而“Clear Data”按钮会用 setText()从 JTextArea 中删除所有文字。

JTextArea 添加到小程序后，我们把它封装到一个 JScrollPane 里，从而在屏上文字过多的情况下，对其进行滚动显示。注意此时你得到的将是一套完整的文字滚动功能，而其间涉及到的操作又是那么简单！对比一下其他 GUI 程序开发环境的做法，你便会大大叹服于 JScrollPane 精简、高效、体贴用户的设计！

13.6 布 局 控 制

在 Java 里，如果你想把组件放到一个窗体内，做法和其他 GUI 系统恐怕有所不同。首先，它全部采用编码方式，并没有什么“资源”来控制组件的放置。其次，组件在窗体内的位置并不是由绝对定位来控制的，而是由某个“层管理器”加以控制。参照各组件添加（add()）的顺序，这咱管理器决定了应该把组件摆在哪里。随着层管理器的不同，组件的大小、形状和摆放也有着显著的区别。除此以外，层管理器会自动跟踪你的小程序或应用程序窗口的尺寸大小——一旦窗口大小发生改变，组件的大小、形状和位置也会相应地改变。

调用 getContentPane()之后，JApplet、JFrame、JWindow 和 JDialog 全部都会产生一个容器，再由这个容器负责组件的容纳和显示。在容器（Container）中，提供了一个名为 setlayout()的方法，专门用于选择不同的“层管理器”。而对其他类来说——比如 JPanel——则由它们直接负责组件的容纳与显示，因此你也必须为它们直接设置层管理器，而不是使用“内容区”（Content Pane）。

在这一节里，我们将通过在上面放置按钮的方式（这是最简单的实验手段），对不同的布局管理器进行探讨。在此，我们并不打算捕获任何按钮事件，因为我们唯一的目的是理解按钮的布局。

13.6.1 BorderLayout

小程序都有一个默认的布局方案，即 BorderLayout（前面许多例子已改换成 FlowLayout 这种布局管理器）。假如没有其他指示，我们用 add()添加的所有组件都会自动居中，并会随

着内容的增多，向四周伸展，直至窗口边界。这正是“BorderLayout”（边界布局）这个名称的来历。

不过，关于 BorderLayout，我们还有更多的话要说。使用这种布局管理器时，心中要有一个四边形以及一个中央区域的概念。将某样东西添加到采用了 BorderLayout 布局的一个窗口时，可使用一个重载的 add()方法，并指定一个常数值作为它的第一个参数。这个值可为下述常数之一：

BorderLayout.NORTH（靠在顶部）
BorderLayout.SOUTH（靠在底部）
BorderLayout.EAST（靠在右侧）
BorderLayout.WEST（靠在左侧）
BorderLayout.CENTER（位于中央，伸展到其他组件，或直接碰到边框）

假如没有具体指出要将组件放在哪里，就默认为 CENTER。

下面是一个简单的例子，采用默认布局——因为 JApplet 会默认使用 BorderLayout：

```
//: c13: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 之外的每个位置，我们添加的组件都会自动压缩——沿一边只占、用最小的空间；而沿另一边则伸展至最大长度。CENTER 则与众不同，它的两边都会自动伸展，保持随时“居中”的特点。

13.6.2 FlowLayout

“Flow”是“流”的意思；换言之，所有组件会一个接一个地“流”到窗体里，按着从左到右的顺序，直至顶部空间被填满。然后，自动移到下一行，继续“流”进去。

这里有一个例子，它将布局管理器指定为 FlowLayout，然后把按钮放到窗体上。大家会注意到，在使用 FlowLayout 的时候，组件会采用它们的“自然”大小——例如，对一个 JButton 来说，它的大小就是上面显示的字串的大小。

```
//: c13: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);
  }
} ///:~
```

所有组件将在 FlowLayout 中都会被压缩成它们的最小尺寸，所以最后的结果可能点儿“古怪”。例如，由于 JLabel 会是它的字串的大小，所以在使用 FlowLayout 的情况下，对它的文本进行右对齐的企图会造成一个无法改变的显示外观。

13.6.3 GridLayout

GridLayout 允许我们建立一个组件表。添加那些组件时，它们会按从左到右、从上到下的顺序在网格中排列。在构造函数里，需要指定自己希望的行、列数，它们将按正比例展开。

```
//: c13: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);
  }
} ///:~
```

由于网格的存在，这个例子里共准备了 21 个空位，但却只放了 20 个按钮，最后一个位置只好成为空白——对 GridLayout 来说，它并不会采取什么“自动伸缩”的措施，所以不能保证占满所有空间。

13.6.4　GridBagLayout

GridBagLayout 主要用来控制窗口中的指定区域；窗口的大小改变后，那些区域也会自动改变大小。它最大的特点在于——对各个区域的控制非常准确！不过，这也是最复杂的一种布局管理器。如果不用心、不多加实践的话，恐怕很难真正地掌握它。它的主要设计宗旨是让一个 GUI 构建工具能自动地、准确地生成代码（任何好的 GUI 构建工具都会用 GridBagLayout，而不是“绝对定位”）。如果你设计的系统非常复杂，那么也应该用一个 GUI 构建工具来帮助自己。当然，如果非要知道 GridBagLayout 背后的细节，这里也建议你阅读由 Horstmann 和 Cornel 编著的《Core Java 2》一书，美国 Prentice-Hall 出版社于 1999 年出版；或者专门找一本讲 Swing 的书看看。

13.6.5　绝对定位

任何图形化组件也可以设置它们的绝对位置，步骤如下：

(1) 为你的容器（Container）设置一个 null（空的）布局管理器，即 setLayout(null)。

(2) 为每个组件都调用 setBounds()或 reshape()——具体哪一个由语言版本决定。然后，采用像素坐标，传递一个约束矩形。可在构造函数中操作，亦可在 paint()中进行——依你希望达到的目标而定。

某些 GUI 构建工具大量采用了这种方式，但对于代码的生成来说，这恐怕并不是一个好办法。更合理的做法是使用前述的 GridBagLayout！

13.6.6　BoxLayout

正是考虑到许多人都不愿意如此麻烦地学习和运用 GridBagLayout，所以 Swing 同时也提供了一个 BoxLayout——它具有 GridBagLayout 的诸多特色，但又不至于牵涉到太多的复杂性。因此，在你希望手工控制布局的时候，往往都会首先想到它（再提醒大家：一旦你设计的系统过于复杂，请务必考虑换用一个 GUI 辅助构建工具，用它帮你生成 GridBagLayout）。BoxLayout 允许我们在垂直或水平方向控制组件的位置，并可用所谓的“撑木”（Strut）和“胶水（Glue）”，来控制各组件之间的间距。首先，让我们来看看如何直接使用 BoxLayout——与此同时，还会演示其他布局管理器的用法：

```
//: c13:BoxLayout1.java
// 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("" + i));
    JPanel jph = new JPanel();
    jph.setLayout(
      new BoxLayout(jph, BoxLayout.X_AXIS));
    for(int i = 0; i < 5; i++)
      jph.add(new JButton("" + 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 的构造函数和其他布局管理器的有所不同——你得将打算用 BoxLayout 控制的那个容器（Container）作为第一个参数传递给构造函数，同时传递的还有布局的方位（作为第二个参数）。

为使问题简化，我们还有另一个特殊的容器——名为 Box（盒子）——它将 BoxLayout 作为自己的自然管理器使用。下例将用 Box 水平和垂直地排列组件。在 Box 中，我们用它的两个静态方法来创建垂直和水平对齐的“盒子”：

```
//: c13: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("" + i));
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++)
      bh.add(new JButton("" + 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);
  }
} ///:~
```

有了 Box 之后，当我们向内容区（Content Pane）添加组件的时候，便同时向其传递第二个参数。

“撑木”（Strut）会在组件之间添加适当的空白（以像素为单位）。要想使用一个“撑木”，只需在需要分隔的两个组件的添加过程之间，把它加进去就可以了：

```
//: c13: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("" + i));
      bv.add(Box.createVerticalStrut(i*10));
    }
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++) {
      bh.add(new JButton("" + 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);
  }
} ///:~
```

“撑木”会使组件间隔固定的距离，“胶水”（Glue）则恰巧相反——它会使两个组件离得尽可能地远。换句话说，它其实更象是一个“弹簧”，而不是什么“胶水”。（所以整套设计根本就应该叫作“撑木和弹簧”，而不是“撑木和胶水”。不知道 Swing 的设计人员当时到底是怎么想的!）

```
//: c13: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);
  }
} ///:~
```

“撑木”只能在一个方向上工作，但一个精密的区域在两个方向上都修正了这个间距：

```
//: c13: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);
  }
} ///:~
```

当然，使用“精密区域”并不见得是一件好事。由于使用的是绝对值，所以它们以后会带来的麻烦，可能会远远大于它们当初的好处！

13.6.7　哪种更佳？

Swing 本身是种潜力无限的工具；只需几行代码，便可实现令人惊叹的效果。本书展示的例子都是最简单的，而且考虑到学习的目的，亲手写这些代码是一种比较恰当的做法。通过将简单布局合并起来，你其实还可以做相当多的事情。不过，等到了某个时间之后，再手工编写 GUI 窗体便显得有些不大适合了——它会越来越复杂，不值得为此浪费大量宝贵的编程时间。Java 和 Swing 的设计者制作了专门用来支持 GUI 构建工具的语言和库。利用那些 GUI 构建工具，你写起程序来会更加轻松。只要理解了布局的基本原理，以及如何对事件进行控制（后文还会详述），那么很快就可以放弃用手工方式来控制布局——改为用一个恰当的工具来帮助你（无论如何，Java 本来就是设计用来提高程序员的工作效率的）。

13.7 Swing 事件模型

在 Swing 事件模型中，一个组件可以发起（“触发”）一个事件。每类事件都由一个不同的类表示。一个事件“触发”之后，一个或者多个“监听器”（Listener）就会接收到它，后者会针对事件采取行动。因此，一个事件的“发起地”和事件最后得以控制的地方可以完美地分隔开。使用 Swing 组件是一回事，组件接收到一个事件后用代码去控制它又是另一回事——这是“区分接口和实现”的一个绝佳范例！

每个事件监听器都属于一个特殊类的对象，那个类实现了特定类型的监听器接口。因此作为程序员，我们要做的唯一事情就是创建一个监听器对象，然后向“触发”事件的那个组件注册它的存在。这个注册是通过在事件触发组件中调用一个 addXXXListener()方法来完成的。其中的“XXX”代表你想“监听”的事件类型。因此，只需观察一下“addListener”方法的名字，就可轻易地知道哪种类型的事件会得到控制（处理）。而且假如你试图监听错误的事件，那么在编译时间就会报错。在本章的后面部分，大家还会知道 JavaBeans 也利用了“addListener”方法的名字，来判断一个 Bean 到底可以控制什么事件。

事件逻辑之后，紧接着便要进入一个监听器（Listener）类的内部。在你创建一个监听器类的时候，唯一的限制就是它必须实现正确的接口。你当然可以创建一个全局性的监听器类，但在这种情况下，内部类可以发挥出更大的用处——因为内部类不仅能在用户界面（UI）或者事务逻辑类中对你的监听器进行逻辑分组，而且可以保持到自己的父对象的一个引用，从而完美地跨越类和子系统边界进行调用。

本章迄今为止的所有例子其实都采用了 Swing 事件模型。在本节剩余的部分，则会对那个模型的细节进行完善。

13.7.1 事件和监听器类型

所有 Swing 组件都同时包括了 addXXXListener()和 removeXXListener()这两个方法，以便从每个组件中增删恰当类型的监听器。大家会注意到，每种情况下的“XXX”同时也代表着为方法指定的参数，例如 addMyListener(MyListener m)。下表总结了基本的事件、监听器以及方法，同时还介绍了通过提供 addXXXListener()和 removeXXListener()方法，为那些特定事件提供支持的基本组件。同时，大家也要注意这样一个问题：事件模型设计的一项基本宗旨是便于以后的扩展，所以假如以后看到了未在本表中列出的事件和监听器类型，那么也丝毫不必觉得惊讶。

事件、监听器接口和添加/删除方法	支持该事件的组件
ActionEvent ActionListener addActionListener() removeActionListener()	Jbutton、JList、JTextField、JMenuItem 及其后裔——包括 JCheckboxMenuItem、JMenu 和 JpopupMenu
AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener()	JScrollbar 以及你创建的、实现了 Adjustable 接口的任何东西
ComponentEvent ComponentListener addComponentListener() removeComponentListener()	☆Component 及其后裔——包括 JButton、JCanvas、JCheckbox、JChoice、JContainer、JPanel、JApplet、JScrollPane、Window、JDialog、JFileDialog、JFrame、JLabel、JList、JScrollbar、JTextArea 和 JTextField

事件、监听器接口和添加/删除方法	支持该事件的组件
ContainerEvent ContainerListener addContainerListener() removeContainerListener()	Container 及其后裔，包括 JPanel、JApplet、JScrollPane、Window、JDialog、JFileDialog 和 JFrame
FocusEvent FocusListener addFocusListener() removeFocusListener()	Component 及其后裔*
KeyEvent KeyListener addKeyListener() removeKeyListener()	Component 及其后裔*
MouseEvent（同时包括点击和移动） MouseListener addMouseListener() removeMouseListener()	Component 及其后裔*
MouseEvent[64]（同时包括点击和移动，注释） MouseMotionListener addMouseMotionListener() removeMouseMotionListener()	Component 及其后裔*
WindowEvent WindowListener addWindowListener() removeWindowListener()	Window 及其后裔，包括 JDialog、JFileDialog 和 JFrame
ItemEvent ItemListener addItemListener() removeItemListener()	JCheckbox、JCheckboxMenuItem、JComboBox、JList 以及实现了 ItemSelectable 接口的所有东西
TextEvent TextListener addTextListener() removeTextListener()	自 TextComponent 派生的任何东西，包括 JTextArea 和 JTextField

可以看到，每种类型的组件都只支持特定类型的事件。因此，要为每个组件都找出它们支持的所有事件，就显得颇为困难。一个更简便的做法修改第 12 章的 ShowMethodClear.java 程序，使其显示出由你指定的任意 Swing 组件支持的全部事件。

第 12 章向大家介绍了“反射”的概念，并当场就利用它为一个指定的类查找对应的方法——结果要么是一个完整的方法列表，要么是那些方法的一个子集（它们的名字与你指定的关键字相符）。这里最有意思的地方在于，它能自动展现出一个类的所有方法，同时并不要求你遍历整个继承树，并在每一个“分权”都对基类进行检查。换言之，这个工具能节省我们大量宝贵的编程时间。由于大多数 Java 方法的名字都取得非常好，仅从名字便能看出它的全部含义，所以在你需要某种功能的时候，试着输入一个重要的单词，往往都能反馈回

[64]尽管想起来似乎是理所当然的，但其实并没有什么 MouseMotionEvent。点击（无论用左键还是右键）和鼠标指针在屏幕上的移动都被划归 MouseEvent 一类，都属于一种“鼠标事件。因此，MouseEvent 在这张表里再度出现并非偶然，也并非一处错误。

恰当的方法名。取得需要的方法名之后，再根据它检索联机文档，调阅详细说明。

不过，在第 12 章的时候，大家还不了解 Swing，所以那一章的示范程序只能是“基于命令行”的。不过，我们现在终于可以对其进行完善，制作一个更有用的 GUI 版本。它针对在 Swing 组件中对“addListener”方法的查找，进行了特别的设计：

```
//: c13: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.io.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class ShowAddListeners extends JApplet {
  Class cl;
  Method[] m;
  Constructor[] ctor;
  String[] n = new String[0];
  JTextField name = new JTextField(25);
  JTextArea results = new JTextArea(40, 65);
  class NameL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nm = name.getText().trim();
      if(nm.length() == 0) {
        results.setText("No match");
        n = new String[0];
        return;
      }
      try {
        cl = Class.forName("javax.swing." + nm);
      } catch(ClassNotFoundException ex) {
        results.setText("No match");
        return;
      }
      m = cl.getMethods();
      // Convert to an array of Strings:
      n = new String[m.length];
      for(int i = 0; i < m.length; i++)
```

```
        n[i] = m[i].toString();
      reDisplay();
    }
  }
  void reDisplay() {
    // Create the result set:
    String[] rs = new String[n.length];
    int j = 0;
    for (int i = 0; i < n.length; i++)
      if(n[i].indexOf("add") != -1 &&
        n[i].indexOf("Listener") != -1)
          rs[j++] =
            n[i].substring(n[i].indexOf("add"));
    results.setText("");
    for (int i = 0; i < j; i++)
      results.append(
        StripQualifiers.strip(rs[i]) + "\n");
  }
  public void init() {
    name.addActionListener(new NameL());
    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));
  }
  public static void main(String[] args) {
    Console.run(new ShowAddListeners(), 500,400);
  }
} ///:~
```

其中，StripQualifers 类原本是在第 12 章里定义的，这里通过导入 com.bruceeckel.util 库，对其进行了“复用”。

GUI 包含了一个名为“name”的文本字段（JTextField name），可在其中输入你想查找的类名。结果显示于一个 JTextArea 中。

大家会注意到，程序中并没有指定一个特殊的按钮或其他组件，指出自己希望从它那里开始搜索。那是由于 JTestField 会自动达到一个 ActionListener 的监视。只要作出了一项改变，或者按下了回车键，列表马上就会更新。假如发现有文字输入，就在 Class.forName()方法中用它查找相应的类。如输入的名字不正确，Class.forName()也会失败——它会产生一个违例。违例会被 JTextArea“捉”住，它会指出“No match”（没有匹配项）。但假如键入的是一个正确的名字（包括大小写），ClassforName()就会成功执行，而 getMethods()方法会返回由 Method 对象构成的一个数组。数组中的每个对象都会通过 toString()方法转变成一个

字串（从而产生了完整的方法签名），并将结果添加到一个名为 n 的字串数组里。n 数组属于 ShowAddListeners 类的一名成员；调用 reDisplay()的时候，会根据那个数组对显示进行更新。

reDisplay()会创建一个字串数组，名为 rs（“rs”代表“结果集”，即 Result Set）。结果集会有条件地从 n 的字串内复制——要求名字里含有“add”和“Listener”字样。随后，我们用 indexOf()和 substring()删除象 public、static 这样的限定词。最后，StripQualifiers.strip()用于删除多余的名字限定词。

利用这个程序，可以方便地调查一个 Swing 组件的功能。只要知道了一个组件支持的事件，便不用再费心去查询该如何对那个事件作出响应。你只需要：

(1) 取得事件类的名字，删除其中的“Event”字样。在剩下的部分中，再添加“Listener”字样。这便是你在自己的内部类中必须实现的监听器接口！

(2) 实现上述接口，然后为自己想要捕捉的事件编写恰当的方法。例如，假如你想捕捉鼠标移动事件，那么就要为 MouseMotionListener 接口的 mouseMoved()方法编写代码（当然，还必须实现其他方法；但大家不久就会看到，对此还有更便利的办法可供采用）。

(3) 为步骤(2)的监听器类创建一个对象，向你的组件注册它——在监听器名字的前面加上“add”字样，便得到了最终的方法名，比如 addMouseMotionListener()。

下表总结了部分监听器接口：

带有适配器（Adapter）的监听器接口	接口中的方法
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)

当然，这并不是一张非常详尽的表格——部分原因是由于事件模型允许我们创建自己的

事件类型以及对应的监听器。因此，大家经常都会碰到一些实现了自己的专用事件的库。不过，只要掌握了本章的知识，对付任何“陌生”事件都应该是不费吹灰之力的。

用监听器适配器简化操作

从上表可以看出，有的监听器接口只有一个方法。这些接口实现起来非常容易，因为只有在希望编写那个特定的方法时，才需要实现它们。不过，同时带有多个方法的监听器接口用起来可就没那么轻松了。举个例子来说，在我们创建一个应用程序的时候，无论如何都要向 JFrame 提供一个 WindowListener。这样一来，以后一旦发生了 windowClosing()事件，就可调用 System.exit()来退出应用程序。但由于 WindowListener 本身是一个“接口”，所以必须同时实现它的所有方法——即使你并不打算用那些方法做什么。显然，这非常麻烦！

为解决这个问题，含有多个方法的某些（但非全部）监听器接口是通过“适配器”（Adapters）来提供的，它们的名字可在上表中看到。每个“适配器”都为每个接口方法提供了默认的空方法。这样一来，我们就可以从适配器继承，然后只覆盖自己想改变的那些方法就可以了，不必一古脑儿地全部实现——不管到底有用还是没用。例如，我们通常可以象下面这样使用 WindowListener（请记住它已封装到由 com.bruceeckel.swing 提供的 Console 类里）：

```
class MyWindowListener extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

“适配器”唯一的用途就是使监听器类的创建变得更加容易。

不过，它也有一个缺点。假定我们象上面那样写一个 WindowAdapter：

```
class MyWindowListener extends WindowAdapter {
  public void WindowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

但它事实上是没有用的。而且假如不知根由，最后或许会把你给急死——因为一切都看似正常，既能编译，又能运行，只是关闭窗口的同时不能退出程序！现在，你知道问题出在哪儿了吗？关键在于使用的方法名是 WindowClosing()，而不是 windowClosing()。仅仅是一处小小的大小写错误，便造就了一个全新的方法！然而，当窗口关闭的时候，调用的可不是这个全新的方法，所以结果当然会令你“意外”。另一方面，尽管有点儿不方便，但“接口”可以保证方法都得以正确的实现！

13.7.2　跟踪多个事件

为证明这些事件都被真正地“触发”，而且也作为一个有趣的实验，我们有必要创建一个小程序，用它跟踪在 JButton 里发生的其他行为（而不仅仅是它是否被按下）。本例同时

向大家揭示出如何继承自己的按钮对象，因为对你感兴趣的所有事件来说，那才是它们的“终极”目的。好了，为了达到该目的，我们只需从 JButton 中继承就可以了[65]。

MyButton 类属于 TrackEvent 的一个内部类，所以 MyButton 能同父窗口打交道，并对它的文本字段（TextField）进行操作。要想把状态信息写到父窗口的“字段”里，这一点是必需的！当然，这也并非一种最完美的方案，它仍然存在限制，因为 myButton 只能同 TrackEvent 联合使用。这种形式的代码有时也叫作“高度耦合”：

```
//: c13: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 {
  HashMap h = new HashMap();
  String[] event = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited","mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  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) {
```

[65] 在 Java 1.0/1.1 中，从按钮对象继承并没有多大用处；那是老版本的一系列基本设计缺陷之一。

```
        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 构造函数中，按钮颜色通过对 setBackground()的一个调用来设置。所有监听器都是用简单的方法调用来安装的。

TrackEvent 类包含了一个 HashMap，它用于容纳表示事件类型的字串；同时还包含 JTextField，与事件有关的信息便保存在其中。当然，你也可以用静态方式来创建它们，而不是把它们放到一个 HashMap 里。但我们也不得不认为，HashMap 在这儿显得方便得多——既易于使用，也易于修改。特别要指出的是，假如你需要在 TrackEvent 中增添或删除一种新类型的事件，那么只需在 even 数组里增删一个字串就可以了——其他操作都是自动的，不必再由你来操心。

调用 report()的时候，为向它传递事件名以及来自事件的参数字串。收到这些信息后，它会在自己的外层类里用 HashMap h 来查找同那个事件名对应的实际 JTextField。找到后，将参数字串放到那个 JTextField 里。

这个例子其实比这儿说的还要有趣的得多，因为你可真正体验到对程序中的事件进行操纵的“快感”。

13.8 Swing 组件一览

现在，大家已理解了布局管理器和事件模型。掌握了这两个概念之后，接着应该来看看具体如何运用不同的 Swing 组件。本节并不是对 Swing 组件及其特性的一个“彻底剖析”，我们讲述的只是它们最基本的一些东西。每个例子都尽可能地精简，便于你移植这些代码，把它们用到自己的程序中去。

要想看到这些例子实际运行的样子，可以直接浏览相应的 HTML 网页——它们以本章的源码形式提供，你可以从 www.BruceEckel.com 或者本书的配套光盘上找到。

同时请注意：

(1) 来自 java.sun.com 的 HTML 版用户联机文档详尽讲述了所有 Swing 类和方法（其中只有少数在本章进行了演示）。

(2) 由于 Swing 事件采用令人相当舒适的命名规范，所以针对一种特定类型的事件，即

使你对它是完全陌生的，也往往能够轻松地“猜”出如何为它编写和安装一个控制模块。利用本章早些时候的 ShowAddListeners.java 检索程序，可以方便地调查由一个组件提供的功能。

(3) 如果你要构建的系统较为复杂，请考虑使用一个 GUI 构造函数，简化自己的工作。

13.8.1 按钮

Swing 提供了多种不同类型的按钮。所有按钮、复选框和单选钮（甚至包括菜单项）都是从 AbstractButton 继承的（由于其中也包括了“菜单项”，并不仅仅是“按钮”，所以再叫作“AbstractButton”恐怕不太合适，应该用“AbstractChooser”或类似的说法才是）。大家不久便会看到菜单项的运用，下例只展示了各类按钮的运用：

```
//: c13: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 {
  JButton jb = new JButton("JButton");
  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);
  }
} ///:~
```

程序首先从来自 javax.swing.plaf.basic 的 BasicArrowButton 开始，然后展示不同类型的按钮。运行本例的时候，会看到开关按钮记着它上一次的位置（按下或松开）。但复选框和单选钮相互间的行为是一模一样的，只是“开”或“关”（它们自 JToggleButton 继承）。

按钮组

如果想让单选钮产生“异或”行为（按下了这个，那个便要松开——同时只能按下一个！），就必须把它们划分到一个按钮组（ButtonGroup）内。但是，就象下例演示的那样，其实任何 AbstractButton 都能添加到一个按钮组内。

为避免重复过多的代码，这个例子利用“反射”机制来生成由不同类型的按钮构成的按钮组。具体操作是在 makeBPanel()中进行的，它会创建一个按钮组以及一个 JPanel。makeBPanel()的第二个参数是一个字串数组。对于每个字串，由第一个参数表示的那个类的一个按钮都会添加到 JPanel 中：

```
//: c13: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 {
  static String[] ids = {
    "June", "Ward", "Beaver",
    "Wally", "Eddie", "Lumpy",
  };
  static JPanel
  makeBPanel(Class bClass, String[] ids) {
    ButtonGroup bg = new ButtonGroup();
    JPanel jp = new JPanel();
    String title = bClass.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 = bClass.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 " +
          bClass);
      }
      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);
  }
} ///:~
```

边框标题从类名中取出，再剔去其中的所有路径信息。随后，将 AbstractButton 初始化成一个 JButton，并为其指定了一个叫作“Failed”（失败）的文本标签。因此，即使你忽略违例消息，仍然可在屏幕上看到发生了什么问题。getConstructor()方法用于产生一个构造函数对象，它根据由传递给 getConstructor()的 Class 数组保存的一系列类型，取得自己的参数数组。随后，我们只需调用 newInstance()，为其传递一个对象数组，并在数组中包括自己实际的参数就可以了——在这种情况下，实际参数就是来自 ids 数组的字串。这便为一个原本简单的操作添加了少许复杂性。要想使按钮具有“异或”行为，我们需要创建一个按钮组，然后将希望具有该行为的每一个按钮都添加到这个组中。以后运行程序的时候，就会发现除 JButton 之外的所有按钮都表现出了这种“异或”行为。

13.8.2 图标

我们可在一个 JLable 或从 AbstractButton 继承的任何东西（包括 JButton、JCheckbox、JradioButton 及不同类型的 JMenuItem）内使用一个图标（Icon）。为 JLabel 使用图标是个非常简单、直观的（如后例所示）。下面这个例子展示了随按钮及其后裔使用图标的各种方法。

可用任何 gif 图形文件来制作自己的图标，本例使用的是这本书源码的一部分，可从 www.BruceEckel.com 下载。要想打开一个文件导入图像，只需创建一个 ImageIcon，并向其传递文件名即可。在这之后，就可在自己的程序中使用结果生成的 Icon 了。

注意路径信息在这个例子中是“硬编码”进去的；根据你的实际情况，可能要修改一下路径，使其和图像文件的位置对应。

```
//: c13:Faces.java
// Icon behavior in Jbuttons.
// <applet code=Faces
//  width=250 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Faces extends JApplet {
  // The following path information is necessary
  // to run via an applet directly from the disk:
  static String path =
    "C:/aaa-TIJ2-distribution/code/c13/";
  static Icon[] faces = {
    new ImageIcon(path + "face0.gif"),
    new ImageIcon(path + "face1.gif"),
    new ImageIcon(path + "face2.gif"),
    new ImageIcon(path + "face3.gif"),
    new ImageIcon(path + "face4.gif"),
  };
  JButton
    jb = new JButton("JButton", faces[3]),
    jb2 = new JButton("Disable");
  boolean mad = false;
  public void init() {
    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);
  }
} ///:~
```

Icon 可在大量构造函数中使用，但也可用 setIcon()添加或修改一个 Icon。这个例子同时展示了当一个按钮身上发生不同的事件时，如何利用 JButton（或任何 AbstractButton）来设置不同样式的图标。也就是说，当按钮被按下、禁用或者鼠标指针移过它的时候，可以分别用不同的图标进行显示，从而营造出一种十分好看的动态效果。

13.8.3　工具提示

前例还为按钮添加了一个“工具提示”（Tool Tip）。我们用来创建用户接口的几乎所有类都是从 JComponent 派生来的，后者包括了一个名叫 setToolTipText(String)的方法。因此，对于我们放置在窗体上的几乎任何东西来说，唯一要做的事情便是执行（这里假定一个名为 jc 的对象，它代表任意“从 JComponent 派生的类”）：

```
jc.setToolTipText("My tip");
```

其中的“My tip”可换成你要显示的工具提示的正文。这样一来，只要将鼠标指针停放

在 JComponent 上方，那么稍待片刻，就会在指针旁边弹出一个小方框，里面显示出对该按钮的说明（工具提示）。

13.8.4 文本字段

下例展示了 JTextField 具有的其他功能：

```
//: c13: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 {
  JButton
    b1 = new JButton("Get Text"),
    b2 = new JButton("Set Text");
  JTextField
    t1 = new JTextField(30),
    t2 = new JTextField(30),
    t3 = new JTextField(30);
  String s = new String();
  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 {
  boolean upperCase = true;
  public void setUpperCase(boolean flag) {
    upperCase = flag;
  }
  public void insertString(int offset,
    String string, AttributeSet attributeSet)
```

```
    throws BadLocationException {
      if(upperCase)
        string = string.toUpperCase();
      super.insertString(offset,
        string, attributeSet);
  }
} ///:~
```

其中，我们新增了 JTextField t3。一旦 JTextField t1 的行动监听器被触发，就在这里进行报告。可以看到，只有在我们按下了回车键之后，JTextField 的行动监听器才会被“触发”。

JTextField t1 同时设置了多个监听器。其中，T1 是一个 DocumentListener，用于对“文档”内的任何变动（这里是 JTextField 的内容发生改变）作出响应。它会将所有文本自动从 t1 复制到 t2。除此以外，t1 的文档被设置成 PlainDocument 的一个派生类，名为 UpperCaseDocuement，它会强迫所有字符都变成大写形式。它可自动侦测退格键，并执行删除，也可对插入光标进行控制。总之，它能按我们希望的样子，进行全面、完美的控制。

13.8.5 边框

JComponent 包含了一个名为 setBorder()的方法，用它可为任何可见的组件加上自己喜欢的边框。下面这个例子演示了各式各样的边框，我们用一个名为 showBorder()的方法来创建一个 JPanel，后在每种情况下都加上不同的边框。另外，它用 RTTI 查找当前正在使用的边框的名字（剔除所有路径信息），然后在面板中央的 JLabel 上显示出这个名字。

```
//: c13: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 FlowLayout());
    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);
  }
} ///:~
```

当然，你也可创建自己的边框，并将它们放在按钮、标签以及从 JComponent 派生出来的其他任何东西内部。

13.8.6　JScrollPane

大多数时候，JScrollPane 的默认行为已足以满足我们的要求。但是，你完全可以自行指定希望的滚动条种类——垂直、水平、两个都要或者两个都不要。

```
//: c13:JScrollPanes.java
// Controlling the scrollbars in a JScrollPane.
// <applet code=JScrollPanes 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 JScrollPanes extends JApplet {
  JButton
    b1 = new JButton("Text Area 1"),
    b2 = new JButton("Text Area 2"),
    b3 = new JButton("Replace Text"),
    b4 = new JButton("Insert Text");
```

```
  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);
  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 B1L 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);
  }
} ///:~
```

通过在 JScrolPane 构造函数中使用不同的参数，可对允许的滚动条进行控制。本例也利用了边框，使显示效果更加悦目。

13.8.7　一个小型编辑器

JTextPane 控件为文字编辑提供了众多支持，而且你不用费太大的劲就可以运用自如。下例对这些功能进行了简单运用：

```
//: c13:TextPane.java
// The JTextPane control is a little editor.
// <applet code=TextPane width=475 height=425>
// </applet>
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class TextPane extends JApplet {
  JButton b = new JButton("Add Text");
  JTextPane tp = new JTextPane();
  static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public void init() {
    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);
  }
} ///:~
```

按钮的作用只是添加一些随机性文字。JTextPane 的设计宗旨是实现文字的“现场”编辑，所以你会看到并没有什么 append()方法。在这种情况下，必须用 setText()将文字捕捉下来，进行修改，再送回窗体——显然，尽管 JTextPane 有着丰富的功能，但我们在这里并没有十分完美地加以运用。

就象早先指出的那样，一个小程序的默认布局行为是采用 BorderLayout（边框布局）。假如在未指定任何细节的前提下，将什么东西送入窗体，那么它会先占据窗体的中央区域，再向四周延展。不过，假如你象上述程序那样，指定了一个限定区域（NORTH、SOUTH、EAST 或 WEST），那么组件的范围就被限定在那个区域之内——就目前来说，按钮会在屏幕底部不断地堆叠（SOUTH）。

同时请注意一下由 JTextPane 提供的内建特性，比如自动换行等等。查阅 JDK 文档，你还能了解到它的其他大量特性。

13.8.8 复选框

每个“复选框”（CheckBox）只有两种状态：选定或者未选定（或者开与关）。它由一个小方框和一个标签构成。如果选定，小框里通常会出现一个“X”标记；如果未选定，则留空。

我们通常用一个构造函数来创建一个 JCheckBox，该构造函数需要取得一个标签作为自

己的参数。创建好 JCheckBox 之后，我们可以查询和设置它的状态。如果愿意，也能取得或修改它的标签。

无论设置还是清除一个 JCheckBox，都会马上产生一个事件。和按钮一样，利用 ActionListener，你可以捕捉这个事件。下面的例子用一个 JTextArea 来罗列已被选中（设置）的所有复选框：

```
//: c13: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 {
  JTextArea t = new JTextArea(6, 15);
  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);
  }
```

```
  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);
  }
} ///:~
```

通过 append()，trace()方法将选中的 JCheckBox 的名字及其当前状态发给 JTextArea。因此，我们可以一目了然地看出当前选中的复选框以及它们的状态。

13.8.9 单选钮

单选钮（Radio Button）在 GUI 程序设计中的概念来自于老式的电子管汽车收音机的机械按钮：按下一个按钮后，其它按钮便会弹起。换句话说，它强迫我们只能在一系列选择中作出一个选择。

为了设置由相关的 JRadioButton 构成的一个按钮组，我们唯一要做的就是把那些按钮添加到一个 ButtonGroup 里（一个窗体里，允许同时存在任意数量的 ButtonGroup）。可选择将其中一个按钮的初始状态设为 true（用构造函数的第二个参数）。但是，假如将多个单选钮都设成 true，那么按钮组内只有最后设置的那个才会有效。

下面是运用单选钮的一个简单例子。注意对单选钮事件的捕捉和其他组件并无区别：

```
//: c13: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 {
  JTextField t = new JTextField(15);
  ButtonGroup g = new ButtonGroup();
  JRadioButton
    rb1 = new JRadioButton("one", false),
    rb2 = new JRadioButton("two", false),
    rb3 = new JRadioButton("three", false);
  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);
  }
} ///:~
```

为了显示出状态，我们使用了一个文本字段（Text Field）。这个字段被设置成“不可编辑”，因为我们只想用它来显示数据，而不是编辑它。换言之，它在这里可以取代 JLabel 的地位，达到一模一样的效果。

13.8.10　组合框（下拉列表）

和单选按钮组类似，下拉列表也可以强迫用户“多选一”。不过，它采用的是一种更简化的方式，而且能够更方便地修改列表中的元素，不容易使用户感到“意外”（你当然也可以动态改变单选钮，但那样做显得过于“明目张胆”，对用户并不“礼貌”）。

Java 的 JComboBox（组合框）和大家在 Windows 中常见的“组合框”有所区别，后者允许用户从一个列表中作出选择，或者直接键入自己的选择（注意是“键入”）。但在使用 JComboBox 的时候，你无法自己键入选择文字，只能从列表中选择，而且只能选择一个。在下例中，JComboBox 框首先会准备好一系列选择项，然后在一个按钮被按下之后，在框内增添新的选择项：

```
//: c13:ComboBoxes.java
// Using drop-down lists.
// <applet code=ComboBoxes
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class ComboBoxes extends JApplet {
```

```
  String[] description = { "Ebullient", "Obtuse",
    "Recalcitrant", "Brilliant", "Somnescent",
    "Timorous", "Florid", "Putrescent" };
  JTextField t = new JTextField(15);
  JComboBox c = new JComboBox();
  JButton b = new JButton("Add items");
  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, 100);
  }
} ///:~
```

JTextField 会显示出“选中的索引”——亦即当前选中项目的序列编号；同时显示的还有单选钮上的标签。

13.8.11 列表框

“列表框”（List Box）和 JComboBox 组合框有着显著区别，而且并不仅仅是外观上的区别。在你选中一个 JComboBox 之后，它会马上“下拉”显示出一个列表；而 JList 无论如何都占据着屏幕上固定的行数，而且不会改变。如果想看到一个列表里的项目，只需调用 getSelectedValues()，它会产生由当前选中项目构成的一个字串数组。

JList 允许多重选定——可以按住 Ctrl 键不放，然后点击多个项目（它们不必挨在一起）。

被选中的项目会进入突出显示状态，而且你可根据需要，同时选定任意多个项目。假如先选中一个项目，然后按住 Shift 键不放，再点击另一个，那么两个项目之间的所有项目（包括它们两个）都会被连续选中。选中了多个项目之后，假如对其中的一个不满意，可以再按住 Ctrl 键，点击它一下就可以了。

```
//: c13: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 {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  DefaultListModel lItems=new DefaultListModel();
  JList lst = new JList(lItems);
  JTextArea t = new JTextArea(flavors.length,20);
  JButton b = new JButton("Add Item");
  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);
      }
    }
  };
  ListSelectionListener ll =
    new ListSelectionListener() {
      public void valueChanged(
        ListSelectionEvent e) {
          t.setText("");
          Object[] items=lst.getSelectedValues();
          for(int i = 0; i < items.length; i++)
            t.append(items[i] + "\n");
        }
```

```
    };
  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);
  }
} ///:~
```

按下按钮后，它会将新项目添加到列表的顶部（因为 addItem()的第二个参数是 0）。

同时，我们也为列表添加了装饰性的边框。

如果只是想把一个字串数组放到一个 JList 里，那么还有一个更简单的办法：将数组传递给 JList 构造函数，然后让它自动完成列表的构建。在上例中，之所以要采用“列表模型”，唯一的原因就是在程序执行期间，能对列表进行操作。

JList 并未提供对文字滚动的直接支持。当然，要想增添这一功能，我们只需将 JList 封装到一个 JScrollPane 里，其他事情便不用操心了！

13.8.12 卡片式对话框

JTabbedPane 允许我们创建一个“卡片式对话框”。在其最顶部，可见一排卡片标签。按下一个标签，便可显示出一个不同的对话框。这是节省空间的一个好办法，而且也非常形象和直观。

```
//: c13:TabbedPane1.java
// Demonstrates the Tabbed Pane.
// <applet code=TabbedPane1
// width=350 height=200> </applet>
```

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class TabbedPane1 extends JApplet {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  JTabbedPane tabs = new JTabbedPane();
  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 TabbedPane1(), 350, 200);
  }
} ///:~
```

在 Java 中，“卡片式对话框”显得异常重要，因为编写小程序（Applet）的时候，并不鼓励你使用“弹出式对话框”——对于从小程序上弹出的任何对话框，都会自动加上一点儿警告消息。

运行上述程序时，可见假如卡片标签数量太多，一行内装不下，JTabbedPane 就会另起一行。从外表上看，就是数量众多的卡片一张一张地“堆叠”起来了。要想看到这样的效果，在你从控制台命令行运行程序的时候，只需简单地改变一下窗口的大小就可以了。

13.8.13　消息框

视窗环境通常提供了一系列标准消息框，以便向用户快速发布信息，或者从用户那里取得消息。在 Swing 中，这些消息框都包含在 JOptionPane 中。尽管有着众多的选择（有的还相当复杂），但最常用的恐怕只有“消息对话框”和“确认对话框”，它们分别用 static JOptionPane.showMessageDialog()和 JOptionPane.showConfirmDialog()这两个方法调用。下例展示了一部分由 JOptionPane 提供的消息框：

```
//: c13: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 {
  JButton[] b = { new JButton("Alert"),
    new JButton("Yes/No"), new JButton("Color"),
    new JButton("Input"), new JButton("3 Vals")
  };
  JTextField txt = new JTextField(15);
  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);
  }
} ///:~
```

为了能写一个 ActionListener，我在这里采用了比较冒险的一种做法——检查按钮上的字串标签。它的问题在于很容易得到错误的标签（通常是大小写不对），而且这个 Bug 往后是很难检查出来的。

注意 showOptionDialog()和 showInputDialog()都提供了返回对象，其中包含了由用户输入的值。

13.8.14　菜单

对于能容纳菜单的每一个组件来说（其中包括 JApplet、JFrame、JDialog 以及它们的后裔），它们都提供了一个 setJMenuBar()方法，可接收一个 JMenuBar（一个组件同时只能有一个 JMenuBar）。我们将 JMenu 添加到 JMenuBar，并将 JMenuItem 添加到 JMenu。每个 JMenuItem 都可对应一个 ActionListener——一旦菜单项被选中，便自动触发这个“监听器”

和那些使用“资源”的系统不同，在 Java 和 Swing 中，必须在源代码中亲手合并所有菜单。下面是一个非常简单的菜单示例：

```
//: c13: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 {
  JTextField t = new JTextField(15);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      t.setText(
        ((JMenuItem)e.getSource()).getText());
    }
  };
  JMenu[] menus = { new JMenu("Winken"),
    new JMenu("Blinken"), new JMenu("Nod") };
  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);
  }
} ///:~
```

通过在“i%3”中利用模数运算符，可将菜单项分布到三个 JMenu 中。每个 JMenuItem 都必须有一个 ActionListener。不过要注意的是，尽管这儿使用的都是同一个 ActionListener，但在实际应用中，通常应该为每个 JMenuItem 都设置一个不同的 ActionListener。

从 JMenuItem 中继承了 AbstractButton，所以它也具有某些和按钮相似的行为。就其本身来说，它提供的是一个可放到下拉菜单里的选择项。但另外还有三种特殊的类型都是从 JMenuItem 继承的——JMenu 用于容纳其他 JMenuItem（用于实现级联式子菜单）；JCheckBoxMenuItem 用于产生一个勾号（√），指出当前菜单项是否被选中；而

JRadioButtonMenuItem 包含了一个单选钮。

下面是一个稍微复杂一些的例子，我们打算将不同口味的冰淇淋添加到菜单中。这个例子同时演示了级联式菜单、键盘助记符以及 JCheckBoxMenuItem 的效果，另外也揭示了如何对菜单进行动态修改：

```
//: c13: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 {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  JTextField t = new JTextField("No flavor", 30);
  JMenuBar mb1 = new JMenuBar();
  JMenu
    f = new JMenu("File"),
    m = new JMenu("Flavors"),
    s = new JMenu("Safety");
  // Alternative approach:
  JCheckBoxMenuItem[] safety = {
    new JCheckBoxMenuItem("Guard"),
    new JCheckBoxMenuItem("Hide")
  };
  JMenuItem[] file = {
    new JMenuItem("Open"),
  };
  // A second menu bar to swap to:
  JMenuBar mb2 = new JMenuBar();
  JMenu fooBar = new JMenu("fooBar");
  JMenuItem[] other = {
    // Adding a menu shortcut (mnemonic) is very
    // simple, but only JMenuItems can have them
    // in their constructors:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
    // No shortcut:
```

```
    new JMenuItem("Baz"),
  };
  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 cold? " + 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[0].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 Menus(), 300, 100);
  }
} ///:~
```

在这个程序中，我将菜单项放到数组里，然后遍历每一个数组，针对每个 JMenuItem 都调用 add()。这样做，可使一个菜单项的增减变得稍微容易一些。

该程序同时创建了两个 JMenuBar（而不是一个），以揭示出在程序运行期间，菜单条可以主动地交换。大家可以看到一系列 JMenu 是如何构成一个 JMenuBar 的，而且每个 JMenu 又是如何由 JMenuItem、JCheckBoxMenuItem 乃至其他的 JMenu（用于生成子菜单）构成的。一个 JMenuBar“装配”好之后，可以用 setJMenuBar()方法把它安装到当前程序里。注意当按钮按下之后，它会调用 getJMenuBar()，从而检查当前安装的是哪个菜单，然后会将其他菜单条放到它们应该在的位置。

测试“Open”（打开）操作时，注意拼写和大小写都是必须区分的。但是，即使没有发现和“Open”相符的，Java 也不会报错。这类漫不经心的字串比较是许多程序错误的根源。

菜单项的选定和撤消选定是自动进行的。对 JCheckBoxMenuItem 进行控制的代码揭示出我们可采用两个不同的办法来判断到底选中了哪一个：第一个办法是字串比较（但就象上面说的，这并非最安全的一种做法；它只是“可行”，但却不是最佳的）；第二个办法是对事件的目标对象进行对比。就象程序中展示的那样，我们用 getState()方法来揭示出状态。用 setState()，大家也可以改变一个 JCheckBoxMenuItem 的状态。

不过，菜单事件运用得有点儿不一致，往往会使人感动困惑：JMenuItem 用的是 ActionListener；但 JCheckBoxMenuItem 用的是 ItemListener。JMenu 对象也可以支持

ActionListener，但那样做通常没多大用处。通常，应该为每个 JMenuItem、JCheckBoxMenuItem 以及 JRadioButtonMenuItem 都指定相应的“监听器”，但这个例子却为各个菜单组件关联了 ItemListener 和 ActionListener。

Swing 提供了对“菜单助记符”——或者说“快捷键”——的支持。这样一来，你就能用键盘代替鼠标，选择自 AbstractButton 派生出来的任何一个东西（按钮、菜单项等等）。具体操作其实非常简单——对 JMenuItem 来说，可以使用一个重载的构造函数，为它的第二个参数赋予某个键的标识符。不过，大多数 AbstractButton 都没有象这样的构造函数，所以一种更常见的做法是使用专门的 setMnemonic()方法（“Mnemonic”就是“助记符”的意思）。上例不仅为按钮添加了助记符，也为部分菜单项添加了这样的设计；快捷按键会在组件上自动显现出来。

在这个例子中，大家也可看到 setActionCommand()的运用。从表面看，大家或许会觉得有点儿奇怪，因为在每一种情况下，“行动命令”和菜单组件上的标签都是完全一致的。那么，何不干脆用标签来代替这种附加的字串呢？其中最关键的原因便在于“国际化”。假如你的程序还想发布针对其他国家的语言版本，那么通常只希望修改菜单内的标签就可以了，不想再去修改代码（否则极易带来新的错误）。因此，为了便于代码检查同一个菜单组件关联在一起的文本字串，“行动命令”在菜单标签发生改变的时候可以丝毫不加改动。由于所有代码都直接同“行动命令”打交道，所以你可以任意修改菜单标签，而不会对代码造成任何干扰。注意在这个程序中，也并不是对所有菜单组件都检查它们的“行动命令”。那些没有检查的，表明它们没有对应的行动命令集。

监听器里还进行了“批处理”。BL 负责执行 JMenuBar 的交换。在 ML 中，我们采取了“断定谁在按铃”的手段——获得 ActiveEvent 的源，把它强制转型成一个 JMenuItem，然后获得行动命令字串，通过一个嵌套的 if 语句传递它。

尽管 FL 监听器控制着 Flavor 菜单里的不同“口味”，但它本身其实是非常简单的。假如你的程序逻辑足够简单，那么采用这种做法是非常恰当的。但在一般情况下，应考虑采用随 FooL、BarL 和 BazL 采用的方法，它们每一个都只同一个菜单组件联系在一起，所以不必增添额外的侦测逻辑，而且我们可以确定地知道是谁调用了监听器。以往这样做会牵涉到许多类，但内部代码可以变得更精简，而且也更安全。

可以看到，菜单代码很快就会变得非常冗长，越来越难辨读。在这种情况下，一个 GUI 构造函数应该是你的首要选择。只要工具称手，对菜单的维护也会简单许多。

13.8.15　弹出式菜单

实现一个 JPopupMenu（弹出式菜单）时，最直接的办法便是创建一个内部类，令其对 MouseAdapter 进行扩展，然后针对自己希望产生弹出行为的每一个组件，都添加那个内部类的一个对象：

```
//: c13: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 {
  JPopupMenu popup = new JPopupMenu();
  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(
          e.getComponent(), e.getX(), e.getY());
      }
    }
  }
```

```
  public static void main(String[] args) {
    Console.run(new Popup(), 300, 200);
  }
} ///:~
```

同样的 ActionListener 被添加到每个 JMenuItem 里，以便从菜单标签里提取出正文，并把它们插入 JTextField 里。

13.8.16 画图

在任何一个优秀的 GUI 框架中，画图操作应该非常容易——Swing 提供的正是这样一个出色的 GUI 框架。对几乎所有画图问题来说，对画图位置的计算都要比调用画图例程本身复杂得多。而且这些计算通常又与画图调用混合到一起，所以最后使接口显得异常复杂。

为简化起见，请考虑在屏幕上表示数据的问题——在这里，我们的“数据”由内建的 Math.sin()方法提供，它是一个数学正弦函数。为了让我们要做的事情变得有趣一点儿，而且也为了进一步演示 Swing 组件的易用性，我们在窗体底部放置一个滑杆，用它动态控制屏幕上显示的正弦曲线的数量。除此以外，假如你重新改变窗口大小，会发现正弦曲线也会自动调整，以适应新窗口大小。

尽管任何你可以描绘任何 JComponent，并把它作为一张“画布”使用，但假如你只想得到一个直观的画面表面，那么通常还是应该从一个 JPanel 中继承。在这里，唯一需要覆盖的方法是 paintComponent()——组件必须重画的时候，便会调用该方法（通常不必关心这方面的问题，因为重画的决定是由 Swing 自己作出的）。调用时，Swing 会向其传递一个 Graphics 对象。随后，我们就可用这个对象在表面上画图了。

在下面的例子中，所有与画图有关技巧都在 SineDraw 类中反映出来了；SineWave 的作用很简单，只是用于配置程序以及那个滑杠控件。而在 SineDraw 中，setCycles()方法提供了一个挂钩，允许另一个对象（此时是滑杆控件）对周期数进行控制。

```
//: c13: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 {
  static final int SCALEFACTOR = 200;
  int cycles;
  int points;
  double[] sines;
  int[] pts;
  SineDraw() { setCycles(5); }
  public void setCycles(int newCycles) {
    cycles = newCycles;
```

```
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    pts = new int[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();
    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 {
  SineDraw sines = new SineDraw();
  JSlider cycles = new JSlider(1, 30, 5);
  public void init() {
    Container cp = getContentPane();
    cp.add(sines);
    cycles.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        sines.setCycles(
          ((JSlider)e.getSource()).getValue());
      }
    });
    cp.add(BorderLayout.SOUTH, cycles);
  }
  public static void main(String[] args) {
    Console.run(new SineWave(), 700, 400);
```

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

所有数据成员和数组都在正弦曲线点的计算过程中使用，其中包括：cycles 指出完整的正弦波形有几个（周期）；points 包含了构成正弦曲线的点数；sines 包含了正弦函数值；而 pts 包含了在 JPanel 上描绘的那些点的 y 坐标。setCycles()方法根据需要的点数来创建数组，并用数字填充 sines 数组。通过调用 rapaint()，setCycles()会强制 paintComponent()的调用，从而进行剩余的计算和重画操作。

覆盖 paintComponent()时，要做的第一件事情是调用方法的基类版本。然后，就可以按照自己的意愿做任何事情了；通常，我们需要使用一系列 Graphics 方法在 JPanel 中描绘像素。要了解那些方法，可以参考 java.awt.Graphics 的说明文档（请从 java.sun.com 下载完整的 Java HTML 版用户文档）。在这儿，大家可以看到几乎所有代码都在执行计算；唯一两个在屏幕上画图的方法是 setColor()和 drawLine()。在大家创建自己的程序时，通常也会产生这种“无奈”的感觉——大多数时间都不得不花在计算上面。反之，实际的画图操作却简单得不能再简单了！

在我创建这个程序的时候，大多数时间都在“努力”地让正弦曲线显示出来。完成之后，我又发现假如能动态地改变波形数，那么岂不是更加完美？不过，根据我以前用其他语言做这种事情的经验，刚开始并不是很情愿尝试这个。但最后，等我勉强去做了之后，却发现它原来是整个项目中最轻松的一部分！首先，我创建了一个 JSlider（构造函数参数分别为 JSlider 最左边的值、最右边的值以及起始值，但也可以使用其他构造函数），然后把这个新建的“滑杆”控件丢到 JApplet 里。随后，我查阅 HTML 文档，注意到唯一能用的“监听器”就是 addChangeListener——只要对滑杆进行了变动，使其足够产生一个不同的值，便会触发这个“监听器”。另外，唯一可用的方法则是 stateChanged()——这个名字猜都能够猜到！它提供了一个 ChangeEvent 对象，以便我们能回过头去，找出造成改变的“源头”，并调查新值是什么。通过调用 sines 对象的 setCycles()方法，新值便会生效，而 JPanel 会被重画。

通常，我们遇到的大多数 Swing 问题都可用与上面类似的步骤加以解决，而且会发现整个过程一般都非常简单——即便以前从未使用过一个特定的组件！

当然，假如你的问题真的很复杂，那么还有另一个更高级的画图工具可供利用，其中包括由其他厂商开发的 JavaBeans 组件以及专用的 Java 2D API（用于画 2D 图）。不过，这些“另类”方案已超出了本书的范围。在你真正需要的时候，可以自己查阅相关资料。

13.8.17　对话框

“对话框”（Dialog Box）是我们几乎每天都在打交道的东西。但是，你对它的本质了解吗？事实上，“对话框”的本质仍然是一个窗口，但它必须从另一个窗口中弹出。其目的是解决特定的问题，同时避免将那些问题的细节把原来的窗口搞得一团糟。对话框在视窗化的编程环境中得到了普遍采用，但在“小程序”中却很少用到。

为了创建一个对话框，你需要从 JDialog 继承。JDialog 只不过是另一种形式的窗口（Window），这和 JFrame 是差不多的。它提供了一个布局管理器（默认为 BorderLayout），我们可添加事件监听器，以便对事件进行处理。为其调用窗口关闭方法 windowClosing()时，一个显著的区别在于：我们并不打算同时将整个应用程序关闭！相反，我们通过调用 dispose()，只释放由对话框窗口占用的资源。下面是一个非常简单的例子：

```
//: c13: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 {
  JButton b1 = new JButton("Dialog Box");
  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);
  }
} ///:~
```

创建好 JDialog 后，必须调用 show()方法，来显示并激活它。要想关闭对话框，必须调用 dispose()。

大家会看见，从小程序上弹出的所有东西（包括对话框）都是“不受信任的”。也就是说，在弹出的窗口里，你会看到一条警告消息。这是由于在理论上，不怀好意的人会利用这

个弹出窗口糊弄用户，让他们相信这是一个标准的应用程序，并欺骗他们输入自己的信用卡号码或者其他敏感信息等等。由于小程序是面向整个 Web 的，所以这就显得更加危险。小程序肯定要同一个网页联系在一起，而且可在你的 Web 浏览器内部看见，但与此同时，一个对话框却可能与之分离（从理论上来说）。这正是小程序通常不使用对话框的原因，它会带来太大的安全隐患。

下面这个例子则要稍微复杂一些；对话框由一个特殊类型按钮的网格构成（使用 GridLayout 布局）。这个按钮在这里定义为 ToeButton 类，它会围绕自己画一个框，而且在中间可能画上一个“x”，一个“o”，或者什么都不画——具体由它当前的状态决定。最开始，它的中间当然是什么都不画的。以后随着顺序，会改变成一个“x”或者一个“o”。不过，假如你点击按钮，它也会在“x”和“o”之间来回切换。除此以外，通过在主应用程序窗口中修改数字，对话框也可以设置任意的行列数。

```
//: c13:TicTacToe.java
// Demonstration of 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 {
  JTextField
    rows = new JTextField("3"),
    cols = new JTextField("3");
  static final int BLANK = 0, XX = 1, OO = 2;
  class ToeDialog extends JDialog {
    int turn = XX; // Start with x's turn
    // w = number of cells wide
    // h = number of cells high
    public ToeDialog(int w, int h) {
      setTitle("The game itself");
      Container cp = getContentPane();
      cp.setLayout(new GridLayout(w, h));
      for(int i = 0; i < w * h; i++)
        cp.add(new ToeButton());
      setSize(w * 50, h * 50);
      // JDK 1.3 close dialog:
      //#setDefaultCloseOperation(
      //#  DISPOSE_ON_CLOSE);
      // JDK 1.2 close dialog:
      addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e){
```

```
        dispose();
      }
    });
  }
  class ToeButton extends JPanel {
    int state = BLANK;
    public ToeButton() {
      addMouseListener(new ML());
    }
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      int x1 = 0;
      int y1 = 0;
      int x2 = getSize().width - 1;
      int y2 = getSize().height - 1;
      g.drawRect(x1, y1, x2, y2);
      x1 = x2/4;
      y1 = y2/4;
      int wide = x2/2;
      int 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 ? OO : 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);
  }
} ///:~
```

注意“静态”（static）的东西只能放在类的外层，所以内部类不可能拥有静态数据或者静态内部类。

paintComponent()方法用于围绕面板（Panel）画一个方框，并负责“x”和“o”的描绘。尽管计算过程非常乏味，但幸运的是，整个过程还算比较直观。

鼠标点击事件由 MouseListener 这个监听器负责捕捉，它首先检查面板上是否写入了什么东西。如果没有，则对父窗口进行查询，调查是谁打开这个面板的，并据此建立 ToeButton 的状态。通过内部类机制，ToeButton 随后可自由地同自己的“父”沟通，并改变次序。假如按钮上已经显示了一个“x”或“o”，则翻转这一次序。注意在这些计算过程中，我们利用了由第 3 章讲述的三元 if-else 技术带来的便利。一个状态发生改变后，便重画一遍 ToeButton。

ToeDialog 的构造函数十分简单：它根据我们的意愿，在 GridLayout 里添加指定数量的按钮；然后针对每个按钮，按一边 50 像素的规格，改变它们的大小。

TicTacToe 通过创建 JTextField（这些文本字段用于输入按钮网格的行数和列数）以及“go”按钮（并加上它的 ActionListener 监听器），从而建立起一个完整的应用程序。按下按钮后，必须取得 JTextField 中的数据。而且由于数据原本采用的是字串格式，所以还得转换成 int 数值——具体转换是由 static Integer.parseInt()方法来完成的。

13.8.18 文件对话框

只有部分操作系统才提供了一些内建的对话框，以便用户选择象字体、颜色和打印机等等常规性的东西。然而，几乎所有图形化操作系统都支持文件的打开与保存。为此，Java 专门提供了一个 JFileChooser 类，用于简化这些文件操作。

下面这个应用程序为大家展示了两种形式的 JFileChooser 对话框，一个用于打开，另一个则用于保存。对于其中的大多数代码来说，大家现在都应该比较熟悉了。另外，所有真正有趣的操作都是在针对两个不同按钮点击的行动监听器中进行的：

```
//: c13:FileChooserTest.java
// Demonstration of File dialog boxes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class FileChooserTest extends JFrame {
  JTextField
    filename = new JTextField(),
    dir = new JTextField();
  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);
  }
} ///:~
```

注意你可以向 JFileChooser 应用许多特殊处理，其中包括利用过滤器，限定只能选择你规定的文件名。

要想显示一个“打开文件”对话框，我们需要调用 showOpenDialog()；要想显示“保存文件”对话框，则需调用 showSaveDialog()。除非对话框关闭，否则这些方法是不会自己返回的。JFileChooser 对象依然存在，所以我们能够从中读取数据。getSelectedFile()和 getCurrentDirectory()这两个方法用于查询用户操作的结果——他们选择了什么文件？当前的目录是什么？假如这些方法返回的是 null 值，表明用户已经取消了在对话框里的操作（按了 Esc 键，或者按了“取消”按钮）。

13.8.19 Swing 组件上的 HTML

对于任何组件来说，只要它能获取普通纯文本，那么必然也能获取 HTML 文本。假如取得的是 HTML 格式的文本，那么必须参照 HTML 的规则，对其进行重新格式化，再将结果显示出来——就象一个“Web 浏览器”那样。也就是说，我们可非常轻松地在一个 Swing 组件里显示出美妙的、五彩缤纷的文字。例如：

```
//: c13:HTMLButton.java
// Putting HTML text on Swing components.
// <applet code=HTMLButton width=200 height=500>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class HTMLButton extends JApplet {
  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);
  }
} ///:~
```

当然，你的文本必须用一个“<html>”标记起头，然后才能添加各种标准的 HTML 标记。要注意的是，平常用来“结束”的 HTML 标记并一定非要添加（比如，</html>这个标记大可不必加入其中）。

ActionListener 在窗体里添加了一个新的 JLabel 标签，它包含的也是 HTML 文本。不过，这个标签并不是在 init()期间添加的，所以必须调用容器的 validate()方法，从而强制对组件进行重新布局（这样才能让新标签真正显示出来）。

其他还可采用 HTML 格式文本的组件包括：JTabbedPane、JMenuItem、JToolTip、JRadioButton 和 JCheckBox。

13.8.20　滑杆和进度条

“滑杆”（Slider）已在前面那个正弦曲线例子中得到了运用。利用它，用户可操纵上面的一个“滑块”，通过来回的移动，从而改变数据。大家想象一下家用电器上的音量调节杆，便知道它是什么样子了。“进度条”（Progress Bar）则采用相对形式来显示从“空”到“满”的一个过程，从而使用户得到“一件事情正在逐渐完成”的概念，比如大家经常看到的“文件正在下载”进度条。我个人比较喜欢的一个例子是直接用滑杆来控制进度条——移动滑块的时候，进度条也会相应地改变。一箭双雕，何乐而不为？

```
//: c13: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 {
  JProgressBar pb = new JProgressBar();
  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);
  }
} ///:~
```

为了将两个组件“钩”到一起，关键在于共享它们的模型，就象下面这一行：

```
pb.setModel(sb.getModel());
```

当然，还需要用一个监听器来同时控制两个。但在并不复杂的环境中，这样的操作是非常轻松的。

JProgressBar 十分简单，但 JSlider 却有许多选项，例如拖动方向（水平左右拖，还是垂直上下拖）、那小滑块的钝头和尖头各有多大等等。在这个例子中，请注意添加一个带标题的边框（TitledBorder）是多么的容易！

13.8.21 树

要想使用一个 JTree（树），可以象下面这样简单地说明：

```
add(new JTree(
  new Object[] {"this", "that", "other"}));
```

这样显示出来的是一个非常初级的“树”形结构。不过，真正的树形结构 API 是非常复杂的——在 Swing 中，它无疑是最大的 API 之一。从表面看，它似乎可以无所不能。但事实上，在你需要做一些更复杂的事情时，还是得先进行一番研究和实验。

尽管这个库本身非常复杂，但幸运的是，它为我们“贴心”地提供了一些“默认”的树组件。平常，只需使用这些默认的东西，便足够满足自己的需要了。只有在特殊情况下，才需要从更大的深度去考察 JTree。

下面的例子采用了“默认”的树组件，目的是在一个小程序中显示一个“树”形结构。按下按钮后，就会在当前选中的“节点”增加了一个新的“子树”（如尚未选择任何节点，则从根节点开始派生）：

```
//: c13:Trees.java
// Simple Swing tree example. Trees can
// be made vastly more complex than this.
// <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 {
  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 {
  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" },
  };
  static int i = 0;
  DefaultMutableTreeNode root, child, chosen;
  JTree tree;
  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);
          // This puts the new node on the
          // currently 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);
  }
} ///:~
```

第一个类是 Branch（分支）。利用这个工具，我们可取得一个字串数组，然后将其中的第一个字串作为“根”，剩下的字串作为“叶”，构建起一个 DefaultMutableTreeNode 对象。在这之后，可调用 node()方法，来产生这个“分支”的根。

在 Tree 类中，包含了一个由字串构成的二级数组。根据它，便可构建起 Branch（分支）。同时还有一个 static int i，通过它可对该数组进行遍历（i 值会递增）。DefaultMutableTreeNode 对象容纳着各个节点。但是，那些节点在屏幕上的物理性表示却是由 JTree 及其对应模型——DefaultTreeModel——来控制的。请注意，当 JTree 添加到小程序之后，它会封装到一个 JScrollPane 里——原因不言而喻，是为了自动获得滚动显示控制。

JTree 通过它自己的模型来控制。当我们修改这个模型时，模型会产生一个事件，从而导致 JTree 对整个棵树中可见（会显示出来）的东西进行必要的更新。在 init()中，我们通过调用 getModel()方法，从而捕捉到这个模型。按钮按下后，便会创建一个新的“分支”。随后，会找到当前选定的组件（如果没有选定任何组件，则默认为根），然后由 insertNodeInfo()方法完成对树进行修改的所有工作，最后促使它进行更新。

平常，象这样的一个例子便可为我们提供需要在一个“树”里用到的全部东西。不过有些时候，“树”可能还有更大的用处——而不仅仅是上例显示的那些“默认”行为。为此，你可以替换自己的类，从而实验不同的行为。但必须注意的是，几乎所有这些类都有一个“巨大”的接口，不花大量时间、不费许多周折，你最后往往会无功而返。因此，在对“树”进行进一步的探索之前，请务必提醒自己小心谨慎。当然了，在普通应用中，你也没必要费那么大的劲儿。只要真正理解了上面那个例子，平常便能够从容地运用“树”了。

13.8.22 表格

和“树”一样，表格在 Swing 中相当庞大，功能也非常强。它们最主要的设计宗旨就是通过“Java 数据库连接”（即 JDBC，第 15 章有介绍），成为数据库的一种直观的、流行的“格子”接口。也就是说，和数据库保存数据的方式来个“一一对应”。因此，它被赋予了令人难以想象的灵活性——当然代价也是高昂的，整个类也显得异常复杂！如果真的对它来个“彻底剖析”，那么你学到的知识完全可用来自己做一个全功能的“电子表格”程序。不过，那可能需要一大本书来讨论才足够！因此，和往常一样，这里只打算让大家理解它的基础，并据此创建一个简单的 JTable。

JTable 控制着数据的显示方式，而 TableModel 控制着数据本身。因此，为了创建一个 JTable，通常首先得创建一个 TableModel。当然，你可以完整实现 TableModel 接口，但从 AbstractTableModel 这个“助手”类中继承，却往往是一种更简单、更流行的做法：

```
//: c13:Table.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 Table extends JApplet {
  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 Table(), 350, 200);
  }
} ///:~
```

DateModel 包括一个数据数组，但我们也能从其它地方得到数据，比如数据库等等。构造函数增加了一个 TableModelListener 监听器，用它在每次表格改变后，打印出数组。剩下的方法都遵守 Beans 命名规范；当 JTable 希望展示 DataModel 中的数据时，就会使用那些方法。AbstractTableModel 为 setValueAt()和 isCellEditable()设置了默认方法，它们都禁止对数据作出修改。因此，假如你希望自己能对数据进行编辑，便必须覆盖这些方法。

有了一个 TableModel 后，只需要把它传给 JTable 构造函数就可以了。至于显示、编辑和更新的细节，便不用你操心了。一切都会自动进行！在这个例子中，我们同样把 JTable 放到一个 JScrollPane 里。

13.8.23 选择外观与感觉

对 Swing 来说，它最有趣的概念之一就是“可插式外观与感觉”（Pluggable Look and Feel）。利用它，你的程序可以模仿出任何操作系统环境的“外观与感觉”。另外，你甚至可在程序执行期间，动态地改变它的“外观与感觉”。不过，通常两件事情中只能选做一个——要么选择能够“跨平台”的外观与感觉（这是 Swing 的基本设计宗旨），要么为当前使用的系统选择相应的外观与感觉，使你的 Java 程序看起来就象是专为那个系统设计的。具体用来选择的代码是非常简单的——只是在创建任何可视组件之前，必须先执行那些代码。这是由于后续的所有组件都会依据当前的外观与感觉而产生。而且即使在程序执行期间改变了外观和感觉，那些组件也不会自动地发生改变（这方面的主题过于复杂，也并不常用，请参考相应的 Swing 参考书）。

实际上，假如想使用跨平台的外观与感觉（这是最流行的做法，名称是“METAL”），那么其实根本不必专门做任何事情——这属于你的“默认”选择。但假如要使用当前操作系统的外观与感觉，就需要插入下列代码。通常，这些代码应该放在你的 main()之前；但某些时候，应该放在你添加的任何组件之前：

```
try {
```

```
    UIManager.setLookAndFeel(UIManager.
      getSystemLookAndFeelClassName());
  } catch(Exception e) {}
```

catch 从句其实不需要添加任何东西，因为假如你设置失败，UIManager 会自动选用默认的跨平台外观与感觉。不过，它对于违例的调试仍然是很有帮助的，所以你可考虑至少在 catch 从句中加一条 print 语句。

下面这个程序根据命令行参数来选择一种“外观与感觉”。注意它演示了在你选择的“外观与感觉”中，几个不同组件显示出来的样子：

```
//: c13: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 {
  String[] choices = {
    "eeny", "meeny", "minie", "moe", "toe", "you"
  };
  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(System.err);
      }
    } else if(args[0].equals("system")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getSystemLookAndFeelClassName());
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
    } else if(args[0].equals("motif")) {
      try {
        UIManager.setLookAndFeel("com.sun.java."+
          "swing.plaf.motif.MotifLookAndFeel");
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
    } else usageError();
    // Note the look & feel must be set before
    // any components are created.
    Console.run(new LookAndFeel(), 300, 200);
  }
} ///:~
```

从中看到，我们可明确指定一个字串，从而代表一种“外观与感觉”。就象 MotifLookAndFeel 那样，它表明要使用的是“Motif”。不过，它和默认的“Metal”外观与感觉是在任何平台上都能通用的唯一选择。尽管还存在着专用于 Windows 和 Macintosh 平台的“外观与感觉”字串，但它们却并非通用的，只能在规定的平台上使用（用 getSystemLookAndFeelClassName()查一查，便可知道一种特定的平台到底支持哪些“外观与感觉”）。

当然，你可以根据自己的偏爱，创建自己的一套“外观与感觉”。这样做有时是必需的。举个例子来说，在你为一家公司做事时，他们可能要求你设计具有该公司独特风格的一套程序。不过，这个题目可太“大”了。由于篇幅的限制，这里不可能再深入下去。请大家自己查阅相关的 Swing 参考书。

13.8.24 剪贴板

JFC 支持用系统剪贴板进行有限的操作（在 Java.awt.datatransfer 封装里）。我们可以将字串对象作为文本“复制”到剪贴板里，也可以将剪贴板中的文本“粘贴”到字串对象中。当然，剪贴板的设计宗旨是用来容纳任意类型的数据，但至于数据在剪贴板上是如何表达的，

却要由实际进行剪切和粘贴操作的程序来决定。Java 的剪贴板 API 通过“Flavor”（形式）的概念，提供了对剪贴板的扩展性支持。数据离开剪贴板时，它会被赋予一系列“形式”——根据需要，该数据可直接转换成那种“形式”（比如，一张数据图表既可表示成一串数字，也可以表示成一幅图像）。通过专用的方法进行检查，我们便知道特定的剪贴板数据是不是能够转换成我们目前感兴趣的那种“形式”。

针对一个 JTextArea 中的字串数据，下面这个程序简单地演示了如何对它们进行剪切（Cut）、复制（Copy）和粘贴（Paste）操作。在这里，大家会发现平常用来加快剪切、复制和粘贴操作速度的键盘快捷键同样可以使用。不过，这种功能并非该程序独有的。你再去看看其他任何程序中的 JTextField 或 JTextArea，便会发现它们其实也全都支持剪贴板的快捷键操作。这个例子只是简单地添加了对剪贴板的程序控制。假如你希望把剪贴板中的文本捕捉到除 JTextComponent 之外的其他东西里，那么也完全可以采用这儿介绍的技术。

```
//: c13: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  {
  JMenuBar mb = new JMenuBar();
  JMenu edit = new JMenu("Edit");
  JMenuItem
    cut = new JMenuItem("Cut"),
    copy = new JMenuItem("Copy"),
    paste = new JMenuItem("Paste");
  JTextArea text = new JTextArea(20, 20);
  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);
  }
} ///:~
```

通过前面的一系列实践，创建和增加菜单及 TextArea 到如今似乎已变成一种单调的活动。不同之处仅仅在于 Clipboard 字段 clipbd 的创建——这是通过 Toolkit 来完成的。

所有行动都在监听器内进行。CopyL 和 CutL 监听器几乎一模一样——除了 CutL 的最后一行是新增的之外。它的作用是删除已经复制的那一行文本。值得注意的是，我们根据字串创建了一个 StringSelection 对象；然后利用这个 StringSelection 对象，发出了对 setContents()的一个调用，以便将一个字串放到剪贴板上。

在 PasteL 中，我们用 getContents()方法将数据从剪贴板上拿下。最后得到的是一个“笼罩着一团迷雾”的 Transferable 对象——此时根本不知道其中到底包含了什么。要想知道其中的内容，一个办法是调用 getTransferDataFlavors()，它会返回由 DataFlavor 对象构成的一个数组，从而指出这个特定的对象到底支持哪些“形式”（Flavor）。另外，也可利用 isDataFlavorSupported()直接询问它是否支持自己感兴趣的一种“形式”。不过在这儿，我们采用的是最大胆的做法——直接调用 getTransferData()——因为可以认定剪贴板的内容支持“字串”（String）这种“形式”——它应该不会造成违例。

未来，完全可能有更多的“形式”获得支持，且让我们拭目以待。

13.9　将小程序打包到 JAR 文件里

JAR 工具的一项重要用途就是对小程序的装载过程进行优化。在 Java 1.0 中，人们倾向于将自己的所有代码都装到一个小程序类里，从而使客户机只需连接一次服务器，便能下载回全部的小程序代码。但这样做不仅会造成混乱、难以阅读和维护的程序，而且.class 文件采用的仍然是未经压缩的格式，所以仍然会浪费下载时间，不会使效率有多大的提高。

JAR 文件则不同，它解决了上述全部问题，而且做法也非常简单——将你的所有.class 文件都压缩到一个文件里，然后指示浏览器下载那个文件。换句话说，你再也不用担心自己生成过多的.class 文件。而且由于是压缩的，所以用户的下载速度可以快上许多。

现在以前述的 TicTacToe.java 为例。它看起来是一个单独的类，但实际上由于包含了五个内部类，所以总共要用到六个类。编译好程序之后，请用下面这个命令把它打包到一个 JAR 文件里：

```
jar cf TicTacToe.jar *.class
```

该命令假定当前目录下的.class 文件全都是 TicTacToe.java 要用到的（否则你会打包进去多余的东西）。

接下来，可创建一个 HTML 网页，在其中加上新的 archive 标记，指出要下载的 JAR 文件名是什么。注意这个标记采用的是标准的 HTML 标记格式，如下所示：

```
<head><title>TicTacToe Example Applet
</title></head>
<body>
<applet code=TicTacToe.class
       archive=TicTacToe.jar
       width=200 height=100>
</applet>
</body>
```

你需要把它放到本章早些时候演示的那个新的（同时也是麻烦的、复杂的）窗体里，否则它无法工作。

13.10 编程技术综述

由于 Java 的 GUI 编程是一种正在迅速进化的技术，在 Java 1.0/1.1 和 Java 2 的 Swing 库之间，存在着一些非常显著的改变，所以一些老的编程手段在前面一些新的 Swing 例子中仍然不得不采用。不过尽管如此，Swing 仍然允许我们采用比老模型更好的一些手段来进行编程。在这一节里，我打算介绍和探讨这样的一些编程手段，从而演示一下这些问题。

13.10.1 事件的动态绑定

对 Swing 事件模型来说，它的一项好处就是具有巨大的灵活性。只需单个方法调用，便可完成事件行为的添加与删除。下面这个例子演示了这一点：

```
//: c13: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 {
  ArrayList v = new ArrayList();
  int i = 0;
  JButton
    b1 = new JButton("Button1"),
    b2 = new JButton("Button2");
  JTextArea txt = new JTextArea();
  class B implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("A button was pressed\n");
    }
  }
  class CountListener implements ActionListener {
    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++);
      v.add(a);
      b2.addActionListener(a);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("Button2 pressed\n");
      int end = v.size() - 1;
      if(end >= 0) {
        b2.removeActionListener(
          (ActionListener)v.get(end));
        v.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);
  }
} ///:~
```

这个例子新作出的调整包括：

(1) 每个按钮都不止对应着一个监听器。通常，组件把事件作为“多强制转型”（Multicast）处理，这意味着我们可为单个事件注册多个监听器。但对特殊的组件来说，假如一个事件被当作“单强制转型”（Unicast）来处理，就会得到一个 TooManyListenersException（监听器过多）违例。

(2) 程序执行期间，监听器在按钮 B2 中动态地增加和删除。“增加”通过我们前面见到过的方法完成，但每个组件同时还有一个 removeXXXListener()方法，可用来删除各种类型的监听器。

这种灵活性为我们的编程提供了更强大的能力。

我们注意到，事件监听器调用时，并一定完全按它们当初添加时的顺序（尽管大多数实

现事实上都是那样做的）。

13.10.2 事务逻辑和 UI 逻辑的分离

设计类时，通常应保证每个类“只做一件事”。如果你特别关心用户界面的代码，这一点尤其重要。因为只要稍不留神，“想做什么”和“怎样显示”两个问题便被混为一谈了。如果你真把这两种逻辑合为一体了，那么代码以后便很难复用。因此，更正确的做法是将你的“事务逻辑”同 GUI 分开。这样一来，以后不仅程序的内部逻辑可方便地挪到其他地方使用，而且就连整个 GUI 也可以照搬过去。因此，为将来作想，请你务必慎重！

另一个问题是“分布式系统”。在这种系统中，“事务逻辑”驻留在另一台完全独立的机器中。这样一来，位于中心位置的事务规则便可服务于所有客户。要想修改它的话，也显得更加简单——只需要在一个地方修改就可以了，不用你到处跑。不过，由于这些事务对象会在不同的应用程序中利用，所以在设计它的时候，你不应该把它同任何特定的显示模式“挂钩”。只让它执行纯粹的事务操作，不要再让它负责其他事情！

下面这个例子证明了将事务逻辑同 GUI 代码分开是多么容易：

```
//: c13:Separation.java
// 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 {
  JTextField
    t = new JTextField(15),
    mod = new JTextField(15);
  BusinessLogic bl = new BusinessLogic(2);
  JButton
    calc1 = new JButton("Calculation 1"),
    calc2 = new JButton("Calculation 2");
  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);
  }
} ///:~
```

可以看到，BusinessLogic（事务逻辑）是一个非常“单纯”的类，它只管埋头做自己的工作，你在其中找不到它会在一个 GUI 环境中使用的哪怕是一丁点儿暗示。

Separation 用于跟踪所有 GUI 细节，它只通过自己的 public 接口同 BusinessLogic 打交道。所有操作都围绕信息在用户界面和 BusinessLogic 对象之间的来回传递而展开。所以同样地，Separation 其实也只在“专心”地做自己的工作。由于 Separation 只知道自己是在同一个 BusinessLogic 对象打交道（换言之，它们两者并没有紧密地“耦合”到一起），所以你用不着十分麻烦，便能让它换跟其他类型的对象打交道——“对不起，我可不是你 BusinessLogic 的奴隶!”。

留意着将 UI（用户界面）和事务逻辑分开，在你将那些老代码转移到 Java 中来的时候，事情也会变得简单许多。

13.10.3 正宗形式

内部类、Swing 事件模型、老事件模型依然支持以及有时候不得不利用老编程手段来运用新库特性……诸如此类的东西揉合到一堆，为代码的编写带来了更大程度的混淆。假如心里没有准备，你最后很容易写出令人不“爽”的代码。

这里要提醒大家的是，除非迫不得已，无论如何都应该坚持采用最简单、最明确的方式——用监听器类（通常写成内部类）来满足自己对事件进行控制的需要。这是本章大多数例子都在采用的“正宗”形式。

在这一模型的指导下，程序便可避免许多无谓的烦恼，不至于经常出现“我不知道发生这个事件的原因是什么”的情况。你的每一部分代码都要关心“怎样做”，而不要去关心“类型检查”。写程序时，假如能坚持遵循这一思路，往往可以起到事半功倍的效果——你的程序不仅能更容易地“概念化”，也能更容易地阅读和维护。

13.11 可视编程和 Bean

通过本书到目前为止的学习，大家已体验到在创建“可重复使用”的代码段时，Java

发挥了多么大的功劳！“最能重复使用”的代码单元应该是“类”，因为它由通用的“特征”（字段）和“行为”（方法）构成——你要么通过“合成”直接进行重复使用，要么通过“继承”来重复使用。

在面向对象的程序设计中，继承和多态是两个最基本的要素。不过在大多数情况下，当你构建一个特定的应用程序时，真正希望的却是一个组件只做你希望做的一件事情，而不要求它“什么都能做”。此时，我们希望能将不同的组件合并到自己的设计中，从而得到一个符合自己要求的设计——就象电子工程师将各种标准芯片装配到一块电路板上，让它发挥某种特定功能那样。看起来，似乎应该有一种手段，能帮助我们加快这种“模块化装配”式的编程速度。

随着微软的 Visual Basic（VB）的问世，“可视化编程”首次取得了成功——而且是巨大的成功！后来，我们又看到了第二代的可视化编程工具，它们以 Borland 公司的 Delphi 为代表——JavaBeans 的创作灵感其实大部分来源于此。利用这些编程工具，组件可采用“图形化”的方式表达出来。由于它们最后通常都要以某种可视组件的形式显示在屏幕上（比如按钮或者文本字段），所以这样的编程方式显得非常直观！事实上，对可视组件来说，无论在编程还是在实际运行的时候，你看到的通常都是一模一样的样子。要想添加一个组件，只需从工具盒里把它“拖”到自己的窗体上就可以了。在进行这样的拖放操作时，应用程序构造函数工具也会帮你自动编码，完全不用你来操心！

不过，要想得到一个完整的程序，仅把组件放到窗体上还不够。通常，你必须修改一个组件的“特征”，比如它的颜色是什么，在上面显示什么文字，连接哪个数据库……等等。在设计期间，这样的“特征”通常叫作“属性”（Properties）。在应用程序构建工具内部，你可以修改任意组件的“属性”。创建好程序后，这些配置数据会保存下来，程序启动时又会复原。

迄今为止，大家可能已习惯了“对象并非仅仅是一系列特征”这种说法——它还代表着一系列行为。在设计期间，一个可视组件的行为通常叫作“事件”（Events）。它的意思是说：“有一件事情在它（组件）身上发生了！”以前，我们需要对一个事件进行编码，从而在那个事件发生的时候，决定该采取什么样的操作。

这样便来到了最关键的一个部分：应用程序构建工具利用“反射”机制来动态查询组件，了解它支持哪些属性和事件。获得了需要的信息后，便可显示出属性，并允许你改变那些属性（构建程序时状态会保存下来），同时也显示出事件。例如，我们可双击一个组件，应用程序构建工具便会自动创建代码主体，并将它同那个特定的“双击”事件对应起来。这样一来，我们剩下的唯一工作就是编写那个事件发生之后要执行的代码。

所有这些都极大简化了我们的工作。因此，我们可将更多的精心放在程序的外观设计以及它要做的事情上——同时依赖应用程序构建工具来管理有关“连接”的所有细节。可视编程工具之所以会取得如此大的成功，它们对应用程序构建速度的大幅提高功不可没——用户界面的构建是明显的受益者，但应用程序的其他部分也同样从这种编程工具身上获得了好处！

13.11.1　什么是 Bean

拨去表面笼罩的所有迷雾，我们终于看清“组件”原来仅仅是一个代码块，它通常用一个“类”的形式表现出来。这儿的关键问题在于应用程序构建工具有能力查清楚一个组件的属性和事件是什么。为了创建一个 VB 组件，程序员必须写一个相当复杂的代码块，然后根据约定，揭示出它具有的属性和事件。Delphi 则不同，作为第二代可视编程工具的“老大”，它针对可视编程进行了更大程度的优化，所以你用它能够更轻松地创建出一个可视组件。然而，对 Java 来说，可视组件的创建才能在它的 JavaBeans 里才真正发挥到了巅峰——因为一

个 Bean 就是一个类！如果要把一样东西变成一个 Bean，我们完全不必特意去写任何额外的代码，或者使用某些特殊的语言扩展。事实上，唯一要做的事情就是稍微修改一下方法的命名方式。现在，我们完全依靠方法名来告诉应用程序构建工具：这到底是一个属性，一个事件，还是一个标准方法！

在 Java 的用户文档中，这种命名规范讲得有点儿不清楚，居然统一叫作“设计范式”（Design Pattern）。这显然是非常不幸的，因为即便没有这种命名上的混淆，真正的“设计范式”（参见《Thinking in Patterns with Java》一书，请到 www.BruceEckel.com 下载）本身已够让人头疼的了！注意，这里讲的并不是真正的设计范式，充其量不过是一种命名规范而已，而且它非常简单：

(1) 针对名叫 xxx 的一种属性，通常需要创建两个方法：getXxx()和 setXxx()。注意“get”或“set”之后的第一个字母会自动小写，从而得到属性名。由“get”方法产生的类型与“set”方法的参数（参数）类型是一样的。属性的名字和“get”及“set”的类型并不存在任何必然的关系。

(2) 对于 boolean（布尔）属性来说，除了可象上面那样使用“get”和“set”方法之外，还可用“is”来代替“get”。

(3) Bean 的标准方法不以上述命名规则为准，但它们是“公共的”（public）。

(4) 对事件来说，我们要使用 Swing 的“监听器”方法。它和我们以前见过的完全一样：用 addFooBarListener(FooBarListener) 和 removeFooBarListener(FooBarListener) 对一个 FooBarEvent 事件进行控制（增加和删除）。大多数时候，内建的事件和监听器已完全能满足我们的需要。但是，你仍然完全可以自行创建新的事件和监听器接口。

对比新老两种代码时，上面的第(1)点或许能帮你解答心中的一个疑惑：为什么许多方法的名字都发生子改动呢？而且改动极微，好象根本便“没什么意义”！不过，大家现在知道了，大多数这样的改动都是为了与“get”和“set”命名规则相符，以便那个特定的组件变成一个真正的 Bean！

我们可利用上面那些规则来创建一个简单的 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，我们注意到一个 Bean 不过就是一个类。通常，我们的所有字段都要设为“私有的”（private），而且只能通过方法进行访问。根据命名规范，这里用到的“属性”包括 jumps、color、spots 和 jumper（注意在属性名中，首字母大小写形式的改变）。尽管内部标识符的名字和前三种情况下的属性名一样，但在第四种情况下（jumper），大家可看到属性名并没有强迫我们用任何特定内部变量的标识符来表达（事实上，你可为那个属性使用任何内部变量）。

这个 Bean 能控制的“事件”包括 ActionEvent 和 KeyEvent——具体取决于对应监听器的“add”和“remove”方法命名。最后，我们看到标准方法 croak()依然是整个 Bean 的一部分。原因很简单，因为它是一个“公共的”方法——而不是因为它符合任何一种命名机制。

13.11.2　用 Introspector 提取 Beanifo

对 Bean 机制来说，它最吸引人的地方在于你将一个“Bean”从工具盒拖到窗体时，背后都会发生哪些事情！应用程序构建工具必须能够创建 Bean（只要有默认构造函数便行）。随后，在不必访问 Bean 的源代码的前提下，提取出必要的信息，来创建属性表和事件控制器。

为达此目的，所采取方案的一部分其实已在第 12 章要结束的时候显现出来了，那就是“反射”（Reflection）。利用 Java 的“反射”机制，一个匿名类的所有方法都可以被调查出来。为了在不必使用其他关键字的前提下（它们在其他可视编程语言里都是必须使用的），从容地解决 Bean 的问题，这一机制无疑是非常完美的！事实上，之所以要在 Java 里加入“反射”，其中很重要的一个原因便是为了提供对 Bean 的支持（当然，“反射”还支持对象序列化以及远程方法调用）。

那么，这是否便意味着，应用程序构建工具的设计者必须对每一个 Bean 进行反射，并在它的方法中检索，从而找到那个 Bean 的各个属性及事件呢？这完全是有可能的，但 Java 的设计者希望提供的是一个标准工具，除了能让 Bean 更易使用之外，还希望建立起一种标准方式，帮你进行更复杂的 Bean 的创建！这个工具便是这里要讲述的“Introspector”类。在这个类中，最重要的方法则是 static getBeanInfo()。首先，我们需要向该方法传递一个 Class 引用。接着，该方法会对你指定的类进行全面查询，最后返回一个 BeanInfo 对象。稍微处理一下这个对象，便可找到自己需要的属性、方法以及事件等等。

其实，我们一般也用不着关心这方面的问题。由于大多数时候都是从各个开发公司那里获取现成的 Bean，所以完全不需要知道背后发生的一切事情——只要能用它就成！那么，怎么用呢？你只需要将 Bean 拖放到一个窗体上，然后配置它们的属性，并为自己感兴趣的事件编写相应的控制模块（事件控制器）就 OK 了。不过，无论从兴趣还是学习的角度出发，我在这里仍然觉得有必要展示一个实际的例子。在这个例子中，我打算用 Introspector 来显示与一个 Bean 有关的信息：

```
//: c13:BeanDumper.java
// Introspecting a Bean.
// <applet code=BeanDumper width=600 height=500>
// </applet>
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 JApplet {
  JTextField query =
    new JTextField(20);
  JTextArea results = new JTextArea();
  public void prt(String s) {
    results.append(s + "\n");
  }
  public void dump(Class bean){
    results.setText("");
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(
        bean, java.lang.Object.class);
```

```
    } catch(IntrospectionException e) {
      prt("Couldn't introspect " +
        bean.getName());
      return;
    }
    PropertyDescriptor[] properties =
      bi.getPropertyDescriptors();
    for(int i = 0; i < properties.length; i++) {
      Class p = properties[i].getPropertyType();
      prt("Property type:\n  " + p.getName() +
        "Property name:\n  " +
        properties[i].getName());
      Method readMethod =
        properties[i].getReadMethod();
      if(readMethod != null)
        prt("Read method:\n  " + readMethod);
      Method writeMethod =
        properties[i].getWriteMethod();
      if(writeMethod != null)
        prt("Write method:\n  " + writeMethod);
      prt("====================");
    }
    prt("Public methods:");
    MethodDescriptor[] methods =
      bi.getMethodDescriptors();
    for(int i = 0; i < methods.length; i++)
      prt(methods[i].getMethod().toString());
    prt("======================");
    prt("Event support:");
    EventSetDescriptor[] events =
      bi.getEventSetDescriptors();
    for(int i = 0; i < events.length; i++) {
      prt("Listener type:\n  " +
        events[i].getListenerType().getName());
      Method[] lm =
        events[i].getListenerMethods();
      for(int j = 0; j < lm.length; j++)
        prt("Listener method:\n  " +
          lm[j].getName());
      MethodDescriptor[] lmd =
        events[i].getListenerMethodDescriptors();
      for(int j = 0; j < lmd.length; j++)
        prt("Method descriptor:\n  " +
          lmd[j].getMethod());
```

```
      Method addListener =
        events[i].getAddListenerMethod();
      prt("Add Listener Method:\n  " +
         addListener);
      Method removeListener =
        events[i].getRemoveListenerMethod();
      prt("Remove Listener Method:\n  " +
        removeListener);
      prt("====================");
    }
  }
  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 void init() {
    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);
  }
} ///:~
```

其中，BeanDumper.dump()几乎包揽了所有实质性的工作。首先，它试着创建一个 BeanInfo 对象。若成功，则调用 BeanInfo 的方法，产生与属性、方法以及事件有关的信息。在 Introspector.getBeanInfo()中，大家还可看到另一个参数。它的作用是告诉 Introspector 该在分级继承结构的什么位置停下来。在这个程序中，它会在从 Object 里解析出所有方法之前停下来——因为我们对那些方法已经不感兴趣了。

针对属性，getPropertyDescriptors()会返回一个由一系列 PropertyDescriptor 构成的数组。针对每一个 PropertyDescriptor，我们都可调用 getPropertyType()，找到通过属性方法传入和传出的对象类。随后，针对每一种属性，我们用 getName()获得它的“假名”（从方法名中提取而来）；用于“读”的方法是 getReadMethod()；用于“写”的方法则是 getWriteMethod()。最后这两个方法会返回一个 Method 对象，我们实际上可利用这个对象，调用对象上对应的方法（这正是“反射”机制在起作用）。

针对那些“公共”（public）方法——包括属性方法——getMethodDescriptors()会返回由 MethodDescriptor 构成的一个数组。针对每个 MethodDescriptor，我们都可获得相应的 Method 对象，并把它的名字打印出来。

针对事件，getEventSetDescriptors()可返回由 EventSetDescriptor 构成的一个数组。然后可对每个 EventSetDescriptor 进行查询，找出监听器的类以及那个类的所有方法。另外，也可在其中对监听器方法进行增删操作。BeanDumper 程序最后打印出所有这些信息。

启动后，程序会强制调查 frogbean.Frog。输出如下（剔除了一些无关紧要的细节）：

```
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)
====================
```

根据这些输出，我们可知道当 Introspector 根据你的 Bean 产生一个 BeanInfo 时，它都“看”到了哪些东西。从中可以看出，属性类型和它的名字是完全无关的。也请注意属性名会自动小写（唯一例外的情况就是属性名在一行内用多个大写字母起头）。另外，请记住这里看到的方法名（比如读和写方法）实际是从一个 Method 对象产生的，我们可利用那个对象，调用对象上对应的方法。

public 方法列表包含了未与属性或事件关联在一起的那些方法，比如 croak()等等。我们可针对一个 Bean，在自己的程序中调用所有这些方法。而且有的应用程序构建工具会在你进行方法调用的时候帮你列出所有这些方法，从而简化你的工作。

最后，我们看到事件被进行了非常彻底的分析，列出了它的监听器、它的方法以及用于增加或删除监听器的方法等等。从根本上说，BeanInfo 对象是一种“一旦拥有，别无所求”的东西。利用它，一个 Bean 的“祖宗十八代”都可以被调查得清清楚楚——即便你手上除了这个对象之外根本没有别的任何资料（同样地，这完全要归功于“反射”）。

13.11.3 一个更复杂的 Bean

接下来的例子要复杂一些——但显得更加有趣。它主要是一个 JPanel；当鼠标移动的时候，便围绕鼠标指针画一个小圆。若按下鼠标，一个单词“Bang!”（嘭）便重重地显示在屏幕中部，同时触发一个行动监听器。

可以自行改变的属性包括圆的大小，以及那个单词的颜色、大小以及文字内容。BangBean 还有它自己的 addActionListener()以及 removeActionListener()方法，以便联系上我们自己的监听器，令它在用户点击 BangBean 的时候触发：

```
//: 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 {
  protected int xm, ym;
  protected int cSize = 20; // Circle size
  protected String text = "Bang!";
  protected int fontSize = 48;
  protected Color tColor = Color.red;
  protected 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);
  }
} ///:~
```

我们注意到的第一件事情是，BangBean 实现了 Serializable（可序列化）接口。也就是说，应用程序构建工具可在程序设计人员调整好属性值后，利用“序列化”的手段，将 BangBean 的所有信息都“暂存”（pickle）下来。以后一旦将这个 Bean 作为正在运行的应用程序的一部分加以创建，便可立即恢复成与原先“暂存”下来时一模一样的属性。

我们看到，所有字段都被设为“私有的”（private）。操作一个 Bean 时，这应该是一个良好的习惯——以保证只有通过方法才能访问（通常利用“属性”机制）。

留意一下 addActionListener()的签名，大家会发现它能产生一个 TooManyListeners-

Exception（监听器过多）违例。这就表明它肯定是“单强制转型”（Unicast）的。也就是说，当事件发生时，只能向一个监听器发出通知。许多人仍然喜欢采用“多强制转型”（Multicast）事件，使多个监听器都能同时收到发生了一个事件的通知。不过，这种做法极易带来问题。具体是什么问题，要了解了下一章的知识后才会真正明白。所以，我们在那个时候还会重提这个话头（参见下一章的“再论 JavaBeans”小节）。

点击鼠标后，就将文本放到 BangBean 的中央；而且假如 actionListener 字段不为“空”（null），就会调用它的 actionperformed()，从而创建一个新的 ActionEvent 对象。任何时候只要鼠标移动了位置，它的新坐标就会被捕捉下来，并对整个“画布”进行重画（当然，事先得删除上面原有的任何文本）。

下面是一个 BangBeanTest 类，用它可对这个 Bean 进行测试（小程序或应用程序的形式均可）。

```
//: c13:BangBeanTest.java
// <applet code=BangBeanTest
// width=400 height=500></applet>
import bangbean.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BangBeanTest extends JApplet {
  JTextField txt = new JTextField(20);
  // During testing, report actions:
  class BBL implements ActionListener {
    int count = 0;
    public void actionPerformed(ActionEvent e){
      txt.setText("BangBean action "+ count++);
    }
  }
  public void init() {
    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);
```

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

一个 Bean 还处在开发环境中时，我们不会使用这个类，但它有助于为你的每个 Bean 都执行一次快速测试。BangBeanTest 会将一个 BangBean 放在小程序里，为 BangBean 连接一个简单的 ActionListener，然后在发生了那个 ActionEvent 事件后，将一个事件计数值打印到 JTestField 文本字段里。当然，等你真正要使用 Bean 时，应用程序构建工具通常都可以帮你自动生成大多数的相关代码。

假如通过 BeanDumper 运行 BangBean，或者把 BangBean 放到一个支持 Bean 的开发环境中，大家便会注意到比上述代码多得多的属性及行动被揭示出来。这是由于 BangBean 是从 JPanel 继承来的，而 JPanel 本身也是一个 Bean，所以……

13.11.4　Bean 的打包

将一个 Bean 拿到支持 Bean 功能的可视程序构造函数中使用之前，必须将其放到一个标准的 Bean 容器里——这个容器就是一个 JAR 文件。在 JAR 文件中，包含了所有的 Bean 类，同时还有一个“详情表”（Manifest）文件，文件中要指出：“这是一个 Bean。”详情表文件实际是一个纯文本文件，只是采用了特殊的形式。对 BangBean 来说，它的详情表文件看起来就象下面这样（请自己剔去第一行和最后一行）：

```
//:! :BangBean.mf
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
///:~
```

在剔去了首尾行的详情表主体中，第一行指出你所采用的详情表方案的版本。在得到 Sun 公司的进一步通知之前，请设置为 1.0；第二行（不计那个空行）为 BangBean.class 文件命名；第三行的意思则是：“这是一个 Bean。”假如没有这第三行，程序构建工具便没法子将该类看作是一个“Bean”。

这里唯一要注意的地方就是务必保证“Name:”字段里设置了正确的路径。如果回过头去研究一下 BangBean.java，就会发现它在一个名为 bangbean 的封装里（package bangbean）——所以必然在从类路径分支出去的、一个名叫“bangbean”的子目录下。在详情表文件中，你指定的名字必须包括这个封装的信息。除此以外，还必须针详情表文件放在“封装”所在路径的根之上。换句话说，在当前的情况下，必须将其放在“bangbean”子目录之上的一个目录内。这样一来，我们必须在与详情表文件同样的目录下，调用 jar 命令，如下所示：

```
jar cfm BangBean.jar BangBean.mf bangbean
```

它假定你希望将结果生成的 JAR 文件命名为 BangBean.jar，而且已将详情表放到一个名为 BangBean.mf 的文件里。

这个时候，大家也许会感到有一点疑惑：“在我编译好 BangBean.java 之后，生成的其他那些类又怎么样了呢？”事实上，它们最终都进入了 bangbean 子目录内。在上述 jar 命令行中，大家可以看到最后一个参数指定的就是 bangbean 子目录。为 jar 指定了一个子目录的名字之后，它就会将整个子目录都“打包”到 jar 文件里（就目前来说，其中包含了最初的 BangBean.java 源代码文件——但在你制作自己的 Bean 时，通常并不希望源码也包括进去，

这里只是一个例子）。除此以外，假如你对刚才创建的 JAR 文件进行解包处理，便会发现自己的详情表文件并不在内，反而是由 jar 创建了它自己的详情表文件（当然部分以你的为基础），名为 MANIFEST.MF，并把它放在了子目录 META-INF 下（这个名字的全称是“meta-information”，即“元信息”）。打开这个新的详情表文件，你还会发现 jar 已为每个文件都添加了数字化签名，采用下列形式：

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=
MD5-Digest: O4NcS1hE3Smnzlp2hj6qeg==
```

通常，我们并不需要关心上面说的那些。假如你进行了改动，只需修改最开始的那个详情表文件，然后再执行一遍 jar 命令，为你的 Bean 创建一个新的 JAR 文件就可以了。另外，要想在 JAR 文件里加入其他 Bean，只需把它们的信息加到自己的详情表便 OK 了。

这里要注意的一件事情是，我们有时希望将每个 Bean 都放到它们自己的子目录下——因为在创建一个 JAR 文件的时候，需要向 jar 工具传递一个子目录的名字，而它会将那个子目录下的所有东西都放到一个 JAR 文件里。大家可以看到，无论 Frog 还是 BangBean 都在它们自己的子目录下。

一旦你的 Bean 正确地进入了一个 JAR 文件，便可把它带到支持 Bean 功能的程序构建环境使用。但对于不同的工具，具体做这件事情的方式却是不同的。不过，Sun 公司提供了一个免费的 JavaBeans 测试床——在他们的“Beans DevelopmentKit”（BDK）中，有一个名为“beanbox”的工具，便是专门用来对你的 Bean 进行测试的（BDK 请从 java.sun.com/beans 下载）。为了把你的 Bean 放到 beanbox 中，只需在启动 beanbox 工具之前，将 JAR 文件复制到 BDK 的“jars”子目录下就可以了。

13.11.5 更复杂的 Bean 支持

通过前面的学习，我们知道制作一个 Bean 是多么的容易！而且更令人高兴的是，除了这里演示的之外，你还可以用它做更多的事情！JavaBeans 体系不仅提供了一个非常容易上手的“入口”，而且只要你需要，还可以任意进行扩展，以适应更复杂的应用场合的需要。不过，对那些问题的详细讨论已超出了本书的范围，这里只能说个大概（详情可参见 java.sun.com/beans）。

首先就拿属性来“开刀”吧，你可以把它弄得更加复杂！上面的例子只演示了单一的属性，但我们实际上也可用一个数组来表示多个属性。这叫作“索引化属性”（Indexed Property）。只需提供恰当的方法（同样要遵守方法的命名规范），Introspector 便能识别出一个索引化属性。进而，你的应用程序构建工具也能正常地识别出它。

属性可以“绑定”（Bound）。换言之，它们会通过一个 PropertyChangeEvent 来通知其他对象。随后，其他对象可根据对 Bean 的改动，完成对自己的修改。

属性可以“受限”（Constrained）。也就是说，其他对象可以否决（Veto）那个属性的一处改变——假如它无法接受那处改变的话。在这种情况下，改变首先通过一个 PropertyChangeEvent 来通知给其他对象。随后，假如觉得不合适的话，那个对象可产生一个 PropertyVetoException，防止那个改变的发生，并恢复成原先的值。

你甚至可以改变一个 Bean 在设计时的表示方式：

(1) 可为特定的 Bean 提供一个自定义属性表。原来那个属性表仍然适用于其他所有 Bean，但一旦选中了那个特定的 Bean，就会自动调用你专门提供的那个属性表。

(2) 可为特定属性创建一个自定义的编辑器。同样地，平时仍然使用原来那个属性表，

但在试图编辑那个特定的属性时，便会自动调你创建的那个编辑器。

(3) 可为自己的 Bean 提供一个自定义的 BeanInfo 类，令其产生与通过 Introspector 默认产生的不同的信息。

(4) 还有可能在所有 FeatureDescriptor 中打开或关闭“专家”模式，以区分基本特性和那些较复杂的高级特性。

13.11.6　Bean 更多的话题

更多的话题显然不能再在这里仔细说了。不过最起码要提醒你一点，在你创建的一个 Bean 的时候，无论如何都该假定它以后会到一个多线程的环境中运行。也就是说，在这之前，你还得先掌握好多线程的知识，这将是第 14 章的主题。在那一章中，其中甚至有一个“再论 JavaBeans”小节，专门探讨这个问题及其对策。

JavaBeans 的许多参考书对你可能都有用处；比如由 Elliotte Rusty Harold 编著的、IDG 于 1998 年出版的《JavaBeans》一书。

13.12　总　　结

Java 1.0 升级到 Java 2 后，对 Java 的所有库而言，GUI 库是其中改动幅度最大的，也是最令人激动的！Java 1.0 推出的“AWT”受到了当时众多的非议，人们认为它是“有史以来最糟的设计之一”。尽管也许能用它创建“便于移植”的程序，但最终生成的 GUI 却落得个“在所有平台上都同样平庸”的下场。另外，同由其他平台提供的专用程序开发工具相比，AWT 还存在着这样那样的限制，用起来十分不“爽”！

Java 1.1 问世时，同时也引入了新的事件模型和 JavaBeans 的概念。从此，一个可供我们大展拳脚的舞台搭建起来了——我们“有希望”用它创建出“真正”的 GUI 组件，它们应该能在可视程序构建工具中方便地“拖放”。除此以外，事件模型和 Bean 也经过了精心的设计，特别强调了如何最方便地进行编程与维护（这在 1.0 的 AWT 里几乎是做不到的）。但到那时为止，我们看到的还仅仅是一个架子，等到 JFC/Swing 类问世之后，全部工作才算真正告了一个段落！利用 Swing 组件，跨平台的 GUI 编程终于“有可能”实现！

万事俱备，只欠东风。事实上，我们现在唯一缺少的就是应用程序构建工具——而这才是会真正发生编程领域“革命”的地方！微软的 Visual Basic 和 Visual C++需要用到微软的应用程序构建工具；Borland 的 Delphi 和 C++ Builder 也与此类似。假如你希望应用程序构建工具变得更好，那么只好“祈求上帝保佑”，希望那些厂家带给你需要的东西。不过，Java 是一种开放环境，所以它不仅允许出现多种应用程序构建工具的竞争，甚至还鼓励出现这样的情况。当然了，你真正需要的工具应该是能够支持 JavaBeans 的。换句话说，这是一个对谁都公平的竞争领域：假如有一个更好的应用程序构建工具问世，那么为了提高效率和不落后于别人，你通常都会毫不犹豫地放弃手上正在用的那个。对 GUI 应用程序构建工具来说，这样的竞争环境以前可从来没出现过！象基于开放标准的 Linux 那样，不管最终由此而产生了什么样的市场空间，都只能是对用户、对我们这些程序员有利！

本章的宗旨只是为了使大家对 Swing 的巨大威力有一个初步的认识。入了门之后，再去学习一些更具体的东西就会感动相对简单一些。不过，大家迄今为止学到的东西已足够应付日常应用——最起码为你最终设计出一个真正优秀的 UI 打下了良好的基础。不过，这里讲到的仍然只是 Swing 这座“大冰山”的一角。要知道，Swing 是被当作一个全功能的 UI 设计工具“套餐”来制作的。利用它，你可做到自己能够想象的几乎任何事情！

假如本章的内容还不能满足你的需要，完全可以自行查阅 Sun 公司提供的联机文档，或者在整个 Web 上搜索自己需要的内容。如果这些都还不够，那么去买一本专门讲 Swing

的参考书吧！“每一本书都总会有用的。”你说呢？这儿向你推荐一本入门读物，名叫《The JFC Swing Tutorial》，Walrath 和 Campione 合著，AddisonWesley 出版社于 1999 年出版。

13.13 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 根据本章的提示，用 Console 类创建一个小程序／应用程序（兼具两种身份）。在其中包括一个文本字段及三个按钮。按下每个按钮时，都在文本字段中显示不同的文字。

(2) 为练习(1)的小程序添加一个复选框，捕捉它的事件，并在文本字段里插入不同的文字。

(3) 用 Console 类创建一个小程序／应用程序。在由 java.sun.com 提供的 HTML 用户文档中，找到 JPasswordField（密码输入字段），把它添加到程序中。假如用户键入正确的密码，就用 Joptionpane 向用户显示一条表示成功的消息。

(4) 用 Console 类创建一个小程序／应用程序，在其中添加拥有一个 addActionListener() 方法的所有组件（请查阅 java.sun.com 的 HTML 文档。提示：请使用索引）。捕捉它们各自的事件，并在一个文本字段里，为每个事件都显示一条消息。

(5) 用 Console 类创建一个小程序／应用程序，在其中添加一个 JButton 和一个 JTextField。编写和连接恰当的监听器，使那个按钮一旦获得“焦点”，键入的字符便会在 JTextField 上显示出来。

(6) 用 Console 创建一个小程序／应用程序。在其窗帧中添加本章介绍的所有组件，包括菜单和一个对话框。

(7) 修改 TextFields.java 程序，使 t2 中的字符保留它们原来键入时的大小写形式，而不是被迫全部转换成大写。

(8) 上网寻找并下载一个或几个免费的 GUI 构建开发环境；或者购买一个商业产品。探索一下要将 BangBean 添加到这种环境中，并实际地使用它，那么都需要哪些前提，需要做哪些事情。

(9) 根据本章的提示，在详情表文件里添加 Frog.class。运行 jar 工具，创建一个 JAR 文件，在其中同时包含 Frog 和 BangBean。接下来，要么从 Sun 公司下载并安装 BDK，要么用你自己的、支持 Bean 的程序构建工具，将 JAR 文件添加到其中，以便对两个 Bean 进行测试。

(10) 自行创建一个名为 Valve 的 JavaBean，其中应该包括两个属性：一个是 boolean 属性，名为“on”；另一个是 int 属性，名为“level”。建立一个详情表文件，用 jar 工具将你的 Bean“打包”，然后将其载入 beanbox 工具或者一个支持 Bean 的程序构建工具中，以便对其进行测试。

(11) 修改 MessageBoxes.java，使其为每个按钮都准备一个 ActionListener（而不是与按钮上的文本匹配）。

(12) 在 TrackEvent.java 中，添加新的事件控制代码，对一种新类型的事件进行监视。在这种情况下，你得自己查询出想要监视的事件的类型。

(13) 从 JButton 继承一个新类型的按钮。每次按下该按钮，它都应将自己的颜色变成一个随机挑选的值。请参考第 14 章的 ColorBoxes.java 程序，那是如何产生随机颜色值的一个经典例子。

(14) 修改 TextPane.java，将 JTextPane 换成一个 JTextArea。

(15) 修改 Menus.java，将菜单上的复选框换成单选钮。

(16) 对 List.java 进行简化：将数组传递给构造函数，免去在列表里动态添加元素的步骤。

(17) 修改 SineWave.java，将 SineDraw 变成一个 JavaBean——你只需要添加新的“getter”和“setter”方法就可以了。

(18) 记得一种叫作“素描盒”的玩具吗？上面有两个手柄，一个画笔的垂直移动，另一个控制水平移动（事实上，绘图仪和打印机也是基于这一原理设计的）。请自己设计一个，刚开始可拿 SineWave.java 作为参考。但是，不要用手柄，换成两个滑杆。另外，请添加一个按钮，用于一次性删除你的所有“涂鸦”。

(19) 创建一个“渐近进度指示条”，在其抵达终点之前，它应该“走”得越来越慢。添加随机性的“不定”行为，使其产生定时的速度加快，但马上又恢复成普通速度。

(20) 修改 Progress.java，使其不再共享模型，而是用一个监听器来连接滑杆和进度条。

(21) 根据“将小程序打包到 JAR 文件里”小节的提示，将 TicTacToe.java 放到一个 JAR 文件里。然后，创建一个 HTML 页，采用 applet 标记那个麻烦、复杂的版本，对其进行修改，加上 archive 标记，以便能使用新建的 JAR 文件。（提示：可在本书源码包里为 TicTacToe.java 配套提供的那个 HTML 页的基础上修改。）

(22) 用 Console 创建一个小程序／应用程序。在其中放置三个滑杆，分别用于控制 java.awt.Color 里的红、绿、蓝（RGB）三种颜色值。在窗体剩下的部分，设置一个 JPanel，用它显示当前由三个滑杆决定的颜色。另外，加上三个不可编辑的文本字段，在其中分别显示当前指定的 RGB 值。

(23) 在 javax.swing 的 HTML 用户文档中，找到 JColorChooser（选色器）。写一个程序，在其中安放一个按钮，按下这个按钮后，让选色器以一个对话框的形式显示出来。

(24) 几乎所有 Swing 组件都是从 Component 派生的，后者提供了一个 setCursor()方法，用于设置光标／鼠标指针。请在 Java HTML 文档查阅它的用法。然后，创建一个小程序，将当前光标改变成 Cursor 类自带的某种光标样式。

(25) 以 ShowAddListeners.java 为基础，创建一个新程序，在其中运用由第 12 章介绍的 ShowMethodsClean.java 所提供的全部功能。

第 14 章 多线程

利用对象，可将一个程序分割成相互独立的区域。我们通常也需要将一个程序转换成多个独立运行的子任务。

象这样的每个子任务都叫作一个“线程”（Thread）。编写程序时，可将每个线程都想象成独立运行，而且都有自己的专用 CPU。一些基础机制实际会为我们自动分割 CPU 的时间。我们通常不必关心这些细节问题，所以多线程的代码编写是相当简便的。

这时理解一些定义对以后的学习很有帮助。“进程”是指一种“自己管自己”的运行程序，它有自己的地址空间。“多任务”操作系统能同时运行多个进程（程序）——但实际是由于 CPU 分时机制的作用，使每个进程都能循环获得自己的 CPU 时间片。但由于轮换速度非常快，使得所有程序好象是在“同时”运行一样。“线程”是进程内部单一的一个顺序控制流。因此，一个进程可能容纳了多个同时执行的线程。

多线程的应用范围很广。但在一般情况下，程序的一些部分同特定的事件或资源联系在一起，同时又不想为它而暂停程序其他部分的执行。这样一来，就可考虑创建一个线程，令其与那个事件或资源关联到一起，并让它独立于主程序运行。一个很好的例子便是“Quit”或“退出”按钮——我们并不希望在程序的每一部分代码中都轮询这个按钮，同时又希望该按钮能及时地作出响应（使程序看起来似乎经常都在轮询它）。事实上，多线程最主要的一个用途就是构建“反应灵敏”的用户界面——在你使用界面中的一个控件时，马上就能得到响应。

14.1 反应灵敏的用户界面

作为我们的起点，请思考一个需要执行某些 CPU 密集型计算的程序。由于 CPU“全心全意”为那些计算服务，所以对用户的输入十分迟钝，几乎没有什么反应。在这里，我们用一个兼具小程序和应用程序两种身份的一个例子来简单显示出一个计数器的结果：

```
//: c14:Counter1.java
// A non-responsive user interface.
// <applet code=Counter1 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Counter1 extends JApplet {
  private int count = 0;
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");
```

```
  private JTextField t = new JTextField(10);
  private boolean runFlag = true;
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public void go() {
    while (true) {
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      if (runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      go();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public static void main(String[] args) {
    Console.run(new Counter1(), 300, 100);
  }
} ///:~
```

在这个程序中，Swing 和小程序代码都应该是大家非常熟悉的，第 13 章对此已有很详细的交待。go()方法正是程序全心全意服务的对象：它将当前的 count（计数）值置入 JTextField t，然后使 count 的值递增。

对 go()内的无限循环来说，它的一部分工作是调用 sleep()。sleep()必须同一个 Thread（线程）对象关联到一起，而且似乎每个应用程序都有部分线程同它关联（事实上，Java 本身就是建立在线程基础上的，肯定有一些线程会伴随我们写的应用一起运行）。所以无论我们是否明确使用了线程，都可利用 Thread 和静态的 sleep()方法来产生当前由程序使用的线程。

注意 sleep()可能“掷”出一个 InterruptException（中断违例）——尽管产生这样的违例被认为是中止线程的一种“恶意”手段，而且应该尽可能地杜绝这一做法。再次提醒大家，违例是为异常情况而产生的，而不是为了本来正常的控制流程。在这里包含了对一个“睡眠”线程的中断，以支持未来的一种语言特性。

一旦按下 start 按钮，就会调用 go()。研究一下 go()，你可能会很自然地（就象我一样）认为它该支持多线程，因为它会进入“睡眠”状态。也就是说，尽管方法本身“睡着”了，CPU 仍然应该忙于监视其他按钮“按下”事件。但有一个问题，那就是 go()是永远都不会返回的，因为它在一个无限循环里。这意味着 actionPerformed()根本不会返回。由于在第一个按键以后便陷入 actionPerformed()中，所以程序不能再对其他任何事件进行控制。如果想出来，必须以某种方式“杀死”进程——最简单的做法就是在控制台窗口按 Ctrl＋C 键（假如你从控制台运行），或者关闭浏览器窗口（假如你通过浏览器运行）。

这里最基本的问题是 go()需要继续执行自己的操作，而与此同时，它也需要返回，使 actionPerformed()能够完成，而且用户界面也能继续响应用户的操作。但对象 go()这样的传统方法来说，它却不能在继续的同时将控制权返回给程序的其他部分。这听起来似乎是一件不可能做到的事情，就象 CPU 必须同时位于两个地方一样，但线程可以解决一切。

“线程模型”（以及它在 Java 中的编程支持）是一种程序编写规范，可在单独一个程序里实现几个操作的同时进行。根据这一机制，CPU 可为每个线程都分配自己的一部分时间。每个线程都“感觉”自己好象拥有整个 CPU，但 CPU 的计算时间实际却是在所有线程间分摊的。唯一的例外就是你用来运行程序的机器本来就安装了多个 CPU。但对于线程来说，最令人欣慰的事实之一就是我们不必关心这方面的问题。也就是说，我们的代码不必知道自己到底是单 CPU 系统中运行，还是在多 CPU 系统中运行。因此，利用线程，我们可以方便地创建出能自由扩展的程序，这种扩展是“透明”的。

线程机制多少降低了一些计算效率，但无论程序的设计，资源的均衡，还是用户操作的方便性，都从中获得了巨大的利益。综合考虑，这一机制还是非常有价值的。当然，如果本来就安装了多个 CPU，那么操作系统能够自行决定为不同的 CPU 分配哪些线程，程序的总体运行速度也会变得更快（所有这些都要求操作系统以及应用程序的支持）。多线程和多任务是充分发挥多处理机系统能力的一种最有效的方式。

14.1.1 从 Thread 继承

为创建一个线程，最简单的做法就是从 Thread 类继承。这个类包含了创建和运行线程所需的一切东西。Thread 最重要的方法是 run()。但为了使用 run()，必须对其进行覆盖，使其能充分按自己的吩咐行事。因此，run()属于那些会与程序中的其他线程“并发”或“同时”执行的代码。

下面这个例子可创建任意数量的线程，并通过为每个线程分配一个独一无二的编号（由一个静态变量产生），从而对不同的线程进行跟踪。Thread 的 run()方法在这里得到了覆盖，每通过一次循环，计数就减 1——计数为 0 时则完成循环（此时一旦 run()方法返回，线程便中止运行）。

```
//: c14:SimpleThread.java
// Very simple Threading example.

public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
```

```
  private int threadNumber = ++threadCount;
  public SimpleThread() {
    System.out.println("Making " + threadNumber);
  }
  public void run() {
    while(true) {
      System.out.println("Thread " +
        threadNumber + "(" + countDown + ")");
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread().start();
    System.out.println("All Threads Started");
  }
} ///:~
```

run()方法几乎肯定含有某种形式的循环——它们会一直持续到线程不再需要为止。因此，我们必须规定特定的条件，以便中断并退出这个循环（或者在上述的例子中，简单地从run()返回即可）。run()通常采用一种无限循环的形式。也就是说，通过阻止外部可能发出的对 run()调用，它会永远运行下去（直到程序完成）。

在 main()中，大家可看到我们创建并运行了大量线程。Thread 包含了一个特殊的方法，叫作 start()，它的作用是对线程进行特殊的初始化，然后调用 run()。所以完整的步骤包括：调用构造函数来构建对象，然后用 start()配置线程，再调用 run()。假如不调用 start()——假如合适的话，可在构造函数中那样做——线程便永远不会启动。

下面是该程序某一次运行的输出（注意每次运行都会不同）：

```
Making 1
Making 2
Making 3
Making 4
Making 5
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 1(1)
All Threads Started
```

```
Thread 3(5)
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
```

我们注意到，这个例子中没有任何地方调用了 sleep()，然而输出结果指出每个线程都获得了属于自己的那一部分 CPU 执行时间。从中可以看出，尽管 sleep()依赖一个线程的存在来执行，但却与允许或禁止线程无关。它只不过是另一个不同的方法而已。

亦可看出线程并不是按它们创建时的顺序运行的。事实上，CPU 处理一个现有线程集的顺序是不确定的——除非我们亲自介入，并用 Thread 的 setPriority()方法调整它们的优先级。

main()创建 Thread 对象时，它并未捕获任何一个对象的引用。普通对象对于垃圾收集来说是一种“公平竞赛”，但线程却并非如此。每个线程都会“注册”自己，所以某处实际存在着对它的一个引用。这样一来，垃圾收集器便只好对它“瞠目以对”了。

14.1.2 如何用多线程实现反应灵敏的界面

现在，我们也许能用一个线程解决在 Counter1.java 中出现的问题。采用的一个技巧便是在一个线程的 run()方法中放置“子任务”——亦即位于 go()内的循环。一旦用户按下 Start 按钮，线程就会启动，但马上结束线程的创建。这样一来，尽管线程仍在运行，但程序的主要工作却能得以继续（等候并响应用户界面的事件）。下面是具体的代码：

```
//: c14:Counter2.java
// A responsive user interface with threads.
// <applet code=Counter2 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter2 extends JApplet {
  private class SeparateSubTask extends Thread {
    private int count = 0;
```

```
  private boolean runFlag = true;
  SeparateSubTask() { start(); }
  void invertFlag() { runFlag = !runFlag; }
  public void run() {
    while (true) {
     try {
      sleep(100);
    } catch(InterruptedException e) {
      System.err.println("Interrupted");
    }
     if(runFlag)
       t.setText(Integer.toString(count++));
    }
  }
}
private SeparateSubTask sp = null;
private JTextField t = new JTextField(10);
private JButton
  start = new JButton("Start"),
  onOff = new JButton("Toggle");
class StartL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if(sp == null)
      sp = new SeparateSubTask();
  }
}
class OnOffL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if(sp != null)
      sp.invertFlag();
  }
}
public void init() {
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  cp.add(t);
  start.addActionListener(new StartL());
  cp.add(start);
  onOff.addActionListener(new OnOffL());
  cp.add(onOff);
}
public static void main(String[] args) {
  Console.run(new Counter2 (), 300, 100);
}
```

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

现在，Counter2 变成了一个相当直观的程序，它唯一的任务就是设置并维持用户界面。但假若用户现在按下 Start 按钮，事件控制代码却不会调用一个方法。相反，它会创建 SeparateSubTask 的一个线程，然后继续 Counter2 事件循环。

SeparateSubTask 类是对 Thread 的一个简单扩展，它带有一个构造函数，可通过调用 start()来运行线程；另外还要调用一个 run()，其中实际包含了来自 Counter1.java 的“go()”代码。

由于 SeparateSubTask 是一个内部类，所以能够直接访问 Counter2 中的 JTextField t；在 run()中，大家可以看到这种情况的发生。外部类的 t 字段是“私有的”，因为 SeparateSubTask 可在不必任何特别授权的前提下，“从容”地访问它。而且，将字段“尽可能私有化”是我们的一项基本设计准则，这样可避免由于来自类外的某种力量，造成对它的不慎改动。

按下 onOff 按钮后，它会对 SeparateSubTask 对象内的 runFlag 标记进行切换。随后，那个线程（在它检查了标记之后）就会启动，并自己停下来。按下 onOff 按钮后，感觉程序是马上作出了响应。当然，这种响应实际并不是“马上”作出的，这和那些“由中断驱动”的系统有所区别。只有在线程拥有对 CPU 的控制，而且 CPU 注意到标记发生改变之后，计数器才会停下来。

注意 SeparateSubTask 这个内部类是“私有的”（private）。也就是说，它的字段和方法可被赋予默认访问权限（run()是一个例外，它必须是“公共的”，因为它在基类里便是“公共的”）。除 Counter2 之外，其他任何人都不能访问私有的内部类，而且两个类是紧密结合在一起的。这里要提醒大家：不管在什么时候，只要发现类和类之间紧密地结合在一起，就要考虑是不是使用内部类技术，来方便代码的编写以及以后对它的维护。

14.1.3 线程和主类的合并

在上面的例子中，我们看到线程类与程序的主类是严格区分开的。这样做非常合理，而且易于理解。然而，还有另一种方式也是经常要用到的。尽管它不大容易理解，但一般都要显得更简洁一些（这也解释了它为什么更流行）。这种“另类”的形式要求将主程序类变成一个线程，从而将主程序类和线程类合并到一起。由于对一个 GUI 程序来说，主程序类必须从 Frame 或 Applet 继承，所以必须用一个接口加入额外的功能。这个接口叫作 Runnable，其中包含了与 Thread 一致的基本方法。事实上，Thread 也实现了 Runnable，它只指出有一个 run()方法。

对合并后的程序／线程来说，它的用法并不是那么一目了然的。当我们启动程序时，会创建一个 Runnable（可运行的）对象，但不会自行启动线程。线程的启动必须明确进行。下面这个程序向我们演示了这一点，它再现了 Counter2 的功能：

```
//: c14:Counter3.java
// Using the Runnable interface to turn the
// main class into a thread.
// <applet code=Counter3 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
```

```
public class Counter3
    extends JApplet implements Runnable {
  private int count = 0;
  private boolean runFlag = true;
  private Thread selfThread = null;
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");
  private JTextField t = new JTextField(10);
  public void run() {
    while (true) {
      try {
        selfThread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      if(runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(selfThread == null) {
        selfThread = new Thread(Counter3.this);
        selfThread.start();
      }
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public static void main(String[] args) {
    Console.run(new Counter3(), 300, 100);
```

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

现在，run()位于类内，但它在 init()结束以后仍处在“睡眠”状态。若按下启动（start）按钮，便会用多少有点儿暧昧的一个表达式来创建线程（如果它尚不存在的话）：

```
new Thread(Counter3.this);
```

若某样东西有一个 Runnable 接口，实际只是意味着它有一个 run()方法，但不存在与之相关的任何特殊东西——它不具有任何天生的线程处理能力，这与那些从 Thread 继承的类是不同的。所以为了从一个 Runnable 对象产生线程，必须象上面那样单独创建一个线程，将 Runnable 对象传递给特殊的 Thread 构造函数。随后便可为那个线程调用 start()，如下所示：

```
selfThread.start();
```

它的作用是执行标准的初始化操作，再调用 run()。

Runnable 接口最大的一个优点是所有东西都从属于相同的类。若需访问什么东西，只需简单地访问它即可，不需要再经过一个单独的对象。然而，就象前例展示的那样，这种访问和使用一个内部类一样简单[66]。

14.1.4 制作多个线程

现在考虑一下创建多个不同的线程的问题。我们不可用前面的例子来做到这一点，所以必须倒退回去，利用从 Thread 继承的多个独立类来封装 run()。但这是一种更常规的方案，而且更易理解，所以尽管前例揭示的是我们经常都能看到的、流行的一种编码样式，但并不推荐你在大多数情况下都真的那样做，因为它显得稍微复杂一些，而且灵活性稍低。

下面这个例子用计数器和切换按钮再现了前面的编码样式。但这一次，一个特定计数器的所有信息（按钮和文本字段）都位于它自己的、从 Thread 继承的对象内。Ticker 中的所有字段都具有 private（私有）属性，这意味着 Ticker 的具体实现可根据实际情况任意修改，其中包括修改用于获取和显示信息的数据组件的数量及类型。创建好一个 Ticker 对象以后，构造函数便请求将它的可视组件添加到外层对象的内容区（Content Pane）内：

```
//: c14:Counter4.java
// By keeping your thread as a distinct class,
// you can have as many threads as you want.
// <applet code=Counter4 width=200 height=600>
// <param name=size value="12"></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter4 extends JApplet {
  private JButton start = new JButton("Start");
```

66 Runnable 自 Java 1.0 便有了，但内部类是从 Java 1.1 才开始采用的。这可以部分解释 Runnable 存在的原因。另外，传统的多线程体系将重点放在要运行的一个函数上，而不是一个对象上。就我自己来说，肯定更愿意从 Thread 继承；它更容易使人明白，也更加灵活。

```
  private boolean started = false;
  private Ticker[] s;
  private boolean isApplet = true;
  private int size = 12;
  class Ticker extends Thread {
    private JButton b = new JButton("Toggle");
    private JTextField t = new JTextField(10);
    private int count = 0;
    private boolean runFlag = true;
    public Ticker() {
      b.addActionListener(new ToggleL());
      JPanel p = new JPanel();
      p.add(t);
      p.add(b);
      // Calls JApplet.getContentPane().add():
      getContentPane().add(p);
    }
    class ToggleL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
      }
    }
    public void run() {
      while (true) {
        if (runFlag)
          t.setText(Integer.toString(count++));
        try {
          sleep(100);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for (int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
  }
  public void init() {
```

```
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Get parameter "size" from Web page:
    if (isApplet) {
      String sz = getParameter("size");
      if(sz != null)
        size = Integer.parseInt(sz);
    }
    s = new Ticker[size];
    for (int i = 0; i < s.length; i++)
      s[i] = new Ticker();
    start.addActionListener(new StartL());
    cp.add(start);
  }
  public static void main(String[] args) {
    Counter4 applet = new Counter4();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    if(args.length != 0)
      applet.size = Integer.parseInt(args[0]);
    Console.run(applet, 200, applet.size * 50);
  }
} ///:~
```

Ticker 不仅包括了自己的线程处理机制，也提供了控制与显示线程的工具。可按自己的意愿创建任意数量的线程，毋需明确地创建窗口化组件。

在 Counter4 中，有一个名为 s 的 Ticker 对象数组。为获得最大的灵活性，我们利用在 Web 页里采用的小程序（applet）参数来初始化数组大小。下面便是 size 参数的大致样子，它需要嵌入“applet”这个 HTML 标记中：

```
<param name=size value="20">
```

其中，param、name 和 value 都是 HTML 关键字。其中，指定了“name”（名字）之后，以后就可以在程序中方便地引用；“value”（值）可以是任何字串，并不一定非要是数字。

我们注意到数组 s 的长度是在 init()内决定的，它没有作为 s 的内嵌定义的一部分提供。换言之，不可将下述代码作为类定义的一部分使用（在任何方法的外部）：

```
int size = Integer.parseInt(getParameter("size"));
Ticker[] s = new Ticker[size];
```

可把它编译出来，但会在运行时间得到一个“空指针违例”。但假如将 getParameter()初始化模块移入 init()内部，则可正常工作。小程序框架会进行必要的启动工作，以便在进入 init()前收集好一些参数。

此外，上述代码被同时设置成一个小程序和一个应用程序。在它是应用程序的情况下，size 参数可从命令行里提取出来（否则就提供一个默认值）。

数组长度决定好以后，就可以开始创建新的 Ticker 对象；作为 Ticker 构造函数的一部分，用于每个 Ticker 的按钮和文本字段会加入小程序。

按下 Start 按钮后，会在整个 Ticker 数组里遍历，并为每个 Ticker 都调用 start()。记住，start()会进行必要的线程初始化工作，再为那个线程调用 run()。

ToggleL 监视器只是简单地反转一下 Ticker 中的标记，一旦对应的线程以后需要修改这个标记，它便会作出相应的反应。

这个例子的一个好处是它使我们能够方便地创建由独立子任务构成的大型集合，并以监视它们的行为。在这种情况下，我们会发现随着子任务数量的增多，机器显示出来的数字可能会出现更大的分歧，这是由于为线程提供服务的方式造成的。

亦可试着体验一下 sleep(100)在 Ticker.run()中的重要作用。若删除 sleep()，那么在按下一个切换按钮前，情况仍然会进展良好。按下按钮以后，那个特定的线程就会出现一个错误的 runFlag，而且 run()会深深陷入一个无限循环——很难在多任务处理期间中断退出。因此，程序对用户操作的反应灵敏度会大幅度降低。

14.1.5 Daemon 线程

“Daemon”线程的作用是在程序运行期间，在后台提供一种“常规”服务，但它并不属于程序的一个基本部分。因此，一旦所有非 Daemon 线程完成，程序也会中止运行。相反，假若有任何非 Daemon 线程仍在运行（比如还有一个正在运行 main()的线程），则程序的运行不会中止。

通过调用 isDaemon()，可调查一个线程是不是一个 Daemon，而且能用 setDaemon()打开或者关闭一个线程的 Daemon 状态。如果是一个 Daemon 线程，那么它创建的任何线程也会自动具备 Daemon 属性。

下面这个例子演示了 Daemon 线程的用法：

```
//: c14:Daemons.java
// Daemonic behavior.
import java.io.*;

class Daemon extends Thread {
  private static final int SIZE = 10;
  private Thread[] t = new Thread[SIZE];
  public Daemon() {
    setDaemon(true);
    start();
  }
  public void run() {
    for(int i = 0; i < SIZE; i++)
      t[i] = new DaemonSpawn(i);
    for(int i = 0; i < SIZE; i++)
      System.out.println(
        "t[" + i + "].isDaemon() = "
        + t[i].isDaemon());
    while(true)
      yield();
  }
}
```

```
class DaemonSpawn extends Thread {
  public DaemonSpawn(int i) {
    System.out.println(
      "DaemonSpawn " + i + " started");
    start();
  }
  public void run() {
    while(true)
      yield();
  }
}

public class Daemons {
  public static void main(String[] args)
  throws IOException {
    Thread d = new Daemon();
    System.out.println(
      "d.isDaemon() = " + d.isDaemon());
    // Allow the daemon threads to
    // finish their startup processes:
    System.out.println("Press any key");
    System.in.read();
  }
} ///:~
```

Daemon 线程可将自己的 Daemon 标记设置成“真”，然后产生一系列其他线程，而且认为它们也具有 Daemon 属性。随后，它进入一个无限循环，在其中调用 yield()，放弃对其他进程的控制。在这个程序早期的一个版本中，无限循环会使 int 计数器增值，但会使整个程序都好象陷入停顿状态。换用 yield()后，却可使程序充满“活力”，不会使人产生停滞或反应迟钝的感觉。

一旦 main()完成自己的工作，便没有什么能阻止程序中断运行，因为这里运行的只有 Daemon 线程。所以能看到启动所有 Daemon 线程后显示出来的结果，System.in 也进行了相应的设置，使程序中断前能等待一个回车。如果不进行这样的设置，就只能看到创建 Daemon 线程的一部分结果（试试将 read()代码换成不同长度的 sleep()调用，看看会有什么效果）。

14.2 共享有限的资源

可将单线程程序想象成一种孤立的实体，它能遍历我们的问题空间，而且一次只能做一件事情。由于只有一个实体，所以永远不必担心会有两个实体同时试图使用相同的资源，就象两个人同时都想停到同一个车位，同时都想通过一扇门，甚至同时发话等等。

进入多线程环境后，它们则再也不是孤立的。可能会有两个甚至更多的线程试图同时同一个有限的资源。必须对这种潜在资源冲突进行预防，否则就可能发生两个线程同时访问一个银行帐号、打印到同一台计算机以及对同一个值进行调整等等不利情况。

14.2.1 不正确的资源访问方法

现在考虑换成另一种方式来使用本章频繁见到的计数器。在下面的例子中，每个线程都包含了两个计数器，它们在 run()里增值以及显示。除此以外，我们使用了 Watcher 类的另一个线程。它的作用是监视计数器，检查它们的值是否总是相同。这表面上是一项无意义的行动，因为许多人都会认为，假如查看代码，那么会发现计数器“肯定”是相同的。但实际情况却不一定如此，希望你看到结果后不至于过份惊讶！下面是程序的第一个版本：

```
//: c14:Sharing1.java
// Problems with resource sharing while threading.
// <applet code=Sharing1 width=350 height=500>
// <param name=size value="12">
// <param name=watchers value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Sharing1 extends JApplet {
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  private JButton
    start = new JButton("Start"),
    watcher = new JButton("Watch");
  private boolean isApplet = true;
  private int numCounters = 12;
  private int numWatchers = 15;
  private TwoCounter[] s;
  class TwoCounter extends Thread {
    private boolean started = false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    // Add the display components as a panel:
    public TwoCounter() {
```

```
    JPanel p = new JPanel();
    p.add(t1);
    p.add(t2);
    p.add(l);
    getContentPane().add(p);
  }
  public void start() {
    if(!started) {
      started = true;
      super.start();
    }
  }
  public void run() {
    while (true) {
      t1.setText(Integer.toString(count1++));
      t2.setText(Integer.toString(count2++));
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
  public void synchTest() {
    Sharing1.incrementAccess();
    if(count1 != count2)
      l.setText("Unsynched");
  }
}
class Watcher extends Thread {
  public Watcher() { start(); }
  public void run() {
    while(true) {
      for(int i = 0; i < s.length; i++)
        s[i].synchTest();
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}
class StartL implements ActionListener {
```

```
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class WatcherL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numWatchers; i++)
        new Watcher();
    }
  }
  public void init() {
    if(isApplet) {
      String counters = getParameter("size");
      if(counters != null)
        numCounters = Integer.parseInt(counters);
      String watchers = getParameter("watchers");
      if(watchers != null)
        numWatchers = Integer.parseInt(watchers);
    }
    s = new TwoCounter[numCounters];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new TwoCounter();
    JPanel p = new JPanel();
    start.addActionListener(new StartL());
    p.add(start);
    watcher.addActionListener(new WatcherL());
    p.add(watcher);
    p.add(new JLabel("Access Count"));
    p.add(aCount);
    cp.add(p);
  }
  public static void main(String[] args) {
    Sharing1 applet = new Sharing1();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
```

```
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numCounters * 50);
  }
} ///:~
```

和往常一样，每个计数器都包含了自己的显示组件：两个文本字段以及一个标签。根据它们的初始值，可知道计数是相同的。这些组件在 TwoCounter 构造函数中加入外层类对象的“内容区”（Content Pane）。

由于一个 TwoCounter 线程是通过用户按键的行动而启动的，所以 start()可能被多次调用。但对一个线程来说，对 Thread.start()的多次调用是非法的（会产生违例）。在 started 标记和被覆盖的 start()方法中，大家可看到针对这一情况采取的防范措施。

在 run()中，count1 和 count2 的增值与显示方式表面上似乎能保持它们完全一致。随后会调用 sleep()；假如没有这个调用，程序便会“凝”在那里，因为 CPU 会变得难于交换任务。

synchTest()方法采取的似乎是没有任何意义的一项行动：它检查 count1 是否等于 count2；如果不等，就把标签设为“Unsynched”（不同步）。但是首先，它调用的是类 Sharing1 的一个静态成员，以便增值和显示一个访问计数器，指出这种检查已成功进行了多少次（这样做的理由会在本例的其他版本中变得非常明显）。

Watcher 类是一个线程，它的作用是为处于活动状态的所有 TwoCounter 对象都调用 synchTest()。其间，它会对 Sharing1 对象中容纳的数组进行遍历。可将 Watcher 想象成它从 TwoCounter 对象的“肩膀”上，不断地“偷看”。

Sharing1 包含了 TwoCounter 对象的一个数组，它在 init()中进行初始化，并在我们按下“start”按钮后作为线程启动。以后若按下“Watch”（监视）按钮，就会创建一个或者多个监视者（Watcher），并对毫不设防的 TwoCounter 进行调查。

注意为了让它作为一个“小程序”在浏览器中运行，Web 页的 applet 标记中需要包含下面这几行：

```
<param name=size value="20">
<param name=watchers value="1">
```

可自行改变宽度、高度以及参数，根据自己的意愿进行试验。若改变了 size 和 watchers，程序的行为也会发生变化。我们也注意到，通过从命令行接受参数（否则就用默认值），它被设计成一个独立的应用程序运行。

下面才是最让人“不可思议”的部分。在 TwoCounter.run()中，无限循环只是不断地重复相邻的行：

```
t1.setText(Integer.toString(count1++));
t2.setText(Integer.toString(count2++));
```

（和“睡眠”一样，不过那在这里并不重要）。然而，在程序运行的时候，你会发现 count1 和 count2 被“观察”（用 Watcher 观察）的次数是不相等的！这是由线程的本质造成的——它们可在任何时候挂起（暂停）。所以在上述两行的执行时刻之间，有时会出现执行暂停现象。同时，Watcher 线程也正好跟随着进来，并恰好在这个时候进行比较，造成计数器值不同的情况。

本例揭示了使用线程时一个非常基本的问题。我们跟无从知道一个线程什么时候运行。想象自己坐在一张桌子前面，桌上放有一把叉子，准备叉起自己的最后一块肉。当叉子要碰到肉时，肉却突然消失了（因为你的线程已经“挂起”，同时有另一个线程进来“偷”走了

那块肉）。这便是我们要解决的问题。

有时候，当我们试着访问一个资源时（就象去吃摆在别人盘子里的肉），并不介意它当前正在由其他人访问。但为了让多线程机制能够正常运转，还是要采取一些措施，防止两个线程同时访问相同的资源——至少在敏感的时期。

为防止出现这样的冲突，只需在线程使用一个资源时为其加锁即可。访问资源的第一个线程会其加上锁以后，其他线程便不能再使用那个资源，除非被解锁。想象一下，这就好比车子的前座属于“有限的资源”，而高喊着“这是我的!”的孩子会坚决地主张把它锁起来。

14.2.2 Java 如何共享资源

针对一种特殊的资源——对象中的内存——Java 提供了内建的机制来防止它们的冲突。由于我们通常将数据元素设为从属于 private（私有）类，然后只通过方法访问那些内存，所以只需将一个特定的方法设为 synchronized（同步的），便可有效地防止冲突。在任何时刻，只可有一个线程调用特定对象的一个 synchronized 方法（尽管那个线程可以调用多个对象的同步方法）。下面列出简单的 synchronized 方法：

```
synchronized void f() { /* ... */ }
synchronized void g(){ /* ... */ }
```

每个对象都包含了一把锁（也叫作“监视器”），它自动成为对象的一部分（不必为此写任何特殊代码）。调用任何 synchronized 方法时，对象就会被锁定，不可再调用那个对象的其他任何 synchronized 方法，除非第一个方法完成了自己的工作，并解除锁定。在上面的例子中，如果为一个对象调用 f()，便不能再为同样的对象调用 g()，除非 f()完成并解除锁定。因此，一个特定对象的所有 synchronized 方法都共享着一把锁，而且这把锁能防止多个方法对通用内存同时进行写操作（比如同时有多个线程）。

每个类也有自己的一把锁（作为类的 Class 对象的一部分），所以 synchronized static 方法可在一个类的范围内被相互间锁定起来，防止与 static 数据的接触。

注意假如想保护其他某些资源不被多个线程同时访问，可通过 synchronized 方法强行访问那些资源。

1. 计数器的同步

有了这个新关键字的帮助后，我们可以采取的方案便更加灵活了：可以只为 TwoCounter 中的方法简单地使用 synchronized 关键字。下面这个例子是对前例的改版，其中加入了新的关键字：

```
//: c14:Sharing2.java
// Using the synchronized keyword to prevent
// multiple access to a particular resource.
// <applet code=Sharing2 width=350 height=500>
// <param name=size value="12">
// <param name=watchers value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
```

```
public class Sharing2 extends JApplet {
  TwoCounter[] s;
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  private JButton
    start = new JButton("Start"),
    watcher = new JButton("Watch");
  private boolean isApplet = true;
  private int numCounters = 12;
  private int numWatchers = 15;

  class TwoCounter extends Thread {
    private boolean started = false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    public TwoCounter() {
      JPanel p = new JPanel();
      p.add(t1);
      p.add(t2);
      p.add(l);
      getContentPane().add(p);
    }
    public void start() {
      if(!started) {
        started = true;
        super.start();
      }
    }
    public synchronized void run() {
      while (true) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
```

```
        System.err.println("Interrupted");
      }
    }
  }
  public synchronized void synchTest() {
    Sharing2.incrementAccess();
    if(count1 != count2)
      l.setText("Unsynched");
  }
}

class Watcher extends Thread {
  public Watcher() { start(); }
  public void run() {
    while(true) {
      for(int i = 0; i < s.length; i++)
        s[i].synchTest();
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}
class StartL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    for(int i = 0; i < s.length; i++)
      s[i].start();
  }
}
class WatcherL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    for(int i = 0; i < numWatchers; i++)
      new Watcher();
  }
}
public void init() {
  if(isApplet) {
    String counters = getParameter("size");
    if(counters != null)
      numCounters = Integer.parseInt(counters);
    String watchers = getParameter("watchers");
    if(watchers != null)
```

```
        numWatchers = Integer.parseInt(watchers);
    }
    s = new TwoCounter[numCounters];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new TwoCounter();
    JPanel p = new JPanel();
    start.addActionListener(new StartL());
    p.add(start);
    watcher.addActionListener(new WatcherL());
    p.add(watcher);
    p.add(new Label("Access Count"));
    p.add(aCount);
    cp.add(p);
  }
  public static void main(String[] args) {
    Sharing2 applet = new Sharing2();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numCounters * 50);
  }
} ///:~
```

我们注意到无论 run()还是 synchTest()都是“同步的”。假如只同步其中的一个方法，那么另一个就可以从容地忽略挂在面前的“对象锁”，并可毫无妨碍地调用。所以必须记住一个重要的规则：对于要访问某个关键共享资源的所有方法来说，都必须把它们设为 synchronized，否则便不能正常地工作。

现在又遇到了一个新问题。Watcher 永远都不能看到正在进行的事情，因为整个 run()方法已设为“同步”。而且由于肯定要为每个对象运行 run()，所以锁永远不能打开，而 synchTest()永远不会得到调用。根据 accessCount 毫无变化这一事实，可以很轻易地看出上述问题。

为解决这个问题，我们能采取的一个办法是只将 run()中的一部分代码隔离出来。想用这个办法隔离出来的那部分代码叫作“关键区域”，而且要用不同的方式来使用 synchronized 关键字，以设置一个关键区域。Java 通过“同步块”提供对关键区域的支持；这一次，我们用 synchronized 关键字指出对象的锁用于对其中封闭的代码进行同步。如下所示：

```
synchronized(syncObject) {
  // This code can be accessed
  // by only one thread at a time
}
```

在能进入同步块之前，必须在 synchObject 上取得锁。如果已有其他线程取得了这把锁，块便不能进入，必须等候那把锁被释放。

可从整个 run()中删除 synchronized 关键字，换成用一个同步块包围两个关键行，从而完成对 Sharing2 例子的修改。但什么对象应作为锁来使用呢？那个对象已由 synchTest()标记出来了——也就是当前对象（this）！所以修改过的 run()方法看起来象下面这个样子：

```
  public void run() {
    while (true) {
      synchronized(this) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
      }
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
```

这是唯一必须对 Sharing2.java 作出的修改，我们会看到尽管两个计数器永远不会脱离同步（取决于允许 Watcher 什么时候检查它们），但在 run()执行期间，仍然向 Watcher 提供了足够的访问权限。

当然，所有同步都取决于程序员是否勤奋：会访问共享资源的每一部分代码都必须封装到一个恰当的同步块里。

2. 同步的效率

由于要为同样的数据编写两个方法，所以无论如何都不会给人留下效率很高的印象。看来似乎更好的一种做法是将所有方法都设为自动同步，并完全消除 synchronized 关键字（当然，含有 synchronized run()的例子显示出那样做也是行不通的）。但它也揭示出获取一把锁并非一种“廉价”方案——为一次方法调用付出的代价（进入和退出方法，不执行方法主体）至少要累积成原先的四倍，而且根据我们的具体现方案，这一代价还有可能变得更高。所以假如已知某个方法不会造成冲突，最明智的做法便是撤消其中的 synchronized 关键字。不过这里要警告大家，千万不要仅仅为了提高性能而撤消该关键字。假如你认为它是一个性能瓶颈，并“希望”它不会出现任何冲突，那么无疑是为程序以后的崩溃埋下了伏笔！

14.2.3 再论 JavaBeans

我们现在已理解了同步，接着可换从另一个角度来考察 JavaBeans。无论什么时候创建

了一个 Bean，就必须假定它以后会在一个多线程的环境中运行。这意味着：

(1) 只要可能，Bean 的所有公共方法都应同步。当然，这也带来了“同步”在运行时间的开销。若特别在意这个问题，在关键区域中不会造成问题的方法就可保留为“不同步”，但注意这通常都不是十分容易判断。真正设计优秀的方法应倾向于“个头”很小（如下例的 getCircleSize()），而且只负责解决一个特定的问题（将问题尽可能地分细一点儿）。也就是说，方法调用在数量如此少的代码里执行，在执行期间，不可能造成对象的改动。对这样的方法来说，即使没有把它设成“同步”，对程序的执行速度也不会有什么明显影响。你可能还要将一个 Bean 的所有 public 方法都设为 synchronized，并只有在迫不得已的情况下（真的会造成性能上的差异），才应考虑删去 synchronized 关键字。

(2) 假如同时为一系列监听器（Listener）触发了一个多强制转型（Multicast）事件，那么在列表中遍历时，必须假定那些监听器也有可能在其中添加或删除。

第 1 点很容易处理，但第 2 点需要考虑更多的东西。让我们以前一章提供的 BangBean.java 为例。在那个例子中，我们忽略了 synchronized 关键字（那时还没有引入呢），并将强制转型设为单强制转型，从而回避了多线程的问题。在下面这个修改过的版本中，我们使其能在多线程环境中工作，并为事件采用了多强制转型技术：

```
//: 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);
  }
} ///:~
```

很容易就可以为方法添加 synchronized。但注意在 addActionListener()和 removeActionListener()中，现在添加了 ActionListener，并从一个 ArrayList 中移去，所以能够根据自己愿望使用任意多个。

我们注意到，notifyListeners()方法并未设为“同步”。可从多个线程中发出对这个方法的调用。另外，在对 notifyListeners()调用的中途，也可能发出对 addActionListener()和 removeActionListener()的调用。这显然会造成问题，因为它否定了 ArrayList actionListeners。为缓解这个问题，我们在一个 synchronized 从句中“克隆”了 ArrayList，并对克隆得到的那个 ArrayList 进行遍历（“克隆”的详情请参考附录 A）。这样便可在不影响 notifyListeners()的前提下，对 ArrayList 进行操纵。

paintComponent()方法也没有设为“同步”。与单纯地添加自己的方法相比，决定是否对重载的方法进行同步要困难得多。在这个例子中，无论 paint()是否“同步”，它似乎都能正常地工作。但必须考虑的问题包括：

(1) 方法会在对象内部修改“关键”变量的状态吗？为判断一个变量是不是“关键”的，必须知道它是否会被程序中的其他线程读取或设置（就目前的情况看，读取或设置几乎肯定是通过“同步”方法进行的，所以可以只对它们进行检查）。对 paint()的情况来说，不会发生任何修改。

(2) 方法要以这些“关键”变量的状态为基础吗？如果一个“同步”方法修改了一个变量，而我们的方法要用到这个变量，那么一般都愿意把自己的方法也设为“同步”。基于这一前提，大家可观察到 cSize 由“同步”方法进行了修改，所以 paint()应当是“同步”的。但在这里，我们可以问：“假如 cSize 在 paint()执行期间发生了变化，会发生的最糟糕的事情是什么呢？”如果发现情况不算太坏，而且仅仅是暂时的效果，那么最好保持 paint()的“不同步”状态，以避免同步方法调用带来的额外开销。

(3) 要留意的第三条线索是 paint()基类版本是否“同步”，在这里它不是同步的。这并不是一个非常严格的参数，仅仅是一条“线索”。比如在目前的情况下，通过同步方法改变的一个字段（cSize）已合成到 paint()公式里，而且可能已改变了情况。但请注意，synchronized 不能继承——也就是说，假如一个方法在基类中是“同步”的，那么在派生类重载版本中，它不会自动进入“同步”状态。

TestBangBean2 中的测试代码已在前一章的基础上进行了修改，在其中加入了额外的监听器，从而演示了 BangBean2 的多强制转型能力。

14.3 堵　　塞

一个线程可以有四种状态：

(1) 新（New）：线程对象已经创建，但尚未启动，所以不可运行。

(2) 可运行（Runnable）：意味着一旦时间分片机制有空闲的 CPU 周期提供给一个线程，那个线程便可立即开始运行。因此，线程可能在、也可能不在运行当中，但一旦条件许可，没有什么能阻止它的运行——它既没有“死”掉，也未被“堵塞”——只是在等着运行。

(3) 死亡（Dead）：从线程的 run()方法中返回后，那个线程便已“死”掉。亦可直接调用 stop()令其强行“死亡”，但这样会产生一个违例——属于 Error 的一个子类（也就是说，我们通常不捕获它）。记住违例的产生应当当作一种特殊事件考虑，而不能把它当作正常程序运行的一部分。所以在 Java 2 中，根本不建议你使用 stop()。另外还有一个 destroy()方法（它从来没实现过），应该尽可能地避免调用它，因为它非常武断，根本不会解除对象的锁定。

(4) 堵塞（Blocked）：线程可以运行，但有某种东西阻碍了它。若线程处于堵塞状态，调度机制可以简单地跳过它，不给它分配任何 CPU 时间。除非线程再次进入“可运行”状态，否则不会采取任何操作。

14.3.1 为何会堵塞

“堵塞”状态是前述四种状态中最有趣的，值得我们作进一步的探讨。线程被堵塞可能是由下述五方面的原因造成的：

(1) 调用 sleep(毫秒数)，使线程进入“睡眠”状态。在规定的时间内，这个线程是不会运行的。

(2) 用 suspend()暂停（挂起）了线程的执行。除非线程收到 resume()消息，否则不会恢复“可运行”状态（Java 2 里不赞成这样，后文还会详细讨论）。

(3) 用 wait()暂停了线程的执行。除非线程收到 nofify()或者 notifyAll()消息，否则不会变成“可运行”（是的，这看起来同原因(2)非常相象，但有一个明显的区别是我们马上就要揭示的）。

(4) 线程正在等候一些 I/O（输入输出）操作完成。

(5) 线程试图调用另一个对象的“同步”方法，但那个对象处于锁定状态，暂时无法使用。

亦可调用 yield()（Thread 类提供的一个方法），从而自动放弃 CPU，以便其他线程能够运行。然而，假如调度机制觉得我们的线程已拥有足够的时间，并跳转到另一个线程，就会发生同样的事情。也就是说，没有什么能防止调度机制重新启动我们的线程。线程被堵塞后，便有一些原因造成它不能继续运行。

下面这个例子展示了进入堵塞状态的全部五种途径。它们全都存在于名为 Blocking.java 的一个文件中，但在这儿将利用一系列不连贯的片断进行解释——大家可注意到片断前后的“Continued”（待续）以及“Continuing”（续前文）标志。一个代码组织工具可利用这些标志，将不连续的代码片断连成一个完整的程序。

由于该例演示的是一些不连续的方法，所以在编译的时候，肯定会获得“不推荐使用”（Deprecated）的消息提示，这是正常的：

首先来看看基本框架：

```
//: c14:Blocking.java
// Demonstrates the various ways a thread
// can be blocked.
// <applet code=Blocking width=350 height=550>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import com.bruceeckel.swing.*;

////////////// The basic framework ///////////
class Blockable extends Thread {
  private Peeker peeker;
  protected JTextField state = new JTextField(30);
  protected int i;
  public Blockable(Container c) {
    c.add(state);
    peeker = new Peeker(this, c);
  }
  public synchronized int read() { return i; }
  protected synchronized void update() {
    state.setText(getClass().getName()
      + " state: i = " + i);
  }
  public void stopPeeker() {
    // peeker.stop(); Deprecated in Java 1.2
    peeker.terminate(); // The preferred approach
  }
}

class Peeker extends Thread {
  private Blockable b;
  private int session;
  private JTextField status = new JTextField(30);
  private boolean stop = false;
  public Peeker(Blockable b, Container c) {
    c.add(status);
    this.b = b;
    start();
  }
  public void terminate() { stop = true; }
  public void run() {
    while (!stop) {
```

```
      status.setText(b.getClass().getName()
        + " Peeker " + (++session)
        + "; value = " + b.read());
       try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
} ///:Continued
```

Blockable 类的意图是成为本例所有类的一个基类。一个 Blockable 对象包含了一个名为 state 的 JTextField（文本字段），用于显示出对象有关的信息。用于显示这些信息的方法叫作 update()。我们发现它用 getClass.getName()来产生类名，而不是仅仅把它打印出来；这是由于 update()不知道自己为其调用的那个类的准确名字，因为那个类是从 Blockable 派生出来的。

在 Blockable 中，变动指示符是一个 int i；派生类的 run()方法会为其增值。

针对每个 Bloackable 对象，都会启动 Peeker 类的一个线程。Peeker 的任务是调用 read()方法，检查与自己关联的 Blockable 对象，看看 i 是否发生了变化，最后用它的 status 文本字段报告检查结果。注意 read()和 update()都是同步的，要求对象的锁定能自由解除，这一点非常重要。

1. 睡眠

这个程序的第一项测试是用 sleep()进行的：

```
///:Continuing
////////////// Blocking via sleep() ///////////
class Sleeper1 extends Blockable {
  public Sleeper1(Container c) { super(c); }
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}

class Sleeper2 extends Blockable {
```

```
  public Sleeper2(Container c) { super(c); }
  public void run() {
    while(true) {
      change();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
  public synchronized void change() {
      i++;
      update();
  }
} ///:Continued
```

在 Sleeper1 中，整个 run()方法都是“同步的”。我们可看到与这个对象关联在一起的 Peeker 可以正常运行——在我们启动线程之前。随后，Peeker 便会完全停止。这正是“堵塞”的一种形式：因为 Sleeper1.run()是同步的，而且一旦线程启动，它就肯定在 run()内部，方法永远不会放弃对象锁定，造成 Peeker 线程的堵塞。

Sleeper2 则没有将 run()方法设为“同步的”，从而提供了一种解决方案。只有 change()方法才是同步的，所以尽管 run()位于 sleep()内部，Peeker 仍然能访问自己需要的同步方法——read()。在这里，我们可看到在启动了 Sleeper2 线程以后，Peeker 会持续运行下去。

2. 暂停和恢复

这个例子接下来的一部分引入了“挂起”或者“暂停”（Suspend）的概述。Thread 类提供了一个名为 suspend()的方法，可临时中止线程；以及一个名为 resume()的方法，用于从暂停处开始恢复线程的执行。显然，我们可以推断出 resume()是由暂停线程外部的某个线程调用的。在这种情况下，需要用到一个名为 Resumer（恢复器）的独立类。演示暂停 / 恢复过程的每个类都有一个对应的恢复器。如下所示：

```
///:Continuing
////////////// Blocking via suspend() //////////
class SuspendResume extends Blockable {
  public SuspendResume(Container c) {
    super(c);
    new Resumer(this);
  }
}

class SuspendResume1 extends SuspendResume {
  public SuspendResume1(Container c) { super(c);}
  public synchronized void run() {
```

```
    while(true) {
      i++;
      update();
      suspend(); // Deprecated in Java 1.2
    }
  }
}

class SuspendResume2 extends SuspendResume {
  public SuspendResume2(Container c) { super(c);}
  public void run() {
    while(true) {
      change();
      suspend(); // Deprecated in Java 1.2
    }
  }
  public synchronized void change() {
      i++;
      update();
  }
}

class Resumer extends Thread {
  private SuspendResume sr;
  public Resumer(SuspendResume sr) {
    this.sr = sr;
    start();
  }
  public void run() {
    while(true) {
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      sr.resume(); // Deprecated in Java 1.2
    }
  }
} ///:Continued
```

SuspendResume1 也提供了一个同步的 run()方法。同样地，当我们启动这个线程以后，就会发现与它关联的 Peeker 进入“堵塞”状态，等候对象锁被释放——但那永远不会发生。和往常一样，这个问题在 SuspendResume2 里得到了解决，它并不同步整个 run()方法，而是采用了一个独立的同步 change()方法。

在 Java 2 中，请注意 suspend()和 resume()的使用是被强烈反对的。由于 suspend()包含了对象锁，所以极易出现“死锁”现象。换言之，很容易就会看到许多被锁住的对象在傻乎乎地等待对方。这会造成整个应用程序的“凝固”。尽管在一些老程序中还能看到它们的踪迹，但在你写自己的程序时，无论如何都应避免。本章稍后便会讲述正确的方案是什么。

3. 等待和通知

通过前两个例子的实践，我们知道无论 sleep()还是 suspend()都不会在自己被调用的时候解除锁定。需要用到对象锁时，请务必注意这个问题。在另一方面，wait()方法在被调用时却会解除锁定，这意味着可在执行 wait()期间调用线程对象中的其他同步方法。但在接着的两个类中，我们看到 run()方法都是“同步”的。在 wait()期间，Peeker 仍然拥有对同步方法的完全访问权限。这是由于 wait()在挂起内部调用的方法时，会解除对象的锁定。

我们也可以看到 wait()的两种形式。第一种形式采用一个以毫秒为单位的参数，它具有与 sleep()中相同的含义：暂停这一段规定时间。区别在于在 wait()中，对象锁已被解除，而且能够自由地退出 wait()，因为一个 notify()可强行使时间流逝。

第二种形式不采用任何参数，这意味着 wait()会持续执行，直到 notify()介入为止。而且在一段时间以后，不会自行中止。

wait()和 notify()比较特别的一个地方是这两个方法都属于基类 Object 的一部分，不象 sleep()、suspend()以及 resume()那样属于 Thread 的一部分。尽管这表面看有点儿奇怪——居然让专门进行线程处理的东西成为通用基类的一部分——但仔细想想又会释然，因为它们操纵的对象锁也属于每个对象的一部分。因此，我们可将一个 wait()置入任何同步方法内部，无论在那个类里是否准备进行涉及线程的处理。事实上，我们能调用 wait()的唯一地方是在一个同步的方法或代码块内部。若在一个不同步的方法内调用 wait()或者 notify()，尽管程序仍会编译，但在运行它的时候，会得到一个 IllegalMonitorStateException（非法监视器状态违例），而且会出现一条多少有点儿让人莫名其妙的一条消息：“current thread notowner”（当前线程不是所有人）。注意 sleep()、suspend()以及 resume()都能在非同步的方法内调用，因为它们不需要对锁定进行操作。

只能为自己的锁定调用 wait()和 notify()。同样地，仍然可以编译那些试图使用错误锁定的代码，但和往常一样会产生同样的 IllegalMonitorStateException 违例。我们没办法用其他人的对象锁来愚弄系统，但可要求另一个对象执行相应的操作，对它自己的锁进行操作。所以一种做法是创建一个同步方法，令其为自己的对象调用 notify()。但在 Notifier 中，我们会看到一个同步方法内部的 notify()：

```
synchronized(wn2) {
  wn2.notify();
}
```

其中，wn2 是类型为 WaitNotify2 的对象。尽管并不属于 WaitNotify2 的一部分，这个方法仍然获得了 wn2 对象的锁。在这个时候，它为 wn2 调用 notify()是合法的，不会产生什么 IllegalMonitorStateException 违例。

```
///:Continuing
////////// Blocking via wait() //////////
class WaitNotify1 extends Blockable {
  public WaitNotify1(Container c) { super(c); }
```

```
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        wait(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}

class WaitNotify2 extends Blockable {
  public WaitNotify2(Container c) {
    super(c);
    new Notifier(this);
  }
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        wait();
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}

class Notifier extends Thread {
  private WaitNotify2 wn2;
  public Notifier(WaitNotify2 wn2) {
    this.wn2 = wn2;
    start();
  }
  public void run() {
    while(true) {
       try {
        sleep(2000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
```

```
      synchronized(wn2) {
        wn2.notify();
      }
    }
  }
} ///:Continued
```

若必须等候其他某些条件（从线程外部加以控制）发生变化，同时又不想在线程内一直傻乎乎地等下去，一般就需要用到 wait()。wait()允许我们将线程置入“睡眠”状态，同时又“积极”地等待条件发生改变。而且只有在一个 notify()或 notifyAll()发生变化的时候，线程才会被唤醒，并检查条件是否有变。因此，我们认为它提供了在线程间进行同步的一种手段。

4. I/O 堵塞

若一个数据流必须等候一些 I/O 活动结束，便会自动进入“堵塞”状态。在下面的代码片断中，有两个类协同标准的 Reader 及 Writer 对象工作。但在测试框架中，我们将设置一个管道化的数据流，使两个线程相互间能安全地传递数据（这正是使用管道流的目的）。

Sender 将数据置入 Writer，并“睡眠”随机长短的时间。然而，Receiver 本身并没有包括 sleep()、suspend()或者 wait()方法。但在执行 read()的时候，如果没有数据存在，它会自动进入“堵塞”状态。如下所示：

```
///:Continuing
class Sender extends Blockable { // send
  private Writer out;
  public Sender(Container c, Writer out) {
    super(c);
    this.out = out;
  }
  public void run() {
    while(true) {
      for(char c = 'A'; c <= 'z'; c++) {
        try {
          i++;
          out.write(c);
          state.setText("Sender sent: "
            + (char)c);
          sleep((int)(3000 * Math.random()));
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        } catch(IOException e) {
          System.err.println("IO problem");
        }
      }
    }
  }
```

```
}

class Receiver extends Blockable {
  private Reader in;
  public Receiver(Container c, Reader in) {
    super(c);
    this.in = in;
  }
  public void run() {
    try {
      while(true) {
        i++; // Show peeker it's alive
        // Blocks until characters are there:
        state.setText("Receiver read: "
          + (char)in.read());
      }
    } catch(IOException e) {
      System.err.println("IO problem");
    }
  }
} ///:Continued
```

这两个类也将信息送入自己的 state 字段，并修改 i 值，使 Peeker 知道线程仍在运行。

5. 测试

令人惊讶的是，主要的小程序（Applet）类实际非常简单，这是由于大多数工作都已置入 Blockable 框架的缘故。大概地说，我们创建了一个由 Blockable 对象构成的数组。而且由于每个对象都是一个线程，所以在按下“start”按钮后，它们会采取自己的行动。还有另一个按钮和 actionPerformed()从句，用于中止所有 Peeker 对象。由于 Java 2“反对”使用 Thread 的 stop()方法，所以可考虑采用这种折衷形式的中止方式。

为了在 Sender 和 Receiver 之间建立一个连接，我们创建了一个 PipedWriter 和一个 PipedReader。注意 PipedReader in 必须通过一个构造函数参数同 PipedWriterout 连接起来。在那以后，我们在 out 内放进去的所有东西都可从 in 中提取出来——似乎那些东西是通过一个“管道”传过去的。随后将 in 和 out 对象分别传递给 Receiver 和 Sender 构造函数；后者将它们当作任意类型的 Reader 和 Writer 看待（也就是说，它们被“向上”强制转型了）。

由 Blockable 引用构成的数组 b 在定义之初并未得到初始化，因为管道化的数据流是没法子在定义前设置好的（对 try 块的需要会成为障碍）：

```
///:Continuing
////////// Testing Everything //////////
public class Blocking extends JApplet {
  private JButton
    start = new JButton("Start"),
    stopPeekers = new JButton("Stop Peekers");
```

```
  private boolean started = false;
  private Blockable[] b;
  private PipedWriter out;
  private PipedReader in;
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < b.length; i++)
          b[i].start();
      }
    }
  }
  class StopPeekersL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // Demonstration of the preferred
      // alternative to Thread.stop():
      for(int i = 0; i < b.length; i++)
        b[i].stopPeeker();
    }
  }
  public void init() {
     Container cp = getContentPane();
     cp.setLayout(new FlowLayout());
     out = new PipedWriter();
    try {
      in = new PipedReader(out);
    } catch(IOException e) {
      System.err.println("PipedReader problem");
    }
    b = new Blockable[] {
      new Sleeper1(cp),
      new Sleeper2(cp),
      new SuspendResume1(cp),
      new SuspendResume2(cp),
      new WaitNotify1(cp),
      new WaitNotify2(cp),
      new Sender(cp, out),
      new Receiver(cp, in)
    };
    start.addActionListener(new StartL());
    cp.add(start);
    stopPeekers.addActionListener(
      new StopPeekersL());
```

```
    cp.add(stopPeekers);
  }
  public static void main(String[] args) {
    Console.run(new Blocking(), 350, 550);
  }
} ///:~
```

在 init()中，注意循环会遍历整个数组，并为页添加 state 和 peeker.status 文本字段。

第一次创建好各个 Blockable 线程后，每个这样的线程都会自动创建并启动自己的 Peeker。所以我们会看到各个 Peeker 都在 Blockable 线程启动之前运行起来。这一点非常重要，因为在 Blockable 线程启动的时候，部分 Peeker 会被堵塞，并停止运行。弄懂这一点，将有助于我们加深对“堵塞”这一概念的认识。

14.3.2 死锁

由于线程可能进入堵塞状态，而且由于对象可能拥有“同步”方法——除非同步锁定被解除，否则线程不能访问那个对象——所以一个线程完全可能等候另一个对象，而另一个对象又在等候下一个对象，以此类推。这个“等候”链最可怕的情形就是进入封闭状态——最后那个对象等候的是第一个对象！此时，所有线程都会陷入无休止的相互等待状态，大家都动弹不得。我们将这种情况称为“死锁”。尽管这种情况并非经常出现，但一旦碰到，程序的调试将变得异常艰难。

就语言本身来说，尚未直接提供防止死锁的帮助措施，需要我们通过谨慎的设计加以避免。如果有谁需要调试一个死锁的程序，他是没有任何窍门可用的。

Java 2 反对使用 stop()、suspend()、resume()以及 destroy()

为减少出现死锁的可能，Java 2 的一项改动便是“反对”使用 Thread 的 stop()、suspend()、resume()以及 destroy()方法。

之所以反对使用 stop()，是因为它会解除由线程获取的所有锁定，而且假如对象处于一种不连贯状态（“被破坏”），那么其他线程能在那种状态下检查和修改它们。结果便造成了一种微妙的局面，我们很难检查出真正的问题所在。所以应尽量避免使用 stop()，应该采用 Blocking.java 里的那种方法，用一个标志告诉线程什么时候通过退出自己的 run()方法来中止自己的执行。

如果一个线程被堵塞，比如在它等候输入的时候，那么一般都不能象在 Blocking.java 中那样轮询一个标志。但在这些情况下，我们仍然不该使用 stop()，而应换用由 Thread 提供的 interrupt()方法，以便中止并退出堵塞的代码。

```
//: c14:Interrupt.java
// The alternative approach to using
// stop() when a thread is blocked.
// <applet code=Interrupt width=200 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
```

```
class Blocked extends Thread {
  public synchronized void run() {
    try {
      wait(); // Blocks
    } catch(InterruptedException e) {
      System.err.println("Interrupted");
    }
    System.out.println("Exiting run()");
  }
}

public class Interrupt extends JApplet {
  private JButton
    interrupt = new JButton("Interrupt");
  private Blocked blocked = new Blocked();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(interrupt);
    interrupt.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          System.out.println("Button pressed");
          if(blocked == null) return;
          Thread remove = blocked;
          blocked = null; // to release it
          remove.interrupt();
        }
      });
    blocked.start();
  }
  public static void main(String[] args) {
    Console.run(new Interrupt(), 200, 100);
  }
} ///:~
```

Blocked.run()内部的 wait()会产生堵塞的线程。当我们按下按钮以后，blocked（堵塞）的引用就会设为 null，使垃圾收集器能够将其清除，然后调用对象的 interrupt()方法。如果是首次按下按钮，我们会看到线程正常退出。但在没有可供“杀死”的线程以后，看到的便只是按钮被按下而已。

suspend()和 resume()方法天生容易死锁。调用 suspend()时，目标线程会停下来，但却仍然持有在这之前获得的锁定。此时，其他任何线程都不能访问锁定的资源，除非被“挂起”

的线程恢复运行。对任何线程来说，如果它们想恢复目标线程，同时又试图使用任何一个锁定的资源，就会造成令人难堪的死锁。所以我们不应该使用 suspend()和 resume()，而应在自己的 Thread 类中置入一个标志，指出线程应该活动还是挂起。若标志指出线程应该挂起，便用 wait()命其进入等待状态。若标志指出线程应当恢复，则用一个 notify()重新启动线程。我们可以修改前面的 Counter2.java 来实际体验一番。尽管两个版本的效果是差不多的，但大家会注意到代码的组织结构发生了很大的变化——为所有“听众”都使用了匿名的内部类，而且 Thread 是一个内部类。这使得程序的编写稍微方便一些，因为它取消了 Counter2.java 中一些额外的记录工作。

```
//: c14:Suspend.java
// The alternative approach to using suspend()
// and resume(), which are deprecated in Java 2.
// <applet code=Suspend width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Suspend extends JApplet {
  private JTextField t = new JTextField(10);
  private JButton
    suspend = new JButton("Suspend"),
    resume = new JButton("Resume");
  private Suspendable ss = new Suspendable();
  class Suspendable extends Thread {
    private int count = 0;
    private boolean suspended = false;
    public Suspendable() { start(); }
    public void fauxSuspend() {
      suspended = true;
    }
    public synchronized void fauxResume() {
      suspended = false;
      notify();
    }
    public void run() {
      while (true) {
        try {
          sleep(100);
          synchronized(this) {
            while(suspended)
              wait();
          }
```

```
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
        t.setText(Integer.toString(count++));
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    suspend.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          ss.fauxSuspend();
        }
      });
    cp.add(suspend);
    resume.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          ss.fauxResume();
        }
      });
    cp.add(resume);
  }
  public static void main(String[] args) {
    Console.run(new Suspend(), 300, 100);
  }
} ///:~
```

Suspendable 中的 suspended（已挂起）标志用于开关“挂起”或者“暂停”状态。为挂起一个线程，只需调用 fauxSuspend()将标志设为 true（真）即可。对标志状态的侦测是在 run()内进行的。就象本章早些时候提到的那样，wait()必须设为“同步”（synchronized），使其能够使用对象锁。在 fauxResume()中，suspended 标志被设为 false（假），并调用 notify()——由于这会在一个“同步”从句中唤醒 wait()，所以 fauxResume()方法也必须同步，使其能在调用 notify()之前取得对象锁（这样一来，对象锁可由要唤醒的那个 wait()使用）。如果遵照本程序展示的样式，可以避免使用 wait()和 notify()。

Thread 的 destroy()方法根本没有实现；它类似一个根本不能恢复的 suspend()，所以会发生与 suspend()一样的死锁问题。然而，这一方法没有得到明确的“反对”，也许会在 Java 以后的版本（Java 2 之后）实现，用于一些可以承受死锁危险的特殊场合。

大家可能会奇怪当初为什么要实现这些现在又被“反对”的方法。之所以会出现这种情

况，大概是由于 Sun 公司主要让技术人员来决定对语言的改动，而不是让那些市场销售人员。通常，技术人员比搞销售的更能理解语言的实质。当初犯下了错误以后，也能较为理智地正视它们。这意味着 Java 能够继续进步，即便这使 Java 程序员多少感到有些不便。就我自己来说，宁愿面对这些不便之处，也不愿看到语言停滞不前。

14.4 优 先 级

线程的“优先级”（Priority）告诉调试程序该线程的重要程度有多大。如果有大量线程都被堵塞，都在等候运行，调试程序会首先运行具有最高优先级的那个线程。然而，这并不表示优先级较低的线程不会运行（换言之，不会由于优先级的存在而导致死锁）。若线程的优先级较低，只不过表示它被准许运行的机会小一些而已。

尽管优先级表面看起来非常有趣，但在实际应用中，几乎从来用不着由自己设置优先级。所以假如你对它不感兴趣，那么尽管跳过这一节。

14.4.1 读取和设置优先级

可用 getPriority()方法读取一个线程的优先级，并用 setPriority()改变它。前述的“计数器”例子仍然可以拿来展示改变优先级的效果。在下面的小程序中，大家会发现计数器的计数速度慢了下来，因为它们对应的线程被分配了较低的优先级：

```
//: c14:Counter5.java
// Adjusting the priorities of threads.
// <applet code=Counter5 width=450 height=600>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Ticker2 extends Thread {
  private JButton
    b = new JButton("Toggle"),
    incPriority = new JButton("up"),
    decPriority = new JButton("down");
  private JTextField
    t = new JTextField(10),
    pr = new JTextField(3); // Display priority
  private int count = 0;
  private boolean runFlag = true;
  public Ticker2(Container c) {
    b.addActionListener(new ToggleL());
    incPriority.addActionListener(new UpL());
    decPriority.addActionListener(new DownL());
    JPanel p = new JPanel();
    p.add(t);
```

```
    p.add(pr);
    p.add(b);
    p.add(incPriority);
    p.add(decPriority);
    c.add(p);
  }
  class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  class UpL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() + 1;
      if(newPriority > Thread.MAX_PRIORITY)
        newPriority = Thread.MAX_PRIORITY;
      setPriority(newPriority);
    }
  }
  class DownL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() - 1;
      if(newPriority < Thread.MIN_PRIORITY)
        newPriority = Thread.MIN_PRIORITY;
      setPriority(newPriority);
    }
  }
  public void run() {
    while (true) {
      if(runFlag) {
        t.setText(Integer.toString(count++));
        pr.setText(
          Integer.toString(getPriority()));
      }
      yield();
    }
  }
}

public class Counter5 extends JApplet {
  private JButton
    start = new JButton("Start"),
    upMax = new JButton("Inc Max Priority"),
    downMax = new JButton("Dec Max Priority");
```

```
  private boolean started = false;
  private static final int SIZE = 10;
  private Ticker2[] s = new Ticker2[SIZE];
  private JTextField mp = new JTextField(3);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new Ticker2(cp);
    cp.add(new JLabel(
      "MAX_PRIORITY = " + Thread.MAX_PRIORITY));
    cp.add(new JLabel("MIN_PRIORITY = "
      + Thread.MIN_PRIORITY));
    cp.add(new JLabel("Group Max Priority = "));
    cp.add(mp);
    cp.add(start);
    cp.add(upMax);
    cp.add(downMax);
    start.addActionListener(new StartL());
    upMax.addActionListener(new UpMaxL());
    downMax.addActionListener(new DownMaxL());
    showMaxPriority();
    // Recursively display parent thread groups:
    ThreadGroup parent =
      s[0].getThreadGroup().getParent();
    while(parent != null) {
      cp.add(new Label(
        "Parent threadgroup max priority = "
        + parent.getMaxPriority()));
      parent = parent.getParent();
    }
  }
  public void showMaxPriority() {
    mp.setText(Integer.toString(
      s[0].getThreadGroup().getMaxPriority()));
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
```

```
  }
  class UpMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int maxp =
        s[0].getThreadGroup().getMaxPriority();
      if(++maxp > Thread.MAX_PRIORITY)
        maxp = Thread.MAX_PRIORITY;
      s[0].getThreadGroup().setMaxPriority(maxp);
      showMaxPriority();
    }
  }
  class DownMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int maxp =
        s[0].getThreadGroup().getMaxPriority();
      if(--maxp < Thread.MIN_PRIORITY)
        maxp = Thread.MIN_PRIORITY;
      s[0].getThreadGroup().setMaxPriority(maxp);
      showMaxPriority();
    }
  }
  public static void main(String[] args) {
    Console.run(new Counter5(), 450, 600);
  }
} ///:~
```

Ticker2 采用本章前面构造好的形式，但这里新增了一个 JTextField（文本字段），用于显示线程的优先级；同时新增了两个按钮，用于人为地提高及降低优先级。

也要注意 yield()的用法，它将控制权自动返回给调度机制。若不进行这样的处理，多线程机制仍会工作，但我们会发现它的运行速度慢了下来（试试删去对 yield()的调用，体验一下实际效果）。亦可调用 sleep()，但假若那样做，计数频率就会改由 sleep()的持续时间控制，而不是由优先级控制。

Counter5 中的 init()创建了由 10 个 Ticker2 构成的一个数组；它们的按钮以及输入字段（文本字段）由 Ticker2 构造函数置入窗体。Counter5 增加了新的按钮，用于启动一切，以及用于提高和降低线程组的最大优先级。除此以外，还有一些标签用于显示一个线程可以采用的最大及最小优先级；以及一个特殊的文本字段，用于显示线程组的最大优先级（在下一节里，我们将全面讨论线程组的问题）。最后，父线程组的优先级也作为标签显示出来。

按下“up”（上）或“down”（下）按钮的时候，会先取得 Ticker2 当前的优先级，然后相应地提高或者降低。

运行该程序时，我们会注意到几件事情。首先，线程组的默认优先级是 5。即使在启动线程之前（或者在创建线程之前，这要求对代码进行适当的修改）将最大优先级降到 5 以下，每个线程都会有一个 5 的默认优先级。

最简单的测试是取得一个计数器，将它的优先级降低至 1，此时应观察到它的计数频率显著放慢。接着试一试再次提高优先级，可以升高回线程组的优先级，但不能再高了。现在

将线程组的优先级降低两次。线程的优先级不会改变，但假若试图提高或者降低它，就会发现这个优先级自动变成线程组的优先级。此外，新线程仍然具有一个默认优先级，即使它比组的优先级还要高（换句话说，不要指望利用组优先级来防止新线程拥有比现在更高的优先级）。

最后，试着提高整个组的最大优先级。可以发现，这样做是没有效果的。我们只能降低线程组的最大优先级，而不能增大它。

14.4.2 线程组

所有线程都隶属于一个线程组。那可以是一个默认线程组，亦可是一个创建线程时明确指定的组。在创建之初，线程被限制到一个组里，而且不能改变到一个不同的组。每个应用都至少有一个线程从属于系统线程组。若创建多个线程而不指定一个组，它们就会自动归属于系统线程组。

线程组也必须从属于其他线程组。必须在构造函数里指定新线程组从属于哪个线程组。若在创建一个线程组的时候没有指定它的归属，则同样会自动成为系统线程组的一名属下。因此，一个应用程序中的所有线程组最终都会将系统线程组作为自己的“父”。

之所以要提出“线程组”的概念，原因是一、两句话很难说清的。如果真的要说清楚，便需要很长的篇幅，最后只会分散大家的注意力，适得其反。一般地说，我们认为是由于“安全”或者“保密”方面的理由才使用线程组的[67]。按照 Arnold 和 Gosling 的说法：“线程组中的线程可以修改组内的其他线程，包括那些位于分级结构最深处的。一个线程不能修改位于自己所在组或者下属组之外的任何线程”。然而，我们很难判断“修改”在这儿的具体含义是什么。下面这个例子展示了位于一个“叶子组”内的线程能修改它所在线程组树的所有线程的优先级，同时还能为这个“树”内的所有线程都调用一个方法。

```
//: c14:TestAccess.java
// How threads can access other threads
// in a parent thread group.

public class TestAccess {
  public static void main(String[] args) {
    ThreadGroup
      x = new ThreadGroup("x"),
      y = new ThreadGroup(x, "y"),
      z = new ThreadGroup(y, "z");
    Thread
      one = new TestThread1(x, "one"),
      two = new TestThread2(z, "two");
  }
}

class TestThread1 extends Thread {
  private int i;
  TestThread1(ThreadGroup g, String name) {
```

67 《The Java Programming Language》第 179 页。该书由 Arnold 和 Jams Gosling 编著，Addison-Wesley 出版社于 1996 年出版。

```
    super(g, name);
  }
  void f() {
    i++; // modify this thread
    System.out.println(getName() + " f()");
  }
}

class TestThread2 extends TestThread1 {
  TestThread2(ThreadGroup g, String name) {
    super(g, name);
    start();
  }
  public void run() {
    ThreadGroup g =
      getThreadGroup().getParent().getParent();
    g.list();
    Thread[] gAll = new Thread[g.activeCount()];
    g.enumerate(gAll);
    for(int i = 0; i < gAll.length; i++) {
      gAll[i].setPriority(Thread.MIN_PRIORITY);
      ((TestThread1)gAll[i]).f();
    }
    g.list();
  }
} ///:~
```

在 main()中，我们创建了几个 ThreadGroup（线程组），每个都位于不同的“叶”上：x 没有参数，只有它的名字（一个 String），所以会自动放到“system”（系统）线程组里；而 y 位于 x 下方，z 而位于 y 下方。注意初始化是按照文字顺序进行的，所以代码合法。

有两个线程创建之后进入了不同的线程组。其中，TestThread1 没有一个 run()方法，但有一个 f()，用于通知线程以及打印出一些东西，以便我们知道它已被调用。而 TestThread2 属于 TestThread1 的一个子类，它的 run()非常详尽，要做许多事情。首先，它获得当前线程所在的线程组，然后利用 getParent()在继承树中向上移动两级（这样做是有道理的，因为我想把 TestThread2 在分级结构中向下移动两级）。随后，我们调用方法 activeCount()，查询这个线程组以及所有子线程组内有多少个线程，从而创建由指向 Thread 的引用构成的一个数组。enumerate()方法将指向所有这些线程的引用置入数组 gAll 里。然后在整个数组里遍历，为每个线程都调用 f()方法，同时修改优先级。这样一来，位于一个“叶子”线程组里的线程就修改了位于父线程组的线程。

调试方法 list()打印出与一个线程组有关的所有信息，把它们作为标准输出。在我们对线程组的行为进行调查的时候，这样做是相当有好处的。下面是程序的输出：

```
java.lang.ThreadGroup[name=x,maxpri=10]
    Thread[one,5,x]
```

```
    java.lang.ThreadGroup[name=y,maxpri=10]
       java.lang.ThreadGroup[name=z,maxpri=10]
          Thread[two,5,z]
one f()
two f()
java.lang.ThreadGroup[name=x,maxpri=10]
   Thread[one,1,x]
   java.lang.ThreadGroup[name=y,maxpri=10]
      java.lang.ThreadGroup[name=z,maxpri=10]
         Thread[two,1,z]
```

list()不仅打印出 ThreadGroup 或者 Thread 的类名，也打印出了线程组的名字以及它的最高优先级。对于线程，则打印出它们的名字，并接上线程优先级以及所属的线程组。注意 list()会对线程和线程组进行缩排处理，指出它们是未缩排的线程组的“子”。

大家可看到 f()是由 TestThread2 的 run()方法调用的，所以很明显，组内的所有线程都是相当脆弱的。然而，我们只能访问那些从自己的 system 线程组树分支出来的线程，而且或许这就是所谓“安全”的意思——我们不能访问其他任何人的系统的线程组分级树！

线程组的控制

抛开安全问题不谈，线程组最有用的一个地方就是控制：只需用单个命令即可完成对整个线程组的操作。下面这个例子演示了这一点，并对线程组内优先级的限制进行了说明。括号内的注释数字便于大家比较输出结果：

```
//: c14:ThreadGroup1.java
// How thread groups control priorities
// of the threads inside them.

public class ThreadGroup1 {
  public static void main(String[] args) {
    // Get the system thread & print its Info:
    ThreadGroup sys =
      Thread.currentThread().getThreadGroup();
    sys.list(); // (1)
    // Reduce the system thread group priority:
    sys.setMaxPriority(Thread.MAX_PRIORITY - 1);
    // Increase the main thread priority:
    Thread curr = Thread.currentThread();
    curr.setPriority(curr.getPriority() + 1);
    sys.list(); // (2)
    // Attempt to set a new group to the max:
    ThreadGroup g1 = new ThreadGroup("g1");
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    // Attempt to set a new thread to the max:
    Thread t = new Thread(g1, "A");
```

```
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (3)
    // Reduce g1's max priority, then attempt
    // to increase it:
    g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    g1.list(); // (4)
    // Attempt to set a new thread to the max:
    t = new Thread(g1, "B");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (5)
    // Lower the max priority below the default
    // thread priority:
    g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
    // Look at a new thread's priority before
    // and after changing it:
    t = new Thread(g1, "C");
    g1.list(); // (6)
    t.setPriority(t.getPriority() -1);
    g1.list(); // (7)
    // Make g2 a child Threadgroup of g1 and
    // try to increase its priority:
    ThreadGroup g2 = new ThreadGroup(g1, "g2");
    g2.list(); // (8)
    g2.setMaxPriority(Thread.MAX_PRIORITY);
    g2.list(); // (9)
    // Add a bunch of new threads to g2:
    for (int i = 0; i < 5; i++)
      new Thread(g2, Integer.toString(i));
    // Show information about all threadgroups
    // and threads:
    sys.list(); // (10)
    System.out.println("Starting all threads:");
    Thread[] all = new Thread[sys.activeCount()];
    sys.enumerate(all);
    for(int i = 0; i < all.length; i++)
      if(!all[i].isAlive())
        all[i].start();
    // Suspends & Stops all threads in
    // this group and its subgroups:
    System.out.println("All threads started");
    sys.suspend(); // Deprecated in Java 2
    // Never gets here...
    System.out.println("All threads suspended");
```

```
    sys.stop(); // Deprecated in Java 2
    System.out.println("All threads stopped");
  }
} ///:~
```

下面的输出结果已进行了适当的编辑，以便用一页能够装下（“java.lang.”字样已被删去），而且添加了适当的编号，与前面程序列表中括号中的编号对应：

```
(1) ThreadGroup[name=system,maxpri=10]
     Thread[main,5,system]
(2) ThreadGroup[name=system,maxpri=9]
     Thread[main,6,system]
(3) ThreadGroup[name=g1,maxpri=9]
     Thread[A,9,g1]
(4) ThreadGroup[name=g1,maxpri=8]
     Thread[A,9,g1]
(5) ThreadGroup[name=g1,maxpri=8]
     Thread[A,9,g1]
     Thread[B,8,g1]
(6) ThreadGroup[name=g1,maxpri=3]
     Thread[A,9,g1]
     Thread[B,8,g1]
     Thread[C,6,g1]
(7) ThreadGroup[name=g1,maxpri=3]
     Thread[A,9,g1]
     Thread[B,8,g1]
     Thread[C,3,g1]
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
(10)ThreadGroup[name=system,maxpri=9]
     Thread[main,6,system]
     ThreadGroup[name=g1,maxpri=3]
       Thread[A,9,g1]
       Thread[B,8,g1]
       Thread[C,3,g1]
       ThreadGroup[name=g2,maxpri=3]
         Thread[0,6,g2]
         Thread[1,6,g2]
         Thread[2,6,g2]
         Thread[3,6,g2]
         Thread[4,6,g2]
Starting all threads:
All threads started
```

所有程序都至少有一个线程在运行，而且 main()采取的第一项行动便是调用 Thread 的一个 static（静态）方法，名为 currentThread()。从这个线程开始，线程组将被创建，而且会为结果调用 list()。输出如下：

```
(1) ThreadGroup[name=system,maxpri=10]
      Thread[main,5,system]
```

我们可以看到，主线程组的名字是 system，而主线程的名字是 main，而且它从属于 system 线程组。

第二个实验显示出 system 组的最高优先级可以减小，而且 main 线程可以增大自己的优先级：

```
(2) ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
```

第三个实验创建一个新的线程组，名为 g1；它自动从属于 system 线程组，因为并没有明确指定它的归属关系。我们在 g1 内部放置了一个新线程，名为 A。随后，我们试着将这个组的最大优先级设到最高的级别，并将 A 的优先级也设到最高一级。结果如下：

```
(3) ThreadGroup[name=g1,maxpri=9]
      Thread[A,9,g1]
```

可以看出，不可能将线程组的最大优先级设为高于它的父线程组。

第四个实验将 g1 的最大优先级降低两级，然后试着把它升至 Thread.MAX_PRIORITY。结果如下：

```
(4) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
```

同样可以看出，提高最大优先级的企图是失败的。我们只能降低一个线程组的最大优先级，而不能提高它。此外，注意线程 A 的优先级并未改变，而且它现在高于线程组的最大优先级。也就是说，线程组最大优先级的变化并不能对现有线程造成影响。

第五个实验试着将一个新线程设为最大优先级。如下所示：

```
(5) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
      Thread[B,8,g1]
```

因此，新线程不能变到比最大线程组优先级还要高的一级。

这个程序的默认线程优先级是 6；若新建一个线程，那就是它的默认优先级，而且不会发生变化，除非对优先级进行了特别的处理。练习六将把线程组的最大优先级降至默认线程优先级以下，看看在这种情况下新建一个线程会发生什么事情：

```
(6) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,6,g1]
```

尽管线程组现在的最大优先级是 3，但仍然用默认优先级 6 来创建新线程。所以，线程组的最大优先级不会影响默认优先级（事实上，似乎没有办法可以设置新线程的默认优先级）。

改变了优先级后，接下来试试将其降低一级，结果如下：

```
(7) ThreadGroup[name=g1,maxpri=3]
     Thread[A,9,g1]
     Thread[B,8,g1]
     Thread[C,3,g1]
```

因此，只有在试图改变优先级的时候，才会强迫遵守线程组最大优先级的限制。

我们在(8)和(9)中进行了类似的试验。在这里，我们创建了一个新的线程组，名为 g2，将其作为 g1 的一个子组，并改变了它的最大优先级。大家可以看到，g2 的优先级无论如何都不可能高于 g1：

```
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
```

也要注意在 g2 创建的时候，它会被自动设为 g1 的线程组最大优先级。

经过所有这些实验以后，整个线程组和线程系统都会被打印出来，如下所示：

```
(10)ThreadGroup[name=system,maxpri=9]
     Thread[main,6,system]
     ThreadGroup[name=g1,maxpri=3]
       Thread[A,9,g1]
       Thread[B,8,g1]
       Thread[C,3,g1]
       ThreadGroup[name=g2,maxpri=3]
         Thread[0,6,g2]
         Thread[1,6,g2]
         Thread[2,6,g2]
         Thread[3,6,g2]
         Thread[4,6,g2]
```

所以由线程组的规则所限，一个子组的最大优先级在任何时候都只能低于或等于它的父组的最大优先级。

本程序的最后一个部分演示了用于整组线程的方法。程序首先遍历整个线程树，并启动每一个尚未启动的线程。例如，system 组随后会被挂起（暂停），最后被中止（尽管用 suspend()和 stop()对整个线程组进行操作看起来似乎很有趣，但应注意这些方法在 Java 1.2 里都是被“反对”的）。但在挂起 system 组的同时，也挂起了 main 线程，而且整个程序都会关闭。所以永远不会达到让线程中止的那一步。实际上，假如真的中止了 main 线程，它会“掷”出一个 ThreadDeath 违例，所以我们通常不这样做。由于 ThreadGroup 是从 Object 继承的，其中包含了 wait()方法，所以也能调用 wait(秒数×1000)，令程序暂停运行任意秒数的时间。当然，事前必须在一个同步块里取得对象锁。

ThreadGroup 类也提供了 suspend()和 resume()方法，所以能中止和启动整个线程组和它的所有线程，也能中止和启动它的子组，所有这些只需一个命令即可（再次提醒，suspend()和 resume()都是 Java 1.2 所“反对”的）。

从表面看，线程组似乎有些让人摸不着头脑，但请注意我们很少需要直接使用它们。

14.5 回顾 runnable

在本章早些时候，我曾建议大家在将一个小程序或主 Frame 变作 Runnable 的一种实现形式之前，一定要好好地想一想。当然，如果必须从一个类继承，而且想使类具有线程处理能力，那么 Runnable 确实是一种正确的方案。本章最后一个例子对这一点进行了剖析，制作了一个 Runnable JPanel 类，用于为自己涂上不同的颜色。这个程序被设计成从命令行获得参数值，以决定颜色网格有多大，以及颜色发生变化之间的 sleep()时间有多久。通过运用这些值，大家能体验到线程一些有趣、但同时可能令人费解的特性：

```
//: 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 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.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  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) {
        System.err.println("Interrupted");
      }
    }
  }
}

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 是一个标准的小程序兼应用程序，它用一个 init()来设置 GUI。首先，init()会设置一个 GridLayout（网格布局），使每个方向上都有 grid 单元。随后，它添加适当数量的 CBox 对象，用它们填充网格，并为每一个都传递 pause 值。在 main()中，我们可看到如何通过命令行参数的传递，对 pause 和 grid 的默认值进行修改；或者也可以通过小程序参数加以修改。

CBox 是进行正式工作的地方。它是从 JPanel 继承的，并实现了 Runnable 接口，使每个 JPanel 也可以是一个 Thread。请记住，在实现 Runnable 的时候，并没有实际产生一个 Thread 对象，只是产生了一个拥有 run()方法的类。因此，我们必须明确创建一个 Thread 对象，并将 Runnable 对象传递给构造函数，随后调用 start()（在构造函数里进行）。在 CBox 里，这个线程的名字叫作 t。

请留意 colors（颜色）数组，它对 Color 类中的所有颜色进行了遍历。随后，收集到的信息将用于 newColor()，以便产生一种随机选择的颜色。当前的单元（格）颜色是 cColor。

paintComponent()则相当简单——只是将颜色设为 cColor，然后用那种颜色填充整个面板（JPanel）。

在 run()中，我们看到一个无限循环，它将 cColor 设为一种新的随机颜色，然后调用 repaint()把它显示出来。随后，对线程执行 sleep()，使其“休眠”由命令行指定的时间长度。

由于这种设计方案非常灵活，而且线程处理同每个 JPanel 元素都紧密结合在一起，所以在理论上可以生成任意多的线程（但在实际应用中，这要受到 JVM 本身的能力限制。数量多了，它恐怕不能再从容应付）。

这个程序也为我们提供了一个有趣的评测基准，因为它揭示了由于不同 JVM 采用了不同的线程实现，所以在性能上也会产生巨大的差异。

14.5.1 过多的线程

有些时候，我们会发现 ColorBoxes 几乎陷于停顿状态。在我自己的机器上，这一情况在产生了 10×10 的网格之后发生了。为什么会这样呢？自然地，我们有理由怀疑 Swing 对它做了什么事情。所以这里有一个例子能够检验那个猜测，它产生了较少的线程。代码经过了重新组织，使一个 ArrayList 实现了 Runnable，而且那个 ArrayList 容纳了数量众多的色块，并随机挑选一些进行更新。随后，我们大量创建这些 ArrayList 对象——具体数量大致参照我们挑选的网格维数。结果便是我们得到比色块少得多的线程。所以假如出现了速度加快的现象，那么我们就能立即知道，因为前例的线程数量太多了。如下所示：

```
//: c14:ColorBoxes2.java
// Balancing thread use.
// <applet code=ColorBoxes2 width=600 height=500>
// <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 CBox2 extends JPanel {
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  void nextColor() {
    cColor = newColor();
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
}

class CBoxList
  extends ArrayList implements Runnable {
  private Thread t;
  private int pause;
  public CBoxList(int pause) {
    this.pause = pause;
    t = new Thread(this);
  }
  public void go() { t.start(); }
  public void run() {
    while(true) {
      int i = (int)(Math.random() * size());
      ((CBox2)get(i)).nextColor();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {
```

```
        System.err.println("Interrupted");
      }
    }
  }
  public Object last() { return get(size() - 1);}
}

public class ColorBoxes2 extends JApplet {
  private boolean isApplet = true;
  private int grid = 12;
  // Shorter default pause than ColorBoxes:
  private int pause = 50;
  private CBoxList[] v;
  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));
    v = new CBoxList[grid];
    for(int i = 0; i < grid; i++)
      v[i] = new CBoxList(pause);
    for (int i = 0; i < grid * grid; i++) {
      v[i % grid].add(new CBox2());
      cp.add((CBox2)v[i % grid].last());
    }
    for(int i = 0; i < grid; i++)
      v[i].go();
  }
  public static void main(String[] args) {
    ColorBoxes2 applet = new ColorBoxes2();
    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);
  }
```

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

在 ColorBoxes2 中，我们创建了 CBoxList 的一个数组，并对其初始化，使其容下各个 CBoxList 网格。每个网格都知道自己该“休眠”多长的时间。随后为每个 CBoxList 都添加等量的 Cbox2 对象，而且将每个 List 都告诉给 go()，用它来启动自己的线程。

CBox2 类似 CBox——能用一种随机选择的颜色描绘自己。但那就是 CBox2 能够做的全部工作。所有涉及线程的处理都已移至 CBoxList 进行。

CBoxList 也可以拥有继承的 Thread，并有一个类型为 ArrayList 的成员对象。这样设计的好处就是 add()和 get()方法可取得特定的参数以及返回值类型，而非只能取得标准对象（它们的名字也可以变得更短）。然而，这里采用的设计表面上看需要较少的代码。除此以外，它会自动保留一个 ArrayList 的其他所有行为。但是，由于 get()牵涉到大量强制转型和用括号封闭的编码，所以随着代码主体的扩充，最终仍有可能需要编写大量代码。

和以前一样，在我们实现 Runnable 的时候，并没有获得与 Thread 配套提供的所有功能，所以必须创建一个新的 Thread，并将自己传递给它的构造函数，以便正式“启动”——start()——一些东西。大家在 CBoxList 构造函数和 go()里都可以体会到这一点。run()方法简单地选择列表里的一个随机元素编号，并为那个元素调用 nextColor()，令其挑选一种新的随机颜色。

运行这个程序时，大家会发现它确实变得更快，响应也更迅速（例如，在你中断它的时候，它停下来的速度更快了）。而且随着网格尺寸的增大，它也不会经常性地陷入“停顿”状态。因此，线程的处理又多了一项新的考虑因素：必须随时检查自己有没有“太多的线程”（针对特定的程序和运行平台，这可能有各种不同的表现——在这里，ColorBoxes 的速度之所以变慢，是由于只有一个线程，但它负责着所有的绘图操作；如请求过多，反应自然迟顿）。如果线程过多，必须试着使用上面介绍的技术，对程序中的线程数量进行“平衡”。如果在一个多线程的程序中遇到了性能上的问题，那么现在就有诸多因素需要考虑：

(1) 对 sleep、yield()以及／或者 wait()的调用足够多吗？

(2) sleep()的调用时间足够长吗？

(3) 运行的线程数是不是太多？

(4) 试过不同的平台和 JVM 吗？

象这样的一些问题是造成多线程编程成为一种“技术活”的原因之一。

14.6 总　　结

何时使用多线程技术，以及何时避免用它，这是我们要掌握的重要课题。它的主要目的是对大量任务进行有序的管理。通过多个任务的混合使用，可以更有效地利用计算机资源，或者对用户来说显得更方便。资源均衡的经典问题是在 I/O 等候期间如何利用 CPU。至于用户方面的方便性，最经典的问题就是如何在一个长时间的下载过程中监视并灵敏地反应一个“停止”（stop）按钮的按下。

多线程的主要缺点包括：

(1) 等着用共享资源时造成程序运行速度变慢。

(2) 对线程进行管理要求的额外 CPU 开销。

(3) 复杂程度无意义的加大，比如用独立的线程来更新数组内每个元素的愚蠢主意。

(4) 漫长的等待、浪费精力的资源竞争以及死锁等多线程症状。

线程另一个优点是它们用“轻度”执行切换（100 条指令的顺序）取代了“重度”进程场景切换（1000 条指令）。由于一个进程内的所有线程共享相同的内存空间，所以“轻度”

场景切换只改变程序的执行和本地变量。而在“重度”场景切换时，一个进程的改变要求必须完整地交换内存空间。

线程处理看来好象进入了一个全新的领域，似乎要求我们学习一种全新的程序设计语言——或者至少学习一系列新的语言概念。由于大多数计算机操作系统都提供了对线程的支持，所以程序设计语言或者库里也出现了对线程的扩展。不管在什么情况下，涉及线程的程序设计：

(1) 刚开始会让人摸不着头脑，要求改换我们传统的编程思路；

(2) 其他语言对线程的支持看来是类似的。所以一旦掌握了线程的概念，在其他环境也不会有太大的困难。尽管对线程的支持使 Java 语言的复杂程度多少有些增加，但请不要责怪 Java。毕竟，利用线程可以做许多有益的事情。

多个线程可能共享同一个资源（比如一个对象里的内存），这是运用线程时面临的最大的一个麻烦。必须保证多个线程不会同时试图读取和修改那个资源。这要求技巧性地运用 synchronized（同步）关键字。它是一个有用的工具，但必须真正掌握它，因为假若操作不当，极易出现死锁。

除此以外，运用线程时还要注意一个非常特殊的问题。由于根据 Java 的设计，它允许我们根据需要创建任意数量的线程——至少理论上如此（例如，假设为一项工程方面的有限元素分析创建数以百万的线程，这对 Java 来说并非实际）。然而，我们一般都要控制自己创建的线程数量的上限。因为在某些情况下，大量线程会将场面变得一团糟，所以工作都会几乎陷于停顿。临界点并不象对象那样可以达到几千个，而是在 100 以下。一般情况下，我们只创建少数几个关键线程，用它们解决某个特定的问题。这时数量的限制问题不大。但在较常规的一些设计中，这一限制确实会使我们感到束手束脚。

大家要注意线程处理中一个不是十分直观的问题。由于采用了线程“调度”机制，所以通过在 run()的主循环中插入对 sleep()的调用，一般都可以使自己的程序运行得更快一些。这使它对编程技巧的要求非常高，特别是在更长的延迟似乎反而能提高性能的时候。当然，之所以会出现这种情况，是由于在正在运行的线程准备进入“休眠”状态之前，较短的延迟可能造成“sleep()结束”调度机制的中断。这便强迫调度机制将其中止，并于稍后重新启动，以便它能做完自己的事情，再进入休眠状态。必须多想一想，才能意识到事情真正的麻烦程度。

本章遗漏的一样东西是一个播放动画的例子，这是目前小程序最流行的一种应用。然而，在 Java JDK 的演示（Demo）区域，配套提供了解决这个问题的一整套方案（还可播放声音），大家可去 java.sun.com 下载。此外，我们完全有理由相信未来版本的 Java 会提供更好的动画支持——尽管目前的 Web 涌现出了与传统方式完全不同的非 Java、非程序化的许多动画方案。如果想系统学习 Java 动画的工作原理，可参考《Core Java 2》一书，由 Cornell 和 Horstmann 编著，Prentice-Hall 于 1997 年出版；若欲更深入地了解线程处理，请参考《Concurrent Programming in Java——Java 中的并发编程》，由 Doug Lea 编著，Addison-Wiseley 于 1997 年出版；或者《Java Threads——Java 线程》，Oaks 和 Wong 编著，O'Reilly 于 1997 年出版。

14.7 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 从 Thread 继承一个类，并覆盖 run()方法。在 run()内，打印出一条消息，然后调用

sleep()。重复三遍这些操作，然后从 run()返回。在构造函数中放置一条启动消息，并覆盖 finalize()，打印一条关闭消息。创建一个独立的线程类，使它在 run()内调用 System.gc()和 System.runFinalization()，并打印一条消息，表明调用成功。创建这两种类型的几个线程，然后运行它们，看看会发生什么。

(2) 修改 Sharing2.java，在 TwoCounter 的 run()方法内部添加一个 synchronized（同步）块，而不是同步整个 run()方法。

(3) 创建两个 Thread 子类，第一个的 run()方法用于最开始的启动并捕获第二个 Thread 对象的引用，然后调用 wait()。第二个类的 run()应该在几秒之后为第一个线程调用 modifyAll()，使第一个线程能打印一条消息。

(4) 在 Ticker2 的 Counter5.java 中，删除 yield()方法，对结果进行解释。用一个 sleep()代替 yield()，同样对结果进行解释。

(5) 在 ThreadGroup1.java 中，将对 sys.suspend()的调用换成对线程组的一个 wait()调用，令其等候 2 秒钟。为了保证获得正确的结果，必须在一个同步块内取得 sys 的对象锁。

(6) 修改 Daemons.java，使 main()有一个 sleep()，而不是一个 readLine()。实验不同的睡眠时间，看看会有什么发生。

(7) 到第 8 章找到那个 GreenhouseControls.java 例子，它应该由三个文件构成。在 Event.java 中，Event 类建立在对时间进行监视的基础上。修改这个 Event，使其成为一个线程。然后修改其余的设计，使它们能与新的、以线程为基础的 Event 正常协作。

(8) 修改练习(7)，用 JDK 1.3 提供的 java.util.Timer 类来运行系统。

(9) 找到第 13 章的 SineWave.java，根据它创建一个程序（用 Console 类，使其兼具小程序和应用程序的双重身份），生成一条动画形式的正弦曲线。使其在观察窗口上慢慢移动，看起来就象示波器那样。用一个线程来驱动这个动画。动画的播放速度应该可以通过一个 java.swing.JSlider 控件加以调整。

(10) 修改练习(9)，在应用程序内同时创建多个正弦波面板。具体的波形数应该可由 HTML 标记或者命令行参数加以控制。

(11) 修改练习(9)，用 java.swing.Timer 类来驱动动画。请注意它和 java.util.Timer 的差别。

第 15 章　分布式计算

历史上，跨越多台机器的编程都倾向于困难、复杂，而且极易出错。

程序员必须掌握与网络有关的大量细节，有时甚至要对硬件有深刻的认识。一般地，我们需要理解连网协议中不同的“层”（Layer）。而且对于每个连网库来说，其中一般都包含了数量众多的函数，分别涉及信息块的连接、打包和拆包；这些块的来回运输；以及握手等等。这是一项令人痛苦的工作。

但是，分布式计算本身的概念并不是很难的，而且它在 Java 库中得到了非常出色的“抽象”。涉及分布式计算时，我们通常希望：

■从其他地方的机器取得一些信息，然后把移到另一台机器上。这是用基本的网络编程来实现的。

■连接到一个数据库，这个数据库可能通过一个网络生存着。这是用 Java 数据库连接（JDBC）来做到的，它进行了非常完整的抽象，完全剔除了那些麻烦的、与具体平台有关的 SQL 细节（“SQL”是“结构化查询语言”的简称，大多数数据库操作都在使用这种语言）。

■通过一个 Web 服务器提供服务。这是用 Java 的“小服务程序”（Servlet）和“Java 服务器页”（JSP）来进行的。

■针对寄居在远程机器上的 Java 对象，进行“透明”的方法执行。“透明”的意思就是指忽略对象在别处这一事实，当成它就在自己的本机上。这是利用 Java 的“远程访问调用”（RMI）来实现的。

■取得用其他语言编写的代码，在其他软硬件架构上运行。这是利用“通用对象请求代理体系”（CORBA）来完成的，后者由 Java 提供了直接支持。

■将事务逻辑同网络连接的问题隔离开，特别是同本身包括了事务管理和安全机制的数据库连接时。这是用“企业 JavaBean”（EJB）来做到的。EJB 实际并非一种分布式结构，只是用它产生的结果程序往往需要在一个联网的客户机 / 服务器环境中运行。

■从代表一个本地系统的网络中，方便、动态地增删设备。这是用 Java 的 Jini 实现的。

上述每个主题在这一章里都会讲到。请注意对每个主题来说，假如真的要展开讨论，那么每个主题都值得专门写一本书。由于篇幅所限，本章只能提供一个大致性的概括。我们的目的是让大家熟悉这些主题，并不是把你变成能马上熟练运用它们的专家（不过，仅是这儿讲的与网络编程、小服务程序和 JSP 有关的知识，便足够你“忙”上好一阵子了）。

15.1　网 络 编 程

Java 最出色的一个地方就是它的所谓“无痛苦连网”概念。Java 网络库的设计者把各种网络操作设计得象读写一个文件那样简单，只是“文件”存在于远程机器上，而且远程机器可以正确地决定如何对你请求或者发送的信息进行处理。有关连网的基层细节已被尽可能地提取出去，并隐藏在 JVM 以及 Java 的本机安装系统里进行控制。我们使用的编程模型其实是一个文件的模型；事实上，网络连接（一个“套接字”）已被封装到流对象里，所以可象对待其他数据流那样采用相同的方法调用。除此以外，在我们处理另一个连网问题——同时控制多个网络连接——的时候，Java 内建的多线程机制也是十分方便的。

本节将用一系列易懂的例子来解释 Java 的连网支持。

15.1.1 机器的标识

当然，为了分辨来自别处的一台机器，以及为了保证自己连接的是希望的那台机器，必须有一种机制能独一无二地标识出网络内的每台机器。早期网络只解决了如何在本地网络环境中为机器提供唯一的名字。但 Java 面向的是整个 Internet，这就要求用一种机制对来自世界各地的机器进行标识。为达到这个目的，我们采用了 IP（互联网地址）的概念。IP 以两种形式存在着：

(1) 大家最熟悉的 DNS（域名服务）形式。我自己的域名是 bruceeckel.com。所以假定我在自己的域内有一台名为 Opus 的计算机，它的域名全称就可以是 Opus.bruceeckel.com。这正是大家向其他人发送电子函件时采用的名字，而且通常集成到一个万维网（WWW）地址里。

(2) 此外，亦可采用“四点”格式，亦即由点号（.）分隔的四组数字，比如 61.149.32.111。

不管哪种情况，IP 地址在内部都表达成一个由 32 个二进制位（bit）构成的数字[68]，所以 IP 地址的每一组数字都不能超过 255。利用由 java.net 提供的 static InetAddress.getByName()，我们可以让一个特定的 Java 对象表达上述任何一种形式的数字。结果是类型为 InetAddress 的一个对象，可用它构成一个“套接字”（Socket），大家在后面会见到这一点。

作为运用 InetAddress.getByName()一个简单的例子，请假设自己有一家拨号连接因特网服务供应商（ISP），那么会发生什么情况。每次拨号连接的时候，都会分配到一个临时性的 IP 地址。但在连接期间，那个 IP 地址拥有与因特网上其他 IP 地址一样的有效性。如果有人按照你的 IP 地址连接你的机器，他们就有可能使用在你机器上运行的 Web 或者 FTP 服务器程序。当然这也有个前提，那就是对方必须准确地知道你目前分配到的 IP。由于每次拨号连接获得的 IP 都是随机的，怎样才能准确地掌握你的 IP 呢？

下面这个程序利用 InetAddress.getByName()来产生你的 IP 地址。为了让它运行起来，事先必须知道计算机的名字。在 Windows 95/98 操作系统中，请大家依次选择“开始”、“设置”、“控制面板”、“网络”，然后进入“标识”卡片。看到了其中的“计算机名称”吗？它就是你应该在命令行输入的内容。

```
//: c15:WhoAmI.java
// Finds out your network address when
// you're connected to the Internet.
import java.net.*;

public class WhoAmI {
  public static void main(String[] args)
      throws Exception {
    if(args.length != 1) {
      System.err.println(
        "Usage: WhoAmI MachineName");
      System.exit(1);
```

[68] 这意味着最多只能得到 40 亿左右的数字组合，全世界的人很快就会把它用光。但根据目前正在研究的新 IP 编址方案，它将采用 128 bit 的数字，这样得到的唯一性 IP 地址也许在几百年的时间里都用不完。

```
    }
    InetAddress a =
      InetAddress.getByName(args[0]);
    System.out.println(a);
  }
} ///:~
```

就我自己的情况来说，机器的名字叫作“peppy”。所以一旦连通我的 ISP，就可以象下面这样执行程序：

```
java WhoAmI peppy
```

得到的结果象下面这个样子（当然，这个地址可能每次都是不同的）：

```
peppy/199.190.87.75
```

假如我把这个地址告诉一位朋友，他就可以立即登录到我的个人 Web 服务器，只需指定目标地址 http://199.190.87.75 即可（当然，我此时不能断线）。有些时候，这是向其他人发送信息或在自己的 Web 站点正式公开之前进行测试的一种方便手段。

1. 服务器和客户机

网络最基本的精神就是让两台机器连接到一起，并相互“交谈”或者“沟通”。一旦两台机器都发现了对方，就可以展开一次令人愉快的双向对话。但它们怎样才能“发现”对方呢？这就象在游乐园里那样：一台机器不得不停留在一个地方，侦听其他机器说：“嘿，你在哪里呢？”

“停留在一个地方”的机器叫作“服务器”（Server）；到处“找人”的机器则叫作“客户机”（Client）或者“客户”。它们之间的区别只有在客户机试图同服务器连接的时候才显得非常明显。一旦连通，就变成了一种双向通信，谁来扮演服务器或者客户机便显得不那么重要了。

所以服务器的主要任务是侦听建立连接的请求，这是由我们创建的特定服务器对象完成的。而客户机的任务是试着与一台服务器建立连接，这是由我们创建的特定客户机对象完成的。一旦连接建好，那么无论在服务器端还是客户机端，连接只是魔术般地变成了一个 I/O 数据流对象。从这时开始，我们可以象读写一个普通的文件那样对待连接。所以一旦建好连接，我们只需象第 11 章那样使用自己熟悉的 I/O 命令即可。这正是 Java 连网最方便的一个地方。

2. 在没有网络的前提下测试程序

由于多种潜在的原因，我们可能没有一台客户机、服务器以及一个网络来测试自己做好的程序。我们也许是在一个课堂环境中进行练习，或者写出的是一个不十分可靠的网络应用，还不能真正拿到网络上去。IP 的设计者注意到了这个问题，并建立了一个特殊的地址——localhost——来满足非真正网络环境中的测试要求。在 Java 中产生这个地址最一般的做法是：

```
InetAddress addr = InetAddress.getByName(null);
```

如果向 getByName()传递一个 null（空）值，就默认为使用 localhost。我们用 InetAddress 指代一台特定的机器，而且必须在进行进一步的操作之前得到这个 InetAddress（互联网地址）。我们不可以操纵一个 InetAddress 的内容（但可把它打印出来，就象下一个例子要演示的那样）。创建 InetAddress 的唯一途径就是那个类的 static（静态）成员方法 getByName()

（这是最常用的）、getAllByName()或者 getLocalHost()。

为得到本地主机回馈地址，亦可向其直接传递字串"localhost"：

```
InetAddress.getByName("localhost");
```

假如已在你的机器的“hosts”表中配置好了“localhost”，那么也可以使用它的保留 IP 地址（四点形式），就象下面这样：

```
InetAddress.getByName("127.0.0.1");
```

这三种方法得到的结果都是一样的。

3. 端口：机器内独一无二的场所

有些时候，一个 IP 地址并不足以完整标识一个服务器。这是由于在一台物理性的机器中，往往运行着多个服务器（程序）。由 IP 表达的每台机器也包含了“端口”（Port）。我们设置一个客户机或者服务器的时候，必须选择一个无论客户机还是服务器都认可连接的端口。就象我们去拜会某人时，IP 地址是他居住的房子，而端口是他在的那个房间。

注意端口并不是机器上一个物理上存在的场所，而是一种软件抽象（主要是为了表述的方便）。客户程序知道如何通过机器的 IP 地址同它连接，但怎样才能同自己真正需要的那种服务连接呢（一般每个端口都运行着一种服务，一台机器可能提供了多种服务，比如 HTTP 和 FTP 等等）？端口编号在这里扮演了重要的角色，它是必需的一种二级定址措施。也就是说，我们请求一个特定的端口，便相当于请求与那个端口编号关联的服务。“报时”便是服务的一个典型例子。通常，每个服务都同一台特定服务器机器上的一个独一无二的端口编号关联在一起。客户程序必须事先知道自己要求的那项服务的运行端口号。

系统服务保留了使用端口 1 到端口 1024 的权力，所以不应让自己设计的服务占用这些以及其他任何已知正在使用的端口。本书的第一个例子将使用端口 8080（为追忆我第一台机器使用的老式 8 位 Intel 8080 芯片，那是一部运行 CP/M 操作系统的机子）。

15.1.2 套接字

“套接字”或者“插座”（Socket）也是一种软件形式的抽象，用于表达两台机器间一个连接的“终端”。针对一个特定的连接，每台机器上都有一个“套接字”，可以想象它们之间有一条虚拟的“线缆”。线缆的每一端都插入一个“套接字”或者“插座”里。当然，机器之间的物理性硬件以及电缆连接都是完全未知的。抽象的基本宗旨是让我们尽可能不必知道那些细节。

在 Java 中，我们创建一个套接字，用它建立与其他机器的连接。从套接字得到的结果是一个 InputStream 以及 OutputStream（若使用恰当的转换器，则分别是 Reader 和 Writer），以便将连接作为一个 I/O 流对象对待。这有两个基于数据流的套接字类：一个是 ServerSocket，服务器用它“侦听”进入的连接；另一个是 Socket，客户用它初始化一次连接。一旦客户（程序）申请建立一个套接字连接，ServerSocket 就会返回（通过 accept()方法）一个对应的服务器端套接字，以便进行直接通信。从此时起，我们就得到了真正的“套接字－套接字”连接，可以用同样的方式对待连接的两端，因为它们本来就是相同的！此时可以利用 getInputStream()以及 getOutputStream()从每个套接字产生对应的 InputStream 和 OutputStream 对象。这些数据流必须封装到缓冲区内。可按第 11 章介绍的方法对类进行格式化，就象对待其他任何流对象那样。

“ServerSocket”这个名字再一次证明了 Java 库中有时会出现一些“莫名其妙”的命名。通常，大家会认为“ServerSocket”最好叫作“ServerConnector”（服务器连接器），或者其他什么名字，只是不要在其中安插一个“Socket”。另外，大家也许会认为 ServerSocket 和 Socket 都应该从某些通用的基类继承。事实上，这两种类确实包含了几个通用的方法，但它

们还不够资格分派到一个通用的基类。相反，ServerSocket 的主要任务是在那里耐心地等候其他机器同它连接，再返回一个实际的 Socket。这正是“ServerSocket”这个命名不恰当的地方，因为它的目标不是真的成为一个 Socket，而是在其他人同它连接的时候产生一个 Socket 对象。

然而，ServerSocket 确实会在主机上创建一个物理性的“服务器”或者侦听用的套接字。这个套接字会侦听进入的连接，然后利用 accept()方法返回一个“已建立”套接字（本地和远程端点均已定义）。容易混淆的地方是这两个套接字（侦听和已建立）都与相同的服务器套接字关联在一起。侦听套接字只能接收新的连接请求，不能接收实际的数据包。所以尽管 ServerSocket 对于编程并无太大的意义，但它确实是“物理性”的。

创建一个 ServerSocket 时，只需为其赋予一个端口编号。不必把一个 IP 地址分配它，因为它已经在自己代表的那台机器上了。但在创建一个 Socket 时，却必须同时赋予 IP 地址以及要连接的端口编号（另一方面，从 ServerSocket.accept()返回的 Socket 已经包含了所有这些信息）。

一个简单的服务器和客户机程序

这个例子将以最简单的方式运用套接字对服务器和客户机进行操作。服务器的全部工作就是等候建立一个连接，然后用那个连接产生的 Socket 创建一个 InputStream 以及一个 OutputStream。在这之后，它从 InputStream 读入的所有东西都会反馈给 OutputStream，直到接收到行中止（END）为止，最后关闭连接。

客户机连接与服务器的连接，然后创建一个 OutputStream。文本行通过 OutputStream 发送。客户机也会创建一个 InputStream，用它收听服务器说些什么（本例只不过是反馈回来的同样的字句）。

服务器与客户机（程序）都使用同样的端口号，而且客户机利用本地主机地址连接位于同一台机器中的服务器（程序），所以不必在一个物理性的网络里完成测试（在某些配置环境中，可能需要同真正的网络建立连接，否则程序不能工作——尽管实际并不通过那个网络通信）。

下面是服务器程序：

```
//: c15:JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;

public class JabberServer {
  // Choose a port outside of the range 1-1024:
  public static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
    try {
      // Blocks until a connection occurs:
      Socket socket = s.accept();
```

```
      try {
        System.out.println(
          "Connection accepted: "+ socket);
        BufferedReader in =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));
        // Output is automatically flushed
        // by PrintWriter:
        PrintWriter out =
          new PrintWriter(
            new BufferedWriter(
              new OutputStreamWriter(
                socket.getOutputStream())),true);
        while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
          System.out.println("Echoing: " + str);
          out.println(str);
        }
      // Always close the two sockets...
      } finally {
        System.out.println("closing...");
        socket.close();
      }
    } finally {
      s.close();
    }
  }
} ///:~
```

可以看到，ServerSocket 需要的只是一个端口编号，不需要 IP 地址（因为它就在这台机器上运行）。调用 accept()时，方法会暂时陷入停顿状态（堵塞），直到某个客户尝试同它建立连接。换言之，尽管它在那里等候连接，但其他进程仍能正常运行（参考第 14 章）。建好一个连接以后，accept()就会返回一个 Socket 对象，它是那个连接的代表。

清除套接字的责任在这里得到了很艺术的处理。假如 ServerSocket 构造函数失败，则程序简单地退出（注意必须保证 ServerSocket 的构造函数在失败之后不会留下任何打开的网络套接字）。针对这种情况，main()会“掷”出一个 IOException 违例，所以不必使用一个 try 块。若 ServerSocket 构造函数成功执行，则其他所有方法调用都必须到一个 try-finally 代码块里寻求保护，以确保无论块以什么方式留下，ServerSocket 都能正确地关闭。

同样的道理也适用于由 accept()返回的 Socket。若 accept()失败，那么我们必须保证 Socket 不再存在或者含有任何资源，以便不必清除它们。但假若执行成功，则后续的语句必须进入一个 try-finally 块内，以保障在它们失败的情况下，Socket 仍能得到正确的清除。由于套接字使用了重要的非内存资源，所以在这里必须特别谨慎，必须自己动手将它们清除（Java

中没有提供“破坏器”来帮助我们做这件事情）。

无论 ServerSocket 还是由 accept()产生的 Socket 都打印到 System.out 里。这意味着它们的 toString 方法会得到自动调用。这样便产生了：

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

大家不久就会看到它们如何与客户程序做的事情配合。

程序的下一部分看来似乎仅仅是打开文件，以便读取和写入，只是 InputStream 和 OutputStream 是从 Socket 对象创建的。利用两个“转换器”类 InputStreamReader 和 OutputStreamWriter，InputStream 和 OutputStream 对象已经分别转换成为 Reader 和 Writer 对象。当然，你也可以直接使用 Java 1.0 的 InputStream 和 OutputStream 类，但对输出来说，使用 Writer 方式具有明显的优势。这一优势是通过 PrintWriter 表现出来的，它有一个重载的构造函数，能获取第二个参数——一个布尔值标志，指向是否在每一次 println()结束的时候自动刷新输出（但不适用于 print()语句）。每次写入了输出内容后（写进 out），它的缓冲区必须刷新，使信息能正式通过网络传递出去。对目前这个例子来说，刷新显得尤为重要，因为客户和服务器在采取下一步操作之前都要等待一行文本内容的到达。若刷新没有发生，那么信息不会进入网络，除非缓冲区满（溢出），这会为本例带来许多问题。

编写网络应用程序时，需要特别注意自动刷新机制的使用。每次刷新缓冲区时，必须创建和发出一个数据包（数据封）。就目前的情况来说，这正是我们所希望的，因为假如包内包含了还没有发出的文本行，服务器和客户机之间的相互“握手”就会停止。换句话说，一行的末尾就是一条消息的末尾。但在其他许多情况下，消息并不是用行分隔的，所以不如不用自动刷新机制，而用内建的缓冲区判决机制来决定何时发送一个数据包。这样一来，我们可以发出较大的数据包，而且处理进程也能加快。

注意和我们打开的几乎所有数据流一样，它们都要进行缓冲处理。本章末尾有一个练习，清楚展现了假如我们不对数据流进行缓冲，那么会得到什么样的后果（速度会变慢）。

无限 while 循环从 BufferedReader in 内读取文本行，并将信息写入 System.out，然后写入 PrintWriter out。注意 in 和 out 可以是任何数据流，这里只是恰巧连接到一个网络而已。

客户程序发出包含了"END"的行后，程序会中止循环，并关闭 Socket。

下面是客户程序的源码：

```
//: c15:JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;

public class JabberClient {
  public static void main(String[] args)
      throws IOException {
    // Passing null to getByName() produces the
    // special "Local Loopback" IP address, for
    // testing on one machine w/o a network:
    InetAddress addr =
      InetAddress.getByName(null);
```

```
    // Alternatively, you can use
    // the address or name:
    // InetAddress addr =
    //   InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    //   InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket =
      new Socket(addr, JabberServer.PORT);
    // Guard everything in a try-finally to make
    // sure that the socket is closed:
    try {
      System.out.println("socket = " + socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } finally {
      System.out.println("closing...");
      socket.close();
    }
  }
} ///:~
```

在 main()中，大家可看到获得本地主机 IP 地址的 InetAddress 的三种途径：使用 null，使用 localhost，或者直接使用保留地址 127.0.0.1。当然，如果想通过网络同一台远程主机连接，也可以换用那台机器的 IP 地址。打印出 InetAddress addr 后（通过对 toString()方法的自动调用），结果如下：

```
localhost/127.0.0.1
```

通过向 getByName()传递一个 null，它会默认寻找 localhost，并生成特殊的保留地址 127.0.0.1。注意在名为 socket 的套接字创建时，同时使用了 InetAddress 以及端口号。打印

这样的某个 Socket 对象时，为了真正理解它的含义，请记住一次独一无二的因特网连接是用下述四种数据标识的：clientHost（客户主机）、clientPortNumber（客户端口号）、serverHost（服务器主机）以及 serverPortNumber（服务器端口号）。服务程序启动后，会在本地主机（127.0.0.1）上建立为它分配的端口（8080）。一旦客户程序发出请求，机器上下一个可用的端口就会分配给它（这种情况下是 1077），这一行动也在与服务程序相同的机器（127.0.0.1）上进行。现在，为了使数据能在客户及服务程序之间来回传送，每一端都需要知道把数据发到哪里。所以在同一个“已知”服务程序连接的时候，客户会发出一个“返回地址”，使服务器程序知道将自己的数据发到哪儿。我们在服务器端的示范输出中可以体会到这一情况：

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

这意味着服务器刚才已接受了来自 127.0.0.1 这台机器的端口 1077 的连接，同时监听自己的本地端口（8080）。而在客户端：

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

这意味着客户已用自己的本地端口 1077 与 127.0.0.1 机器上的端口 8080 建立了连接。

大家会注意到每次重新启动客户程序的时候，本地端口的编号都会递增。这个编号从 1025（刚好在系统保留的 1-1024 之外）开始，并会一直增加下去，除非我们重启机器。若重新启动机器，端口号仍然会从 1025 开始增值（在 Unix 机器中，一旦超过保留的套按字范围，数字就会再次从最小的可用数字开始）。

创建好 Socket 对象后，将其转换成 BufferedReader 和 PrintWriter 的过程便与在服务器中相同（同样地，两种情况下都要从一个 Socket 开始）。在这里，客户通过发出字串"howdy"，并在后面跟随一个数字，从而初始化通信。注意缓冲区必须再次刷新（这是自动发生的，通过传递给 PrintWriter 构造函数的第二个参数）。若缓冲区没有刷新，那么整个会话（通信）都会被挂起，因为用于初始化的“howdy”永远不会发送出去（缓冲区不够满，不足以造成发送动作的自动进行）。从服务器返回的每一行都会写入 System.out，以验证一切都在正常运转。为中止会话，需要发出一个"END"。若客户程序简单地挂起，那么服务器会“掷”出一个违例。

大家在这里可以看到我们采用了同样的措施来确保由 Socket 代表的网络资源得到正确的清除，这是用一个 try-finally 块实现的。

套接字建立了一个“专用”连接，它会一直持续到明确断开连接为止（专用连接也可能间接性地断开，前提是某一端或者中间的某条链路出现故障而崩溃）。这意味着参与连接的双方都被锁定在通信中，而且无论是否有数据传递，连接都会连续处于开放状态。从表面看，这似乎是一种合理的连网方式。然而，它也为网络带来了额外的开销。本章后面会介绍进行连网的另一种方式。采用那种方式，连接的建立只是暂时的。

15.1.3　服务多个客户

JabberServer 可以正常工作，但每次只能为一个客户程序提供服务。在典型的服务器中，我们希望同时能处理多个客户的请求。解决这个问题的关键就是多线程处理机制。而对于那些本身不支持多线程的语言，达到这个要求无疑是异常困难的。通过第 14 章的学习，大家已经知道 Java 已对多线程的处理进行了尽可能的简化。由于 Java 的线程处理方式非常直接，所以让服务器控制多名客户并不是件难事。

最基本的方法是在服务器（程序）里创建单个 ServerSocket，并调用 accept()来等候一个新连接。一旦 accept()返回，我们就取得结果获得的 Socket，并用它新建一个线程，令其只为那个特定的客户服务。然后再调用 accept()，等候下一次新的连接请求。

对于下面这段服务器代码，大家可发现它与 JabberServer.java 例子非常相似，只是为一个特定的客户提供服务的所有操作都已移入一个独立的线程类中：

```
//: c15:MultiJabberServer.java
// A server that uses multithreading
// to handle any number of clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s)
      throws IOException {
    socket = s;
    in =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Enable auto-flush:
    out =
      new PrintWriter(
        new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())), true);
    // If any of the above calls throw an
    // exception, the caller is responsible for
    // closing the socket. Otherwise the thread
    // will close it.
    start(); // Calls run()
  }
  public void run() {
    try {
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) break;
        System.out.println("Echoing: " + str);
        out.println(str);
      }
      System.out.println("closing...");
    } catch(IOException e) {
      System.err.println("IO Exception");
    } finally {
      try {
        socket.close();
```

```
      } catch(IOException e) {
        System.err.println("Socket not closed");
      }
    }
  }
}

public class MultiJabberServer {
  static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Server Started");
    try {
      while(true) {
        // Blocks until a connection occurs:
        Socket socket = s.accept();
        try {
          new ServeOneJabber(socket);
        } catch(IOException e) {
          // If it fails, close the socket,
          // otherwise the thread will close it:
          socket.close();
        }
      }
    } finally {
      s.close();
    }
  }
} ///:~
```

每次有新客户请求建立一个连接时，ServeOneJabber 线程都会取得由 accept()在 main()中生成的 Socket 对象。然后和往常一样，它创建一个 BufferedReader，并用 Socket 自动刷新 PrintWriter 对象。最后，它调用 Thread 的特殊方法 start()，令其进行线程的初始化，然后调用 run()。这里采取的操作与前例是一样的：从套接字读入某些东西，然后把它原样反馈回去，直到遇到一个特殊的"END"结束标志为止。

同样地，套接字的清除必须进行谨慎的设计。就目前这种情况来说，套接字是在 ServeOneJabber 外部创建的，所以清除工作可以“共享”。若 ServeOneJabber 构造函数失败，那么只需向调用者“掷”出一个违例即可，然后由调用者负责线程的清除。但假如构造函数成功，那么必须由 ServeOneJabber 对象负责线程的清除，这是在它的 run()里进行的。

请注意 MultiJabberServer 是多么的简单！和以前一样，我们创建一个 ServerSocket，并调用 accept()允许一个新连接的建立。但这一次，accept()的返回值（一个 Socket）将传递给用于 ServeOneJabber 的构造函数，由它创建一个新线程，并对那个连接进行控制。连接中断后，线程便可简单地消失。

假如 ServerSocket 创建失败，则再一次通过 main()掷出违例。如果成功，则位于外层的 try-finally 代码块可以担保正确的清除。位于内层的 try-catch 块只负责防范 ServeOneJabber 构造函数的失败；若构造函数成功，则 ServeOneJabber 线程会将对应的套接字关掉。

为了证实服务器代码确实能为多名客户提供服务，下面这个程序将创建许多客户（使用线程），并同相同的服务器建立连接。每个线程的“存在时间”都是有限的。一旦到期，就留出空间以便创建一个新线程。允许创建的线程的最大数量是由 final int MAX_THREADS 决定的。大家会注意到这个值非常关键，因为假如把它设得很大，线程便有可能耗尽资源，并产生不可预知的程序错误。

```
//: c15:MultiJabberClient.java
// Client that tests the MultiJabberServer
// by starting up multiple clients.
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  private static int counter = 0;
  private int id = counter++;
  private static int threadcount = 0;
  public static int threadCount() {
    return threadcount;
  }
  public JabberClientThread(InetAddress addr) {
    System.out.println("Making client " + id);
    threadcount++;
    try {
      socket =
        new Socket(addr, MultiJabberServer.PORT);
    } catch(IOException e) {
      System.err.println("Socket failed");
      // If the creation of the socket fails,
      // nothing needs to be cleaned up.
    }
    try {
      in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Enable auto-flush:
      out =
        new PrintWriter(
```

```
        new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())), true);
      start();
    } catch(IOException e) {
      // The socket should be closed on any
      // failures other than the socket
      // constructor:
      try {
        socket.close();
      } catch(IOException e2) {
        System.err.println("Socket not closed");
      }
    }
    // Otherwise the socket will be closed by
    // the run() method of the thread.
  }
  public void run() {
    try {
      for(int i = 0; i < 25; i++) {
        out.println("Client " + id + ": " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } catch(IOException e) {
      System.err.println("IO Exception");
    } finally {
      // Always close it:
      try {
        socket.close();
      } catch(IOException e) {
        System.err.println("Socket not closed");
      }
      threadcount--; // Ending this thread
    }
  }
}

public class MultiJabberClient {
  static final int MAX_THREADS = 40;
  public static void main(String[] args)
      throws IOException, InterruptedException {
    InetAddress addr =
```

```
      InetAddress.getByName(null);
    while(true) {
      if(JabberClientThread.threadCount()
         < MAX_THREADS)
        new JabberClientThread(addr);
      Thread.currentThread().sleep(100);
    }
  }
} ///:~
```

JabberClientThread 构造函数获取一个 InetAddress，并用它打开一个套接字。大家可能已看出了这样的一个套路：Socket 肯定用于创建某种 Reader 以及/或者 Writer（或者 InputStream 和/或 OutputStream）对象，这是运用 Socket 的唯一方式（当然，我们可考虑编写一、两个类，令其自动完成这些操作，避免大量重复的代码编写工作）。同样地，start()执行线程的初始化，并调用 run()。在这里，消息发送给服务器，而来自服务器的信息则在屏幕上回显出来。然而，线程的“存在时间”是有限的，最终都会结束。注意在套接字创建好以后，但在构造函数完成之前，假若构造函数失败，套接字会被清除。否则，为套接字调用 close()的责任便落到了 run()方法的头上。

threadcount 跟踪计算目前存在的 JabberClientThread 对象的数量。它将作为构造函数的一部分增值，并在 run()退出时减值（run()退出意味着线程中止）。在 MultiJabberClient.main()中，大家可以看到线程的数量会得到检查。若数量太多，则多余的暂不创建。方法随后进入“休眠”状态。这样一来，部分线程最后肯定会中止，从而可以腾出地方创建更多的。大家可试验一下逐渐增大 MAX_THREADS 值，看看对于你使用的系统来说，建立多少线程（连接）才会使您的系统资源降低至危险程度。

15.1.4 数据报

大家迄今看到的所有例子采用的都是“传输控制协议”（TCP），亦称作“基于数据流的套接字”。根据该协议的设计宗旨，它应该具有高度的可靠性，而且能保证数据顺利抵达目的地。换言之，它允许重传那些由于各种原因半路“走失”的数据。而且收到字节的顺序与它们发出来时是一样的。当然，这种控制与可靠性需要我们付出一些代价：TCP 有着非常高的开销！

还有另一种协议，名为“用户数据报协议”（UDP），它并不刻意追求数据包会完全发送出去，也不能担保它们抵达的顺序与它们发出时一样。我们认为这是一种“不可靠协议”（TCP 当然是“可靠协议”）。听起来似乎很糟，但由于它的速度快得多，所以经常还是有用武之地的。对某些应用来说，比如声音信号的传输，如果少量数据包在半路上丢失了，那么用不着太在意，因为传输的速度显得更重要一些。大多数联网游戏，比如 Diablo2 和 Quake3（参见 www.myquake3.com），采用的也是 UDP 协议进行通信，因为网络通信的快慢是实时游戏是否流畅进行的决定性因素。也可以想想一台用来报时的服务器，假如某一次报时信息丢失了，那么也真的不必过份紧张。另外，有些应用也许能向服务器传回一条 UDP 消息，以便以后能够恢复。如果在适当的时间里没有响应，消息就会丢失。

通常，我们大多数的直接网络编程都用 TCP 进行，只有在极少情况下才需要 UDP。在本书的第一版里，对 UDP 进行了更为完整的介绍，甚至还有一个例子供大家参考（详见本书配套光盘，或从 www.BruceEckel.com 下载那本书的电子版）。

15.1.5　从一个小程序里使用 URL

对一个小程序来说，通过它正在其中运行的那个 Web 浏览器，完全可以将任何。URL 显示出来。你只需用下面这一行代码即可：

```
getAppletContext().showDocument(u);
```

其中，u 代表 URL 对象。这是将我们重新定向至另一个 Web 页的一个简单例子。尽管只是重定向到了一个 HTML 页，但实际上也可以重定向至一个 CGI 程序的输出。

```
//: c15:ShowHTML.java
// <applet code=ShowHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class ShowHTML extends JApplet {
  JButton send = new JButton("Go");
  JLabel l = new JLabel();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    send.addActionListener(new Al());
    cp.add(send);
    cp.add(l);
  }
  class Al implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      try {
        // This could be a CGI program instead of
        // an HTML page.
        URL u = new URL(getDocumentBase(),
          "FetcherFrame.html");
        // Display the output of the URL using
        // the Web browser, as an ordinary page:
        getAppletContext().showDocument(u);
      } catch(Exception e) {
        l.setText(e.toString());
      }
    }
  }
  public static void main(String[] args) {
```

```
    Console.run(new ShowHTML(), 100, 50);
  }
} ///:~
```

URL 类的最大的特点就是有效地保护了我们的安全。用它直接同一个 Web 服务器建立连接，毋需知道幕后的任何东西！

从服务器读取一个文件

让我们对上述程序稍加改动，以便读取服务器上的一个文件。在这种情况下，目标文件要由客户机来指定：

```
//: c15:Fetcher.java
// <applet code=Fetcher width=500 height=300>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Fetcher extends JApplet {
  JButton fetchIt= new JButton("Fetch the Data");
  JTextField f =
    new JTextField("Fetcher.java", 20);
  JTextArea t = new JTextArea(10,40);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    fetchIt.addActionListener(new FetchL());
    cp.add(new JScrollPane(t));
    cp.add(f); cp.add(fetchIt);
  }
  public class FetchL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      try {
        URL url = new URL(getDocumentBase(),
          f.getText());
        t.setText(url + "\n");
        InputStream is = url.openStream();
        BufferedReader in = new BufferedReader(
          new InputStreamReader(is));
        String line;
        while ((line = in.readLine()) != null)
```

```
          t.append(line + "\n");
        } catch(Exception ex) {
          t.append(ex.toString());
        }
      }
    }
    public static void main(String[] args) {
      Console.run(new Fetcher(), 500, 300);
    }
} ///:~
```

URL 对象的创建和前例十分相似。和平常一样，getDocumentBase()是我们的起点，但这一次，文件名是从 JTextField 里读取的。一旦 URL 对象创建好之后，它的字串版本就会放到 JTextArea 里，以便我们看到它的样子。随后，我们从 URL 获得一个 InputStream。在这种情况下，它只是简单地产生由文件内的字符构成的一个流。转换成一个 Reader，并进行了缓冲处理之后，每一行都会读入，并追加到 JTextArea 里。要注意的是，JTextArea 已放置到一个 JScrollPane 里，所以你不用担心滚动显示的问题——已经有人帮你做了。

15.1.6　更多的连网问题

实际上还有更多的连网问题需要我们解决。Java 连网机制还为 URL 提供了相当丰富的支持，其中包括协议控制器，用于处理在一个网站上发现的不同类型的内容。在《Java Network Programming》这本书里，大家可找到对其他 Java 连网特性的全面解释。该书由 Elliotte Rusty Harold 编写，O'Reilly 出版社于 1997 年出版。

15.2　Java 数据库连接（JDBC）

据估计，将近一半的软件开发都要涉及客户（机）/ 服务器方面的操作。Java 向程序员作出的一项承诺就是构建与平台无关的客户机 / 服务器数据库应用——这是通过“Java 数据库连接”（Java DataBase Connectivity，JDBC）实现的。

数据库最主要的一个问题就是各家公司之间的规格大战。确实存在一种“标准”数据库语言，即“结构查询语言”（SQL-92），但通常都必须确切知道自己要和哪家数据库公司打交道，否则极易出问题，尽管存在所谓的“标准”。JDBC 是面向“与平台无关”设计的，所以在编程的时候不必关心自己要使用的是什么数据库产品。然而，从 JDBC 里仍有可能发出对某些数据库公司专用功能的调用，所以仍然不可任性妄为。

作为程序员，我们可能要用到 SQL 类型名称的一个地方是 SQL 的“TABLE CREATE”语句——以便新建一张数据库表，并为每一列都定义 SQL 类型。不过不幸的是，不同数据库产品支持的 SQL 类型也有显著的区别。即便两个 SQL 类型具有相同的含义，并采用了相同的结构，但不同的数据库也可能为其采用不同的类型名称。大多数主流数据库都支持一种 SQL 数据类型，以便存放大的二进制值：在 Oracle 中，这个类型叫作 LONG RAW；Sybase 把它叫作 IMAGE；Informix 把它叫作 BYTE；而 DB2 把它叫作 LONGVARCHAR FOR BIT DATA。因此，假如你想设计一个便于移植的数据库，就只应该使用标准的 SQL 类型标识符。

对一本书的作者来说，移植性也是一个必须考虑的问题，因为读者也许会用各种各样难以预料的数据仓库软件来测试其中的例子。我尽自己的最大努力保证这些例子的可移植能力。大家还应注意到，书中牵涉到具体数据库的代码都被隔离出来。我把它们集中到一块儿，

便于你针对自己的环境，统一进行必要的修改。

和 Java 中的其他许多 API 一样，JDBC 的设计宗旨也是简化你的工作。进行方法调用时，你会发现整个过程完全符合你从数据库里收集数据时的“习惯思维”——先连接到数据库，创建一条语句并执行查询，然后对结果集进行检索。

为实现这一“与平台无关”的特性，JDBC 为我们提供了一个“驱动程序管理器”，它能动态维护数据库查询所需的所有驱动程序对象。所以假如要连接由三家公司开发的不同种类的数据库，就需要三个单独的驱动程序对象。驱动程序对象会在装载时由“驱动程序管理器”自动注册，并可用 Class.forName()强行装载。

要想打开一个数据库，必须先创建一个“数据库 URL”，它要指定下述三方面的内容：

(1) 用“jdbc”指出要使用 JDBC。

(2) “subprotocol”（子协议）：驱动程序的名字或者一种数据库连接机制的名称。由于 JDBC 的设计从 ODBC 吸取了许多灵感，所以可选用的第一种子协议就是“jdbc-odbc bridge”，它用“odbc”关键字即可指定。

(3) 数据库标识符：随使用的数据库驱动程序的不同而变化，但一般都提供了一个比较符合逻辑的名称，由数据库管理软件映射（对应）到保存了数据表的一个物理目录。要想使自己的数据库标识符真正有用，必须用自己的数据库管理软件为自己喜欢的名字注册（注册的具体过程又随运行平台的不同而变化）。

所有这些信息都统一合并到一个字串里，即“数据库 URL”。举个例子来说，要想通过 ODBC 子协议同一个标定为“people”的数据库连接，数据库 URL 可设为：

```
String dbUrl = "jdbc:odbc:people";
```

如通过一个网络连接，数据库 URL 还需要包含对远程机器进行标识的连接信息，这样便显得有点儿麻烦了。下面是一个 CloudScape 数据库的例子，我们利用 RMI 从一个远程客户机那里调用它：

```
jdbc:rmi://192.168.170.27:1099/jdbc:cloudscape:db
```

数据库 URL 实际是将两个 jdbc 调用合并成一个。其中，第一部分是“jdbc:rmi://192.168.170.27:1099/”，它通过 RMI 连接到正在监听端口 1099 的远程数据库引擎，该引擎的地址是 192.168.170.27。这个 URL 的第二部分是“jdbc:cloudscape:db”，它用子协议和数据库名称传递更多的常见设置。但在这种情况下，只有在第一部分通过 RMI 建立了同远程机器的连接之后，才能开始这样的传递。

准备好建立同数据库的连接后，可调用静态方法 DriverManager.getConnection()，并向它传递数据库 URL、用户名以及一个密码，以便进入数据库。结果，我们会得到一个 Connection 对象，随后便可用它查询和操作数据库。

下面这个例子将打开一个联络信息数据库，并根据命令行提供的参数查询一个人的姓（Last Name）。它只选择那些有 E-mail 地址的人的名字，然后列印出符合查询条件的所有人：

```
//: c15:jdbc:Lookup.java
// Looks up email addresses in a
// local database using JDBC.
import java.sql.*;

public class Lookup {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException {
```

```
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    // Load the driver (registers itself)
    Class.forName(
      "sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c = DriverManager.getConnection(
      dbUrl, user, password);
    Statement s = c.createStatement();
    // SQL code:
    ResultSet r =
      s.executeQuery(
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE " +
        "(LAST='" + args[0] + "') " +
        " AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    while(r.next()) {
      // Capitalization doesn't matter:
      System.out.println(
        r.getString("Last") + ", "
        + r.getString("fIRST")
        + ": " + r.getString("EMAIL") );
    }
    s.close(); // Also closes ResultSet
  }
} ///:~
```

可以看到，数据库 URL 的创建过程与我们前面讲述的完全一样。在该例中，数据库未设密码保护，所以用户名和密码都是空串。

用 DriverManager.getConnection()建好连接后，接下来可根据结果生成的 Connection 对象创建一个 Statement（语句）对象，这是用 createStatement()方法实现的。拿到结果 Statement 后，我们可调用 executeQuery()，向其传递包含了 SQL-92 标准 SQL 语句的一个字串（不久就会看到如何自动创建这类语句，所以没必要在这里知道关于 SQL 更多的东西）。

executeQuery()方法会返回一个 ResultSet（结果集）对象，它与“迭代器”（Iterator）非常相似：next()方法将迭代器移至语句中的下一条记录；如果已抵达结果集的末尾，则返回 null。我们肯定能从 executeQuery()返回一个 ResultSet 对象，即使查询结果是个空集（也就是说，不会产生一个违例）。注意在试图读取任何记录数据之前，都必须调用一次 next()。若结果集为空，那么对 next()的这个首次调用就会返回 false。对于结果集中的每条记录，都可将字段名作为字串使用（当然还有其他方法），从而选择不同的字段。另外要注意的是字段名的大小写是无关紧要的——SQL 数据库不在乎这个问题。为决定返回的类型，可调用 getString()，getFloat()等等。到这个时候，我们已用 Java 的“正宗”格式得到了自己的数据库数据，接下去便可用标准 Java 代码做自己想做的任何事情了。

15.2.1 让例子工作起来

就 JDBC 来说，代码本身是很容易理解的。最令人迷惑的部分是如何使它在自己特定的系统上运行起来。之所以会感到迷惑，是由于它要求我们掌握如何才能使 JDBC 驱动程序正确装载，以及如何用我们的数据库管理软件来设置一个数据库。

当然，具体的操作过程在不同的机器上也会有所区别。但这儿提供的在 32 位 Windows 环境下的操作过程可有效帮助大家理解在其他平台上的操作。

1. 步骤 1：寻找 JDBC 驱动程序

上述程序包含了下面这条语句：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

这似乎暗示着一个目录结构，但大家不要被它蒙骗了。在我手上这个 JDK 1.1 安装版本中，根本不存在叫作 JdbcOdbcDriver.class 的一个文件。所以假如在看了这个例子后去寻找它，那么必然会徒劳而返。另一些人提供的例子使用的是一个假名字，如“myDriver.ClassName”，但人们从名字上得不到任何帮助。事实上，上述用于装载 jdbc-odbc 驱动程序（实际是与 JDK 1.1 配套提供的唯一驱动）的语句只在联机文档的少数几个地方才能看到（特别是在一个标题为“JDBC-ODBC Bridge Driver”的页内）。若上面的装载语句不能工作，那么它的名字可能已随着 Java 新版本的发布而改变了；此时应到联机文档里寻找新的表述方式。

若装载语句出错，会在这个时候得到一个违例。为了检验驱动程序装载语句是不是能正常工作，请将该语句后面直到 catch 从句之间的代码暂时设为注释。如果程序运行时未出现违例，表明驱动程序的装载是正确的。

2. 步骤 2：配置数据库

同样地，我们只限于在 32 位 Windows 环境中工作；你可能需要研究一下自己的操作系统，找出适合自己平台的配置方法。

首先打开控制面板。其中可能有两个图标都含有“ODBC”字样，必须选择那个“32 位 ODBC”，因为另一个是为了保持与 16 位软件的向后兼容而设置的，和 JDBC 混用没有任何作用。双击“32 位 ODBC”图标后，看到的应该是一个卡片式对话框，上面一排有多个卡片标签，其中包括“用户 DSN”、“系统 DSN”、“文件 DSN”等等。其中，“DSN”代表“数据源名称”（Data Source Name）。它们都与 JDBC-ODBC 桥有关，但设置数据库时唯一重要的地方“系统 DSN”。尽管如此，由于需要测试自己的配置以及创建查询，所以也需要在“文件 DSN”中设置自己的数据库。这样便可让 Microsoft Query 工具（与 Microsoft Office 配套提供）正确地找到数据库。注意一些软件公司也设计了自己的查询工具。

最有趣的数据库是我们已经使用过的那一个。标准 ODBC 支持多种文件格式，其中包括由不同公司专用的一些格式，如 dBASE。然而，它也包括了简单的“逗号分隔 ASCII 列表”格式，它几乎是每种数据工具都能生成的。就目前的例子来说，我只选择自己的“people”数据库。这是我多年来一直在维护的一个数据库，中间使用了各种联络管理工具。我把它导出成为一个逗号分隔的 ASCII 文件（一般有个.csv 扩展名，用 Outlook Express 导出通信簿时亦可选用同样的文件格式）。在“文件 DSN”区域，我按下“添加”按钮，选择用于控制逗号分隔 ASCII 文件的文本驱动程序（Microsoft Text Driver），然后撤消对“使用当前目录”的选择，以便导出数据文件时可以自行指定目录。

大家会注意到在进行这些工作的时候，并没有实际指定一个文件，只是一个目录。那是因为数据库通常是由某个目录下的一系列文件构成的（尽管也可能采用其他形式）。每个文

件一般都包含了单个“数据表”，而且 SQL 语句可以产生从数据库中多个表摘取出来的结果（这叫作“联合”，或者 join）只包含了单张表的数据库（就象目前这个）通常叫作“平面文件数据库”。对于大多数问题，如果已经超过了简单的数据存储与获取力所能及的范围，那么必须使用多个数据表。通过“联合”，从而获得希望的结果。我们把这些叫作“关系型”数据库。

3. 步骤 3：测试配置

为了对配置进行测试，需用一种方式核实数据库是否可由查询它的一个程序“见到”。当然，可简单地运行上述的 JDBC 示范程序，并加入下述语句：

```
Connection c = DriverManager.getConnection(
  dbUrl, user, password);
```

若产生一个违例，表明你的配置有误。

然而，此时很有必要使用一个自动化的查询生成工具。我使用的是与 Microsoft Office 配套提供的 Microsoft Query，但你完全可以自己选择一个。查询工具必须知道数据库在什么地方，而 Microsoft Query 要求我进入 ODBC Administrator 的“文件 DSN”卡片，并在那里新添一个条目。同样指定文本驱动程序以及保存数据库的目录。虽然可将这个条目命名为自己喜欢的任何东西，但最好还是使用与“系统 DSN”中相同的名字。

做完这些工作后，再用查询工具创建一个新查询时，便会发现自己的数据库可以使用了。

4. 步骤 4：建立自己的 SQL 查询

我用 Microsoft Query 创建的查询不仅指出目标数据库存在且次序良好，也会自动生成 SQL 代码，以便将其插入我自己的 Java 程序。我希望这个查询能够检查记录中是否存在与启动 Java 程序时在命令行键入的相同的“姓”（Last Name）。所以作为一个起点，我搜索自己的姓“Eckel”。另外，我希望只显示出有对应 E-mail 地址的那些名字。创建这个查询的步骤如下：

(1) 启动一个新查询，并使用查询向导（Query Wizard）。选择“people”数据库（等价于用适应的数据库 URL 打开数据库连接）。

(2) 选择数据库中的“people”表。从这张数据表中，选择 FIRST、LAST 和 EMAIL 数据列。

(3) 在“Filter Data”（过滤器数据）下，选择 LAST，并选择“equals”（等于），加上参数 Eckel。点选“And”单选钮。

(4) 选择 EMAIL，并选中“Is not Null”（不为空）。

(5) 在“Sort By”下，选择 FIRST。

查询结果会向我们展示出是否能得到自己希望的东西。

现在可以按下 SQL 按钮。不需要我们任何方面的介入，正确的 SQL 代码会立即弹现出来，以便我们粘贴和复制。对于这个查询，相应的 SQL 代码如下：

```
SELECT people.FIRST, people.LAST, people.EMAIL
FROM people.csv people
WHERE (people.LAST='Eckel') AND
(people.EMAIL Is Not Null)
ORDER BY people.FIRST
```

若查询比较复杂，手工编码极易出错。但利用一个查询工具，就可交互式地测试自己的

查询，并自动获得正确的代码。事实上，亲手为这些事情编码是令人难以接受的。

5. 步骤 5：在自己的查询中修改和粘贴

我们注意到上述代码与程序中使用的代码是有所区别的。那是由于查询工具对所有名字都进行了限定，即便涉及的仅有一个数据表（若真的涉及多个数据表，这种限定可避免来自不同表的同名数据列发生冲突）。由于这个查询只需要用到一个数据表，所以可考虑从大多数名字中删除“people”限定符，就象下面这样：

```
SELECT FIRST, LAST, EMAIL
FROM people.csv people
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

此外，我们不希望“硬编码”这个程序，以免被限死只能查找一个固定的名字。相反，它应该能查找我们在命令行动态指定的一个名字。所以还要进行必要的修改，并将 SQL 语句转换成一个动态生成的字串。如下所示：

```
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE " +
"(LAST='" + args[0] + "') " +
" AND (EMAIL Is Not Null) " +
"ORDER BY FIRST");
```

SQL 还有一种方式可将名字插入一个查询，名为“存储进程”（Stored Procedures），它的速度非常快。但对于我们的大多数实验性数据库操作，以及一些初级应用，用 Java 构建查询字串已经很不错了。

从这个例子可以看出，利用目前找得到的工具——特别是查询构建工具——涉及 SQL 及 JDBC 的数据库编程是非常简单和直观的。

15.2.2 改编成 GUI 版本

最理想的结果就是让刚才那个检索程序一直运行，要查找什么东西时只需简单地切换到它，键入要查找的名字即可。通过下列代码，我们的程序有了一个 GUI 版本，它被创建成兼具小程序和应用程序两种身份。同时，新增了名字自动完成功能——不必键入完整的姓，即可看到数据：

```
//: c15:jdbc:VLookup.java
// GUI version of Lookup.java.
// <applet code=VLookup
// width=500 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.event.*;
import java.sql.*;
import com.bruceeckel.swing.*;

public class VLookup extends JApplet {
  String dbUrl = "jdbc:odbc:people";
  String user = "";
  String password = "";
  Statement s;
  JTextField searchFor = new JTextField(20);
  JLabel completion =
    new JLabel("                    ");
  JTextArea results = new JTextArea(40, 20);
  public void init() {
    searchFor.getDocument().addDocumentListener(
      new SearchL());
    JPanel p = new JPanel();
    p.add(new Label("Last name to search for:"));
    p.add(searchFor);
    p.add(completion);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    cp.add(results, BorderLayout.CENTER);
    try {
      // Load the driver (registers itself)
      Class.forName(
        "sun.jdbc.odbc.JdbcOdbcDriver");
      Connection c = DriverManager.getConnection(
        dbUrl, user, password);
      s = c.createStatement();
    } catch(Exception e) {
      results.setText(e.toString());
    }
  }
  class SearchL implements DocumentListener {
    public void changedUpdate(DocumentEvent e){}
    public void insertUpdate(DocumentEvent e){
      textValueChanged();
    }
    public void removeUpdate(DocumentEvent e){
      textValueChanged();
    }
  }
  public void textValueChanged() {
```

```
    ResultSet r;
    if(searchFor.getText().length() == 0) {
      completion.setText("");
      results.setText("");
      return;
    }
    try {
      // Name completion:
      r = s.executeQuery(
        "SELECT LAST FROM people.csv people " +
        "WHERE (LAST Like '" +
        searchFor.getText()  +
        "%') ORDER BY LAST");
      if(r.next())
        completion.setText(
          r.getString("last"));
      r = s.executeQuery(
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE (LAST='" +
        completion.getText() +
        "') AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    } catch(Exception e) {
      results.setText(
        searchFor.getText() + "\n");
      results.append(e.toString());
      return;
    }
    results.setText("");
    try {
      while(r.next()) {
        results.append(
          r.getString("Last") + ", "
          + r.getString("fIRST") +
          ": " + r.getString("EMAIL") + "\n");
      }
    } catch(Exception e) {
      results.setText(e.toString());
    }
  }
  public static void main(String[] args) {
    Console.run(new VLookup(), 500, 200);
  }
```

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

数据库的许多逻辑都是相同的，但大家可看到这里添加了一个 DocumentListener，用于监视在 JTextField（文本字段）的输入。所以只要键入一个新字符，它首先就会试着查找数据库中的“姓”，并显示出与当前输入相符的第一条记录（将其置入 completion JLabel，并用它作为要查找的文本）。因此，只要我们键入了足够的字符，使程序能找到与之相符的唯一一条记录，就可以停手了。

15.2.3 JDBC API 为何如此复杂

阅览 JDBC 的联机帮助文档时，我们往往会产生畏难情绪。特别是 DatabaseMetaData 接口——与 Java 中看到的大多数接口相反，它的体积显得非常庞大。它的作用是帮你查询各种数据库产品的信息。其中有数量众多的方法，比如 dataDefinitionCausesTransactionCommit()、getMaxColumnNameLength()、getMaxStatementLength()、storesMixedCaseQuotedIdentifiers()、supportsANSI92IntermediateSQL()、supportsLimitedOuterJoins()等等。我们有必要使用这些查询方法吗？

正如早先指出的那样，数据库起初一直处于一种混乱状态。这主要是由于各种数据库应用提出的要求造成的，所以数据库工具显得非常“强大”——换言之，“庞大”。只是近几年才涌现出了 SQL 的通用语言（常用的还有其他许多数据库语言）。但即便象 SQL 这样的“标准”，也存在无数的变种，所以 JDBC 必须提供一个巨大的 DatabaseMetaData 接口，使我们的代码能真正利用当前要连接的一种“标准”SQL 数据库的能力。简言之，我们可编写出简单的、能移植的 SQL。但如果想优化代码的执行速度，那么为了适应不同数据库类型的特点，我们的编写代码的麻烦就大了。

当然，这并不是 Java 的缺陷。数据库产品之间的差异是我们和 JDBC 都要面对的一个现实。但是，如果能编写通用的查询，而不必太关心性能，那么事情就要简单得多。即使必须对性能作一番调整，只要知道最终面向的平台，也不必针对每一种情况都编写不同的产品查询代码。

15.2.4 一个更复杂的例子

这里再向大家介绍一个更复杂、但也更有趣的例子[69]，它同时牵涉到服务器上一个数据库里的多个数据表。在这里，数据库的宗旨是记录各种社区活动，并允许登记参加这些活动，所以我们把它叫作“社区公益数据库”（Community Interests Database，CID）。不过要注意的是，这个例子只想让大家对数据库及其实现有一个整体性的认识，不打算成为更深层次的数据库开发教程。有大量参考书、学习班和软件都可以帮助你熟练掌握数据库的设计和开发。

除此以外，这个例子假定已在服务器上安装好了 SQL 数据库（尽管它也可以在一台本地机器上运行）。另外，我们也为数据库找到了一个恰当的 JDBC 驱动程序，并证明它能可靠使用。目前有几个免费的 SQL 数据库产品可供你免费使用，有的甚至能在你安装某种 Linux 操作系统的时候自动装好。你的工作仅仅是选择数据库，并找到它的 JDBC 驱动程序。这个例子建立在一个名为“Cloudscape”的 SQL 数据库系统基础上。

为方便我们对连接信息进行改动，数据库驱动程序、数据库 URL、用户名以及密码都放在单独一个类里：

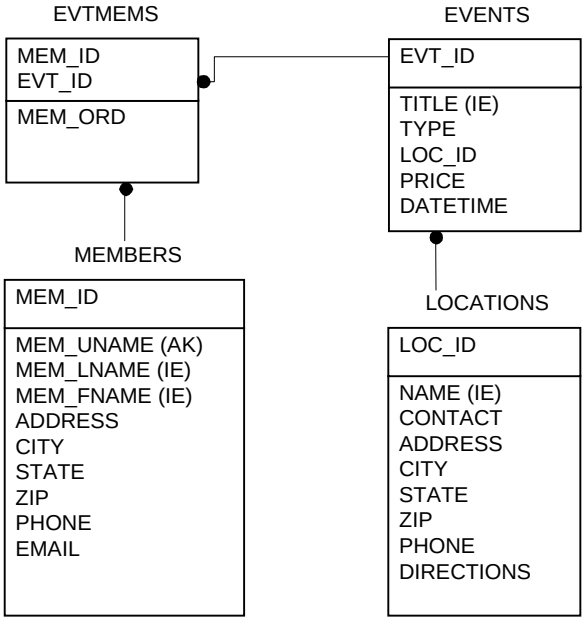
---

[69] 该例由 Dave Bartlett 创建。

```
//: c15:jdbc:CIDConnect.java
// Database connection information for
// the community interests database (CID).

public class CIDConnect {
  // All the information specific to CloudScape:
  public static String dbDriver =
    "COM.cloudscape.core.JDBCDriver";
  public static String dbURL =
    "jdbc:cloudscape:d:/docs/_work/JSapienDB";
  public static String user = "";
  public static String password = "";
} ///:~
```

在这个例子中，不打算为数据库加上密码保护，所以用户名和密码均为空串。

该数据库由一系列数据表构成，结构如下：



其中，“MEMBERS”（成员）表包含了社区成员的个人资料；“EVENTS”（活动）和“LOCATIONS”（地点）表包含了与社区活动有关的信息，并指出在什么地方进行；而“EVTMEMS”表用于将活动与愿意参加那个活动的成员连接到一起（EVTMEMS＝EVENTS+MEMBERS）。大家可以看到，一张表内的一个数据成员在另一个表里产生了一个键。

下面这个类包含了用于创建这些数据库表的 SQL 字串（查阅相关 SQL 指南，了解 SQL 代码语法）：

```
//: c15:jdbc:CIDSQL.java
// SQL strings to create the tables for the CID.

public class CIDSQL {
```

```
public static String[] sql = {
  // Create the MEMBERS table:
  "drop table MEMBERS",
  "create table MEMBERS " +
  "(MEM_ID INTEGER primary key, " +
  "MEM_UNAME VARCHAR(12) not null unique, "+
  "MEM_LNAME VARCHAR(40), " +
  "MEM_FNAME VARCHAR(20), " +
  "ADDRESS VARCHAR(40), " +
  "CITY VARCHAR(20), " +
  "STATE CHAR(4), " +
  "ZIP CHAR(5), " +
  "PHONE CHAR(12), " +
  "EMAIL VARCHAR(30))",
  "create unique index " +
  "LNAME_IDX on MEMBERS(MEM_LNAME)",
  // Create the EVENTS table
  "drop table EVENTS",
  "create table EVENTS " +
  "(EVT_ID INTEGER primary key, " +
  "EVT_TITLE VARCHAR(30) not null, " +
  "EVT_TYPE VARCHAR(20), " +
  "LOC_ID INTEGER, " +
  "PRICE DECIMAL, " +
  "DATETIME TIMESTAMP)",
  "create unique index " +
  "TITLE_IDX on EVENTS(EVT_TITLE)",
  // Create the EVTMEMS table
  "drop table EVTMEMS",
  "create table EVTMEMS " +
  "(MEM_ID INTEGER not null, " +
  "EVT_ID INTEGER not null, " +
  "MEM_ORD INTEGER)",
  "create unique index " +
  "EVTMEM_IDX on EVTMEMS(MEM_ID, EVT_ID)",
  // Create the LOCATIONS table
  "drop table LOCATIONS",
  "create table LOCATIONS " +
  "(LOC_ID INTEGER primary key, " +
  "LOC_NAME VARCHAR(30) not null, " +
  "CONTACT VARCHAR(50), " +
  "ADDRESS VARCHAR(40), " +
  "CITY VARCHAR(20), " +
  "STATE VARCHAR(4), " +
```

```
    "ZIP VARCHAR(5), " +
    "PHONE CHAR(12), " +
    "DIRECTIONS VARCHAR(4096))",
    "create unique index " +
    "NAME_IDX on LOCATIONS(LOC_NAME)",
  };
} ///:~
```

下面这个程序利用 CIDConnect 和 CIDSQL 信息来装载 JDBC 驱动程序，建立同数据库的一个连接，然后创建如上图所示的表结构。要想连接数据库，我们需要调用名为 DriverManager.getConnection()的一个静态方法，向其传递目标数据库的 URL、用户名以及密码信息，以便获得对数据库的访问权。成功之后，会返回一个 Connection 对象，利用它就可开始查询和操作数据库了。连接建立之后，我们可以简单地将 SQL“推”到数据库中——在目前的情况下，我们是通过在 CIDSQL 数组里进行的遍历。不过，程序首次运行时，“drop table”命令会失败，从而导致一个违例。这个违例会被捕捉下来，作出报告，然后被简单地忽略。之所以要执行“drop table”命令，原因是为了更方便地进行实验——我们可以修改对表进行定义的 SQL，再运行程序，使老表被新表替代。

在这个例子中，有必要将违例“扔”到控制台上：

```
//: c15:jdbc:CIDCreateTables.java
// Creates database tables for the
// community interests database.
import java.sql.*;

public class CIDCreateTables {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException,
  IllegalAccessException {
    // Load the driver (registers itself)
    Class.forName(CIDConnect.dbDriver);
    Connection c = DriverManager.getConnection(
      CIDConnect.dbURL, CIDConnect.user,
      CIDConnect.password);
    Statement s = c.createStatement();
    for(int i = 0; i < CIDSQL.sql.length; i++) {
      System.out.println(CIDSQL.sql[i]);
      try {
        s.executeUpdate(CIDSQL.sql[i]);
      } catch(SQLException sqlEx) {
        System.err.println(
          "Probably a 'drop table' failed");
      }
    }
    s.close();
```

```
    c.close();
  }
} ///:~
```

注意数据库中的所有改动都可通过修改 CIDSQL 表内的字串来加以控制，而不必修改 CIDCreateTables。

executeUpdate()通常会返回受 SQL 语句影响的行数。修改一个或多个数据行的时候，我们更经常地通过 executeUpdate()来执行 INSERT、UPDATE 或者 DELETE 语句。对于象 CREATE TABLE、DROP TABLE 以及 CREATE INDEX 这样的语句，executeUpdate()返回的肯定是 0。

为了对数据库进行测试，它装载的时候有一个示范性数据。这便要求用一系列 INSERT 和 SELECT 语句（注意执行顺序）来产生结果集。为了方便地添加和修改测试数据，我们将测试数据设置成一个二维的对象数组。这样一来，executeInsert()方法就可利用表内一行内的信息来创建恰当的 SQL 命令。

```
//: c15:jdbc:LoadDB.java
// Loads and tests the database.
import java.sql.*;

class TestSet {
  Object[][] data = {
    { "MEMBERS", new Integer(1),
      "dbartlett", "Bartlett", "David",
      "123 Mockingbird Lane",
      "Gettysburg", "PA", "19312",
      "123.456.7890",  "bart@you.net" },
    { "MEMBERS", new Integer(2),
      "beckel", "Eckel", "Bruce",
      "123 Over Rainbow Lane",
      "Crested Butte", "CO", "81224",
      "123.456.7890", "beckel@you.net" },
    { "MEMBERS", new Integer(3),
      "rcastaneda", "Castaneda", "Robert",
      "123 Downunder Lane",
      "Sydney", "NSW", "12345",
      "123.456.7890", "rcastaneda@you.net" },
    { "LOCATIONS", new Integer(1),
      "Center for Arts",
      "Betty Wright", "123 Elk Ave.",
      "Crested Butte", "CO", "81224",
      "123.456.7890",
      "Go this way then that." },
    { "LOCATIONS", new Integer(2),
      "Witts End Conference Center",
```

```
      "John Wittig", "123 Music Drive",
      "Zoneville", "PA", "19123",
      "123.456.7890",
      "Go that way then this." },
    { "EVENTS", new Integer(1),
      "Project Management Myths",
      "Software Development",
      new Integer(1), new Float(2.50),
      "2000-07-17 19:30:00" },
    { "EVENTS", new Integer(2),
      "Life of the Crested Dog",
      "Archeology",
      new Integer(2), new Float(0.00),
      "2000-07-19 19:00:00" },
    // Match some people with events
    {  "EVTMEMS",
      new Integer(1),  // Dave is going to
      new Integer(1),  // the Software event.
      new Integer(0) },
    { "EVTMEMS",
      new Integer(2),  // Bruce is going to
      new Integer(2),  // the Archeology event.
      new Integer(0) },
    { "EVTMEMS",
      new Integer(3),  // Robert is going to
      new Integer(1),  // the Software event.
      new Integer(1) },
    { "EVTMEMS",
      new Integer(3), // ... and
      new Integer(2), // the Archeology event.
      new Integer(1) },
  };
  // Use the default data set:
  public TestSet() {}
  // Use a different data set:
  public TestSet(Object[][] dat) { data = dat; }
}

public class LoadDB {
  Statement statement;
  Connection connection;
  TestSet tset;
  public LoadDB(TestSet t) throws SQLException {
    tset = t;
```

```
    try {
      // Load the driver (registers itself)
      Class.forName(CIDConnect.dbDriver);
    } catch(java.lang.ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    connection = DriverManager.getConnection(
      CIDConnect.dbURL, CIDConnect.user,
      CIDConnect.password);
    statement = connection.createStatement();
  }
  public void cleanup() throws SQLException {
    statement.close();
    connection.close();
  }
  public void executeInsert(Object[] data) {
    String sql = "insert into "
      + data[0] + " values(";
    for(int i = 1; i < data.length; i++) {
      if(data[i] instanceof String)
        sql += "'" + data[i] + "'";
      else
        sql += data[i];
      if(i < data.length - 1)
        sql += ", ";
    }
    sql += ')';
    System.out.println(sql);
    try {
      statement.executeUpdate(sql);
    } catch(SQLException sqlEx) {
      System.err.println("Insert failed.");
      while (sqlEx != null) {
        System.err.println(sqlEx.toString());
        sqlEx = sqlEx.getNextException();
      }
    }
  }
  public void load() {
    for(int i = 0; i< tset.data.length; i++)
      executeInsert(tset.data[i]);
  }
  // Throw exceptions out to console:
  public static void main(String[] args)
```

```
  throws SQLException {
    LoadDB db = new LoadDB(new TestSet());
    db.load();
    try {
      // Get a ResultSet from the loaded database:
      ResultSet rs = db.statement.executeQuery(
        "select " +
        "e.EVT_TITLE, m.MEM_LNAME, m.MEM_FNAME "+
        "from EVENTS e, MEMBERS m, EVTMEMS em " +
        "where em.EVT_ID = 2 " +
        "and e.EVT_ID = em.EVT_ID " +
        "and m.MEM_ID = em.MEM_ID");
      while (rs.next())
        System.out.println(
          rs.getString(1) + "  " +
          rs.getString(2) + ", " +
          rs.getString(3));
    } finally {
      db.cleanup();
    }
  }
} ///:~
```

TestSet 类包含了一个默认的数据集；假如你使用的是默认构造函数，便会产生这个数据集。不过，你也可以通过第二个构造函数，在另一个数据集的基础上创建 TestSet 对象。之所以要用一个二维对象数组来容纳数据集，是由于它可以是任意类型——包括字串或数值类型。executeInsert()方法根据数据构建 SQL 命令时，利用 RTTI 对字串数据（这种数据有引号）和非字串数据加以区分。将这个命令打印到控制台后，便用 executeUpdate()把它发送给数据库。

LoadDB 的构造函数负责建立真正的连接，而 load()会遍历这些数据，并为每条记录都调用 executeInsert()。cleanup()负责关闭语句和连接；为保证它得到调用，我们把它放在一条 finally 从句中。

数据库载入后，一条 executeQuery()语句便会产生示范结果集。由于查询同时合并了几个表，所以它是“连接”（Join）操作的一个典型例子。

在 Sun 发布的 Java 产品中，配套提供了一系列电子文档，它们包含有更丰富的 JDBC 信息。除此以外，在《JDBC Database Access with Java》这本书中，你也能找到更加深入的内容。该由 Hamilton、Cattel 和 Fisher 编写，Addison-Wesley 出版社于 1997 年出版。其他 JDBC 参考书也常有面市。

15.3 小服务程序（Servlet）

显然，来自 Internet 或企业内部网的客户访问是让大量用户同时方便地访问数据和资源

的一种简便方法[70]。这种形式的访问要求客户机使用万维网标准的“超文本标记语言！（HTML）以及“超文本传输协议”（HTTP）。Servlet API 集抽象出了一个通用的方案框架，用以代表 HTTP 请求。

在传统意义上，对于象允许一个 Internet 客户更新数据库这样的问题来说，采取的办法是创建一个 HTML 页（俗称“网页”），在其中放上文本字段（文字输入区）以及一个“提交”（Submit）按钮。用户键入适当的信息，然后按下那个按钮。随后，数据会随同一个 URL 传给服务器，告诉它运行指定地方的一个通用网关接口（CGI）程序，对自己提交的数据进行处理。CGI 程序通常是用 Perl、Python、C、C++或者其他任何一种语言写成的——只要那种语言能读取标准输入，并能写到标准输出。所有这些都由 Web 服务器来提供：调用 CGI 程序，用标准数据流（输入时也可用一个环境变量进行）进行输入和输出。CGI 程序则负责做其他一切事情。首先，它会检查输入的数据，判断它的格式是否正确。如果不正确，CGI 程序必须生成相应的 HTML 代码，以网页的形式，向用户描述其中存在的问题。这个页会传给 Web 服务器（通过 CGI 程序的标准输出），再由服务器回传给用户。在这种情况下，用户通常需要后退一个页，重新输入一遍。假如输入的数据是正确的，CGI 程序就会用恰当的方式来处理数据，或许还要把数据添加到一个数据库。随后，它必须产生一个恰当的 HTML 网页，让 Web 服务器把这个页回传给用户，让他们看到处理后的结果。

假如能完全用 Java 提供的方案来解决上述问题，那么便是最理想的——客户端运行的小程序检查数据格式是否正确，然后发出数据。随后，由服务器端的小服务程序接收和处理这些数据。但不幸的是，尽管小程序是一种可行的技术，而且已赢得了大量支持，但它们在 Web 上使用仍然问题多多。原因很简单，你没办法预测客户机的 Web 浏览器到底采用的是哪一个 Java 版本。事实上，我们甚至没法子肯定一个 Web 浏览器到底有没有提供对 Java 的支持！但内部网就不同了。由于范围较小，你可以保证客户机提供了正确的支持，从而可以更加自由、更加放心地来设计自己的系统。但在 Web 上，最安全的做法还是在服务器端进行所有处理，并只将纯 HTML 回传给客户机。只有这样，你才可以保证不会由于对方安装的软件不对，从而将一些客户机拒之门外！

小服务程序为服务器端的编程提供了一个出色的方案，这使其成为许多人愿意换用 Java 的决定性原因之一。这样一来，不仅可以彻底替换掉传统的 CGI 程序框架（同时避免了许多麻烦的 CGI 问题），而且由于用的是 Java，所以你的所有代码都具有了强大的移植能力。另外，你还能访问到所有的 Java API（当然，那些用于产生 GUI 的除外，比如 Swing）。

15.3.1　基本 Servlet

小服务程序 API 的体系非常“经典”：一个 service()扮演着“服务提供者”的角色。通过小服务程序容器软件通过它发出所有客户机请求；用于初始化和卸载的 init()和 destory()方法。后者只有在小服务程序装载后又卸载的时候才会调用（这种情况很少见）。

```
public interface Servlet {
  public void init(ServletConfig config)
    throws ServletException;
  public ServletConfig getServletConfig();
  public void service(ServletRequest req,
    ServletResponse res)
    throws ServletException, IOException;
```

[70] Dave Bartlett 对本节以及 JSP 那一节作了不少的贡献，在此表示感谢。

```
  public String getServletInfo();
  public void destroy();
}
```

getServletConfig()唯一的用途就是返回一个 ServletConfig 对象，其中包含了这个小服务程序的初始化和启动参数。getServletInfo()可返回一个字串，其中包含与小服务程序有关的信息，比如它的作者、版本和版权声明等等。

GenericServlet 类是该接口的一外壳实现方式，通常不必使用。HttpServlet 类是对 GenericServlet 的一个扩展，专门用来控制 HTTP 协议——HttpServlet 是我们以后会经常用到的一个类。

对小服务程序 API 来说，它设计得最“贴心”的一个地方就是为 HttpServlet 类提供支持的辅助对象。注意一下 Servlet 接口中的 service()方法，可发现它有两个参数：ServletRequest 和 ServletResponse。在 HttpServlet 类中，这两个对象针对 HTTP 进行了相应的扩展，分别变成了 HttpServletRequest 和 HttpServletResponse。下面有一个简单的例子，它展示了 HttpServletResponse 的用法：

```
//: c15:servlets:ServletsRule.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
  int i = 0; // Servlet "persistence"
  public void service(HttpServletRequest req,
  HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.print("<HEAD><TITLE>");
    out.print("A server-side strategy");
    out.print("</TITLE></HEAD><BODY>");
    out.print("<h1>Servlets Rule! " + i++);
    out.print("</h1></BODY>");
    out.close();
  }
} ///:~
```

ServletsRule 和小服务程序本身一样简单。小服务程序只需初始化一道，这种初始化是在小服务程序容器首次启动之后，在装载小服务程序的时候，通过调用它的 init()方法来实现的。假如一个客户机发出对一个 URL 的请求，同时它又恰巧代表一个小服务程序，那么小服务程序容器就会拦截这一请求，并在设置好 HttpServletRequest 和 HttpServletResponse 对象之后，调用 service()方法。

对 service()方法来说，它的主要职责便是同客户机发出的 HTTP 请求打交道，并根据在请求里指定的属性，构建一个 HTTP 响应。ServletRule 只会对响应的对象进行操作，不计客户机发出的到底是什么。

设置好响应的内容类型之后（产生 Writer 或 OutputStream 之前，这个工作必须做好），响应对象的 getWriter()方法便会产生一个 PrintWriter 对象，它用于生成基于字符的响应数据（另外，亦可用 getOutputStream()产生一个 OutputStream，以便用于二进制响应；不过，只有在特别的场合下才这样做）。

程序剩下的部分只是简单地以一个字串序列的形式，将 HTML 传回客户机（这里假定你已理解了 HTML，所以那一部分不再解释）。然而，请注意由变量 i 表示的“点击计数器”。在 print()语句中，它被自动转换成一个字串。

运行程序时，我们发现在向小服务程序发送不同的请求之间，i 的值是保持不变的。这其实反映了小服务程序一个重要的特征：由于一个类只有一个小服务程序载入容器，而且永远不会卸载（除非小服务程序容器中止——通常只有在服务器重启的时候才会出现这种情况），所以那个小服务程序类的所有字段实际上都变成了“持久性对象”！这便意味着，在不同的小服务程序请求之间，我们根本不必担心对值进行保持的问题。但另一方面，CGI 就显得麻烦得多，它要求将值写到磁盘上，否则便没办法保持。这样一来，不仅操作复杂，而且最后得到的方案很难跨平台移植。

当然，Web 服务器必须重新启动（所以小服务程序也得重启），比如在进行系统维护或者断电的时候。为避免丢失任何持久性信息，小服务程序的 init()和 destroy()方法会在小服务程序载入或卸载的时候自动调用，这样便使我们有机会在当机期间将数据保存下来，并在重新启动之后恢复。小服务程序容器会在中止自己的时候调用 destroy()方法，所以只要服务器进行了正确的配置，你总有机会将重要数据保存下来。

使用 HttpServlet 时，还需注意另一个问题。这个类提供了 doGet()和 doPost()方法，可以区分来自客户机的一个 CGI “GET”和一个 CGI “POST”。GET 和 POST 只是具体的数据提交方式有所差异。而对于这种差异，我个人宁愿忽略。不过，我看过的大多数出版物似乎都喜欢创建独立的 doGet()和 doPost()方法，而不是用一个通用的、统一的 service()方法——该方法能同时控制两种类。这应该说是一种“偏见”，而且还似乎颇为流行，但我却认为这是从 CGI 程序员那里继承来的一种不良习惯——他们总是特别留意用的到底是一个 GET，还是一个 POST。所以在“最简单的做法也许最有用”[71]这一准则的指导下，就我自己来说，情愿在这些例子中都只采用 service()方法，让它自己去分辨 GET 和 POST。

一个表单提交给一个小服务程序后，HttpServletRequest 便会预先载入所有表单数据，并用“键－值”对的形式把那些数据保存下来。假如你知道字段名，便可用 getParameter()方法直接使用它们，在其中查找需要的值。另外，就象下例展示的那样，也可以取得对所有字段名的一个列举（Enumeration——“迭代器”的一种老形式）。这个例子也演示了如何在仅仅一个小服务程序中，产生包含了窗体的网页，并同时对网页作出响应（后面还有一个更好的方案，用 JSP 实现的）。假如 Enumeration 是空的，便表明没有字段——换言之，没有表单提交进来。在这种情况下，它会生成表单，而且提交按钮会重新调用同一个小服务程序。不过，假如字段存在，便直接把它们显示出来。

```
//: c15:servlets:EchoForm.java
// Dumps the name-value pairs of any HTML form
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
```

[71] 这是“极端编程”（XP）的一项基本准则。详见 www.xpprogramming.com。

```
public class EchoForm extends HttpServlet {
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    Enumeration flds = req.getParameterNames();
    if(!flds.hasMoreElements()) {
      // No form submitted -- create one:
      out.print("<html>");
      out.print("<form method=\"POST\"" +
        " action=\"EchoForm\">");
      for(int i = 0; i < 10; i++)
        out.print("<b>Field" + i + "</b> " +
          "<input type=\"text\""+
          " size=\"20\" name=\"Field" + i +
          "\" value=\"Value" + i + "\"><br>");
      out.print("<INPUT TYPE=submit name=submit"+
      " Value=\"Submit\"></form></html>");
    } else {
      out.print("<h1>Your form contained:</h1>");
      while(flds.hasMoreElements()) {
        String field= (String)flds.nextElement();
        String value= req.getParameter(field);
        out.print(field + " = " + value+ "<br>");
      }
    }
    out.close();
  }
} ///:~
```

在这里，大家也会注意到一个缺点：Java 的字串处理功能似乎并不理想——需要需要用换行、引号以及加号来构建字串对象，所以返回的页面看起来似乎一团糟。假如网页较大，那么几乎完全不可能把它直接编码到 Java 里。此时，一个解决办法是将网页保持为一个独立的文本文件，然后打开它，并传递给 Web 服务器。假如必须对页内的内容进行一些替换操作，那么实际上并不能从中获得多大的便利，因为 Java 的字串处理能力实在是太可怜了。在这些情况下，你恐怕需要选择另一种更恰当的方案（要我选择的话，就选 Python；它甚至有一个版本是专门嵌入到 Java 中的，名为 JPython），从而生成一个清爽悦目的页。

15.3.2 Servlet 和多线程

小服务程序容器提供了一个线程池，用于对来自客户机的请求进行控制。同时到达的两个客户机请求完全有可能通过你的 service()同时进行处理。因此，service()方法必须用一种保证线程安全的方式来写成。对通用资源（文件、数据库等等）的任何访问都需要用 synchronized 关键字加以保护。

下面这个简单的例子在线程的 sleep()方法外部包裹了一条 synchronized 从句。它会封锁其他所有线程，直至规定时间过去（5 秒钟）。进行测试时，你应该同时启动几个浏览器窗口，然后在每个窗口里都尽可能快地访问这个 Servlet——结果应该是每一个都必须等待规定的时间之后才能访问。

```
//: c15:servlets:ThreadServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ThreadServlet extends HttpServlet {
  int i;
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    synchronized(this) {
      try {
        Thread.currentThread().sleep(5000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
    out.print("<h1>Finished " + i++ + "</h1>");
    out.close();
  }
} ///:~
```

另外，整个小服务程序都有可能进行同步——做法是将 synchronized 关键字放到 service()方法的前面。事实上，换用 synchronized 从句的唯一理由就是关键性的代码位于一条可能永远都不会执行的执行路径上。在那种情况下，通过使用一个 synchronized 从句，我们也有可能免去每次进行同步的开销。

15.3.3　用 Servlet 控制会话

HTTP 是一种“无会话”的协议，所以对于连续产生的服务器点击来说，你无从知道是同一个人在连续查询你的站点，还是完全由不同的人在查询。为此，许多人付出了大量努力，使 Web 开发者能对会话进行跟踪。假如不能对客户进行追踪，那些做电子商务的公司便根本没法子做生意——最起码的一点，他们不能把货物放到用户的“购物车”中。

有几个办法都可对会话进行跟踪，但最常见的还是利用“Cookie”（小甜饼），它已成为 Internet 标准不可分割的一部分。“Internet 工程任务组织”（IETF）下属的“HTTP 工作组”已将 Cookie 制订为正式标准，详见 RFC 2109 文件（ds.internic.net/rfc/rfc2109.txt，或者访问 www.cookiecentral.com）。

“Cookie”是由 Web 服务器发给浏览器的一种非常短小的信息，采用纯文本格式。浏览器把它们保存在用户的磁盘上。以后，只要再次访问与 Cookie 对应的那个 URL，Cookie

便会随那次调用一起，“悄悄地”传给服务器。这样一来，服务器便知道：“哦，上次来过的那个人又来了!”你仔细想想，便知道这实际是保持“通信会话”的一种非常初级的形式。不过，出于安全方面的考虑，几乎所有浏览器都允许用户拒绝接收任何形式的 Cookie，只需要简单地设置一下就可以了。假如你的站点必须对那些关闭了 Cookie 功能的客户机进行跟踪，那么必须亲自设计另一种会话跟踪和保持手段（比如 URL 改写或表单的隐藏字段等等）——这是由于 Servlet API 内置的会话跟踪功能是完全围绕 Cookie 而设计的。

1. Cookie 类

Servlet API（2.0 或更高版本）提供了一个 Cookie 类。这个类集成了所有的 HTTP 头细节，并允许设计各种不同的 Cookie 属性。要想使用 Cookie，只需把它添加到响应对象就可以了。构造函数需要取得 Cookie 的名字作为它的第一个参数，以及一个值作为第二个参数。在你发送任何正式内容之前，Cookie 必须完成它在响应对象里的添加。

```
Cookie oreo = new Cookie("TIJava", "2000");
res.addCookie(cookie);
```

通过调用 HttpServletRequest 对象的 getCookie()方法，便可恢复 Cookie。该方法会返回由 Cookie 对象构成的一个数组：

```
Cookie[] cookies = req.getCookies();
```

随后，便可为每个 Cookie 调用 getValue()，从而生成一个字串，其中包含了实际的 Cookie 内容。在上述例子中，getValue("TIJava")会产生包含了“2000”字样的一个字串。

2. Session 类

“Session”（会话）是指在规定时间内，由一个客户机向 Web 站点发出的一个或多个页面访问请求。例如，假如你从网上购买食品，那么一个会话就应该自首次在“我的购物车”中添加一样食品开始，一直持续到“结帐”为止。每次在购物车里添加一样食品时，都会造成一个新的 HTTP 连接的建立，它对以前的连接或者购物车里已有的东西是没有丝毫概念的。为了弥补这种信息不足的漏洞，你的所有购物记录都会保存在 Cookie 中，从而至少在表面上，实现了“会话跟踪”。

一个小服务程序的 Session 对象寄居在服务器那一端；其目的是在客户机同 Web 站点打交道期间，跟踪记录有用的数据。这种数据可能与父会话有着紧密的联系——比如购物车里的货物；或者可以是一些敏感的用户身份资料（客户机首次登录你的站点时输入的），在以后的特定操作中，同样的资料便不用重新输入。

小服务程序 API 的 Session 类利用 Cookie 类来完成这些工作。不过，所有 Session 对象都必须将某种独一无二的标识符保存在客户机中，并把它传递给服务器。另外，Web 站点也可以采用其他形式的会话跟踪机制——只是这很难实现，因为它们并不是封装在小服务程序 API 中的（换言之，你必须亲手编写它们，以便在客户机屏蔽了 Cookie 支持的前提下，仍然能跟踪用户会话）。

下面是用小服务程序 API 实现会话跟踪的一个例子：

```
//: c15:servlets:SessionPeek.java
// Using the HttpSession class.
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class SessionPeek extends HttpServlet {
  public void service(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException, IOException {
    // Retrieve Session Object before any
    // output is sent to the client.
    HttpSession session = req.getSession();
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<HEAD><TITLE> SessionPeek ");
    out.println(" </TITLE></HEAD><BODY>");
    out.println("<h1> SessionPeek </h1>");
    // A simple hit counter for this session.
    Integer ival = (Integer)
      session.getAttribute("sesspeek.cntr");
    if(ival==null)
      ival = new Integer(1);
    else
      ival = new Integer(ival.intValue() + 1);
    session.setAttribute("sesspeek.cntr", ival);
    out.println("You have hit this page <b>"
      + ival + "</b> times.<p>");
    out.println("<h2>");
    out.println("Saved Session Data </h2>");
    // Loop through all data in the session:
    Enumeration sesNames =
      session.getAttributeNames();
    while(sesNames.hasMoreElements()) {
      String name =
        sesNames.nextElement().toString();
      Object value = session.getAttribute(name);
      out.println(name + " = " + value + "<br>");
    }
    out.println("<h3> Session Statistics </h3>");
    out.println("Session ID: "
      + session.getId() + "<br>");
    out.println("New Session: " + session.isNew()
      + "<br>");
    out.println("Creation Time: "
      + session.getCreationTime());
    out.println("<I>(" +
      new Date(session.getCreationTime())
      + ")</I><br>");
```

```
    out.println("Last Accessed Time: " +
      session.getLastAccessedTime());
    out.println("<I>(" +
      new Date(session.getLastAccessedTime())
      + ")</I><br>");
    out.println("Session Inactive Interval: "
      + session.getMaxInactiveInterval());
    out.println("Session ID in Request: "
      + req.getRequestedSessionId() + "<br>");
    out.println("Is session id from Cookie: "
      + req.isRequestedSessionIdFromCookie()
      + "<br>");
    out.println("Is session id from URL: "
      + req.isRequestedSessionIdFromURL()
      + "<br>");
    out.println("Is session id valid: "
      + req.isRequestedSessionIdValid()
      + "<br>");
    out.println("</BODY>");
    out.close();
  }
  public String getServletInfo() {
    return "A session tracking servlet";
  }
} ///:~
```

在 service()方法内部，我们为请求对象调用 getSession()，它会返回同这个请求对应的 Session 对象。Session 对象并不通过网络进行传输；相反，它寄居在服务器上，并同客户机及其语法建立起联系。

getSession()共提供了两个版本：一个没有参数，就象本例显示的那样；另一个有参数，即 getSession(boolean)——其中，getSession(true)等价于不带布尔值参数的 getSession()。之所以要使用布尔值参数，唯一的目的就是指出在没找到会话对象的前提下，是不是马上创建一个。其中最常用的还是 getSession(true)——亦即 getSession()。

如果面对的并不是一个刚刚新建的 Session 对象，那么它会告诉我们与前一次访问的客户机有关的细节。假如 Session 对象是新建的，那么程序会自这一次访问开始，收集和客户机的活动有关的信息。对客户机活动信息的捕捉是通过由 Session 对象提供的 setAttribute()和 getAttribute()方法来做到的。如下所示：

```
java.lang.Object getAttribute(java.lang.String)
void setAttribute(java.lang.String name,
                  java.lang.Object value)
```

Session 对象采用一种简单的“名字一值”对来载入信息。其中，“名字”是一个字串，而“值”可以是从 java.lang.Object 派生的任何对象。SessionPeek 用于跟踪记录在这一次“会

话”期间，客户机总共返回了多少次。这是用一个名为 sesseek.cntr 的 Integer（整数）对象来完成的。假如指定的名字没有找到，便创建值为 1 的一个整数；否则的话，新的整数值就为以前那个整数值递增（加 1）之后的值。新的 Integer 对象会放到 Session 对象里。假如在一次 setAttribute()调用期间使用相同的键，那么新对象就会覆盖老对象。递增后的计数器用于显示在此次会话期间，客户机总共访问了多少次。

getAttributeNames()和 getAttribute()及 setAttribute()都有关系；它会返回当前放到 Session 对象中的所有对象名的一个“列举”。SessionPeek 中的一个 while 循环展示了这个方法的实际运用。

大家也许会感到迷惑，一个 Session 对象到底要“坚守阵地”多长的时间才算数呢？答案取决于当前正在用着的 Servlet 容器；它们通常默认为 30 分钟（1800 秒）。通过在 ServletPeek 里调用 getMaxInactiveInterval()，便应该看到这样的数值。不过，由于 Servlet 容器的区别，我们的测试似乎产生了混合性的结果。有些时候，Session 对象可能一整晚的时间都“挂”在那里。不过，我还从来没见过一个 Session 对象在规定的时间之前自动“撤退”。对此，大家可以亲自测试一下。你只需将 setMaxInactiveInterval()设为 5 秒，然后观察当过了规定的时间后，Session 对象到底是还“挂”在哪里呢，还是被清除。在决定实际使用的一个小服务程序容器时，这是你应当调查清楚的一种属性。

15.3.4 运行 Servlet 例子

如果你手边还没有一个应用程序服务器可以支持 Sun 的小服务程序和 JSP 技术，那么可下载名叫“Tomcat”（汤姆猫）的一个 Java 小服务程序及 JSP 支持包。它是免费的，而且是“开放源码”的，而且获得 Sun 公司正式授权发布。它的网址在 jakarta.appache.org。

请根据指示安装好 Tomcat，然后编辑 server.xml 文件，指定你放置了小服务程序的那个目录树位置。启动 Tomcat 程序之后，就可开始对自己的小服务程序进行测试了。

这里提供的只是对小服务程序的一个简要介绍；关于这个主题，还有其他许多大部头的书在专门讲它。不过，弄懂了这里的概念，大家就应该算是“入门”了。另外，下一节的许多概念也是“向后兼容”于小服务程序的。

15.4 Java 服务器页（JSP）

“Java 服务器页”（Java Server Pages，JSP）是一种标准的 Java 扩展，定义于小服务器扩展的顶部。JSP 的设计目标是简化动态 Web 页的生成与管理。

前文提到的 Tomcat 参考实现（jakarta.apache.org）自动地提供了对 JSP 的支持。

利用 JSP，我们可将一个 Web 页的 HTML 代码同 Java 代码合并到同一个文档中。在 Java 代码的周围，包裹着一系列特殊标记，指示 JSP 容器应该利用代码来生成一个小服务程序——或至少一部分小服务程序。JSP 的优点在于我们只需维护一个文档，它同时代表着页面内容以及背后提供支持的 Java 代码。缺点则在于 JSP 页的容器必须同时兼容 HTML 和 Java（不过，要到将来才会出现支持 JSP 的 GUI 构建环境）。

JSP 容器首次载入一个 JSP 时（JSP 容器通常同一个 Web 服务器对应在一起，有时甚至就是 Web 服务器的一部分），放在 JSP 标记中的小服务程序代码就会自动生成、编译并载入容器。HTML 页的“静态”部分通过向 write()发送静态字串对象来产生。动态部分则直接包括到小服务程序中。

从这时起，只要未对这个页的 JSP 源码进行修改，它便和一个普通的静态 HTML 页再加一个相应的小服务程序没有什么区别（只是所有 HTML 代码都是由小服务程序生成的）。但假如修改了 JSP 源码，便会自动重新编译，并在这个页下一次被请求的时候重新装载，反

映出最新的变动。当然，所有这些操作都是动态进行的，所以在第一次访问一个 JSP 时，对方的响应速度肯定会显得慢一些。不过，既然一个 JSP 通常使用的次数应该比改变的次数多得多，所以平常也不用过份担心这方面的问题。

JSP 页的结构属于小服务程序和 HTML 页的一种“交集”。JSP 标记的起始和结束都采用尖括号，这和 HTML 标记没什么两样——只是所有标记同时还加上了百分号。因此，所有 JSP 标记都象下面这样表示：

```
<% JSP code here %>
```

在最开头的那个百分号之后，可加上其他字符，指出标记内的 JSP 代码的准确类型是什么。

下面是一个极其简单的 JSP 例子，它利用一个标准的 Java 库调用来获得当前时间（以毫秒为单位），再除以 1000，得到以秒数表示的时间。由于这里采用的是一个 JSP 表达式（<%=），所以计算结构会强制转变成一个字串，并放到最终生成的 Web 页上：

```
//:! c15:jsp:ShowSeconds.jsp
<html><body>
<H1>The time in seconds is:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~
```

在本书的 JSP 例子中，实际的代码文件毋需第一行和最后一行。

Web 服务器首先必须作好设置，以便客户机建立对 JSP 页的一个请求时，将这个请求转发给 JSP 容器。随后，再由容器来调入网页。正如早先指出的那样，页面首次调入时，会生成由页指定的组件，并由 JSP 容器把它编译成一个或者多个小服务程序的形式。在上面的例子中，小服务程序包含了相应的代码来配置 HttpServletResponse 对象，产生一个 PrintWriter 对象（它的名字肯定叫 out），然后将计算出来的时间转换成一个字串，再将字串发给 out。就象大家看到的那样，所有这些都是用一条非常简洁的语句来完成的，但对普通的 HTML/Web 设计人员来说，却往往不具备写这种代码的“功底”。

15.4.1 隐式对象

小服务程序包含了一些类，为我们提供了一系列便利的工具，比如 HttpServletRequest、HttpServletResponse、Session 等等。这些类的对象已构建到 JSP 规范中，并可在你的 JSP 中直接使用，不需要再为此编写任何多余的代码。我们将这样的对象称作“隐式对象”或者“默认对象”。下表对 JSP 中的隐式对象进行了详细总结。

隐式变量	类型（javax.servlet）	说　　明	作用域
Request	HttpServletRequest 的协议相关子类型	请求进行服务调用	请求
response	HttpServletResponse 的协议相关子类型	响应请求	页
pageContext	jsp.PageContext	页面场景（Page Context）封装了与具体产品有关的特性，并为这个 JSP 提供了便利的方法和命名空间	页

续表

隐式变量	类型（javax.servlet）	说　　明	作用域
session	http.HttpSession 的协议相关协议	为发出请求的客户机创建的会话对象。参见小服务程序的 Session 对象	会话
application	ServletContext	从小服务程序配置对象（比如 getServletConfig()和 getContext()）获得的小服务程序场景	应用程序
out	jsp.JspWriter	在输出流里写入的对象	页
config	ServletConfig	针对这个 JSP 的 ServletConfig	页
page	java.lang.Object	对当前请求进行处理的这个页的实现类的实例	页

每个对象的“作用域”都可能发生显著变化。举个例子来说，session 对象的作用域可以超出一个页的作用域，因为它可能同时跨越多个客户机请求和页。application 对象可为一组 JSP 页提供服务，那些页合并到一起，便形成了一个完整的应用程序。

15.4.2 JSP 引导命令

JSP 容器的引导命令其实是一些消息，它们用“@”加以标记：

```
<%@ directive {attr="value"}* %>
```

引导命令并不将任何东西传给 out 流，但在设置你的 JSP 页属性时，它们却是非常重要的。另外，它们要以 JSP 容器为基础。比如下面这一行：

```
<%@ page language="java" %>
```

它的意思是：JSP 页里采用的脚本语言是 Java。事实上，JSP 规范目前只提供了对 Java 的支持，它只描述了语言属性等于“Java”的那些脚本的含义。这一条引导命令的目的是为 JSP 技术带来更大的灵活性，便于将来的扩展。例如，假定以后想选择另一种语言——比如 Python（脚本编制的最佳选择之一）——那么（通过将 Java 技术对象模型揭示到脚本环境）那种语言必须提供对 Java 运行时间环境的支持。特别地，它需要支持前面定义的那些“隐式变量”、JavaBeans 属性以及公共方法等等。

这里最重要的是页引导命令。它定义了大量和具体页有关的属性，并将这些属性告诉给 JSP 容器。这些属性包括 language、extends、import、session、buffer、autoFlush、isThreadSafe、info 以及 errorPage 等等。例如：

```
<%@ page session=”true” import=”java.util.*” %>
```

这一行首先指出这个页要求加入一个 HTTP 会话。由于我们还没有设置语言引导命令，所以 JSP 容器会默认采用 Java，而且对于脚本语言变量 session 来说，它的类型也会默认为 javax.servlet.http.HttpSession。假如引导命令之前执行过，但又失败了，那么隐式变量 session 便没法子使用。假如 session 变量未经指定，那么它会默认为“true”。

import 属性对脚本环境可以使用的类型进行了说明。该属性在这儿的用法和它在 Java 里的用法是一模一样的——也就是说，是一个用逗号分隔的列表，其中包含了一系列标准的 import 表达式。这个列表是由转换过的 JSP 页产品来导入的，而且可在脚本环境中使用。同样地，当前只有在语言引导命令是“java”时，它才得到了定义。

15.4.3 JSP 脚本元素

用引导命令设置好脚本运行环境后，便可开始利用脚本语言元素了。JSP 1.1 共有三个这样的语言元素：declaration（声明）、scriptlet（脚本片）和 expression（表达式）。其中“声

明”用于声明元素；“脚本片”对应一个语句“片断”；而“表达式”更是不言而喻，它代表一个完整的语言表达式。在 JSP 中，每个脚本语言元素都必须用一个“<%”开头，每一个的语法如下：

```
<%! declaration %>
<%  scriptlet   %>
<%= expression  %>
```

其中，“<%!”、“<%”、“<%=”之后以及“%>”之前的空格可要可不要。

所有这些标记都以 XML 为基础；甚至可以这样说：“JSP 页能映射成一个 XML 文档！”对上述脚本元素来说，XML 中对应的语法则变成：

```
<jsp:declaration> declaration </jsp:declaration>
<jsp:scriptlet>   scriptlet   </jsp:scriptlet>
<jsp:expression>  expression  </jsp:expression>
```

除此以外，还有责备类型的注释可供选用：

```
<%-- jsp comment --%>
<!-- html comment -->
```

其中，第一种形式便于我们在 JSP 源码中添加专门的注释，在传给客户机的 HTML 中，它们不会以任何形式显示出来。当然，第二种注释形式并非 JSP 独有的——感觉非常眼熟，对吧？事实上，它就是标准的 HTML 注释！这里最有趣的地方是，我们可将 JSP 代码插到一条 HTML 注释里，而注释会在结果生成的页中显示出来，其中包括来自 JSP 代码的结果。

“声明”用于指出在一个 JSP 页里要用到脚本语言（当前只能是 Java）的哪些变量和方法。声明必须是一条完整的 Java 语句，而且不可在 out 流中产生任何输出。在下面的 Hello.jsp 例子中，对 loadTime、loadDate 和 hitCount 这三个变量的声明均为完整的 Java 语句，它们负责着对新变量的声明及初始化：

```
//:! c15:jsp:Hello.jsp
<%-- This JSP comment will not appear in the
generated html --%>
<%-- This is a JSP directive: --%>
<%@ page import="java.util.*" %>
<%-- These are declarations: --%>
<%!
    long loadTime= System.currentTimeMillis();
    Date loadDate = new Date();
    int hitCount = 0;
%>
<html><body>
<%-- The next several lines are the result of a
JSP expression inserted in the generated html;
the '=' indicates a JSP expression --%>
<H1>This page was loaded at <%= loadDate %> </H1>
```

```
<H1>Hello, world! It's <%= new Date() %></H1>
<H2>Here's an object: <%= new Object() %></H2>
<H2>This page has been up
<%= (System.currentTimeMillis()-loadTime)/1000 %>
seconds</H2>
<H3>Page has been accessed <%= ++hitCount %>
times since <%= loadDate %></H3>
<%-- A "scriptlet" that writes to the server
console and to the client page.
Note that the ';' is required: --%>
<%
   System.out.println("Goodbye");
   out.println("Cheerio");
%>
</body></html>
///:~
```

运行这个程序时，大家会发现在连续对该页发出的两次请求之间，loadTime、loadDate 以及 hitCount 变量的值均得到了保持。换句话说，它们显然是“字段”，而非“本地变量”。

在这个例子的末尾，一个“脚本片”（scriptlet）在 Web 服务器的控制台写上“Goodbye”，并在隐式 JspWriter 对象 out 中，写入“Cheerio”。在“脚本片”中，实际可以包含任何代码片断，只要它们是有效 Java 语句（的一部分）。“脚本片”会在对请求进行处理的时候得以执行。当指定 JSP 内的所有“脚本片”都按它们在 JSP 页内出现的顺序合并好之后，它们应该产生一条有效的语句，就象由标准的 Java 程序语言定义的那样。至于它们是否会在 out 流里产生输出，完全取决于“脚本片”里的代码。应该注意的是，“脚本片”由于能修改它们能“看到”的那些对象，所以有时也会产生不好的副作用。

在 Hello.jsp 的中部，大家可看到 JSP 表达式同 HTML 混合到一起使用的例子。这些表达式当然必须是完整的 Java 语句。这些表达式会被求值，结果会被强制转变成一个字串，然后发给 out。假如表达式的结果不能强行转换成字串，那么会立即产生一个 ClassCastException（类强制转型违例）。

15.4.4　提取字段和值

下例和“小服务程序”那一节的例子有着某些共通之处。首次请求访问一个页时，它会侦测到你当前还没有字段。然后，它会返回一个页，其中包含了一个表单——采用和小服务器例子相同的代码，只不过采用 JSP 格式。填好那些字段，并提交这个表单到相同的 JSP URL 之后，它会侦测到字段的存在，所以会把它们显示出来。这是一种更理想的技术，因为无论是包含了表单（要求用户填写）的页，还是用于那个页的响应代码，现在都可包含到同一个文件中。因此，我们的创建和维护工作都可以变得更简单。

```
//:! c15:jsp:DisplayFormData.jsp
<%-- Fetching the data from an HTML form. --%>
<%-- This JSP also generates the form. --%>
<%@ page import="java.util.*" %>
<html><body>
```

```
<H1>DisplayFormData</H1><H3>
<%
  Enumeration flds = request.getParameterNames();
  if(!flds.hasMoreElements()) { // No fields %>
    <form method="POST"
    action="DisplayFormData.jsp">
<%  for(int i = 0; i < 10; i++) {  %>
      Field<%=i%>: <input type="text" size="20"
      name="Field<%=i%>" value="Value<%=i%>"><br>
<%  } %>
    <INPUT TYPE=submit name=submit
    value="Submit"></form>
<%} else {
    while(flds.hasMoreElements()) {
      String field = (String)flds.nextElement();
      String value = request.getParameter(field);
%>
      <li><%= field %> = <%= value %></li>
<%  }
  } %>
</H3></body></html>
///:~
```

这个例子最有趣的地方在于它演示了“脚本片”(Scriptlet)代码如何同普通的 HTML 代码混合使用——你甚至可以在一个 Java 的 for 循环里生成 HTML。构建任何这样的表单时，这种技术都显得十分实用——因为否则的话，你就得不断地请求 HTML 代码，为服务器带来沉重负担。

15.4.5 JSP 页的属性和作用域

通过查阅小服务程序和 JSP 的 HTML 用户文档，大家可发现一些特性，它们专门用来报告与当前正在运行的小服务程序或 JSP 有关的信息。下面这个例子可显示出其中的一部分信息：

```
//:! c15:jsp:PageContext.jsp
<%--Viewing the attributes in the pageContext--%>
<%-- Note that you can include any amount of code
inside the scriptlet tags --%>
<%@ page import="java.util.*" %>
<html><body>
Servlet Name: <%= config.getServletName() %><br>
Servlet container supports servlet version:
<% out.print(application.getMajorVersion() + "."
+ application.getMinorVersion()); %><br>
<%
```

```
  session.setAttribute("My dog", "Ralph");
  for(int scope = 1; scope <= 4; scope++) {  %>
    <H3>Scope: <%= scope %> </H3>
<%  Enumeration e =
      pageContext.getAttributeNamesInScope(scope);
    while(e.hasMoreElements()) {
      out.println("\t<li>" +
        e.nextElement() + "</li>");
    }
  }
%>
</body></html>
///:~
```

该例同时还展示了如何同时运用嵌入式 HTML，以及向 out 写入，从而输出至最终的 HTML 页中。

产生的第一部分信息是小服务程序的名字。在最简单的情况下，它可能就是“JSP”——但具体要取决于你采用的到底是什么产品。也可以利用应用程序（application）对象，查找小服务程序容器当前的版本是什么。最后，在设置了一个会话属性之后，会显示出一个特定作用域内的“属性名”。不过在大多数 JSP 编程中，我们都不必经常性地使用作用域；它们之所以在这里显示出来，只是为了增加这个例子的趣味性。共有四种属性作用域，包括：page scope（页作用域，即作用域 1）、request scope（请求作用域，即作用域 2）、session scope（会话作用域，即作用域 3——在这儿，会话作用域中唯一能用的元素就是“My dog”，它紧挨在 for 循环的前面添加）以及 application scope（应用程序作用域，即作用域 4）。至于具体是哪个作用域，要取决于 ServletContext 对象。对于每种 Java 虚拟机（JVM）来说，都只能有一个“Web 应用程序”（Web application）；而对每个 Web 应用程序来说，又只能有一个 ServletContext。所谓“Web 应用程序”，是指一系列小服务程序和内容的集合，它们安装在服务器的 URL 名字空间的一个特定子集下，比如/catalog。对此，我们通常要用一个配置文件加以指定。在应用程序作用域内，大家会看到不同的对象，它们分别用于表示工作目录与临时目录的路径。

15.4.6　用 JSP 控制会话

“会话”（Session）的问题已在小服务程序以前的那一节里解释过了，它也能在 JSP 中使用。下面这个例子对 session 对象进行了实验，并允许我们自由更改一个会话的有效时间——过了这个时间，会话便无效了。

```
//:! c15:jsp:SessionObject.jsp
<%--Getting and setting session object values--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H3><li>This session was created at
<%= session.getCreationTime() %></li></H1>
<H3><li>Old MaxInactiveInterval =
  <%= session.getMaxInactiveInterval() %></li>
```

```
<% session.setMaxInactiveInterval(5); %>
<li>New MaxInactiveInterval=
  <%= session.getMaxInactiveInterval() %></li>
</H3>
<H2>If the session object "My dog" is
still around, this value will be non-null:<H2>
<H3><li>Session value for "My dog" =
<%= session.getAttribute("My dog") %></li></H3>
<%-- Now add the session object "My dog" --%>
<% session.setAttribute("My dog",
                 new String("Ralph")); %>
<H1>My dog's name is
<%= session.getAttribute("My dog") %></H1>
<%-- See if "My dog" wanders to another form --%>
<FORM TYPE=POST ACTION=SessionObject2.jsp>
<INPUT TYPE=submit name=submit
Value="Invalidate"></FORM>
<FORM TYPE=POST ACTION=SessionObject3.jsp>
<INPUT TYPE=submit name=submit
Value="Keep Around"></FORM>
</body></html>
///:~
```

session 对象是默认提供的，所以不再需要为它进行额外的编码。通过对 getID()、getCreationTime()以及 getMaxInactiveInterval()的调用，可显示出与这个会话对象有关的信息。

首次启动这个会话后，会看到一个初始化的 MaxInactiveInterval 值，比如 1800 秒（30 分钟）——这就是一个会话的“有效时间”或者“存活时间”。但具体的值应该取决 JSP/小服务程序的配置。当然，等那么长的时间才能看到效果是令人难以接受的。因此，我们将 MaxInactiveInterval 缩短为 5 秒钟。假如在 5 秒结束之前便刷新了这个页，就会看到：

```
Session value for "My dog" = Ralph
```

但假如再多等一段时间，则“Ralph”会变成 null（空）值。

为体验如何将会话信息带到其他页，同时也为了看到强制一个会话对象失效与让它简单地“超时”这两种做法的区别，我们还创建了另外两个 JSP。第一个（通过在 SessionObject.jsp 里按下“invalidate”按钮来访问）会读入会话信息，并明确地指示那个会话“失效”：

```
//:! c15:jsp:SessionObject2.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.invalidate(); %>
</body></html>
```

```
///:~
```

为体验它的效果，可刷新 SessionObject.jsp，然后马上点按“invalidate”（失效）按钮，从而转为访问 SessionObject2.jsp。在这个时候，大家看到的应该仍然是“Ralph”字样。然后，在 5 秒钟的超时设定还没过之前，及时刷新 SessionObject2.jsp。这时，会话会被强制失效，而且“Ralph”字样也会消失。

现在假如返回到 SessionObject.jsp 中，并刷新页面显示（从而得到一个新的 5 秒超时时间），然后按下“Keep Around”按钮，从而进入下一个页——SessionObject3.jsp。后者根本不会主动地让一个会话“失效”：

```
//:! c15:jsp:SessionObject3.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<FORM TYPE=POST ACTION=SessionObject.jsp>
<INPUT TYPE=submit name=submit Value="Return">
</FORM>
</body></html>
///:~
```

由于这个页不会让会话失效，所以“Ralph”字样会在那里一直显示出来——前提是每次在 5 秒的超时到期之前，你总是及时地刷新一下这个页。

15.4.7 创建和修改 Cookie

Cookie（小甜饼）也是在小服务程序之前的一节里讲解过的。同样地，JSP 再一次表现了它的过人之处。在其中使用 Cookie，甚至比在小服务程序里使用还要来得简单。下例对此进行了生动的说明，它将提取随请求一道传来的 Cookie，读取和修改其生存时间（即“截止期限”，或者 MaxAge），同时为传回去的响应付上一个新的 Cookie：

```
//:! c15:jsp:Cookies.jsp
<%--This program has different behaviors under
 different browsers! --%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<%
Cookie[] cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++) { %>
  Cookie name: <%= cookies[i].getName() %> <br>
  value: <%= cookies[i].getValue() %><br>
  Old max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
  <% cookies[i].setMaxAge(5); %>
```

```
  New max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
<% } %>
<%! int count = 0; int dcount = 0; %>
<% response.addCookie(new Cookie(
    "Bob" + count++, "Dog" + dcount++)); %>
</body></html>
///:~
```

由于每种浏览器保存 Cookie 的方式都是不同的，所以用不同的浏览器测试时，看到的结果也可能不同（并非一定不同，到你读到本书的时候，象这样的一些 Bug 也许已经修复了）。另外，假如关闭浏览器再重新启动它（而非只是访问一个不同的页，然后返回到 Cookies.jsp），也可能会看到不同的结果。注意使用会话对象（session）看起来要比直接使用 Cookie 更保险一些。

显示了会话标识符之后，随 request 对象来的 Cookie 数组中的每个元素都会显示出来，同时显示的还有它的 MaxAge。然后对这个 MaxAge 进行修改，并再次显示出来，以验证新值。最后，将一个新的 Cookie 添加到响应对象（response）。不过，你的浏览器也许会忽略这个 MaxAge；因此，有必要实验这个程序，并修改 MaxAge，看看它在不同浏览器下的行为。

15.4.8 JSP 总结

这一节仅仅是对 JSP 的一个简要介绍。不过，仅仅利用这里学到的知识（同时还有在本书其他地方学到的 Java 知识，以及自己对 HTML 本身的掌握），你已经可以通过 JSP 开始编写更加复杂的 Web 应用。JSP 的语法并非特别深奥或复杂，所以假如你理解了本节讲述的内容，便可试着通过实践，来消化和真正掌握 JSP 编程技术。至于更进一步的信息，可参考目前专门讲小服务程序的一些参考书，或者访问 java.sun.com 寻求帮助。

JSP 对你而言应该是一种比较重要的知识，即使你的目标仅仅是制作小服务程序。假如你以后遇到对一种小服务程序的特性不理解的情况，那么通常都能写一个 JSP 测试程序，来解决心中的疑难。这可比专门写一个小服务程序来测试容易并快捷得多。这部分是由于 JSP 允许你编写更少的代码，并可混合使用 HTML 和 Java 代码。但更重要的原因在于，一旦源代码发生改变，JSP 容器便帮你自动控制 JSP 的所有重新编译及重新装载操作，从而简化了我们的许多工作！

不过，大家也要注意在创建 JSP 的时候，要求你拥有比纯粹 Java 编程或制作一般 Web 页更高的水平。除此以外，对一个有问题的 JSP 页进行调试时，要比调试一个纯粹的 Java 程序麻烦得多，因为它的错误消息往往显得有点儿莫名其妙（至少目前是这样的）。当然，这一局面也许会随着开发系统的改进而发生改变。不过除了 JSP 之外，我们将来也许还会看到基于 Java 和 Web 构建起来的其他技术。它们也许会更易使用，更容易由一般的 Web 站点设计人员掌握和运用。

15.5 远程方法调用（RMI）

为通过网络执行其他机器上的代码，传统的方法不仅难以学习和掌握，也极易出错。思考这个问题最佳的方式是：某些对象正好位于另一台机器，我们可向它们发送一条消息，并获得返回结果，就象那些对象位于自己的本地机器一样。“远程方法调用”（RMI）采用的正

是这种抽象。本节将引导大家经历一些必要的步骤，创建自己的 RMI 对象。

15.5.1　远程接口

RMI 大量依赖接口来完成自己的工作。在需要创建一个远程对象的时候，我们通过传递一个接口来隐藏基层的实现。所以客户得到远程对象的一个引用时，它们真正得到的是接口引用。这个引用正好同一些本地的根代码连接，由后者负责通过网络通信。但我们并不关心这些事情，只需通过自己的接口引用发送消息即可。

创建一个远程接口时，必须遵守下列规则：

(1) 远程接口必须为 public 属性（不能有“封装访问”权限；也就是说，它不能是“友好的”）。否则，一旦客户试图装载一个实现了远程接口的远程对象，就会得到一个错误。

(2) 远程接口必须对 java.rmi.Remote 这个接口进行扩展。

(3) 除与应用程序本身有关的违例之外，远程接口中的每个方法都必须在自己的 throws 从句中声明 java.rmi.RemoteException。

(4) 作为参数或返回值传递的一个远程对象（不管是直接的，还是在本地对象中嵌入）必须声明为远程接口，不可声明为具体的实施类。

下面是一个简单的远程接口示例，它表示的是一个精确计时服务：

```
//: c15:rmi:PerfectTimeI.java
// The PerfectTime remote interface.
package c15.rmi;
import java.rmi.*;

interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
} ///:~
```

表面上与其他接口是类似的，但它对 Remote 进行了扩展，而且它的所有方法都会“掷”出 RemoteException（远程违例）。记住接口和它所有的方法都是“公共的”（public）。

15.5.2　远程接口的实施

服务器必须包含一个扩展了 UnicastRemoteObject 的类，并实现远程接口。这个类也可以含有附加的方法，但客户只能使用远程接口中的方法。这是显然的，因为客户得到的只是指向接口的一个引用，而非实现它的那个类。

必须为远程对象明确定义构造函数，即使只准备定义一个默认构造函数，用它调用基类构造函数。必须把它明确地编写出来，因为它必须“掷”出 RemoteException 违例。

下面列出远程接口 PerfectTime 的实现：

```
//: c15:rmi:PerfectTime.java
// The implementation of
// the PerfectTime remote object.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
```

```
import java.net.*;

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
  // Implementation of the interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Must implement constructor
  // to throw RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Called automatically
  }
  // Registration for RMI serving. Throw
  // exceptions out to the console.
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
    new RMISecurityManager());
    PerfectTime pt = new PerfectTime();
    Naming.bind(
      "//peppy:2005/PerfectTime", pt);
    System.out.println("Ready to do time");
  }
} ///:~
```

在这里，main()控制着对服务器进行设置的全部细节。保存 RMI 对象时，必须在程序的某个地方采取下述操作：

(1) 创建和安装一个安全管理器，令其支持 RMI。作为 Java 发行包的一部分，唯一适用于 RMI 的是 RMISecurityManager。

(2) 创建远程对象的一个或多个实例。在这里，大家可看到创建的是 PerfectTime 对象。

(3) 向 RMI 远程对象注册表注册至少一个远程对象。一个远程对象拥有的方法可生成指向其他远程对象的引用。这样一来，客户只需到注册表里访问一次，得到第一个远程对象即可。

1. 设置注册表

在这儿，大家可看到对静态方法 Naming.bind()的一个调用。然而，这个调用要求注册表作为计算机上的一个独立进程运行。注册表服务器的名字是 rmiregistry。在 32 位 Windows 环境中，可使用：

```
start rmiregistry
```

令其在后台运行。在 Unix 中，则使用：

```
rmiregistry &
```

和许多网络程序一样，rmiregistry 位于机器启动它所在的某个 IP 地址处，但它也必须监视一个端口。如果象上面那样调用 rmiregistry，不使用参数，注册表的端口就会默认为1099。若希望它位于其他某个端口，只需在命令行添加一个参数，指定那个端口编号即可。对这个例子来说，端口将位于 2005，所以 rmiregistry 应该象下面这样启动（对于 32 位 Windows）：

```
start rmiregistry 2005
```

对于 Unix，则使用下述命令：

```
rmiregistry 2005 &
```

与端口有关的信息必须传送给 bind()命令，同时传送的还有注册表所在的那台机器的 IP 地址。但假若我们想在本地测试 RMI 程序，就象本章的网络程序一直测试的那样，这样做就会带来问题。在 JDK 1.1.1 版本中，存在着下述两方面的问题[72]：

(1) localhost 不能随 RMI 工作。所以为了在单独一台机器上完成对 RMI 的测试，必须提供机器的名字。为了在 32 位 Windows 环境中调查自己机器的名字，可进入控制面板，选择“网络”，选择“标识”卡片，其中应该列出了你的计算机的名字。就我自己的情况来说，我的机器叫作“Peppy”。不过，似乎大写形式会被忽略。

(2) 除非计算机有一个活动的 TCP/IP 连接，否则 RMI 不能工作，即使所有组件都只需要在本地机器里互相通信。这意味着在试图运行程序之前，必须连接到自己的 ISP（因特网服务供应商），否则会得到一些稀奇古怪的违例消息。

考虑到这些因素，bind()命令变成了下面这个样子：

```
Naming.bind("//peppy:2005/PerfectTime", pt);
```

若使用默认端口 1099，就没有必要指定一个端口，所以可以使用：

```
Naming.bind("//peppy/PerfectTime", pt);
```

通过去掉 IP 地址，只使用标识符，你应该能够执行本地测试：

```
Naming.bind("PerfectTime", pt);
```

服务名是随意的；它在这里只是恰巧为 PerfectTime，看起来和类名一样。但是，你完全可以根据情况任意修改。最重要的是确保它在注册表里是个独一无二的名字，以便客户正常地获取远程对象。若这个名字已在注册表里了，就会得到一个 AlreadyBoundException 违例。为防止这个问题，可考虑坚持使用 rebind()，放弃 bind()。这是由于 rebind()要么会添加一个新条目，要么将同名的条目替换掉。

尽管 main()退出，我们的对象已经创建并注册，所以会由注册表一直保持活动状态，等候客户到达并发出对它的请求。只要 rmiregistry 处于运行状态，而且我们没有为名字调用 Naming.unbind()方法，对象就肯定位于那个地方。考虑到这个原因，在我们设计自己的代码时，需要先关闭 rmiregistry，并在编译远程对象的一个新版本时重新启动它。

并不一定要将 rmiregistry 作为一个外部进程启动。若事前知道自己的是要求用以注册表的唯一一个应用，就可在程序内部启动它，使用下述代码：

```
LocateRegistry.createRegistry(2005);
```

和前面一样，2005 代表我们在这个例子里选用的端口号。这等价于在命令行执行 rmiregistry 2005。但在设计 RMI 代码时，这种做法往往显得更加方便，因为它取消了启动

[72] 为找到这些资料，我不知损伤了多少个脑细胞。

和中止注册表所需的额外步骤。一旦执行完这个代码，就可象以前一样使用 Naming 进行“绑定”——bind()。

15.5.3 创建根与干

若编译和运行 PerfectTime.java，即使 rmiregistry 正确运行，它也无法工作。这是由于 RMI 的框架尚未就位。首先必须创建根和干（stubs and skeletons），以便提供网络连接操作，并使我们将远程对象伪装成自己机器内的某个本地对象。

所有这些幕后的工作都是相当复杂的。我们从远程对象传入、传出的任何对象都必须“implement Serializable”（如果想传递远程引用，而非整个对象，对象的参数就可以“implement Remote”）。因此可以想象，当根和干通过网络“汇集”了所有参数，并返回结果的时候，会自动进行序列化以及数据的重新装配。幸运的是，我们根本没必要了解这些方面的任何细节，但根和干却是必须创建的。一个简单的过程如下：在编译好的代码中调用 rmic，它会创建必需的一些文件。所以唯一要做的事情就是为编译过程新添一个步骤。

然而，rmic 工具与特定的封装和类路径有很大的关系。PerfectTime.java 位于 c15.rmi 这个封装内中，即使我们就在 PerfectTime.class 所在的那个目录下调用 rmic，rmic 也无法找到文件。这是由于它搜索的是类路径，并不关心“当前路径”。因此，我们必须同时指定类路径，就象下面这样：

```
rmic c15.rmi.PerfectTime
```

执行这个命令时，并不一定非要在包含了 PerfectTime.class 的目录中进行，但结果会置于当前目录。

若 rmic 成功运行，目录里就会多出两个新类：

```
PerfectTime_Stub.class
PerfectTime_Skel.class
```

它们分别对应根（Stub）和干（Skeleton）。现在，我们已准备好让服务器与客户机相互沟通了。

15.5.4 使用远程对象

RMI 全部的宗旨就是尽可能简化远程对象的使用。我们在客户程序中要做的唯一一件额外的事情就是查找并从服务器取回远程接口。自此以后，剩下的事情就是普通的 Java 编程：将消息发给对象！下面是运用了 PerfectTime 的一个程序：

```
//: c15:rmi:DisplayPerfectTime.java
// Uses remote object PerfectTime.
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
      new RMISecurityManager());
    PerfectTimeI t =
      (PerfectTimeI)Naming.lookup(
```

```
        "//peppy:2005/PerfectTime");
    for(int i = 0; i < 10; i++)
      System.out.println("Perfect time = " +
        t.getPerfectTime());
  }
} ///:~
```

ID 字串与那个用于向 Naming 注册对象的那个字串是相同的，第一部分指出了 URL 和端口号。由于我们准备使用一个 URL，所以也可以指定 Internet 上的一台机器。

从 Naming.lookup()返回的东西必须强制转型为远程接口，而不是强制转型为类。若强制转型为类，会得到一个违例。

在下述方法调用中：

```
t.getPerfectTime()
```

我们可看到一旦获得远程对象的引用，用它进行的编程和用本地对象进行的编程是非常相似的（仅有一个区别：远程方法会“掷”出一个 RemoteException 违例）。

15.6 CORBA

设计大型的分布式应用时，本章前述的方法可能还无法令人满意。举个例子来说，我们有时希望同那些老的数据仓库打交道，或者需要由一个服务器对象提供服务——而无论它的物理位置到底在哪里。在这些情况下，都要求使用“远程过程调用”（Remote Procedure Call，RPC）的某种形式，而且可能要求与具体的语言无关。这便是 CORBA 发挥用武之地的时候了！

CORBA 并非一种语言特性，它只是一种整合技术。准确地说，它应该只是一种规范，各个厂商根据它来实现与 CORBA 相容的整合产品。CORBA 目前由 OMG 负责开发、制订与维护。其宗旨是面向分布式的、与语言无关的对象互动，定义一个标准的框架。

利用 CORBA 提供的能力，我们可向 Java 与非 Java 对象发出远程过程调用，并可采用一种“与地点无关”的方式，同那些老系统沟通。针对图形化和非图形化的应用程序构建，Java 添加了相应的网络支持以及一套非常出色的面向对象程序设计语言。Java 和 OMG 对象模型相互间有着很好的对应（映射）关系。例如，无论 Java 还是 CORBA 都实现了“接口”的概念，并都采用了一个引用对象模型。

15.6.1 CORBA 基础

由 OMG 开发的“对象互动”规范通常称为“对象管理体系”（ObjectManagement Architecture，OMA）。OMA 定义了两个组件：“核心对象模型”和“OMA 引用体系”。其中，“核心对象模型”（Core Object Model）规范了对象、接口、操作等等基本概念（CORBA 是这个模型的改进版本）；而“OMA 引用体系”（OMA Reference Architecture）定义的是服务和机制一个基础架构，用于真正实现对象的“互动”（即“相互操作”）。在 OMA 引用体系中，包括“对象请求代理”（Object Request Broker，ORB）、“对象服务”（Object Services，也称作“CORBA 服务”）以及一些通用工具。

ORB 相当于一条“通信总线”。利用这条总线，一个对象可从其他对象那里请求服务——无论对方的物理位置到底在哪里。也就是说，尽管看起来好象是在客户代码中的一个普通方法调用，但它背后实际会涉及相当复杂的操作。首先，必须存在同服务器对象的一个连接。

而且为了创建一个连接，ORB 必须知道服务器的实现代码到底“藏”在哪里。建立好连接后，还必须对方法参数汇集到一块儿——也就是说，你得把它们转换成一个二进制位流，以便通过网络传送。其他必须发送的信息还包括：服务器的机器名、服务器进程以及服务器对象在那个进程里的标识。最后，所有这些信息要通过一个低级的电缆级协议发送出去。服务器收到信息后，对其进行解码，最后才能真正地执行调用。幸好，ORB 帮我们将所有这些复杂性都结结实实地隐藏起来。作为程序员，我们只需要直接调用远程对象的方法就可以了，具体操作和调用一个本地对象的方法一样简单！

目前并没有规范指出一个 ORB 核心该如何实现，但为了在不同厂家的 ORB 之间提供一种最起码的兼容性，OMG 定义了一系列服务，它们可通过标准接口加以访问。

1. CORBA 接口定义语言（IDL）

CORBA 的设计宗旨是“与语言无关”。也就是说，一个客户机对象可调用不同类的服务器对象的方法，无论对方是用哪种语言来实现的。当然，客户机对象事先必须知道由服务器对象揭示出来的方法名和签名。这时便需要用到 IDL。CORBA IDL 用于指定数据类型、属性、操作、接口以及其他许多东西，它最大的特点就是不依赖任何一种语言（与语言无关）。IDL 采用的语法类似于 C++或 Java。在这三种语言中，存在着一些通用的概念，但却采用了不同的“说法”。下表对此进行了总结：

CORBA IDL	Java	C++
模块	封装	命名空间
接口	接口	纯抽象类
方法	方法	成员函数

IDL 也支持“继承”的概念，而且和 C++一样，也采用冒号运算符来进行表示。首先，程序员需要编写一系列 IDL 描述，在其中指出已实现并由服务器及客户机使用的属性、方法与接口。随后，用一个由厂家提供的 IDL/Java 编译器对 IDL 进行编译。这种编译器会读取 IDL 源码，并生成相应的 Java 代码。

IDL 编译器是一种非常有用的工具：它不仅生成与 IDL 等价的 Java 源码，还会生成用于汇集方法参数及发出远程调用的代码。我们将后面这种代码称为“根和干”代码，它们通常用多个 Java 源码文件加以组织，并通常属于同一个 Java 封装的一部分。

2. 命名服务

“命名服务”属于 CORBA 的基本服务之一。CORBA 对象是通过一个“引用”来访问的——这种“引用”信息对我们人来说是根本看不懂的。但是，我们仍然可为引用指定一个字串名。这一操作就称为“将引用字串化”。在这期间，我们便要用到一个名为“命名服务”（Naming Service）的 OMA 组件，它专门负责字串到对象以及对象到字串的转换及映射。由于“命名服务”在这儿扮演了一个“电话簿”的角色，无底服务器还是客户机都可对它进行查询和改动，所以它必须作为一个单独的进程来运行。假如创建一个“对象到字串”的映射，那么这种操作叫作“绑定一个对象”；而假如删除刚才的映射，就叫作“撤消绑定”。另外，假如通过调查一个对象引用，得知了字串名，便把这一过程叫作“解析出名字”。

举个例子来说，在启动的时候，服务器应用程序可以创建一个服务器对象，将对象绑定到命名服务里，然后等候客户机发出请求。而对客户机来说，它首先需要获得一个服务器对象的“引用”，再解析出字串名，然后才能利用该引用去调用服务器里的实际对象。

同样地，尽管“命名服务”规范属于 CORBA 的一部分，但具体实现它的应用程序却是

由 ORB 厂商来提供的。因此，随着你选择的产品的不同，访问命名服务的方式也会有所区别。

15.6.2　一个例子

这里为大家准备的示范代码并不“完美”，因为不同的 ORB 会用不同的方式来访问 CORBA。因此，所有例子都应该是“与厂商有关”的。在下面的例子中，我们采用的是 JavaIDL——由 Sun 公司提供的一个免费产品，它提供非常初级的 ORB、一个命名服务以及一个 IDL→Java 编译器。除此以外，由于 Java 本身还非常年轻，而且在不断地进步，所以在各个 Java/CORBA 产品中，并非所有 CORBA 特性都包括全了的。

在此，我们打算实现一个服务器程序，在某台机器上运行它，向它查询当前的准确时间。另外，还要实现一个客户机程序，以便向服务器查询时间。尽管在这里，两个程序均是用 Java 来实现的，但你完全可换用不同的语言来做同样的事情（在实际应用中，恐怕经常都会遇到这种情况）。

1. 编写 IDL 源码

第一步当然是写一段 IDL 描述，指出都打算提供哪些服务。这个操作通常是由服务器程序员来完成的。对他来说，完全可以自由地挑选一种语言来实现服务器，只要其中提供了一个 CORBA IDL 编译工具。随后，IDL 文件可以发布给客户端的程序员，它充当着不同语言和程序员之间的一种“桥梁”作用。

下例列出了我们这个 ExactTime 服务器采用的 IDL 描述：

```
//: c15:corba:ExactTime.idl
//# You must install idltojava.exe from
//# java.sun.com and adjust the settings to use
//# your local C preprocessor in order to compile
//# This file. See docs at java.sun.com.
module remotetime {
   interface ExactTime {
      string getTime();
   };
}; ///:~
```

这是对 remotetime 命名空间内的 ExactTime 接口的一个声明。该接口仅包含了一个方法，用于以字串格式，返回当前时间。

2. 创建根和干

第二步是编译 IDL，创建 Java 的“根和干”代码，以便用它来实现客户端和服务器端程序。在 JavaIDL 产品中，用来干这件事情的工具叫作 idltojava：

```
idltojava remotetime.idl
```

这样便会同时为根和干都生成相应的代码。idltojava 会生成一个 Java“封装”，并在 IDL 模块 remotetime 之后命名。而且，生成的 Java 文件会放到 remotetime 子目录中。_ExactTimeImplBase.java 是我们的“干”，用于实现服务器对象；而_ExactTimeStub.java 是我们的“根”，用于客户机。在 ExactTime.java 中，有着 IDL 接口对应于 Java 的表达方式。同时，还用到了另两个支持文件（例如，有一个可方便我们进行命名服务的各种操作）。

3. 实现服务器和客户机

下面首先列出的是服务器端的代码。服务器对象是在 ExactTimeServer 类里实现的。RemoteTimeServer 是用来创建一个服务器对象的应用程序，它会向 ORB 注册，为对象引用指派一个名字，然后便什么事情也不用操心了——只需在那里等着客户机的请求就行：

```
//: c15:corba:RemoteTimeServer.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Server object implementation
class ExactTimeServer extends _ExactTimeImplBase {
  public String getTime(){
    return DateFormat.
        getTimeInstance(DateFormat.FULL).
          format(new Date(
              System.currentTimeMillis()));
  }
}

// Remote application implementation
public class RemoteTimeServer {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws Exception {
    // ORB creation and initialization:
    ORB orb = ORB.init(args, null);
    // Create the server object and register it:
    ExactTimeServer timeServerObjRef =
      new ExactTimeServer();
    orb.connect(timeServerObjRef);
    // Get the root naming context:
    org.omg.CORBA.Object objRef =
      orb.resolve_initial_references(
        "NameService");
    NamingContext ncRef =
      NamingContextHelper.narrow(objRef);
    // Assign a string name to the
    // object reference (binding):
    NameComponent nc =
      new NameComponent("ExactTime", "");
```

```
    NameComponent[] path = { nc };
    ncRef.rebind(path, timeServerObjRef);
    // Wait for client requests:
    java.lang.Object sync =
      new java.lang.Object();
    synchronized(sync){
      sync.wait();
    }
  }
} ///:~
```

大家可以看到，服务器对象的实现是异常简单的——让一个普通的 Java 类从 IDL 编译器生成的“干”代码中继承就可以了。不过，在同 ORB 和其他 CORBA 服务沟通时，局面才稍微变得复杂起来。

4. 一些 CORBA 服务

针对上述程序，下面让我们对其中那些 JavaIDL 相关代码所做的事情进行一番简要说明（主要忽略了依具体厂商而定的部分 CORBA 代码）。main()的第一行用于启动 ORB，这是理所当然的，因为我们的服务器对象需要同它打交道。完成了对 ORB 的初始化，紧接着创建了一个服务器对象。实际上，它准确名字应该叫作“临时服务对象”。这种对象负责接收来自客户机的请求，而且它的“生存时间”和创建它的进程是一样的。“临时服务对象”创建好之后，便需要向 ORB 进行注册——让 ORB 知道自己的存在，继而可将相关的请求转发给它。

到目前为止，我们手上掌握的只有一个 timeServerObjRef——一个对象引用，而且只有当前的服务器进程才“知道”它。下一步，我们将为这个服务对象分配一个“字串化”的名字；以后，客户机需要用那个名字来定位实际的服务对象。为完成这个操作，我们需要使用“命名服务”。首先，我们需要获得指向“命名服务”的一个对象引用；在对 resolve_initial_references()的调用中，我们指定“命名服务”在 JavaIDL 中的字串化对象引用“NameService”。这样一为，它就会返回相应的一个对象引用。接着，利用 narrow()方法，我们将返回来的引用强制转型为一个特定的 NamingContext 引用。从此开始，便可正常使用命名服务了！

为了将前述的服务对象同一个字串化的对象引用绑定起来，我们首先创建一个 NameComponent 对象，将其初始化成“ExactTime”——亦即我们打算同服务对象绑定到一起的名字字串。随后，我们利用 rebind()方法来指定一个引用——即使它已经存在（另一方面，假如使用的是 bind()，那么在引用已经存在的情况下，它会产生一个违例）。在 CORBA 中，一个名字是由一系列 NameContext 构成的——这正是我们为什么要用一个数组将名字绑定到对象引用的原因。

到此为止，服务对象便作好了由客户机调用的一切准备。此时，服务器进程将进入一种等候状态。同样地，由于它是一个“临时服务对象”，所以其生存时间与服务器进程的生存时间是完全一致的。JavaIDL 目前尚未提供对“持久性对象”的支持——即使创建它的进程执行完毕，那种对象也会一直存在下去！

现在，大家已经知道了服务器代码的功用。接下来，再来看看客户机代码：

```
//: c15:corba:RemoteTimeClient.java
```

```
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class RemoteTimeClient {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws Exception {
    // ORB creation and initialization:
    ORB orb = ORB.init(args, null);
    // Get the root naming context:
    org.omg.CORBA.Object objRef =
      orb.resolve_initial_references(
        "NameService");
    NamingContext ncRef =
      NamingContextHelper.narrow(objRef);
    // Get (resolve) the stringified object
    // reference for the time server:
    NameComponent nc =
      new NameComponent("ExactTime", "");
    NameComponent[] path = { nc };
    ExactTime timeObjRef =
      ExactTimeHelper.narrow(
        ncRef.resolve(path));
    // Make requests to the server object:
    String exactTime = timeObjRef.getTime();
    System.out.println(exactTime);
  }
} ///:~
```

头几行做的事情和服务器进程差不多：初始化 ORB，并解析出对命名服务服务器的一个引用。接下来，我们需要获得服务对象的一个对象引用。为此，我们将字串化对象引用传给 resolve()方法，并用 narrow()，将结果强制转型成为 ExactTime 接口引用。最后，调用 getTime()，取得“准确时间”。

5. 激活命名服务进程

现在，我们已同时拥有了一个服务器和一个客户机应用程序，它们已作好了“互动”的准备。不过大家会注意到，两者均需要通过命名服务来绑定及解析字串化的对象引用。因此，在运行服务器或客户机之前，首先必须启动命名服务进程。在 JavaIDL 中，命名服务属于一种标准的 Java 应用程序，并配套提供了产品封装（但其他产品可能有所区别）。JavaIDL 命名服务在一个 JVM 实例的内部运行。在默认情况下，它会监听 900 这个网络端口。

6. 激活服务器和客户机

现在，我们可以启动服务器和客户机应用程序了（必须按此顺序，因为我们的服务器只

是临时的）。假如所有方面都已正确设置，那么在客户机的控制台窗口中，应该会看到一行输出，指出当前的时间。当然，这种结果本身当然并没有太大的吸引力。不过，大家至少应注意一件事情：即使在同一台物理性的机器上使用，客户机和服务器程序都会在不同的虚拟机内运行，而且通过一个基本的整合层（ORB 和命名服务），它们相互间能够正常地通信。

这只是一个简单的例子，设计时并没有面向真正的网络环境。但既然 ORB 通常都是“和地点无关”的，所以有没有网络似乎也无所谓。若服务器和客户机程序在不同的机器上运行，那么 ORB 能用一个叫作“实现仓库”（Implementation Repository）的组件，解析出远程字串化引用。尽管“实现仓库”属于 CORBA 的一部分，但它目前几乎毫无规范可言，所以不同的厂商有不同的做法。

15.6.3　Java 小程序和 CORBA

Java 小程序可采用 CORBA 客户机的形式来运行。这样一来，小程序就可访问那些表现为 CORBA 对象的远程信息及服务。但对小程序来说，它只能连接自己当初从中下载的那个服务器。因此，小程序与之打交道的所有 CORBA 对象也必须放在那个服务器上。但这样一来，便与 CORBA 当初的设计宗旨背道而驰了——没法子实现完全的“与地点无关”。

还有网络安全的问题。假如你在一个内部网上，那么一个办法是放宽浏览器的安全限制。另外，也可在连接外部服务器的时候，通过一道防火墙来保证安全。

有些 Java ORB 产品提供了解决这一问题的专利性方案。例如，有些产品采用了“HTTP 隧道”技术，另一些则集成了特殊的防火墙功能。

即便放在附录里讨论，这个主题也显得过于复杂了，所以我们不打算再去赘述。只是要提醒大家，它明显是你需要特别加以留意的！

15.6.4　CORBA 与 RMI 的对比

我们知道，CORBA 最主要的一个特点就是支持 RFC。这样一来，我们的本地对象就可调用由远程对象提供的方法。不过，正宗的 Java 已有一个类似的机制可做同样的事情，这就是 RMI——远程方法调用（参见第 15 章）。两者最主要的差别在于，RMI 只能算 RFC 的一个“子集”，因为只有在 Java 对象之间，它才能模拟 RFC 的一些功能；反之，RFC 则显得更加通用，因为在用不同语言实现的对象之间，它都能实现远程调用。

不过，RMI 可以调用位于远程机器上的、用非 Java 语言实现的服务。这时我们唯一要做的就是选用某种形式的封装器 Java 对象，用它将服务器端的非 Java 代码“包裹”在里面就可以了。对 Java 客户机来说，封装器对象是通过 RMI，从“外部”建立与它的连接的；而对非 Java 代码来说，封装器对象是通过前述的某种技术（如 JNI 或 J/Direct），从“内部”建立与它的连接的。

这种方式要求你自己动手编写某种“整合层”，而那正是 CORBA 已经帮你做好的一件事情——只不过，假如亲自动手，可免去在许多 ORB 产品中挑来挑去的麻烦。

15.7　企业 JavaBeans

[73]现在，让我们假定你需要开发一个多层应用程序，以便通过一个 Web 接口，观察与更新某个数据库中的记录。那么，到底该采取何种方案呢？可考虑用 JDBC 写一个数据库应用程序；用 JSP 和小服务程序写一个 Web 接口；或者用 CORBA/RMI 写一个分布式应用。但

[73] 本节内容主要由 Robert Castaneda 负责，同时还要感谢 Dave Bartlett 提供的帮助。

在开发一个分布式对象系统时，除了知道 API 之外，还有其他哪些因素是我们必须考虑在内的呢？下面便是答案：

■性能：我们创建的分布式对象必须性能良好，因为它们同时可能要为许多客户提供服务。也许要考虑采用某种优化技术（比如高速缓存），同时还要对数据库连接这样的资源进行暂存。另外，还必须对分布式对象的生存时间进行管理。

■扩展性：分布式对象也必须具有良好的“伸缩性”，易于扩展。在一个分布式应用中，“扩展性”意味着分布式对象的实例数量可以自由增加，并可移至其他机器上，同时用不着修改任何源码。

■安全性：分布式对象必须经常管理对其进行访问的那些客户机的身份验证和授权。在理想情况下，毋需任何重新编译，便应该完成新用户的添加和授权。

■分布式事务：分布式对象应该能“透明”地引用分布式事务。举个例子来说，假如目前有两个各自独立的数据库，那么应该在同一次事务处理中，同时完成对两个数据库的更新。而且假如未达到指定的标准，则恢复它们原先的状态。

■复用性：理想的分布式对象肯定能够复用”。应该毫无困难地把它们转移到另一个厂商的应用程序服务器里。理想情况下，应该能在不作任何特殊修改的前提下，将自己设计的分布式对象组件卖给别人。另外，也应该在不重新编译或改写的前提下，购买其他人制作的组件，并直接用它。

■可靠性：假如整个系统内的某台机器当掉，客户机应当自动放弃对它的请求，改为使用其他机器上运行的对象备份。

上面这些问题，再加上你必须解决的各种实际性事务问题，表面上会把它“压垮”。然而，除了你真正的事务问题，其他所有问题都是多余的——针对每个分布式事务应用，你设计的事务方案必须能够复用。

Sun 以及其他主流分布式对象厂家都意识到了这一点。他们认为在不久之后，每个开发队伍都会重新用到以前设计过的特定方案，所以它们创建了“企业 JavaBeans”（EJB）规范。EJB 描述的是一个服务器端的组件模型，它采用一种标准方式，解决上面要考虑的所有问题。对开发者来说，可自行创建名为 EJB 的事务组件，令其从底层的“支撑”代码分离出来，将全部的注意力都放在提供事务逻辑上。由于 EJB 是用标准方式定义的，所以“与厂商无关”。

15.7.1 JavaBeans 与 EJB 的对比

由于在名字上颇为相似（都是一种“咖啡豆”），所以许多人都并非十分清楚 JavaBeans 组件模型和企业 JavaBeans 规范的关系。事实上，尽管 JavaBeans 和 EJB 规范的宗旨都是通过标准化方案，在各种不同的开发及展开工具之间，有效地实现代码的“复用”与“移植”，但各种方案要解决的实际是不同的问题。

在 JavaBeans 组件模型中，定义的标准是为了创建可复用的组件，这些组件通常在 IDE 开发工具中采用，而且它们通常（但并非绝对）都是“可视”组件。

企业 JavaBeans（EJB）则不同，规范定义的组件模型主要针对服务器端 Java 代码的开发。由于根据 EJB 的设计宗旨，它以后可能会在多种不同的服务器平台上运行（包括那些不支持可视组件的中心服务器），所以每一个“EJB”都不能利用象 AWT 或 Swing 这样的图形化库。

15.7.2 EJB 规范

EJB 规范描述了一种服务器端的组件模型。它总共定义了六种分担不同任务的人物角色，要求他们在开发及展开（前期配置）阶段执行不同的任务。同时，它还定义了系统组件。

这些角色将在一个分布式系统的开发、展开以及运行阶段用到。厂家、管理员以及开发者都各自有不同的“角色”，对各自的技术类别及工作领域进行划分。其中，厂家提供的是一个技术性框架，而开发者各自负责自己专业领域内的特定组件的创建（比如“财务”组件）。另外，同一个人也可兼任多种角色。EJB 规范中定义的各种角色在下表进行了详细总结：

角　　色	职　　责
EJB 供应商	负责创制可重复使用的 EJB 组件的开发商。这些组件必须打包到一个特殊的 JAR 文件中（ejb-jar 文件）
应用程序装配者	在一系列 ejb-jar 文件的基础上，创建和装配出需要的应用程序。其中包括自己亲手写一些应用程序，在其中利用 EJB 集合（小服务程序、JSP、Swing 等等）。
展开/配置人	从装配者和 Bean 供应商那里取得 ejb-jar 文件集合，并在一个运行时间环境（亦即一个或更多的 EJB 容器）中把它们配置好（展开）
EJB 容器/服务器供应商	提供一种运行时间环境及相关工具，用于配置、管理和运行 EJB 组件
系统管理员	管理不同的组件和服务，担保其正确配置和交互，同时保证系统处在正常运行状态。

15.7.3 EJB 的组件

EJB 组件是一些可重复利用的事务逻辑元素，严格根据 EJB 规范中定义的一系列标准及设计范式而设计。它还允许其他服务（比如安全保护、高速缓存以及分布式事务）采用组件的形式来运行。要由一家“EJB 供应商”负责 EJB 组件的开发。

1. EJB 容器和服务器

“EJB 容器”（EJB Container）是一种运行时间环境，其中包含了并运行着 EJB 组件，并为那些组件提供了一系列标准化服务。EJB 容器所履行的职责必须严格遵守规范的定义，从而保证产品的通用性。EJB 容器提供了最基础层次的 EJB“封装”，包括分布式事务、安全性、各个 Bean 的生存时间管理、缓存、多线程以及会话管理等等。必须由 EJB 容器供应商来负责 EJB 容器的提供。

“EJB 服务器”（EJB Server）被定义成一种“应用程序服务器”，其中包含并运行着一个或多个 EJB 容器。EJB 服务器供应商负责着 EJB 服务器的提供。通常，EJB 容器和 EJB 服务器是相同的。

2. Java 命名和目录接口（JNDI）

针对网络上的 EJB 组件和其他容器服务（比如事务处理），“Java 命名和目录接口”（Java Naming and Directory Interface，JNDI）在 EJB 中作为它们的一种“命名服务”来使用。JNDI 和其他命名及目录标准（比如 CORBA CosNaming）非常相似，而且实际可作为在那些标准顶部的一个“封装器”（或“包裹器”）来实现。

3. Java 事务 API 和 Java 事务服务（JTA/JTS）

JTA/JTS 在 EJB 中作为一种事务 API 使用。EJB 供应商可考虑通过 JTS 来创建事务代码——尽管 EJB 容器通常已在 EJB 中以一个“EJB 组件”的形式，实现了那些事务逻辑。而对负责产品配置（安装）的人来说，他们需要在配置期间定义好一个 EJB 组件的事务属性。EJB 容器负责对本地或分布式的事务进行控制。JTS 规范其实是 CORBA OTS（对象事务服务）在 Java 中的一种对应实现。

4. CORBA 和 RMI/IIOP

通过与 CORBA 协议的兼容，EJB 规范定义了如何同 CORBA 打交道（互动）。为达此目的，象 JTS 和 JNDI 这样的 EJB 服务需要映射至对应的 CORBA 服务，而且要在一种名为“IIOP”的协议顶部，实现 RMI。

在 EJB 中，对 CORBA 和 RMI/IIOP 的使用是在 EJB 容器里进行的，而且要由 EJB 容器供应商来负责。对于 EJB 组件来说，它并不知道、也不关心 EJB 容器里是否正在使用 CORBA 和 RMI/IIOP。这就意味着，EJB 供应商可编写自己的 EJB 组件，并把它们展开到任何 EJB 容器中去——不用担心当前正使用的是什么通信协议。

15.7.4 EJB 组件的构成

一个 EJB 组件由一系列单元构成，其中包括 Bean 本身、对一些接口的实现以及一个信息文件。注意所有东西都要“打包”到一起，放到一个特殊的 jar 文件中。

1. EJB

EJB 本身是一个 Java 类，由 EJB 供应商负责开发。它实现了一个 EJB 接口，并对实际用来“干活”的事务方法进行了实施。注意类本身并未实现任何身份验证、授权、多线程或者交易代码。

2. 主接口

你创建的每个 EJB 都必须有一个对应的 Home 接口（主接口）。该接口作为你的 EJB 的一位“代理”使用。客户机将通过它找寻到你的 EJB 的实例，或者新建那个 EJB 的一个实例。

3. 远程接口

远程接口本质上是一个 Java 接口，它反映了你的 EJB 希望“暴露”给外部世界的方法。远程接口扮演了同 CORBA IDL 接口差不多的一种角色。

4. 配置描述符

这是一个 XML 文件，包含了同你的 EJB 有关的信息。由于用的是 XML，所以配置人可以非常方便地更改 EJB 的属性。在配置描述符中，可供定义的属性包括：

■你的 EJB 需要的主和远程接口名
■要在 JNDI 中为 EJB 主接口注册的名字
■EJB 每个方法的事务属性
■用于身份验证的访问控制

5. ejb-jar 文件

一个普通的 Java JAR 文件，其中包含了你的 EJB、主和远程接口以及配置描述符。

15.7.5 EJB 的工作

最后拿到包含了 Bean 本身、主/远程接口以及配置描述符的一个 ejb-jar 文件后，便可将所有元素合并到一块儿。在这个过程中，大家可真正理解到为什么需要主和远程接口，以及 EJB 容器是如何使用它们的。

EJB 容器实现了在 ejb-jar 文件里的主和远程接口。如早先讲到的那样，主接口提供了用来创建和查找你的 EJB 的方法。也就是说，EJB 容器负责着对你的 EJB 的“生存时间”的

管理。正是由于存在这样的一条“迂回路线”，所以我们可恰当地采取一些优化措施。例如，5 个客户可能会通过主接口同时请求创建一个 EJB，但 EJB 容器实际上只需创建一个 EJB，然后让 5 位客户共同使用它。这是通过“远程接口”来做到的，它也是由 EJB 容器来实现的。对远程对象来说，它实际上相当于 EJB 的一个“代理”对象。

在主和远程接口的帮助下，对 EJB 的所有调用都会通过 EJB 容器进行“代理”。正是由于采取了这样的“迂回路线”，所以 EJB 容器才能“从容”地控制安全及事务处理的行为。

15.7.6　EJB 的类型

EJB 规范定义了不同类型的 EJB，它们各自具有不同的特征和行为。在规范里，总共定义了两个大类的 EJB：会话 Bean 和实体 Bean。每个大类下面又分许多小类。

1. 会话 Bean

会话 Bean 用于从客户机的立场出发，表示“使用场景”（Use-Cases）或者“工作流”（Workflow）。它们代表的是对持久性数据采取的操作，而非代表数据本身。会话 Bean 下面又分两个小类:“无状态的”和“有状态的”。所有会话 Bean 都必须实现 javax.ejb.SessionBean 接口。EJB 容器控制着一个会话 Bean 的“生存时间”。

“无状态会话 Bean”是 EJB 组件最简单的一种类型。在不同的方法调用之间，它们并不维持与客户机进行的任何会话的状态。因此，在服务器那一端，它们极易复用。而且由于它们已被缓存下来，所以可根据需要进行自由扩充。使用无状态的会话 Bean 时，所有状态信息都必须在 EJB 的外部保存。

“有状态会话 Bean”则不同，在不同的调用之间，以前的状态必须保持下来。它们同客户机有着“一对一”的逻辑对应关系，而且能在它们自己内部维持状态。EJB 容器负责对此类会话 Bean 的聚集和缓存——这是通过“屏蔽与激活”（Passivation and Activation）机制来做到的。假如 EJB 容器发生崩溃，那么针对此类会话 Bean 的所有信息都会立即丢失。有些高级的 EJB 容器则提供了容错与恢复机制，可保障此类“有状态会话 Bean”的安全。

2. 实体 Bean

实体 Bean 代表的是持久性数据以及该数据的行为。这种 Bean 可在多个客户机之间共享，采用与共享数据库数据一样的方式。EJB 容器负责对实体 Bean 的缓存，并维持其完整性。实体 Bean 的“生存时间”延展到了 EJB 容器之外；换言之，假如 EJB 容器发生崩溃，那么容器下一次又恢复正常运行之后，实体 Bean 仍然会“活得好好的”。

根据“持久性”到底由谁来管理，实体 Bean 下面又分两个小类：“由容器管理的”和“由 Bean 管理的”。

“由容器管理的持久性”（Container Managed Persisitence，CMP）：CMP 实体 Bean 有其自己的持久性实现，由 EJB 容器进行管理。通过在配置描述符中指定的属性，EJB 容器会将实体 Bean 的属性映射至某些持久性仓库——这通常（但非绝对）是一个数据库。CMP 可缩短 EJB 的开发周期，同时可显著减少需要的代码量。

“由 Bean 管理的持久性”（Bean Managed Persistence，BMP）：BMP 实体 Bean 的持久性实现由 EJB 的供应商来提供。EJB 供应商需要负责用于新建一个 EJB 的逻辑的实现。除此以外，还要实现用于更新 EJB 属性、删除 EJB 以及查找 EJB 所需的逻辑。其间，通常要求编写恰当的 JDBC 代码，来实现同一个数据库或其他持久性数据仓库的沟通。通过 BMP，开发者可对实体 Bean 的持久性方案进行全面性的操控。

在不可能采用 CMP 来实现的前提下，BMP 也为我们赋予了更大的灵活性。举个例子来说，假如你想创建一个 EJB，并在其中封装一个老的大型机系统上的代码，，就必须利用

CORBA，自行控制持久性。

15.7.7 开发一个 EJB

作为一个例子，我们打算对前面 RMI 小节的“Perfect Time”例子进行修改，将其作为一个 EJB 组件来实现。注意这个例子是一个简单的“无状态会话 Bean”。

就象前面说过的那样，EJB 组件至少要由一个类（EJB 类）和两个接口（远程和主接口）构成。为 EJB 创建一个远程接口时，必须遵守下述规则：

(1) 远程接口必须是“公共的”（public）。

(2) 远程接口必须对 javax.ejb.EJBObject 接口进行扩展。

(3) 远程接口中的每个方法都必须在其 throws 从句中声明 java.rmi.RemoteException 违例；同时还要加上其他依你的应用程序而定的违例。

(4) 作为参数或返回值传递的任何对象（要么是直接传递的，要么是在一个本地对象内嵌入而传递的）都必须是一个有效的 RMI-IIOP 数据类型（包括其他 EJB 对象）。

下面是一个简单的远程接口，它用于 PerfectTime 这个 EJB：

```
//: c15:ejb:PerfectTime.java
//# You must install the J2EE Java Enterprise
//# Edition from java.sun.com and add j2ee.jar
//# to your CLASSPATH in order to compile
//# this file. See details at java.sun.com.
// Remote Interface of PerfectTimeBean
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTime extends EJBObject {
  public long getPerfectTime()
    throws RemoteException;
} ///:~
```

主接口是实际创建组件的地方。它可定义 create 方法，用于创建 EJB 的实例；或者定义 finder 方法，用于查找现成的 EJB（只能是实体 Bean）。要为一个 EJB 创建主接口，必须按下述规则行事：

(1) 主接口必须是“公共的”。

(2) 主接口必须对 javax.ejb.EJBHome 接口进行扩展。

(3) 主接口中创建的每个 create 方法都必须在其 throws 从句中声明 java.rmi.RemoteException；同时还要声明一个 javax.ejb.CreateException 违例。

(4) create 方法的返回值必须是一个远程接口。

(5) finder 方法的返回值（仅适用于实体 Bean）必须是一个远程接口、java.utilEnumeration 或者 java.util.Collection。

(6) 作为参数（参数）传送的任何对象（要么直接传送，要么嵌在一个本地对象内部传送）都必须是一个有效的 RMI-IIOP 数据类型（其中包括其他 EJB 对象）。

对主接口来说，标准的命名规范是先取得远程接口的名字，再在末尾加上“Home”字样。下面是用于 PerfectTime 这个 EJB 的主接口：

```
//: c15:ejb:PerfectTimeHome.java
// Home Interface of PerfectTimeBean.
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTimeHome extends EJBHome {
  public PerfectTime create()
    throws CreateException, RemoteException;
} ///:~
```

接下来，我们就可开始实现事务逻辑了。在你创建自己的 EJB 实现类时，必须遵守下述规则（注意你应当查阅 EJB 规范文档，熟悉一下当自己开发企业 JavaBeans 时，应该留意的所有规则）：

(1) 类必须是“公共的”。

(2) 类必须实现一个 EJB 接口（可为 javax.ejb.SessionBean 或者 javax.ejb.EntityBean）。

(3) 类应该定义一系列方法，它们直接对应远程接口中的各个方法。注意类本身并不实现远程接口；它只是对远程接口中的方法进行“镜像”，但不会产生 java.rmi.RemoteException 违例。

(4) 定义一个或者更多的 ejbCreate()方法，以初始化自己的 EJB。

(5) 所有方法的返回值和参数必须是一个有效的 RMI-IIOP 数据类型。

```
//: c15:ejb:PerfectTimeBean.java
// Simple Stateless Session Bean
// that returns current system time.
import java.rmi.*;
import javax.ejb.*;

public class PerfectTimeBean
  implements SessionBean {
  private SessionContext sessionContext;
  //return current time
  public long getPerfectTime() {
     return System.currentTimeMillis();
  }
  // EJB methods
  public void ejbCreate()
  throws CreateException {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void
  setSessionContext(SessionContext ctx) {
    sessionContext = ctx;
  }
```

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

由于这只是一个简单的例子，所以 EJB 方法（ejbCreate()、ejbRemove()、ejbActivate()和 ejbPassivate()）全部空置。这些方法会由 EJB 容器进行调用，并用于控制组件的状态。setSessionContext()方法会传递一个 javax.ejb.SessionContext 对象，其中包含了与组件的使用“场景”（Context）有关的信息，比如当前正在进行的事务处理，以及一些安全信息等等。

创建好 EJB 后，接着便要创建一个配置描述符。这实际上是一个 XML 文件，作用是对 EJB 组件进行描述。配置描述符应当保存在一个名叫 ejb-jar.xml 的文件中。

```
//:! c15:ejb:ejb-jar.xml
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 1.1//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <description>Example for Chapter 15</description>
  <display-name></display-name>
  <small-icon></small-icon>
  <large-icon></large-icon>
  <enterprise-beans>
    <session>
      <ejb-name>PerfectTime</ejb-name>
      <home>PerfectTimeHome</home>
      <remote>PerfectTime</remote>
      <ejb-class>PerfectTimeBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <ejb-client-jar></ejb-client-jar>
</ejb-jar>
///:~
```

从中可看到在<session>标记里定义的组件、远程接口以及主接口。配置描述符可利用一些 EJB 开发工具来自动生成。

根据 EJB 规范的规定，除标准的 ejb-jar.xml 配置描述符之外，各厂商专用的标让应当全部保存到一个单独的文件中。这样一来，在不同的组件和不同“品牌”的 EJB 容器之间，便可实现高度的移植能力。

文件必须打包（压缩）到一个标准的 Java Archive（JAR）文件中。配置描述符应当放到 JAR 文件内部压缩的/META-INF 子目录下。

在配置描述符中定义好 EJB 组件之后，作为“配置人”，便应该将 EJB 组件配置（展开）到 EJB 容器中。迄今本书完稿时为止，这种配置或展开过程已广泛采用了 GUI 图形界面的方式来进行，而且不同的 EJB 容器都有自己不同的做法。所以，我们在这里便不再多说了。

但对每个 EJB 容器来说，针对如何配置或展开一个 EJB，它们都应该提供了详细的文档说明。请自行参考。

由于 EJB 组件属于“分布式”对象，所以展开过程还应创建一些由客户机使用的“干”，以便那些客户机调用 EJB 组件。这些“干”类应该放到客户机应用程序的类路径上。由于 EJB 组件可在 RMI-IIOP(CORBA)或 RMI-JRMP 的顶部实现，因此产生的“干”也会随着 EJB 容器的不同而发生变化——不过不管怎样，它们都是生成好的类，可由客户机直接使用。

一个客户机程序希望调用 EJB 时，首先必须在 JNDI 中查找 EJB 组件，并取得到那个 EJB 组件的一个主接口引用。随后，用主接口创建目标 EJB 的一个实例，拿给客户机使用。

在下例中，客户机上运行的是一个简单的 Java 程序（但请记住，你实际可以使用任何东西，包括小服务程序、JSP——甚至一个 CORBA 或 RMI 分布式对象）：

```
//: c15:ejb:PerfectTimeClient.java
// Client program for PerfectTimeBean

public class PerfectTimeClient {
public static void main(String[] args)
throws Exception {
  // Get a JNDI context using
  // the JNDI Naming service:
  javax.naming.Context context =
    new javax.naming.InitialContext();
  // Look up the home interface in the
  // JNDI Naming service:
  Object ref = context.lookup("perfectTime");
  // Cast the remote object to the home interface:
  PerfectTimeHome home = (PerfectTimeHome)
    javax.rmi.PortableRemoteObject.narrow(
      ref, PerfectTimeHome.class);
  // Create a remote object from the home interface:
  PerfectTime pt = home.create();
  // Invoke  getPerfectTime()
  System.out.println(
    "Perfect Time EJB invoked, time is: " +
    pt.getPerfectTime() );
  }
} ///:~
```

源码中的注释已对该程序进行了很好的解释。请注意在 Java 强制转型执行之前，我们先用 narrow()方法来执行某种形式对象强制转型。类似的操作也在 CORBA 中得到了采用。另外要注意的是，主对象在这里成为专门“制造”PerfectTime 的一个场所。

15.7.8　EJB 总结

“企业 JavaBeans”规范越来越趋于成熟和标准化，同时也针对分布式对象计算进行了最大程度的简化。它是 Java 2 企业版（J2EE）平台的一个主要构成部分，而且也得到了越

来越多的分布式对象团体的支持。目前已有许多相关工具可供选用，未来还会有更多出色的工具出台，它们都可在一定程度上帮助我们简化 EJB 组件的开发。

本节的综述性内容只提供了 EJB 的一个“入门指引”。欲了解 EJB 规范的详情，请访问官方的 EJB 主页（java.sun.com/products/ejb/），可在那里下载到最新的规范文档，同时还有 J2EE 对它的“参考实现”（公版）。有了这些知识储备以及拿到了称手的工具之后，便可着手开发和展开自己的 EJB 组件了！

15.8 Jini：分布式服务

本节[74]针对 Sun 公司的 Jini 技术进行了一番概述。我们简单讲解了 Jini 的优点和缺点，并向大家展示了如何利用 Jini 架构，在自己的分布式系统编程中引入更深层次的“抽象”，从而高效率地将传统的“网络编程”转变成“面向对象的编程”。

15.8.1 Jini 的由来

以前在设计操作系统的时候，人们都假定计算机有“三大件”：一个处理器（CPU）、一些内存以及一个磁盘。启动机器时，它做的第一件事情便是找寻磁盘。如果没找到，那么“Sorry，您就甭费神把我当计算机使了！”不过，随着时代的进步，“此计算机已非彼计算机”，计算机的面貌已发生了很大的变化。就拿嵌入式设备来说吧，它也有处理器、内存以及一个网络连接——只是没有磁盘。再说说手机——当你打开电源的时候，它做的第一件事情就是寻找当地的无线电话网络。这种硬件环境的复杂化——从“磁盘为中心”到“网络为中心”的巨大变化——促使我们必须对软件的组织加以变革。这便是设计 Jini 的目的。

Jini 对计算机架构进行重新思考的一种尝试，它赋予了“网络”足够的重要性，同时也照顾到了那些根本不需要磁盘的设备的需要——而这些设备（它们来自许多不同的厂家）往往需要通过一个网络实现相互间的沟通。就网络本身来说，它应该是变得得非常快的一种环境——设备和服务在上面频繁地添加、移去。Jini 为此提供了相应的机制，便于我们在网络上毫无阻碍地添加、移去和寻找特定的设备和服务。除此以外，Jini 还提供了一个完整的编程模型，程序员可以更方便地控制各种设备相互间的通信。

Jini 以 Java 语言、对象序列化以及 RMI 为基础创建，试图将面向对象程序设计的好处扩展到网络上。同时允许对象通过网络，从一个虚拟机传送到另一个虚拟机。它并不要求设备厂商象以前那样，在形形色色的所谓“网络协议”面前屈服，而是通过接口到对象的形式，让各个设备实现相互间的“交谈”。

15.8.2 什么是 Jini

Jini 代表着一系列 API 和网络协议，可利用它们构建和展开分布式系统，并采用“服务联盟”的形式加以组织。一个“服务”可以是网络上存在的任何东西，只要它能准备好执行一种有用的功能便成。硬件设备、软件、通信信道（甚至人自己）——都可以是“服务”。例如，一个 Jini 相容的磁盘驱动器可提供“存储”服务；一台 Jini 相容的打印机可提供“打印”服务；……等等。一个“服务联盟”便是由这样的一系列服务构成的，所有服务当前都可通过网络使用。一个“客户”（可以是一个程序、服务或者用户）利用这种“服务联盟”，便可达到自己的某个目的。

为执行一个任务，客户必须从“服务”那里寻求帮助。例如，一个客户程序可从一台数

[74] 本节素材由 Bill Venners 提供（www.artima.com）。

码相机提供的存储服务那里下载照片，然后把照片上传到由一个磁盘驱动器提供的持久性存储服务中，最后将照片的“按比例缩小图”打印到由彩色打印机提供的打印服务那里。在这个例子中，客户程序需要构建一个分布式系统，其中包括它自己、图像存储服务、持久性存储服务以及彩色打印服务。客户和这个分布式系统的服务联合在一起，便可完成既定的任务：从数码相机取出和保存照片，并打印所有照片的对比性缩小示意图。

之所以用了“联盟”这个词，还有一个原因是 Jini 眼中的网络并不牵涉到所谓的“中央集权”，毋需什么“中心认证”。由于没有一个服务是收费的，所以网络上所有可用的服务便形成了一个松散的“联盟”——一个人人平等的“对等网”！在 Jini 的运行时间架构中，只提供了让客户和服务相互找着对方的一种途径（通过检索服务，它保存着当前所有可用服务的一个目录）。服务相互找到对方后，便可把对方当作自己的“一份子”使用。客户及其请求的服务会自己执行自己的任务，整个过程与 Jini 运行时间架构是无关的。假如 Jini 检索服务发生崩溃，那么崩溃前已相互“认识”的分布式系统仍会继续自己的工作，不会受到崩溃的影响。Jini 甚至还制订了一个网络协议，以便在没法使用检索服务的时候，客户通过该协议寻找自己需要的服务。

15.8.3　Jini 怎样工作的

Jini 定义了一个“运行时间架构”（Run-time Infrastructure），它驻留在网络上，允许我们通过它添加、移去、寻找以及访问需要的服务。这个架构总共驻留在三个地方：网上的检索服务中；服务提供者那里（比如与 Jini 相容的设备）；以及客户那里。其中“检索服务”（Lookup Services）是所有以 Jini 为基础的系统采用的中央组织机制。一旦网上出现了一种新服务，它便需要向检索服务注册自己的“存在”。假如客户希望寻找某种服务来配合自己的任务，那么也应该向一个检索服务查询。

运行时间架构采用了一种建立在“网络”层基础上的协议，名为 discovery（发现）；同时采用了两种对象一级的协议，名为 join（加入）和 lookup（查找）。

通过“发现”，客户和服务可找到检索服务；“加入”使一种服务能在检索服务中注册自己；而“查找”允许客户在需要某种服务的时候进行查询。

15.8.4　发现过程

“发现”的工作原理是这样的：假定你有一个具有 Jini 功能的磁盘驱动器，它用于提供一种持久性的存储服务。一旦你将该驱动器连接到网络，它就会象一个已知的端口投放一个多波包，从而广播出一个“存在声明”。在这个存在声明中，包括了一个 IP 地址和一个端口号。利用这些信息，磁盘驱动器就可以联络一个检索服务了。

检索服务会监视已知的端口，侦察是否存在一个“存在声明”。当它接收到一个存在声明之后，就会打开并分析数据包。根据包内的信息，检索服务可判断出它是否应该联络这个数据包的发送者。如果答案是肯定的，它就会根据自包内提取出来的 IP 地址和端口号，建立一个直接的 TCP 连接，从而与发送者进行联络。利用 RMI，检索服务可通过网络向包的始发者发出名为“服务注册者”的一个对象。这个服务注册者对象的用途是简化以后同检索服务的通信。通过调用这个对象上的方法，声明包的发送者就可以针对检索服务执行加入和检索操作。在磁盘驱动器的例子中，检索服务会建立同磁盘驱动器的一个 TCP 连接，并向其发送一个服务注册者对象，利用该对象，磁盘驱动器就可以在“加入”过程中，向其注册自己的存储服务。

15.8.5　加入过程

服务提供者拿到一个服务注册者对象（这是发现操作的最终产品）之后，就可以开始执

行加入操作了——以便成为检索服务中注册的那个服务联盟的一部分。为了执行一次加入操作，服务提供者需要在服务注册者对象上调用 register()方法，同时将一个名为“服务项”的对象作为参数传递给它（服务项是指对服务进行描述的一系列对象）。register()方法会将服务项的一个拷贝发送给检索服务，最后将服务项保存在那里。

所有这些操作完成之后，服务提供者就完成了他的加入操作：其服务已在检索服务中注册完毕。

服务项是包含了若干个对象的一个容器。在这些对象中，包括一个所谓的“服务对象”，客户机可利用它与服务沟通。在服务项中，还可以包含任意数量的“属性”，它们可以为任意对象。一些可能的属性包括图标、为服务提供图形界面的类以及用于提供更多服务信息的对象等。

服务对象通常实现了一个或多个接口，通过这些接口，客户机就可以与服务打交道。举个例子来说，一个检索服务就是一个 Jini 服务，而且它的服务对象是服务注册者。在加入期间，由服务提供者调用的 register()方法会在 ServiceRegistrar 接口（它是 net.jini.core.lookup 封装的一个成员）中声明，所有的服务注册者对象都实现了这个接口。客户机和服务提供者通过服务注册者对象建立与检索服务的通信——这是通过调用 ServiceRegistrar 接口中实现的那些方法来完成的。

类似地，一个磁盘驱动器可提供一个服务对象，只要它实现了一些已知的存储服务接口。客户机可查找到由这个存储服务接口指定的磁盘驱动器，并建立同它的联系。

15.8.6 查找过程

服务通过加入操作向一个检索注册了自己的存在之后，对那个检索服务进行查询的客户机就可使用那个服务了。为了构建一个分布式系统，以便让一系列服务统一执行特定的任务，客户机必须定位并请求获得各个服务的帮助。为了找到一个服务，客户机需要通过一个名为“查找”的过程来向检索服务提出查询。

为了执行一次查找操作，客户机需要在一个服务注册者对象上调用 lookup()方法（客户机和服务供应者类似，也通过前述的“发现”过程获得一个服务注册者）。客户机需要以参数的形式，向 lookup()传递一个“服务模板”——这其实是一个特殊的对象，作为查询的搜索标准使用。在服务模板中，可能包含了对一个 Class 对象数组的引用。这些 Class 对象向检索服务指出：客户机需要的服务对象的 Java 类型是什么。服务模板也包含了一个“服务 ID”，它唯一性地标识了一种服务，同时指出了一系列属性——那些属性必须与服务提供者在服务项中上传的属性一一对应。在服务模板中，还可包含通配符，用于指示任何这些字段。例如，服务 ID 字段中的一个通配符可对应任何服务 ID。lookup()方法会将服务模板发给检索服务。随后，由后者执行查询，并向任何相符的服务对象回传一个零值。作为客户机，它会获得对相符的服务对象的一个引用——它是放在 lookup()方法的返回值里传回来的。

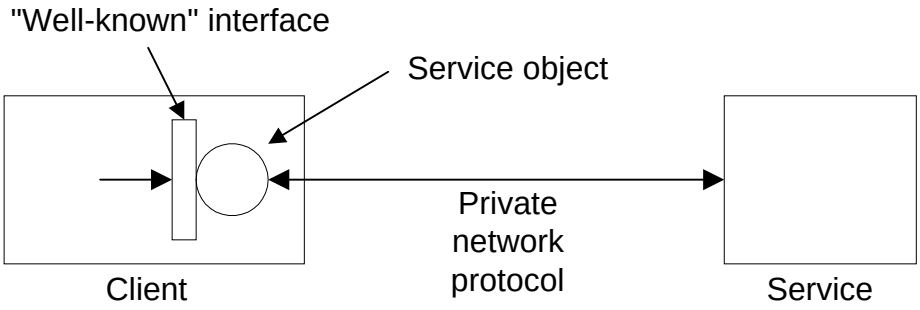
通常，客户机需要根据 Java 类型（一般是一个接口）来查找自己需要的一个服务。举个例子来说，假如客户机需要使用一台打印机，那么它会合成一个服务模板，并为到打印机服务的一个已知接口包括一个 Class 对象。所有打印机服务都实现了这个已知的接口。检索服务会返回实现了该接口的一个或多个服务对象。属性可包括到服务模板中，从而限制与搜索到的匹配项目的数量。客户机将通过从服务对象已知的打印机服务接口上调用方法，从而使用打印机服务。

15.8.7 接口与实现的分离

Jini 体系将面向对象的程序设计带到了网络世界，它使网络服务能够利用对象的一项基本属性：接口与实现的分离。举个例子来说，一个服务对象可授权客户机以多种方式访问服

务。对象实际代表着整个服务，它在查找过程中下载到客户机，然后在其本地执行。另外，服务对象可作一个纯粹的“代理”使用，为远程服务器提供服务。以后，一旦客户机调用服务对象上的某个方法，它会将请求通过网络发给服务器，再由后者执行实际的工作。第三种方式涉及到本地服务对象以及一个远程服务器，它们各自完成一半工作。

采用 Jini 体系之后，一个重要的结果就是：对于客户机来说，它并不需要知道在代理服务对象和远程服务器之间用于通信的网络协议是什么。如下图所示，网络协议属于服务本身实现的一部分。该协议完全是一个私人化的问题，由服务的开发者来决定。对客户机来说，它可通过这种私有协议同服务通信，因为服务会将自己的代码（服务对象）注入客户机的地址空间中。对于注入的服务对象来说，它可利用 RMI、CORBA、DCOM、在套接字/数据流顶部构建起来的专用协议以及其他任何东西，从而实现与服务的通信。客户机只是没必要关心网络协议，因为它可与服务对象实现的已知接口进行沟通。而服务对象又会照管到在网络上通信的其他一切细节。



（客户机通过一个已知接口同服务通信）

对于相同的服务接口来说，它的不同实现亦可能采用了迥异的方法及网络协议。一个服务可利用专门硬件来满足客户机的请求，或者干脆用软件来完成自己的工作。事实上，一个服务采用的实现方式也可能随时时间的推移而发生变化。客户机可保证自己能拿到一个服务对象使用，而那个对象肯定已理解了当前的服务实现细节，这是由于客户机本来就是从服务提供者那里拿到那个服务对象的（通过检索服务）。对客户机来说，一个服务看起来就是一个已知的接口，无论那个服务具体是如何实现的。

15.8.8　分布式系统的抽象

针对分布式系统编程，Jini 试图将对它的抽象级别从网络协议层提升到对象接口层。在那些连接到网络的各种嵌入式/手持式设备中，采用了由不同厂家设计的分布式系统组件。正是由于 Jini 的问世，所以设备厂家并不一定非要遵守某种网络层协议才能实现设备相互间的沟通。相反，它们只需同标准的 Java 接口打交道，便可经由它实现设备间的交互。利用 Jini 运行时间架构的发现、加入及查找机制，网络上的每个设备相互间都能“找到”对方。一旦找到，设备便通过 Java 接口进行通信。

15.9　总　　结

连同用于本地设备网络的 Jini 一道，本章向大家介绍了被 Sun 称为 J2EE（Java2 企业版）的一系列（但非全部）组件。J2EE 的宗旨是构建一系列工具，允许 Java 开发者更快地构建基于服务器的应用程序——而且采用一种“与平台无关”的形式。在传统意义上，假如想构建这样的一个应用程序，那么不仅很难、很花时间，而且也不大容易在能方便移植到其他平台的同时，又能让事务逻辑从实现的基层细节中分离出来。J2EE 正是针对这一系列问题的

一套完整解决方案。利用它所提供的框架，我们可创建出真正令人满意的服务器端应用程序——人们目前已经对这样的程序提出了迫切的要求，而且以后还会越来越迫切！

15.10 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 编译和运行本章中的 JabberServer 和 JabberClient 程序。接着编辑一下程序，删去为输入和输出设计的所有缓冲机制，然后再次编译和运行，观察一下结果。

(2) 创建一个服务器，用它请求用户输入密码，然后打开一个文件，并将文件通过网络连接传送出去。创建一个同该服务器连接的客户，为其分配适当的密码，然后捕获和保存文件。在自己的机器上用 localhost（通过调用 InetAddress.getByName(null)生成本地 IP 地址 127.0.0.1）测试这两个程序。

(3) 修改练习(2)中的程序，令其用多线程机制对多个客户进行控制。

(4) 修改 JabberClient，禁止输出刷新，并观察结果。

(5) 修改 MultiJabberServer，使其采用“线程池”。注意每次一个客户机断开连接之后，不再将线程简单地抛弃，而是把它放到一个“可用线程池”里面。一个新客户机希望连接时，服务器会检查这个池，并让其中的一个空闲线程来负责对该请求的处理。假如池内连一个可用的线程都没有，就新建一个线程。这样一来，需要的线程便会自然地增加到合理的数量。线程池的价值体现在不必为每个新的客户机都创建和破坏一个新线程，它免去了这些方面的开销。

(6) 以 ShowHTML.java 为基础，创建一个小程序，令其成为对自己 Web 站点的特定部分实施密码保护的大门。

(7) 修改 CIDCreateTables.java，使其从一个文本文件（而不是 CIDSQL）中读取 SQL 字串。

(8) 配置你的系统，以便能成功执行 CIDCreateTables.java 和 LoadDB.java。

(9) 对 ServletsRule.java 进行修改。具体做法是覆盖 destory()方法，将 i 值保存到一个文件里，并用 init()方法恢复旧值。通过重新启动小服务程序容器，从而证明自己的设计可行。假如还没有一个现成的小服务程序容器，可考虑从 jakarta.appache.org 下载、安装并运行 Tomcat。

(10) 创建一个小服务程序，令其在响应对象中添加一个 Cookie，从而将其保存到客户机上。在小服务程序里添加适当的代码，用于获取并显示 Cookie。假如还没有一个现成的小服务程序容器，可考虑从 jakarta.appache.org 下载、安装并运行 Tomcat。

(11) 创建一个小服务程序，令其用 Session 对象来保存自己选择的会话信息。在同一个小服务程序中，请获取并显示那些会话信息。假如还没有一个现成的小服务程序容器，可考虑从 jakarta.appache.org 下载、安装并运行 Tomcat。

(12) 创建一个小服务程序，令其将一个会话的失效周期变成 5 秒钟，这是通过调用 getMaxInactiveInterval()来实现的。请测试这个程序，证实会话在 5 秒钟之后确实会失效。假如还没有一个现成的小服务程序容器，可考虑从 jakarta.appache.org 下载、安装并运行 Tomcat。

(13) 创建一个 JSP 页，用<H1>标记打印一行文本。利用 JSP 页内嵌入的 Java 代码，将文本颜色设为随机值。假如还没有一个现成的 JSP 容器，可考虑从 jakarta.appache.org 下载、安装并运行 Tomcat。

(14) 修改 Cookies.jsp 中的最大存在时间值，并观察在两种不同的浏览器中，它的具体表现行为。同时也请注意简单地重新访问网页和关闭后再重启浏览器这两种做法之间的差异。假如还没有一个现成的 JSP 容器，可考虑从 jakarta.appache.org 下载、安装并运行 Tomcat。

(15) 创建一个 JSP，让用户在一个字段输入会话截止期限，另一个字段则用于容纳保存在会话中的数据。提交按钮可对页面进行刷新，并取得当前过期时间和会话数据，并把它们作为默认值放到前述的字段里。假如还没有一个现成的 JSP 容器，可考虑从 jakarta.appache.org 下载、安装并运行 Tomcat。

(16) （选做题）对 VLookup.java 程序作一番修改，使我们能点击得到的结果名字，然后程序会自动取得那个名字，并把它复制到剪贴板（以便我们方便地粘贴到自己的 E-mail）。可能需要回过头去研究一下第 13 章，回忆该如何使用 JFC 剪贴板。

附录 A 传递和返回对象

到目前为止，读者应对对象的“传递”有了一个较为深刻的认识，记住实际传递的只是一个引用。

在许多程序设计语言中，我们可用语言的“普通”方式到处传递对象，而且大多数时候都不会遇到问题。但有些时候却不得不采取一些非常做法，这便使情况突然变得有点儿复杂起来（在 C++中则会变得非常复杂）。Java 亦不例外，我们十分有必要准确认识在对象传递和赋值时所发生的一切。这正是本附录的宗旨。

若读者是从某些特殊的程序设计环境中转移过来的，那么一般都会问到：“Java 有指针吗？”有些人认为指针的操作很困难，而且十分危险，所以一厢情愿地认为它没什么好处。同时由于 Java 有如此好的口碑，所以应该很轻易地免除自己以前编程中的麻烦，其中不可能夹带有指针这样的“危险品”。但令人“遗憾”的是，从严格的技术角度说，Java 实际是有指针的！事实上，Java 中每个对象（除原始数据类型以外）的标识符都属于指针的一种。不过，它们的使用已受到了严格的限制和防范，不仅编译器对它们有“戒心”，运行时间系统也不例外。或者换从另一个角度说，Java 有指针，但没有指针算法。我曾一度将这种指针叫做“引用”，但你可以把它想像成“安全指针”。和预备学校为学生提供的安全剪刀类似——除非故意，否则是不可能伤着自己的。只不过，你有时得慢慢来，要先习惯一些沉闷的工作。

A.1 传递引用

将引用传递进入一个方法时，指向的仍然是相同的对象。一个简单的实验可以证明这一点：

```
//: appendixa:PassReferences.java
// Passing references around.

public class PassReferences {
  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);
  }
} ///:~
```

toString()方法会在打印语句里自动调用，而 PassHandles 直接从 Object 继承，不会重新定义 toString()。因此，这里采用的是 toString()的 Object 版本，它会打印出对象的类，接着

打印出那个对象所在的地址（不是引用，而是对象的实际存储位置）。输出结果象下面这样：

```
p inside main(): PassReferences@1653748
h inside f(): PassReferences@1653748
```

可以看到，无论 p 还是 h 引用的都是同一个对象。这比复制一个新的 PassHandles 对象有效多了，使我们能将一个参数发给一个方法。但这样做也带来了另一个重要的问题：别名！

A.1.1　别名问题

“别名”意味着多个引用都试图指向同一个对象，就象前面的例子展示的那样。若有人向那个对象里写入一点什么东西，就会产生别名问题。假如其他引用的所有者并不希望那个对象改变，看到这一点就会感动非常“惊讶”。可用下面这个简单的例子说明：

```
//: appendixa:Alias1.java
// Aliasing two references to one object.

public class Alias1 {
  int i;
  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);
  }
} ///:~
```

对下面这一行代码来说：

```
Alias1 y = x; // Assign the reference
```

它会新建一个 Alias1 引用，但不是把它分配给由 new 创建的一个新对象，而是分配给一个现有的引用。所以引用 x 的内容——即对象 x 指向的地址——被分配给 y，所以无论 x 还是 y 都与相同的对象连接起来。这样一来，一旦 x 的 i 在下述语句中增值：

```
x.i++;
```

y 的 i 值也必然受到影响。从最终的输出就可看出：

```
x: 7
y: 7
Incrementing x
x: 8
y: 8
```

此时最直接的一个解决办法就是干脆不这样做：不要有意将多个引用指向同一作用域内的同一个对象。这样做可使代码更易理解和调试。然而，一旦准备将引用作为一个参数或参数传递——这是 Java 设想的正常方法——别名问题就会自动出现，因为创建的本地引用可能修改“外部对象”（在方法作用域之外创建的对象）。下面是一个例子：

```
//: appendixa:Alias2.java
// Method calls implicitly alias their
// arguments.

public class Alias2 {
  int i;
  Alias2(int ii) { i = ii; }
  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);
  }
} ///:~
```

输出如下：

```
x: 7
Calling f(x)
x: 8
```

方法改变了自己的参数——外部对象。一旦遇到这种情况，必须判断它是否合理，用户是否愿意这样，以及是不是会造成问题。

通常，我们调用一个方法是为了产生返回值，或者用它改变为其调用方法的那个对象的状态（方法其实就是我们向那个对象“发一条消息”的方式）。很少需要调用一个方法来处理它的参数；这叫作利用方法的“副作用”（Side Effect）。所以倘若创建一个会修改自己参数的方法，必须向用户明确地指出这一情况，并警告使用那个方法可能会有的后果以及它的潜在威胁。由于存在这些混淆和缺陷，所以应该尽量避免改变参数。

若需在一个方法调用期间修改一个参数，且不打算修改外部参数，就应在自己的方法内部制作一个副本，从而保护那个参数。本附录的大多数内容都是围绕这个问题展开的。

A.2 制作本地副本

先稍微总结一下：Java 中的所有参数或参数传递都是通过传递引用进行的。也就是说，

当我们传递“一个对象”时，实际传递的只是对方法外部的那个对象的“一个引用”。所以一旦对那个引用进行了任何修改，便相当于修改了外部对象。此外：

■参数传递过程中会自动产生别名问题

■没有本地对象，只有本地引用

■引用有作用域，而对象没有

■对象的“存在时间”在 Java 里不是个问题

■没有语言上的支持（如“常数”）可防止对象被修改（以避免别名的副作用）

若只是从对象中读取信息，而不修改它，传递引用便是参数传递中最有效的一种形式。这种做非常恰当；默认的方法一般也是最有效的方法。然而，有时仍需将对象当作“本地的”对待，使我们作出的改变只影响一个本地副本，不会对外面的对象造成影响。许多程序设计语言都支持在方法内自动生成外部对象的一个本地副本[75]。尽管 Java 不具备这种能力，但允许我们达到同样的效果。

A.2.1　按值传递

首先来解决一下术语的问题，最适合“按值传递”的看起来是参数（参数）。“按值传递”以及它的含义取决于如何理解程序的运行方式。最常见的意思是获得要传递的任何东西的一个本地副本，但这里真正的问题是如何看待自己准备传递的东西。对于“按值传递”的含义，目前存在两种存在明显区别的见解：

(1) Java 按值传递任何东西。若将原始数据类型传递进入一个方法，会明确得到原始数据类型的一个副本。但若将一个引用传递进入方法，得到的是引用的副本。所以人们认为“一切”都按值传递。当然，这种说法也有一个前提：引用肯定也会被传递。但 Java 的设计方案似乎有些超前，允许我们忽略（大多数时候）自己处理的是一个引用。也就是说，它允许我们将引用假想成“对象”，因为在发出方法调用时，系统会自动照管两者间的差异。

(2) Java 主要按值传递（无参数），但对象却是按引用传递的。得到这个结论的前提是引用只是对象的一个“别名”，所以不考虑传递引用的问题，而是直接指出“我准备传递对象”。由于将其传递进入一个方法时没有获得对象的一个本地副本，所以对象显然不是按值传递的。Sun 公司似乎在某种程度上支持这一见解，因为它“保留但未实现”的关键字之一便是 byvalue（按值）。但没人知道那个关键字什么时候可以发挥作用。

尽管存在两种不同的见解，但其间的分歧归根到底是由于对“引用”的不同解释造成的。我打算在本书剩下的部分里回避这个问题。大家不久便会知道，再在这个问题纠缠下去其实是没有任何意义的——最重要的是理解一个引用的传递会造成调用者的对象发生意外的改变。

A.2.2　克隆对象

如果想修改一个对象，但又不想改变调用者的对象，就可考虑制作该对象的一个本地副本。这也是本地副本最常见的一种用途。若决定制作一个本地副本，只需简单地使用 clone() 方法即可。Clone 是“克隆”的意思，即制作一模一样的一个“变身”。这个方法在基类 Object 中定义成“protected”（受保护）模式。但在希望克隆的任何派生类中，必须将其覆盖为“public”模式。例如，由于标准库类 ArrayList 覆盖了 clone()，所以我们可以为 ArrayList 调用 clone()，如下所示：

---

[75] 在 C 语言中，通常控制的是少量数据位，默认操作是按值传递。C++也必须遵照这一形式，但按值传递对象并非肯定是一种有效的方式。此外，在 C++中用于支持按值传递的代码也较难写，是件令人头痛的事情。

```
//: appendixa:Cloning.java
// The clone() operation works for only a few
// items in the standard Java library.
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 {
  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);
  }
} ///:~
```

clone()方法产生了一个 Object，后者必须立即重新强制转型为正确类型。这个例子指出 ArrayList 的 clone()方法不会自动尝试克隆 ArrayList 内包含的每个对象——由于别名问题，老的 ArrayList 和克隆后的 ArrayList 都包含了相同的对象。我们通常把这种情况叫作“简单复制”或者“浅层复制”，因为它只复制了一个对象的“表面”部分。实际对象除包括了这个“表面”以外，还包括了引用指向的所有对象，以及那些对象又指向的其他所有对象……由此类推。这便是“对象网”或“对象关系网”的由来。只有复制整张网，才能叫作“全面复制”或者“深层复制”。

在输出中可看到浅层复制的结果，注意对 v2 采取的行动也会影响到 v：

```
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

一般地说，由于不敢保证 ArrayList 里包含的对象是“可克隆”[76]的，所以最好不要试

[76] “可克隆”的英语是“cloneable”，请注意 Java 库专门保留了这样的一个关键字。

图克隆那些对象。

A.2.3　使类具有克隆能力

尽管克隆方法是在所有类最基本的 Object 中定义的，但克隆仍然不会在每个类里自动进行[77]。这似乎有点儿不可思议，因为基类方法在派生类里是肯定能用的。但 Java 确实有点儿反其道而行之；如果想在一个类里使用克隆方法，唯一的办法就是专门添加一些代码，以保证克隆的正常进行。

1. 使用 protected 时的技巧

为避免我们创建的每个类都默认具有克隆能力，clone()方法在基类 Object 里得到了“保留”（设为 protected）。这样造成的后果就是：对那些简单地使用一下这个类的客户程序员来说，他们不会默认地拥有这个方法；其次，我们不能利用指向基类的一个引用来调用 clone()（尽管那样做在某些情况下特别有用，比如用多态的方式克隆一系列对象）。在编译时，这实际是通知我们对象不可克隆的一种方式——而且最奇怪的是，Java 库中的大多数类都不能克隆。因此，假如我们执行下述代码：

```
Integer x = new Integer(1);
x = x.clone();
```

那么在编译时，就有一条讨厌的错误消息弹出，告诉我们不可访问 clone()——因为 Integer 并没有覆盖它，而且它对 protected 版本来说是默认的。

但是，假若我们是在一个从 Object 派生出来的类中（所有类都是从 Object 派生的），就有权调用 Object.clone()，因为它是“受保护的”，而且我们是一个“继承者”。基类 clone()提供了一个有用的功能——它进行的是对派生类对象的真正“按位”复制，所以相当于标准的克隆行动。然而，我们随后需要将自己的克隆操作设为 public（公共的），否则便无法访问。总之，克隆时要注意的两个关键问题是：

■几乎肯定要调用 super.clone()

■必须将克隆设为 public

有时还想在更深层的派生类中覆盖 clone()，否则就会直接使用我们的 clone()（现在已成为“公共的”），而那并不一定是我们所希望的（然而，由于 Object.clone()已制作了实际对象的一个副本，所以也有可能允许这种情况）。protected 的技巧在这里只能用一次：首次从一个不具备克隆能力的类继承，而且想使一个类变成“能够克隆”。而在从我们的类继承的任何场合，clone()方法都是可以使用的，因为 Java 不可能在派生之后反而缩小方法的访问

---

[77] 为此，我们完全可以创建一个简单的计数器例子来加以证明，就像下面这样：

```
public class Cloneit implements Cloneable {
  public static void main (String[] args)
  throws CloneNotSupportedException {
    Cloneit a=new Cloneit();
    Cloneit b= new Cloneit();
  }
}
```

不过，上述代码之所以能够工作，完全是由于 main()属于 Cloneit 的一个方法，所以就有权调用 clone()这个“受保护”的基类方法。如果从一个不同的类中调用，该程序将无法编译。

范围。换言之，一旦对象变得可以克隆，从它派生的任何东西都是能够克隆的，除非使用特殊的机制（后面讨论）令其“关闭”克隆能力。

2. 实现 Cloneable 接口

为使一个对象的克隆能力功德圆满，还需要做另一件事情：实现 Cloneable 接口。这个接口会使人觉得有些奇怪，因为它是空的！

```
interface Cloneable {}
```

之所以要实现这个空接口，显然不是因为我们准备向上强制转型成一个 Cloneable，以及调用它的某个方法。有的人认为在这里使用接口属于一种“欺骗”行为，因为它打算使用的功能和其初衷是相悖的。Cloneable interface 的实现扮演了一个“标记”的角色，封装到类的类型中。

两方面的原因促成了 Cloneable interface 的存在。首先，可能有一个向上强制转型引用指向一个基类型，而且不知道它是否真的能克隆那个对象。在这种情况下，可用 instanceof 关键字（第 12 章有介绍）检查引用是否确实与一个能克隆的对象建立了连接：

```
if(myReference instanceof Cloneable) // ...
```

第二个原因是考虑到我们可能不愿所有对象类型都能克隆。所以 Object.clone()会验证一个类是否真的是实现了 Cloneable 接口。若答案是否定的，则“掷”出一个 CloneNotSupportedException 违例。所以在一般情况下，我们必须将“implement Cloneable”作为对克隆能力提供支持的一部分。

A.2.4 成功的克隆

理解了实现 clone()方法背后的所有细节后，便可创建出以后能方便复制的类，以便提供一个本地副本：

```
//: appendixa:LocalCopy.java
// Creating local copies with clone().
import java.util.*;

class MyObject implements Cloneable {
  int i;
  MyObject(int ii) { i = ii; }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("MyObject can't clone");
    }
    return o;
  }
  public String toString() {
    return Integer.toString(i);
  }
}
```

```
public class LocalCopy {
  static MyObject g(MyObject v) {
    // Passing a reference, modifies outside object:
    v.i++;
    return v;
  }
  static MyObject f(MyObject v) {
    v = (MyObject)v.clone(); // Local copy
    v.i++;
    return v;
  }
  public static void main(String[] args) {
    MyObject a = new MyObject(11);
    MyObject b = g(a);
    // Testing reference equivalence,
    // not object equivalence:
    if(a == b)
      System.out.println("a == b");
    else
      System.out.println("a != b");
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    MyObject c = new MyObject(47);
    MyObject d = f(c);
    if(c == d)
      System.out.println("c == d");
    else
      System.out.println("c != d");
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
} ///:~
```

不管怎样，clone()必须能够访问，所以必须将其设为 public（公共的）。其次，作为 clone()的初期行动，应调用 clone()的基类版本。这里调用的 clone()是 Object 内部预先定义好的。之所以能调用它，是由于它具有 protected（受到保护的）属性，所以能在派生的类里访问。

Object.clone()会检查原先的对象有多大，再为新对象腾出足够多的内存，将所有二进制位从原来的对象复制到新对象。这叫作“按位复制”，而且按一般的想法，这个工作应该是由 clone()方法来做的。但在 Object.clone()正式开始操作前，首先会检查一个类是否 Cloneable，即是否具有克隆能力——换言之，它是否实现了 Cloneable 接口。若未实现，Object.clone()就掷出一个 CloneNotSupportedException 违例，指出我们不能克隆它。因此，我们最好用一个 try-catch 块将对 super.clone()的调用代码包围（或封装）起来，试图捕获一个应当永不出现的违例（因为这里确实已实现了 Cloneable 接口）。

在 LocalCopy 中，两个方法 g()和 f()揭示出两种参数传递方法间的差异。其中，g()演示的是按引用传递，它会修改外部对象，并返回对那个外部对象的一个引用。而 f()是对参数进行克隆，所以将其分离出来，并让原来的对象保持独立。随后，它继续做它希望的事情。甚至能返回指向这个新对象的一个引用，而且不会对原来的对象产生任何副作用。注意下面这个多少有些古怪的语句：

```
v = (MyObject)v.clone();
```

它的作用正是创建一个本地副本。为避免被这样的一个语句搞混淆，记住这种非常奇怪的编码形式在 Java 中是完全允许的，因为有一个名字的任何东西实际都是一个引用。所以引用 v 用于克隆一个它所指向的副本，而且最终返回指向基类型 Object 的一个引用（因为它在 Object.clone()中是那样定义的），随后必须将其强制转型为正确的类型。

在 main()中，两种不同参数传递方式的区别在于它们分别测试了一个不同的方法。输出结果如下：

```
a == b
a = 12
b = 12
c != d
c = 47
d = 48
```

大家要记住这样一个事实：Java 对“是否等价”的测试并不对所比较对象的内部进行检查，从而核实它们的值是否相同。==和!=运算符只是简单地对比引用的内容。若引用内的地址相同，就认为引用指向同样的对象，所以认为它们是“等价”的。所以运算符真正检测的是：“由于别名问题，引用是否指向同一个对象？”

A.2.5 Object.clone()的效果

调用 Object.clone()时，实际发生的是什么事情呢？当我们在自己的类里覆盖 clone()时，什么东西对于 super.clone()来说是最关键的呢？根类中的 clone()方法负责建立正确的存储容量，并通过“按位复制”将二进制位从原始对象中复制到新对象的存储空间。也就是说，它并不只是预留存储空间以及复制一个对象——实际需要调查出欲复制之对象的准确大小，然后复制那个对象。由于所有这些工作都是在由根类定义之 clone()方法的内部代码中进行的（根类并不知道要从自己这里继承出去什么），所以大家或许已经猜到，这个过程需要用 RTTI 判断欲克隆的对象的实际大小。采取这种方式，clone()方法便可建立起正确数量的存储空间，并对那个类型进行正确的按位复制。

不管我们要做什么，克隆过程的第一个部分通常都应该是调用 super.clone()。通过进行一次准确的复制，这样做可为后续的克隆进程建立起一个良好的基础。随后，可采取另一些必要的操作，以完成最终的克隆。

为确切了解其他操作是什么，首先要正确理解 Object.clone()为我们带来了什么。特别地，它会自动克隆所有引用指向的目标吗？下面这个例子可完成这种形式的检测：

```
//: appendixa:Snake.java
// Tests cloning to see if destination
// of references are also cloned.
```

```
public class Snake implements Cloneable {
  private Snake next;
  private char c;
  // Value of i == number of segments
  Snake(int i, char x) {
    c = x;
    if(--i > 0)
      next = new Snake(i, (char)(x + 1));
  }
  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);
  }
} ///:~
```

一条 Snake（蛇）由数“段”构成，每一段的类型都是 Snake。所以，这是一个一段段链接起来的列表。所有段都是以循环方式创建的，每做好一段，都会使第一个构造函数参数的值递减，直至最终为零。而为给每段赋予一个独一无二的标记，第二个参数（一个 char）的值在每次循环构造函数调用时都会递增。

increment()方法的作用是循环递增每个标记，使我们能看出变化；而 toString 则循环打印出每个标记。输出如下：

```
s = :a:b:c:d:e
s2 = :a:b:c:d:e
after s.increment, s2 = :a:c:d:e:f
```

这意味着只有第一段才是由 Object.clone()复制的，所以此时进行的是一种“浅层复制”。若希望复制整条蛇——即进行“深层复制”——必须在被覆盖的 clone()里采取附加的操作。

通常可在从一个能克隆的类里调用 super.clone()，以确保所有基类行动（包括 Object.clone()）都能进行。随着是为对象内每个引用都明确调用一个 clone()；否则那些引用会别名变成原始对象的引用。构造函数的调用也大致相同——首先构造基类，然后是下一个派生的构造函数……以此类推，直到位于最深层的派生构造函数。区别在于 clone()并不是个构造函数，所以没办法实现自动克隆。为了克隆，必须由自己亲手进行。

A.2.6 克隆复合对象

试图深层复制复合对象时会遇到一个问题。必须假定成员对象中的 clone()方法也能依次对自己的引用进行深层复制，以此类推。这使我们的操作变得复杂。为了能正常实现深层复制，必须对所有类中的代码进行控制，或者至少全面掌握深层复制中需要涉及的类，确保它们自己的深层复制能正确进行。

下面这个例子总结了面对一个复合对象进行深层复制时需要做哪些事情：

```
//: appendixa:DeepCopy.java
// Cloning a composed object.

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(System.err);
    }
    return o;
  }
}

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(System.err);
    }
    return o;
  }
}

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(System.err);
    }
    // Must clone references:
    o.depth = (DepthReading)o.depth.clone();
    o.temperature =
      (TemperatureReading)o.temperature.clone();
    return o; // Upcasts back to Object
  }
}

public class DeepCopy {
  public static void main(String[] args) {
    OceanReading reading =
      new OceanReading(33.9, 100.5);
    // Now clone it:
    OceanReading r =
      (OceanReading)reading.clone();
```

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

其中，DepthReading 和 TemperatureReading 非常相似；它们都只包含了原始数据类型。所以 clone()方法能够非常简单：调用 super.clone()并返回结果即可。注意两个类使用的 clone()代码是完全一致的。

OceanReading 是由 DepthReading 和 TemperatureReading 对象合并而成的。为了对其进行深层复制，clone()必须同时克隆 OceanReading 内的引用。为达到这个目标，super.clone()的结果必须强制转型成一个 OceanReading 对象（以便访问 depth 和 temperature 引用）。

A.2.7 用 ArrayList 进行深层复制

下面让我们复习一下本附录早些时候提出的 ArrayList 例子。这一次 Int2 类是可以克隆的，所以能对 ArrayList 进行深层复制：

```
//: appendixa:AddingClone.java
// You must go through a few gyrations
// to add cloning to your own class.
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;
  }
}

// Once it's cloneable, inheritance
// doesn't remove cloneability:
class Int3 extends Int2 {
  private int j; // Automatically duplicated
  public Int3(int i) { super(i); }
}
```

```
public class AddingClone {
  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();
    // See if it changed v's elements:
    System.out.println("v: " + v);
    System.out.println("v2: " + v2);
  }
} ///:~
```

Int3 自 Int2 继承而来，并添加了一个新的原始数据类型成员：int j。大家也许认为自己需要再次覆盖 clone()，以确保 j 得到复制，但实情并非如此。将 Int2 的 clone()当作 Int3 的 clone()调用时，它会调用 Object.clone()，判断出当前操作的是 Int3，并复制 Int3 内的所有二进制位。只要没有新增需要克隆的引用，对 Object.clone()的一个调用就能完成所有必要的复制——无论 clone()是在层次结构多深的一级定义的。

至此，大家可以总结出对 ArrayList 进行深层复制的先决条件：在克隆了 ArrayList 后，必须在其中遍历，并克隆由 ArrayList 指向的每个对象。为了对 HashMap 进行深层复制，也必须采取类似的做法。

这个例子剩余的部分显示出克隆已实际进行——证据就是在克隆了对象以后，可以自由改变它，而原来那个对象不受任何影响。

A.2.8　通过序列化进行深层复制

若研究一下第 11 章介绍的那个 Java 对象序列化示例，可能发现若在一个对象序列化以后再撤消对它的序列化，那么经历的实际正是一个“克隆”的过程。

那么为什么不用序列化进行深层复制呢？下面这个例子通过计算执行时间对比了这两

种方法：

```
//: 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 {
  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 {
  static final int SIZE = 5000;
  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");
  }
} ///:~
```

其中，Thing2 和 Thing4 包含了成员对象，所以需要进行一些深层复制。一个有趣的地方是尽管 Serializable 类很容易设置，但在复制它们时却要做多得多的工作。克隆涉及到大量的类设置工作，但实际的对象复制是相当简单的。结果很好地说明了一切。下面是几次运行分别得到的结果：

```
Duplication via serialization: 940 Milliseconds
Duplication via cloning: 50 Milliseconds

Duplication via serialization: 710 Milliseconds
Duplication via cloning: 60 Milliseconds
```

```
Duplication via serialization: 770 Milliseconds
Duplication via cloning: 50 Milliseconds
```

除了序列化和克隆之间巨大的时间差异以外，我们也注意到序列化技术的运行结果并不稳定，而每一次克隆所花费的时间都是相同的，它显得更稳定。

A.2.9 使克隆具有更大的深度

若新建一个类，它的基类会默认为 Object，并默认为不具备克隆能力（就象在下一节会看到的那样）。只要不明确地添加克隆能力，这种能力便不会自动产生。但我们可以在任何层添加它，然后便可从那个层开始向下具有克隆能力。如下所示：

```
//: appendixa:HorrorFlick.java
// You can insert Cloneability
// at any level of inheritance.
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 InternalError();
    }
  }
}
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();
```

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

添加克隆能力之前，编译器会阻止我们的克隆尝试。一旦在 Scientist 里添加了克隆能力，那么 Scientist 以及它的所有“后裔”都可以克隆。

A.2.10　为什么有这个奇怪的设计

之所以感觉这个方案的奇特，因为它事实上的确如此。也许大家会奇怪它为什么要象这样运行，而该方案背后的真正含义是什么呢？后面讲述的是一个未获证实的故事——大概是由于围绕 Java 的许多买卖使其成为一种设计优良的语言——但确实要花许多口舌才能讲清楚这背后发生的所有事情。

最初，Java 只是作为一种用于控制硬件的语言而设计，与 Internet 并没有丝毫联系。象这样一类面向大众的语言一样，其意义在于程序员可以对任意一个对象进行克隆。这样一来，clone()就放置在根类 Object 里面，但因为它是一种公用方式，因而我们通常能够对任意一个对象进行克隆。看来这是最灵活的方式了，毕竟它不会带来任何害处。

正当 Java 看起来象一种终级 Internet 程序语言的时候，情况却发生了变化。突然地，人们提出了安全问题，而且理所当然，这些问题与使用对象有关，我们不愿望任何人克隆自己的保密对象。所以我们最后看到的是为原来那个简单、直观的方案添加的大量补丁：clone()在 Object 里被设置成“protected”。必须将其覆盖，并使用“implement Cloneable”，同时解决违例的问题。

只有在准备调用 Object 的 clone()方法时，才没必要使用 Cloneable 接口，因为那个方法会在运行期间得到检查，以确保我们的类实现了 Cloneable。但为保持一致（而且由于 Cloneable 无论如何都是空的），最好还是由自己实现 Cloneable。

A.3　克隆的控制

为消除克隆能力，大家也许认为只需将 clone()方法简单地设为 private（私有）即可，但这样是行不通的，因为不能采用一个基类方法，并使其在派生类中更“私有”。所以事情并没有这么简单。此外，我们有必要控制一个对象是否能够克隆。对于我们设计的一个类，实际有许多种方案都是可以采取的：

(1) 保持中立，不为克隆做任何事情。也就是说，尽管不可对我们的类克隆，但从它继承的一个类却可根据实际情况决定克隆。只有 Object.clone()要对类中的字段进行某些合理的操作时，才可以作这方面的决定。

(2) 支持 clone()，采用实现 Cloneable（可克隆）能力的标准操作，并覆盖 clone()。在被覆盖的 clone()中，可调用 super.clone()，并捕获所有违例（这样可使 clone()不“掷”出任何违例）。

(3) 有条件地支持克隆。若类容纳了其他对象的引用，而那些对象也许能够克隆（集合类便是这样的一个例子），就可试着克隆拥有对方引用的所有对象；如果它们“掷”出了违例，只需让这些违例通过即可。举个例子来说，假设有一个特殊的 Vector，它试图克隆自己容纳的所有对象。编写这样的一个 Vector 时，并不知道客户程序员会把什么形式的对象置入这个 Vector 中，所以并不知道它们是否真的能够克隆。

(4) 不实现 Cloneable()，但是将 clone()覆盖成 protected，使任何字段都具有正确的复制行为。这样一来，从这个类继承的所有东西都能覆盖 clone()，并调用 super.clone()来产生正确的复制行为。注意在我们实现里，可以而且应该调用 super.clone()——即使那个方法本来

预期的是一个 Cloneable 对象（否则会掷出一个违例），因为没有人会在我们这种类型的对象上直接调用它。它只有通过一个派生类调用；对那个派生类来说，如果要保证它正常工作，需实现 Cloneable。

(5) 不实现 Cloneable 来试着防止克隆，并覆盖 clone()，以产生一个违例。为使这一设想顺利实现，只有令从它派生出来的任何类都调用重新定义后的 clone()里的 suepr.clone()。

(6) 将类设为 final，从而防止克隆。若 clone()尚未被我们的任何一个上级类覆盖，这一设想便不会成功。若已被覆盖，那么再一次覆盖它，并“掷”出一个 CloneNotSupportedException（克隆不支持）违例。为担保克隆被禁止，将类设为 final 是唯一的办法。除此以外，一旦涉及保密对象或者遇到想对创建的对象数量进行控制的其他情况，应该将所有构造函数都设为 private，并提供一个或更多的特殊方法来创建对象。采用这种方式，这些方法就可以限制创建的对象数量以及它们的创建条件——一种特殊情况是第 16 章要介绍的 singleton（独子）方案。

下面这个例子总结了克隆的各种实现方法，然后在层次结构中将其“关闭”：

```
//: appendixa:CheckCloneable.java
// Checking to see if a reference can be cloned.

// 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);
  }
}

// Can't inherit from this, so can't override
// the clone method like in BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
  static Ordinary tryToClone(Ordinary ord) {
    String id = ord.getClass().getName();
    Ordinary x = null;
    if(ord instanceof Cloneable) {
      try {
        System.out.println("Attempting " + id);
        x = (Ordinary)((IsCloneable)ord).clone();
        System.out.println("Cloned " + id);
      } catch(CloneNotSupportedException e) {
        System.err.println("Could not clone "+id);
      }
    }
    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, since clone() is
    // protected in Object:
    //! x = (Ordinary)x.clone();
    // tryToClone() checks first to see if
    // a class implements Cloneable:
    for(int i = 0; i < ord.length; i++)
      tryToClone(ord[i]);
  }
} ///:~
```

第一个类 Ordinary 代表着大家在本书各处最常见到的类：不支持克隆，但在它正式应用以后，却也不禁止对其克隆。但假如有一个指向 Ordinary 对象的引用，而且那个对象可能是从一个更深的派生类向上强制转型来的，便不能判断它到底能不能克隆。

WrongClone 类揭示了实现克隆的一种不正确途径。它确实覆盖了 Object.clone()，并将那个方法设为 public，但却没有实现 Cloneable。所以一旦发出对 super.clone()的调用（由于对 Object.clone()的一个调用造成的），便会无情地掷出 CloneNotSupportedException 违例。

在 IsCloneable 中，大家看到的才是进行克隆的各种正确行动：先覆盖 clone()，并实现了 Cloneable。但是，这个 clone()方法以及本例的另外几个方法并不捕获 CloneNotSupportedException 违例，而是任由它通过，并传递给调用者。随后，调用者必须用一个 try-catch 代码块把它包围起来。在我们自己的 clone()方法中，通常需要在 clone()内部捕获 CloneNotSupportedException 违例，而不是任由它通过。正如大家以后会理解的那样，对这个例子来说，让它通过是最正确的做法。

类 NoMore 试图按照 Java 设计者打算的那样“关闭”克隆：在派生类 clone()中，我们掷出 CloneNotSupportedException 违例。TryMore 类中的 clone()方法正确地调用 super.clone()，并解析成 NoMore.clone()，后者掷出一个违例并禁止克隆。

但在已被覆盖的 clone()方法中，假若程序员不遵守调用 super.clone()的“正确”方法，又会出现什么情况呢？在 BackOn 中，大家可看到实际会发生什么。这个类用一个独立的方法 duplicate()制作当前对象的一个副本，并在 clone()内部调用这个方法，而不是调用 super.clone()。违例永远不会产生，而且新类是可以克隆的。因此，我们不能依赖“掷”出一个违例的方法来防止产生一个可克隆的类。唯一安全的方法在 ReallyNoMore 中得到了演示，它设为 final，所以不可继承。这意味着假如 clone()在 final 类中掷出了一个违例，便不能通过继承来进行修改，并可有效地禁止克隆（不能从一个拥有任意继承级数的类中明确调用 Object.clone()；只能调用 super.clone()，它只可访问直接基类）。因此，只要制作一些涉及安全问题的对象，就最好把那些类设为 final。

在类 CheckCloneable 中，我们看到的第一个类是 tryToClone()，它能接纳任何 Ordinary 对象，并用 instanceof 检查它是否能够克隆。若答案是肯定的，就将对象强制转型成为一个 IsCloneable，调用 clone()，并将结果强制转型回 Ordinary，最后捕获有可能产生的任何违例。请注意用运行时间类型鉴定（见第 12 章）打印出类名，使自己看到发生的一切情况。

在 main()中，我们创建了不同类型的 Ordinary 对象，并在数组定义中向上强制转型成为 Ordinary。在这之后的头两行代码创建了一个纯粹的 Ordinary 对象，并试图对其克隆。然而，这些代码不会得到编译，因为 clone()是 Object 中的一个 protected（受到保护的）方法。代码剩余的部分将遍历数组，并试着克隆每个对象，分别报告它们的成功或失败。输出如下：

```
Attempting IsCloneable
Cloned IsCloneable
Attempting NoMore
Could not clone NoMore
Attempting TryMore
Could not clone TryMore
Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Could not clone ReallyNoMore
```

总之，如果希望一个类能够克隆，那么你需要：

(1) 实现 Cloneable 接口

(2) 覆盖 clone()

(3) 在自己的 clone()中调用 super.clone()

(4) 在自己的 clone()中捕获违例

这一系列步骤能达到最理想的效果。

A.3.1　副本构造函数

克隆看起来要求进行非常复杂的设置，似乎还该有另一种替代方案。一个办法是制作特殊的构造函数，令其负责复制一个对象。在 C++中，这叫作“副本构造函数”。刚开始的时候，这好象是非常明显、非常自然的一种解决方案（如果你是 C++程序员，这个方法就更显亲切）。下面是一个实际的例子：

```
//: appendixa:CopyConstructor.java
// A constructor for copying an object of the same
// type, as an attempt to create a local copy.

class FruitQualities {
  private int weight;
  private int color;
  private int firmness;
  private int ripeness;
  private int smell;
  // etc.
```

```
  FruitQualities() { // Default constructor
    // do something meaningful...
  }
  // Other constructors:
  // ...
  // Copy constructor:
  FruitQualities(FruitQualities f) {
    weight = f.weight;
    color = f.color;
    firmness = f.firmness;
    ripeness = f.ripeness;
    smell = f.smell;
    // etc.
  }
}

class Seed {
  // Members...
  Seed() { /* Default constructor */ }
  Seed(Seed s) { /* Copy constructor */ }
}

class Fruit {
  private FruitQualities fq;
  private int seeds;
  private Seed[] s;
  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:
  Fruit(Fruit f) {
    fq = new FruitQualities(f.fq);
    seeds = f.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 {
  Tomato() {
    super(new FruitQualities(), 100);
  }
  Tomato(Tomato t) { // Copy-constructor
    super(t); // Upcast for base copy-constructor
    // Other copy-construction activities...
  }
}

class ZebraQualities extends FruitQualities {
  private int stripedness;
  ZebraQualities() { // Default constructor
    // do something meaningful...
  }
  ZebraQualities(ZebraQualities z) {
    super(z);
    stripedness = z.stripedness;
  }
}

class GreenZebra extends Tomato {
  GreenZebra() {
    addQualities(new ZebraQualities());
  }
  GreenZebra(GreenZebra g) {
    super(g); // Calls Tomato(Tomato)
    // Restore the right qualities:
    addQualities(new ZebraQualities());
  }
  void evaluate() {
    ZebraQualities zq =
      (ZebraQualities)getQualities();
    // Do something with the qualities
```

```
    // ...
  }
}

public class CopyConstructor {
  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); // Hmmm... will this work?
    System.out.println("In slice, f is a " +
      f.getClass().getName());
  }
  public static void main(String[] args) {
    Tomato tomato = new Tomato();
    ripen(tomato); // OK
    slice(tomato); // OOPS!
    GreenZebra g = new GreenZebra();
    ripen(g); // OOPS!
    slice(g); // OOPS!
    g.evaluate();
  }
} ///:~
```

这个例子第一眼看上去有点奇怪。不同水果的质量肯定有所区别，但为什么只是把代表那些质量的数据成员直接置入 Fruit（水果）类？有两方面可能的原因。第一个是我们可能想简便地插入或修改质量。注意 Fruit 有一个 protected（受到保护的）addQualities()方法，它允许派生类来进行这些插入或修改操作（大家或许会认为最合乎逻辑的做法是在 Fruit 中使用一个 protected 构造函数，用它获取 FruitQualities 参数，但构造函数不能继承，所以不可在第二级或级数更深的类中使用它）。通过将水果的质量置入一个独立的类，可以得到更大的灵活性，其中包括可以在特定 Fruit 对象的存在期间中途更改质量。

之所以将 FruitQualities 设为一个独立的对象，另一个原因是考虑到我们有时希望添加新的质量，或者通过继承与多态改变行为。注意对 GreenZebra 来说（这实际是西红柿的一种类型——我已栽种成功，它们看起来简直令人难以置信），构造函数会调用 addQualities()，并为其传递一个 ZebraQualities 对象。该对象是从 FruitQualities 派生出来的，所以能与基类中的 FruitQualities 引用联系在一起。当然，一旦 GreenZebra 使用 FruitQualities，就必须将其向下强制转型成为正确的类型（就象 evaluate()中展示的那样），但它肯定知道类型是 ZebraQualities。

大家也看到有一个 Seed（种子）类，Fruit（大家都知道，水果含有自己的种子）包含了一个 Seed 数组。

最后，注意每个类都有一个副本构造函数，而且每个副本构造函数都必须关心为基类和

成员对象调用副本构造函数的问题，从而获得“深层复制”的效果。对副本构造函数的测试是在 CopyConstructor 类内进行的。方法 ripen()需要获取一个 Tomato 参数，并对其执行副本构建工作，以便复制对象：

```
t = new Tomato(t);
```

而 slice()需要获取一个更常规的 Fruit 对象，而且对它进行复制：

```
f = new Fruit(f);
```

它们都在 main()中伴随不同种类的 Fruit 进行测试。下面是输出结果：

```
In ripen, t is a Tomato
In slice, f is a Fruit
In ripen, t is a Tomato
In slice, f is a Fruit
```

从中可看出一个问题。在 slice()内部对 Tomato 进行了副本构建工作以后，结果便不再是一个 Tomato 对象，而只是一个 Fruit。它已丢失了作为一个 Tomato（西红柿）的所有特征。此外，如果取得一个 GreenZebra，那么 ripen()和 slice()会把它分别转换成一个 Tomato 和一个 Fruit。所以非常不幸，假如想制作对象的一个本地副本，那么 Java 中的副本构造函数并不是特别适合我们。

为什么在 C++的作用比在 Java 中大?

副本构造函数是 C++的一个基本构成部分，因为它能自动产生对象的一个本地副本。但前面的例子确实证明了它不适合在 Java 中使用，为什么呢？在 Java 中，我们操控的一切东西都是引用，而在 C++中，却可以使用类似于引用的东西，也能直接传递对象。这时便要用到 C++的副本构造函数：只要想获得一个对象，并按值传递它，就可以复制对象。所以它在 C++里能很好地工作，但应注意这套机制在 Java 里是很不通的，所以不要用它。

A.4　只　读　类

尽管在一些特定的场合，由 clone()产生的本地副本能够获得我们希望的结果，但程序员（方法的作者）不得不亲自禁止别名处理的副作用。假如想制作一个库，令其具有常规用途，但却不能担保它肯定能在正确的类中得以克隆，这时又该怎么办呢？更有可能的一种情况是，假如我们想让别名发挥积极的作用——禁止不必要的对象复制——但却不希望看到由此造成的副作用，那么又该如何处理呢？

一个办法是创建“不变对象”，令其从属于只读类。可定义一个特殊的类，使其中没有任何方法能造成对象内部状态的改变。在这样的一个类中，别名处理是没有问题的。因为我们只能读取内部状态，所以当多处代码都读取相同的对象时，不会出现任何副作用。

作为“不变对象”一个简单例子，Java 的标准库包含了“封装器”（wrapper）类，可用于所有原始数据类型。大家可能已发现了这一点，如果想在一个象 ArrayList（它只采用 Object 引用）这样的集合里保存一个 int 数值，可以将这个 int 封装到标准库的 Integer 类内部。如下所示：

```
//: appendixa:ImmutableInteger.java
// The Integer class cannot be changed.
```

```
import java.util.*;

public class ImmutableInteger {
  public static void main(String[] args) {
    ArrayList 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（以及所有原始数据类型“封装器”类）用简单的形式实现了“不变性”：它们没有提供可供修改对象的方法。

若确实需要一个容纳了原始数据类型的对象，并想对原始数据类型进行修改，就必须亲自创建它们。幸运的是，操作非常简单：

```
//: appendixa:MutableInteger.java
// A changeable wrapper class.
import java.util.*;

class IntValue {
  int n;
  IntValue(int x) { n = x; }
  public String toString() {
    return Integer.toString(n);
  }
}

public class MutableInteger {
  public static void main(String[] args) {
    ArrayList 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)).n++;
    System.out.println(v);
  }
} ///:~
```

注意 n 在这里简化了我们的编码。

若默认的初始化为零已经足够（这样便不需要构造函数），而且不用考虑把它打印出来（这样便不需要 toString），那么 IntValue 甚至还能变得更加简单。如下所示：

```
class IntValue { int n; }
```

将元素取出来，再对其进行强制转型，这多少显得有些笨拙，但那是 ArrayList 的问题，并不是 IntValue 的错。

A.4.1　创建只读类

完全可以创建自己的只读类，下面是个简单的例子：

```
//: appendixa:Immutable1.java
// Objects that cannot be modified
// are immune to aliasing.

public class Immutable1 {
  private int data;
  public Immutable1(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable1 quadruple() {
    return new Immutable1(data * 4);
  }
  static void f(Immutable1 i1) {
    Immutable1 quad = i1.quadruple();
    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());
  }
} ///:~
```

所有数据都设为 private，可以看到没有任何 public 方法对数据作出修改。事实上，看起来真正需要修改一个对象的方法是 quadruple()，但它的作用是新建一个 Immutable1 对象，初始对象则是原封未动的。

方法 f()需要取得一个 Immutable1 对象，并对其采取不同的操作，而 main()的输出显示出没有对 x 作任何修改。因此，x 对象可别名处理许多次，不会造成任何伤害，因为根据 Immutable1 类的设计，它能保证对象不被改动。

A.4.2　“一成不变”的弊端

从表面看，不变类的建立似乎是一个好方案。但是，一旦真的需要那种新类型的一个修改的对象，就必须辛苦地进行新对象的创建工作，同时还有可能涉及更频繁的垃圾收集。对

有些类来说，这个问题并不是很大。但对其他类来说（比如 String 类），这一方案的代价显得太高了。

为解决这个问题，我们可以创建一个“同志”类，并使其能够修改。以后只要涉及大量的修改工作，就可换为使用能修改的同志类。完事以后，再切换回不可变的类。

因此，上例可改成下面这个样子：

```
//: appendixa:Immutable2.java
// A companion class for making
// changes to immutable objects.

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 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());
  }
} ///:~
```

和往常一样，Immutable2 包含的方法保留了对象不可变的特征，只要涉及修改，就创建新的对象。完成这些操作的是 add()和 multiply()方法。同志类叫作 Mutable，它也含有 add()和 multiply()方法。但这些方法能够修改原有的 Mutable 对象，而不是新建一个。除此以外，Mutable 的一个方法可用它的数据产生一个 Immutable2 对象，反之亦然。

两个静态方法 modify1()和 modify2()揭示出获得同样结果的两种不同方法。在 modify1()中，所有工作都是在 Immutable2 类中完成的，我们可看到在进程中创建了四个新的 Immutable2 对象（而且每次重新分配了 val，前一个对象就成为垃圾）。

在方法 modify2()中，可看到它的第一个行动是获取 Immutable2 y，然后从中生成一个 Mutable（类似于前面对 clone()的调用，但这一次创建了一个不同类型的对象）。随后，用 Mutable 对象进行大量修改操作，同时用不着新建许多对象。最后，它切换回 Immutable2。在这里，我们只创建了两个新对象（Mutable 以及 Immutable2 的结果），而不是四个。

这一方法特别适合在下述场合应用：

(1) 需要不可变的对象，而且

(2) 经常需要进行大量修改，或者

(3) 创建新的不变对象代价太高

A.4.3　不变字串

请观察下述代码：

```
//: appendixa:Stringer.java
```

```
public class Stringer {
  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
  }
} ///:~
```

q 传递进入 upcase()时，它实际是 q 的引用的一个副本。该引用连接的对象实际只在一个统一的物理位置处。引用四处传递的时候，它的引用会得到复制。

若观察对 upcase()的定义，会发现传递进入的引用有一个名字 s，而且该名字只有在 upcase()执行期间才会存在。upcase()完成后，本地引用 s 便会消失，而 upcase()返回结果——还是原来那个字串，只是所有字符都变成了大写。当然，它返回的实际是结果的一个引用。但它返回的引用最终是为一个新对象的，同时原来的 q 并未发生变化。所有这些是如何发生的呢？

1. 隐式常数

若使用下述语句：

```
String s = "asdf";
String x = Stringer.upcase(s);
```

那么真的希望 upcase()方法改变参数或者参数吗？我们通常是不愿意的，因为作为提供给方法的一种信息，参数一般是拿给代码的读者看的，而不是让他们修改。这是一个相当重要的保证，因为它使代码更易编写和理解。

为了在 C++中实现这一保证，需要一个特殊关键字的帮助：const。利用这个关键字，程序员可以保证一个引用（C++叫“指针”或者“引用”）不会被用来修改原始的对象。但这样一来，C++程序员需要用心记住在所有地方都使用 const。这显然容易使人混淆，也不容易记住。

2. 重载"+"和 StringBuffer

利用前面提到的技术，String 类的对象被设计成“不可变”。若查阅联机文档中关于 String 类的内容（本附录稍后还要总结它），就会发现类中能够修改 String 的每个方法实际都创建和返回了一个崭新的 String 对象，新对象里包含了修改过的信息——原来的 String 是原封未动的。因此，Java 里没有与 C++的 const 对应的特性可用来让编译器支持对象的不可变能力。若想获得这一能力，可以自行设置，就象 String 那样。

由于 String 对象是不可变的，所以能够根据情况对一个特定的 String 进行多次别名处理。因为它是只读的，所以一个引用不可能会改变一些会影响其他引用的东西。因此，只读对象可以很好地解决别名问题。

通过修改产生对象的一个崭新版本，似乎可以解决修改对象时的所有问题，就象 String 那样。但对某些操作来讲，这种方法的效率并不高。一个典型的例子便是为 String 对象覆盖

的运算符“+”。“覆盖”意味着在与一个特定的类使用时，它的含义已发生了变化（用于 String 的“+”和“+=”是 Java 中能被覆盖的唯一运算符，Java 不允许程序员覆盖其他任何运算符[78]）。

针对 String 对象使用时，“+”允许我们将不同的字串连接起来：

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

可以想象出它“可能”是如何工作的：字串"abc"可以有一个方法 append()，它新建了一个字串，其中包含"abc"以及 foo 的内容；这个新字串然后再创建另一个新字串，在其中添加"def"；以此类推。

这一设想是行得通的，但它要求创建大量字串对象。尽管最终的目的只是获得包含了所有内容的一个新字串，但中间却要用到大量字串对象，而且要不断地进行垃圾收集。我怀疑 Java 的设计者是否先试过种方法（这是软件开发的一个教训——除非自己试试代码，并让某些东西运行起来，否则不可能真正了解系统）。我还怀疑他们是否早就发现这种做法的最终性能是无法令人接受的。

真正的解决之道是象前面介绍的那样制作一个可变的同志类。对字串来说，这个同志类叫作 StringBuffer，编译器可自动创建一个 StringBuffer，以便计算特定的表达式，特别是面向 String 对象应用覆盖过的运算符+和+=时。下面这个例子对此进行了演示：

```
//: appendixa:ImmutableStrings.java
// Demonstrating StringBuffer.

public class ImmutableStrings {
  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);
  }
} ///:~
```

创建字串 s 时，编译器做的工作大致等价于后面使用 sb 的代码——创建一个 StringBuffer，并用 append()将新字符直接加入 StringBuffer 对象（而不是每次都产生新对象）。尽管这样做更有效，但不值得每次都创建象"abc"和"def"这样的引号字串，编译器会把它们都转换成 String 对象。所以尽管 StringBuffer 提供了更高的效率，但会产生比我们希望的多

[78] C++允许程序员随意重载运算符。由于这通常是一个复杂的过程（参见《Thinking in C++》第二版的第 10 章，Prentice-Hall 出版社 2000 年出版），所以 Java 的设计者认定它是一种“糟糕”的特性，决定不在 Java 中采用。但具有讽刺意味的是，运算符的重载在 Java 中要比在 C++中容易得多。为此，大家可以参考一下 Python（www.Python.org），它不仅提供了垃圾收集机制，也允许直接进行运算符重载。

得多的对象。

A.4.4 String 和 StringBuffer 类

这里总结一下同时适用于 String 和 StringBuffer 的方法，以便对它们相互间的沟通方式有一个印象。但是，这些表格并没有把每个单独的方法都包括进去，只是包括了与本次讨论有重要关系的那些方法。那些已被重载的方法用单独一行总结。

首先总结 String 类的各种方法：

方　　法	参数，重载	用　　途
Constructor	已被重载：默认，String，StringBuffer，char 数组，byte 数组	创建 String 对象
length()	无	String 中的字符数
charAt()	int 索引	位于 String 内某个位置的 char
getChars()，getBytes	开始复制的起点和终点，要向其中复制内容的数组，对目标数组的一个索引	将 char 或 byte 复制到外部数组内部
toCharArray()	无	产生一个 char[]，其中包含了 String 内部的字符
equals()，equalsIgnoreCase()	用于对比的一个 String	对两个字串的内容进行等价性检查
compareTo()	用于对比的一个 String	结果为负、零或正，具体取决于 String 和参数的字典顺序。注意大写和小写不是相等的！
regionMatches()	这个 String 以及其他 String 的位置偏移，以及要比较的区域长度。覆盖加入了“忽略大小写”的特性	一个布尔结果，指出要对比的区域是否相同
startsWith()	可能以它开头的 String。覆盖在参数里加入了偏移	一个布尔结果，指出 String 是否以那个参数开头
endsWith()	可能是这个 String 后缀的一个 String	一个布尔结果，指出参数是不是一个后缀
indexOf()，lastIndexOf()	已重载：char，char 和起始索引，String，String 和起始索引	若参数未在这个 String 里找到，则返回-1；否则返回参数开始处的位置索引。lastIndexOf()可从终点开始回溯搜索
substring()	已重载：起始索引，起始索引和结束索引	返回一个新的 String 对象，其中包含了指定的字符子集
concat()	想连结的 String	返回一个新 String 对象，其中包含了原始 String 的字符，并在后面加上由参数提供的字符
relpace()	要查找的老字符，要用它替换的新字符	返回一个新 String 对象，其中已完成了替换工作。若没有找到相符的搜索项，就沿用老字串
toLowerCase()，toUpperCase()	无	返回一个新 String 对象，其中所有字符的大小写形式都进行了统一。若不必修改，则沿用老字串

续表

方　　法	参数，重载	用　　途
trim()	无	返回一个新的 String 对象，头尾空白均已删除。若毋需改动，则沿用老字串
valueOf()	已重载：object，char[]，char[]和偏移以及计数，boolean，char，int，long，float，double	返回一个 String，其中包含参数的一个字符表现形式
Intern()	无	为每个独一无二的字符顺序都产生一个（而且只有一个）String 引用

可以看到，一旦有必要改变原来的内容，每个 String 方法都小心地返回了一个新的 String 对象。另外要注意的一个问题是，若内容不需要改变，则方法只返回指向原来那个 String 的一个引用。这样做可以节省存储空间和系统开销。

下面列出有关 StringBuffer（字串缓冲）类的方法：

方　　法	参数，重载	用　　途
构造函数	已重载：默认，要创建的缓冲区长度，要根据它创建的 String	新建一个 StringBuffer 对象
toString()	无	根据这个 StringBuffer 创建一个 String
length()	无	StringBuffer 中的字符数量
capacity()	无	返回目前分配的空间大小
ensureCapacity()	用于表示希望容量的一个整数	使 StringBuffer 容纳至少希望的空间大小
setLength()	用于指示缓冲区内字串新长度的一个整数	缩短或扩充前一个字符串。如果是扩充，则用 null 值填充空隙
charAt()	表示目标元素所在位置的一个整数	返回位于缓冲区指定位置处的 char
setCharAt()	代表目标元素位置的一个整数以及元素的一个新 char 值	修改指定位置处的值
getChars()	复制的起点和终点，要在其中复制的数组以及目标数组的一个索引	将 char 复制到一个外部数组。和 String 不同，这里没有 getBytes()可供使用
Append()	已重载：Object，String，char[]，特定偏移和长度的 char[]，boolean，char，int，long，float，double	将参数转换成一个字串，并将其追加到当前缓冲区的末尾。若有必要，同时增大缓冲区的长度
insert()	已重载，第一个参数代表开始插入的位置：Object，String，char[]，boolean，char，int，long，float，double	第二个参数转换成一个字串，并插入当前缓冲区。插入位置在偏移区域的起点处。若有必要，同时会增大缓冲区的长度
reverse()	无	反转缓冲内的字符顺序

其中，最常用的一个方法是 append()。在计算包含了+和+=运算符的 String 表达式时，编译器便会用到这个方法。insert()方法采用类似的形式。这两个方法都能直接对缓冲区进行操作，不需要另建新对象。

A.4.5　字串的特殊性

现在，大家已知道 String 类并非仅仅是 Java 提供的另一个类。String 里含有大量特殊的

类。通过编译器和特殊的覆盖或重载运算符+和+=，可将引号字符串转换成一个 String。在本附录中，大家已见识了剩下的一种特殊情况：用同志 StringBuffer 精心构造的“不可变”能力，以及编译器中出现的一些有趣现象。

A.5 总　　结

由于 Java 中的所有东西都是引用，而且由于每个对象都是在内存堆中创建的——只有不再需要的时候，才会当作垃圾收掉，所以对象的操作方式发生了变化，特别是在传递和返回对象的时候。举个例子来说，在 C 和 C++中，如果想在一个方法里初始化一些存储空间，可能需要请求用户将那片存储区域的地址传递进入方法。否则就必须考虑由谁负责清除那片区域。因此，这些方法的接口和对它们的理解就显得要复杂一些。但在 Java 中，根本不必关心由谁负责清除，也不必关心在需要一个对象的时候它是否仍然存在。因为系统会为我们照料一切。我们的程序可在需要的时候创建一个对象。而且更进一步地，根本不必担心那个对象的传输机制的细节：只需简单地传递引用即可。有些时候，这种简化非常有价值，但另一些时候却显得有些多余。

可从两个方面认识这一机制的缺点：

(1) 肯定要为额外的内存管理付出效率上的损失（尽管损失不大），而且对于运行所需的时间，总是存在一丝不确定的因素（因为在内存不够时，垃圾收集器可能会被强制采取行动）。对大多数应用来说，优点显得比缺点重要，而且部分对时间要求非常苛刻的段落可以用 native 方法写成（参见附录 A）。

(2) 别名处理：有时会不慎获得指向同一个对象的两个引用。只有在这两个引用都假定指向一个“明确”的对象时，才有可能产生问题。对这个问题，必须加以足够的重视。而且应该尽可能地“克隆”一个对象，以防止另一个引用被不希望的改动影响。除此以外，可考虑创建“不可变”对象，使它的操作能返回同种类型或不同种类型的一个新对象，从而提高程序的执行效率。但千万不要改变原始对象，使对那个对象别名的其他任何方面都感觉不出变化。

有些人认为 Java 的克隆方案有点儿“笨拙”，所以他们干脆实现了自己的克隆方案[79]，永远杜绝调用 Object.clone() 方法，从而消除了实现 Cloneable 和捕获 CloneNotSupportException 违例的需要。这一做法是合理的，而且由于 clone()在 Java 标准库中很少得以支持，所以这显然也是一种“安全”的方法。只要不调用 Object.clone()，就不必实现 Cloneable 或者捕获违例，所以那看起来也是能够接受的。

A.6 练　　习

这些练习的答案放在《The Thinking in Java Annotated Solution Guide》这份电子文档里，只需少量费用，便可从 www.BruceEckel.com 购得。

(1) 演示第二级的“别名处理”。创建一个方法，令其取得到一个对象的引用，但不要修改那个引用的对象。然而，方法会调用另一个方法，把引用传递给它。最后，由另一个方法来修改对象。

(2) 创建一个 myString 类，在其中包含了一个 String 对象，以便用在构造函数中用构造

[79] Doug Lea 特别重视这个问题，并把这个方法推荐给了我，他说只需为每个类都创建一个名为 duplicate()的函数即可。

函数的参数对其进行初始化。添加一个 toString()方法以及一个 concatenate()方法，令其将一个 String 对象追加到我们的内部字串。在 myString 中实现 clone()。创建两个 static 方法，每个都取得一个 myString x 引用作为自己的参数，并调用 x.concatenate("test")。但在第二个方法中，请首先调用 clone()。测试这两个方法，观察它们不同的结果。

(3) 创建一个名为 Battery（电池）的类，在其中包含一个 int，用它表示电池的编号（采用独一无二的标识符的形式）。接下来，创建一个名为 Toy 的类，其中包含了一个 Battery 数组以及一个 toString，用于打印出所有电池。为 Toy 写一个 clone()方法，令其自动关闭所有 Battery 对象。克隆 Toy 并打印出结果，完成对它的测试。

(4) 修改 CheckCloneable.java，使所有 clone()方法都能捕获 CloneNotSupportException 违例，而不是把它直接传给调用者。

(5) 使用可变同志类技术，创建一个不可变的类，在其中包括一个 int 值、一个 double 值以及一个 char 数组。

(6) 修改 Compete.java，为 Thing2 和 Thing4 类添加更多的成员对象，看看自己是否能判断计时随复杂性变化的规律——是一种简单的线性关系，还是看起来更加复杂。

(7) 从 Snake.java 开始，创建 Snake 的一个深层复制版本。

(8) 继承一个 ArrayList，让它的 clone()执行一次深层复制。

附录 B Java 固有接口(JNI)

本附录素材由 Andrea Provaglio 赞助提供，在这儿的使用也获得了他们的授权，详情见 www.AndreaProvaglio.com。

JAVA 语言及其标准 API（应用程序编程接口）应付应用程序的编写已绰绰有余。但在某些情况下，还是得用到一些非 JAVA 代码。例如，我们有时要使用操作系统的专属功能，与特殊硬件设备打交道，重复利用原有的非 Java 代码，或者实现对实时性要求严格的功能等等。

与非 Java 代码的沟通要求获得编译器和“虚拟机”的专门支持，并需附加的工具将 Java 代码映射成非 Java 代码。针对调用非 Java 代码的问题，JavaSoft 提供的标准方案叫作“Java 固有接口”（Java Native Interface），本附录将围绕它展开讨论。不过，由于篇幅有限，这里只能讲个大概。而且在某些情况下，还要求你事先掌握了一些预备知识，对我们要说的一些概念和技术做到心中有数。

JNI 是一种包容极广的编程接口，允许我们从 Java 应用程序里调用固有方法。它是自 Java 1.1 开始加入的。在某种程度上，它维持着与 Java 1.0 的“固有方法接口”（NMI）的兼容，两者具有一定的相似性。不过，NMI 设计上一些特点使其不大容易获所有虚拟机的支持。考虑到这个原因，Java 语言将来的版本可能不再提供对 NMI 的支持，这儿也不准备讨论它。

目前，JNI 只能与用 C 或 C++写成的固有方法打交道。利用 JNI，我们的固有方法可以：

■创建、检查及更新 Java 对象（包括数组和字串）

■调用 Java 方法

■捕捉和产生违例

■载入类并获取类信息

■执行运行时间类型检查

所以，原来在 Java 中能对类及对象做的几乎所有事情在固有方法中都同样能够做到。

B.1 调用固有方法

我们先从一个简单的例子开始：由一个 Java 程序调用固有方法，后者再调用一个标准的 C 库函数：printf()。

第一步是写一段 Java 代码，声明一个固有方法及其参数（参数）：

```
//: appendixb:ShowMessage.java
public class ShowMessage {
  private native void ShowMessage(String msg);
  static {
    System.loadLibrary("MsgImpl");
    // Linux hack, if you can't get your library
    // path set in your environment:
    // System.load(
```

```
    //  "/home/bruce/tij2/appendixb/MsgImpl.so");
  }
  public static void main(String[] args) {
    ShowMessage app = new ShowMessage();
    app.ShowMessage("Generated with JNI");
  }
} ///:~
```

在固有方法声明的后面，跟随有一个 static 代码块，它会调用 System.loadLibrary()——你可在任何时候调用它，但目前这样子更恰当。System.loadLibrary()将一个 DLL 载入内存，并建立同它的链接。DLL 必须位于你的系统路径中。取决于具体的平台，JVM 会自动添加适当的文件扩展名。

在上述代码中，我们其实也发出了对 System.load()方法的调用，只是把它变成了注释内容，并未真正执行（代码行前加“//”）。这里指定的属于“绝对”路径，而不是依赖一个环境变量。当然，环境变量是一种更好的做法，它为程序在不同平台间的移植带来了方便。但是，假如找不到环境变量，也可以将 loadLibrary()调用变成注释，同时撤消对 System.load()的注释，改为使用绝对路径，令其指向自己的目录。

B.1.1　头文件生成器：javah

现在编译你的 Java 源文件，并对编译出来的.class 文件运行 javah。注意同时指定-jin 开关参数（在本书配套的源码包中，makefile 工具会自动帮你做这个工作）：

```
javah –jni ShowMessage
```

javah 会读入类文件，并针对每一个固有方法声明，在 C 或 C++头文件里生成一个函数原型。下面是输出结果——ShowMessage.h 源文件（为符合本书的排版要求，稍微进行了一下处理）：

```
/* DO NOT EDIT THIS FILE
   - it is machine generated */
#include <jni.h>
/* Header for class ShowMessage */

#ifndef _Included_ShowMessage
#define _Included_ShowMessage
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     ShowMessage
 * Method:    ShowMessage
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
Java_ShowMessage_ShowMessage
  (JNIEnv *, jobject, jstring);
```

```
#ifdef __cplusplus
}
#endif
#endif
```

从“#ifdef_cplusplus”这个预处理引导命令可以看出，该文件既可由 C 编译器编译，亦可由 C++编译器编译。第一个#include 引导命令包括 jni.h——一个头文件——它的作用之一是定义要在文件其余部分用到的类型；JNIEXPORT 和 JNICALL 是一些宏，它们进行了适当的扩充，以便与那些不同平台专用的引导命令配合；JNIEnv、jobject 以及 jstring 则是 JNI 数据类型定义，稍后还会详加解释。

B.1.2 名字管理和函数签名

JNI 统一了固有方法的命名规则；这一点是非常重要的，因为它属于虚拟机将 Java 调用与固有方法链接起来的机制的一部分。从根本上说，所有固有方法都要以一个“Java”起头，后面跟随 Java 方法的名字；下划线字符则作为分隔符使用。若 Java 固有方法“重载”（即命名重复），那么也把函数签名追加到名字后面。在原型前面的注释里，大家可看到固有的签名。欲了解命名规则和固有方法签名更详细的情况，请参考相应的 JNI 文档。

B.1.3 实现你的 DLL

此时，我们要做的全部事情就是写一个 C 或 C++源代码文件，在其中包含由 javah 生成的头文件；并实现固有方法；然后编译它，生成一个动态链接库（DLL）。这一部分的工作是与具体平台有关的。下面的代码已经编译并链接至一个名为 MsgImpl.dll 的文件（Windows 平台）；或者至 MsgImpl.so（Unix/Linux 平台）——本书源代码包的 makefile 工具包含了做这些工作的指示——可在本书配套光盘上找到它，也可从 www.BruceEckel.com 免费下载：

```
//: appendixb:MsgImpl.cpp
//# Tested with VC++ & BC++. Include path must
//# be adjusted to find the JNI headers. See
//# the makefile for this chapter (in the
//# downloadable source code) for an example.
#include <jni.h>
#include <stdio.h>
#include "ShowMessage.h"

extern "C" JNIEXPORT void JNICALL
Java_ShowMessage_ShowMessage(JNIEnv* env,
jobject, jstring jMsg) {
  const char* msg=env->GetStringUTFChars(jMsg,0);
  printf("Thinking in Java, JNI: %s\n", msg);
  env->ReleaseStringUTFChars(jMsg, msg);
} ///:~
```

传递到固有方法内部的一系列参数是返回 Java 的大门。第一个的类型为 JNIEnv，其中

包含了回调 JVM 所需的所有挂钩（下一节再详细讲述）。第二个参数根据方法的类型，可能有不同的含义。对于象上例那样的非静态方法，该参数等价于 C++的“this”指针，也类似于 Java 的“this”——它是对调用固有方法的那个对象的一个引用。但对于静态方法来说，它就变成了对实现方法的那个 Class 对象的一个引用。

剩余的参数代表传递到固有方法调用里的 Java 对象。原始数据类型也是以这种形式传递的，但它们进行的“按值”传递。

在后面的小节里，我们准备讲述如何从一个固有方法的内部访问和控制 JVM，同时对上述代码进行更详尽的解释。

B.2　访问 JNI 函数：JNIEnv 参数

利用 JNI 函数，程序员可从一个固有方法的内部与 JVM 打交道。正如大家在前面的例子中看到的那样，每个 JNI 固有方法都会接收一个特殊的参数作为自己的第一个参数，即“JNIEnv 参数”——它是指向类型为 JNIEnv_的一个特殊 JNI 数据结构的指针。JNI 数据结构的一个元素是指向由 JVM 生成的一个数组的指针；该数组的每个元素都是指向一个 JNI 函数的指针。可从固有方法的内部发出对 JNI 函数的调用，做法是撤消对这些指针的引用（具体的操作实际很简单）。每种 JVM 都以自己的方式实现了 JNI 函数，但它们的地址肯定位于预先定义好的偏移位置。

利用 JNIEnv 参数，程序员可访问一系列函数。这些函数可划分为下述类别：

- 获取版本信息
- 进行类和对象操作
- 控制对 Java 对象的全局和局部引用
- 访问实例字段和静态字段
- 调用实例方法和静态方法
- 执行字串和数组操作
- 产生和控制 Java 违例

JNI 函数的数量相当多，这里便不再赘述。相反，我会向大家揭示使用这些函数时背后的一些基本原理。欲了解更详细的情况，请参阅自己所用编译器的 JNI 文档。

若观察一下 jni.h 头文件，就会发现在#ifdef _cplusplus 预处理器条件的内部，当 C++编译器编译它时，JNIEnv_结构会被定义成一个类。这个类里包含了大量内嵌函数。通过一种简单而且熟悉的语法，这些函数可让我们从容访问 JNI 函数。例如，前例包含了下面这行代码：

```
env->ReleaseStringUTFChars(jMsg, msg);
```

它在 C 里可改写成下面这个样子：

```
(*env)->ReleaseStringUTFChars(env, jMsg, msg);
```

我们注意到，C 编程样工显得更加复杂（这是理所当然的）——需要连续两次撤消对 env 指针的引用，而且还必须将相同的指针作为第一个参数传给 JNI 函数调用。本附录的例子采用的是 C++样式。

B.2.1　访问 Java 字串

作为访问 JNI 函数的一个例子，请思考 MsgImpl.cpp 中的代码。在这里，我们利用 JNIEnv 的参数 env 来访问一个 Java 字串。Java 字串采取的是 Unicode 格式，所以假若收到这样一个字串，并想把它传给一个非 Unicode 函数（如 printf()），首先必须用 JNI 函数

GetStringUTFChars()将其转换成 UTF-8 字符（这种字符可为 8 位宽度，用于容纳 ASCII 值；亦可为 16 位宽度，用于容纳 Unicode。如果原始字串的内容纯粹是 ASCII 代码，那么最后生成的字串也采用 ASCII 格式）。

GetStringUTFChars()是 JNIEnv 的成员函数之一。为了访问 JNI 函数，我们用传统的 C++语法来调用一个函数（通过指针）。利用上述形式便可实现对所有 JNI 函数的访问。

B.3 传递和使用 Java 对象

在前例中，我们将一个字串传递给固有方法。事实上，亦可将自己创建的 Java 对象传递给固有方法。随后，在我们的固有方法内部，便可访问接收到的那个对象的字段及方法。

要想传递对象，声明固有方法时要采用标准 Java 语法。如下例所示，MyJavaClass 有一个 public（公共）字段，以及一个 public 方法。UseObjects 类声明了一个固有方法，用于接收 MyJavaClass 类的一个对象。为调查固有方法是否能控制自己的参数，我们为参数设置了 public 字段，然后调用固有方法，最后打印出 public 字段的值。

```
//: appendixb:UseObjects.java
class MyJavaClass {
  public int aValue;
  public void divByTwo() { aValue /= 2; }
}

public class UseObjects {
  private native void
    changeObject(MyJavaClass obj);
  static {
    System.loadLibrary("UseObjImpl");
    // Linux hack, if you can't get your library
    // path set in your environment:
    // System.load(
    //"/home/bruce/tij2/appendixb/UseObjImpl.so");
  }
  public static void main(String[] args) {
    UseObjects app = new UseObjects();
    MyJavaClass anObj = new MyJavaClass();
    anObj.aValue = 2;
    app.changeObject(anObj);
    System.out.println("Java: " + anObj.aValue);
  }
} ///:~
```

编译好代码，并运行了 javah 之后，就可开始实现固有方法。在下面的例子中，一旦取得字段和方法 ID，就通过 JNI 函数访问它们：

```
//: appendixb:UseObjImpl.cpp
```

```
//# Tested with VC++ & BC++. Include path must
//# be adjusted to find the JNI headers. See
//# the makefile for this chapter (in the
//# downloadable source code) for an example.
#include <jni.h>
extern "C" JNIEXPORT void JNICALL
Java_UseObjects_changeObject(
JNIEnv* env, jobject, jobject obj) {
  jclass cls = env->GetObjectClass(obj);
  jfieldID fid = env->GetFieldID(
    cls, "aValue", "I");
  jmethodID mid = env->GetMethodID(
    cls, "divByTwo", "()V");
  int value = env->GetIntField(obj, fid);
  printf("Native: %d\n", value);
  env->SetIntField(obj, fid, 6);
  env->CallVoidMethod(obj, mid);
  value = env->GetIntField(obj, fid);
  printf("Native: %d\n", value);
} ///:~
```

忽略“this”的等价物，C++函数会收到一个 jobject，它代表 Java 对象引用“固有”的那一面——那个引用是我们从 Java 代码里传递的。我们简单地读取 aValue，把它打印出来，改变这个值，调用对象的 divByTwo()方法，然后重新打印一遍。

为访问一个字段或方法，首先必须分别用 GetFieldID()或 GetMehtodID()方法获取它的标识符。这些函数要求取得类对象、包含了元素名的一个字串以及用于提供类型信息的一个字串。“类型信息”要么是字段的数据类型，要么是一个方法的签名信息（详情参见 JNI 文档）。这些函数会返回一个标识符，利用它就可访问对应的元素。尽管这一方式显得颇为“曲折”，但我们的固有方法本来就对 Java 对象的内部布局一无所知。因此，它必须通过由 JVM 返回的索引来访问字段和方法。这样一来，不同的 JVM 就可实现不同的内部对象布局，同时不会对你的固有方法造成影响。

若运行 Java 程序，就会发现从 Java 那一侧传来的对象已由我们的固有方法处理。但传递的到底是什么呢？是指针，还是 Java 引用？而且垃圾收集器在固有方法调用期间又在做什么呢？

垃圾收集器会在固有方法执行期间持续运行，但在一次固有方法调用期间，我们的对象可保证不会被当作“垃圾”收去。为确保这一点，事先创建了“局部引用”，并在固有方法调用之后立即清除。由于它们的“生存时间”不能超过调用的范围，所以能保证对象在固有方法调用期间的有效性。

由于这些引用会在每次函数调用的时候创建和破坏，所以不可在 static 变量中制作固有方法的局部副本（本地拷贝）。若希望一个引用在函数存在期间持续有效，就需要一个全局引用。全局引用不是由 JVM 创建的，但通过调用特定的 JNI 函数，程序员可将局部引用扩展为全局引用。创建一个全局引用时，需对引用对象的“生存时间”负责。全局引用（以及它引用的对象）会一直留在内存里，直到用特定的 JNI 函数明确释放了这个引用。它类似于 C 的 malloc()和 free()。

B.4 JNI 和 Java 违例

利用 JNI，可产生、捕捉、打印以及重新产生 Java 违例——就象在一个 Java 程序里那样。但对程序员来说，需自行调用专用的 JNI 函数，以便对违例进行处理。下面列出用于违例处理的一些 JNI 函数：

■Throw()：产生一个现有的违例对象；在固有方法中用于重新产生一个违例

■ThrowNew()：产生一个新的违例对象

■ExceptionOccurred()：判断一个违例是否已经产生，但尚未清除

■ExceptionDescribe()：打印一个违例和堆栈跟踪信息

■ExceptionClear()：清除一个待决的违例

■FatalError()：造成一个严重错误，不返回

在所有这些函数中，最不容忽视的就是 ExceptionOccurred()和 ExceptionClear()。大多数 JNI 函数都能产生违例，而且没有象 Java try 块那样的语言特性可供利用。所以在每一次 JNI 函数调用之后，都必须调用 ExceptionOccurred()，了解违例是否已经产生。若侦测到一个违例，可选择对其加以控制（可能时还要重新产生它）。然而，必须确保违例最终被清除。可在自己的函数中用 ExceptionClear()来完成清除；若违例被重新产生，也可以在其他某些函数中清除。但无论如何，这一工作都是必不可少的。

我们必须保证违例被彻底清除。否则，假若在一个违例待决的情况下调用一个 JNI 函数，获得的结果往往是无法预知的。也有少数几个 JNI 函数可在违例时安全调用——当然，它们全都是专门的违例控制函数。

B.5 JNI 和线程

由于 Java 是一种多线程语言，几个线程可能同时发出对一个固有方法的调用（若另一个线程发出调用，固有方法可能在执行中途暂停）。此时，完全要由程序员来保证固有调用在多线程的环境中安全进行。例如，要防范用一种未进行监视的方法修改共享数据。此时，我们主要有两个选择：将固有方法声明为“同步的”（synchronized），或在固有方法内部采取其他某些策略，确保进行正确、并发的数据处理。

此外，绝对不要通过线程传递 JNIEnv，因为它指向的内部结构在每个线程中都有所不同，其中包含了只对那个特定线程才有意义的信息。

B.6 使用原有代码

为实现 JNI 固有方法，最简单的方法就是在一个 Java 类里编写固有方法的原型，编译那个类，再通过 javah 运行.class 文件。但假若我们已有一个大型的、早已存在的代码库，而且想从 Java 里调用它们，那么又该如何是好呢？注意不可将 DLL 中的所有函数更名，使其符合 JNI 命名规则，这种方案是不可行的。最好的办法是在原来的代码库“外面”写一个作为“封装器”使用的 DLL。Java 代码会调用新 DLL 里的函数，后者再调用原始的 DLL 函数。这个方法并非仅仅是“一种”解决方案；大多数情况下，我们甚至必须这样做！因为必须随对象引用调用 JNI 函数，否则便无法使用它们。

B.7　其 他 信 息

在本书第一版的附录 A，大家还可找到其他大量介绍性资料，其中包括一个用 C（注意不是 C++）写的例子，以及对微软问题的说明。那本书的电子版已包括到本书的配套光盘上，也可从 www.BruceEckel.com 免费下载。更多的资料可以从 java.sun.com 那里拿到（如果用一个搜索引擎查找，可以在“training & tutorials”分类中，搜索“native methods”关键字）。《Core Java 2》第 II 卷的第 11 章也对固有方法进行了全面探讨，那本书由 Horstmann 和 Cornell 合著，Prentice-Hall 出版社于 2000 年出版。

附录 C Java 编程指南

本附录包含了大量有用的建议，帮助大家进行低级程序设计，并对代码的编写提供了一般性的指导：

显然，这里提供的仅仅是一些指导性意见，而不是严格意义上的“规则”。它们的宗旨是为你的程序设计提供一些参考，提醒你一些应该注意的问题。某些情况下，由于条件所限，你可能还必须遵守或者打破这儿的“规则”。

C.1 设 计

(1) 付出必有收获。从短期看，似乎要花不少时间才能想出一个真正合理的方案。但只要你真正得到了这样的一个方案，而且在第一次运行成功之后，以后就可以非常轻松地移植到新环境中——而不需要花数小时、数天乃至数月的时间来苦苦思考。我们的努力工作会带来巨大回报（其价值甚至无法估量）。这不仅使自己的程序易于理解和维护，而且也会更具“经济效益”。这个概念需要大家经过一番实践才能真正理解，因为假如只是部分代码设计得“不错”，那么系统的“总体效益”仍是显出现不出来的。请不要放弃，继续努力！坚决抵制草草完工的诱惑——那样往往会得不偿失！

(2) 首先得运行起来，再考虑如何变得更快。即便你肯定一段代码相当重要，而且以后可能成为系统最大的一个“瓶颈”，也要坚决抵制住马上就去“优化”它的冲动！先用最简单的设计，让你的系统工作起来。然后，觉察到它真的不快，再去优化它。在许多情况下，你最后都会发现原来担心的那个“瓶颈”根本不是一个真正的“瓶颈”。请将时间省下来吧，去做其他更重要的事情！

(3) 记住“问题要细化”。如果要解决的问题实在太复杂，请试着想像程序的基本操作是什么，将一个困难的目标划分为一系列容易达到的小目标。每个“目标”都是一个“对象”——先对付容易的对象，写相应的代码来使用那个对象，然后研究这个对象，把它最困难的部分封装到其他对象里……逐渐细化。

(4) 将类的创建者和类的用户（客户程序员）区分开。自己设计了一个类，它的用户便是自己的“客户”，那些客户通常是另外一些程序员，亦即“客户程序员”。对客户程序员来讲，他们并不希望知道这个类背后的细节，只要能“用”就可以了——而且还要“好用”。作为类的创建者，我们必须精通类设计，并合理地编写自己的类，使其能由大多数仍然是新手的程序员方便地使用——同时要保证它们在应用程序中工作时的“健壮性”。对一个库来说，只有它在对用户来说“透明”的前提下，才有前提成为一个真正“好用”的库。

(5) 创建一个类时，试着为它取一个清楚的名字，甚至完全不需注解就能明白。你的目标是使客户程序员的接口在概念上清晰易懂。因此，应该尽可能地使用方法重载来创建一个直观、易于使用的接口。

(6) 完成了分析与设计阶段后，至少应明确系统中要用到的类、它们的公共接口以及它们与其他类的关系（特别是和基类的关系）。假如在你的设计完成后，实际得到的比上面那些东西还要多，便请考察多出来的东西是否能在程序的“生存时间”里一直有用？如果只是暂时有用，那么维护它们便显得得不偿失。对一个开发小组的成员来说，他们谁都不想维护那些对提高效率没有任何帮助的东西；这正是在分析与设计阶段中，许多设计方法都不会考

虑的原因！

(7) 测试应该自动化。先写好测试代码（在写类之前），然后随着类的设计，同步那些代码。利用 makefile 或其他类似工具，自动运行你的测试。这样一来，只需运行测试代码，许多变动便可自动得到验证。如果有错误的话，马上就能发现！由于有这个自动进行的、安全的测试床“护驾”，所以在需要的时候，可以放心大胆地进行各项实验。请记住，通过类型检查、违例控制等等手段，Java 语言为我们提供了一个从前难以想象的、出色的测试环境。当然，它们提供的帮助也就到此为止了，剩下的工作还得由你自己完成——你得根据要解决的问题，自行思考出一系列不同的解决方案，选出其中最好的。

(8) 先写测试代码（在写类之前），检验类设计的完整性。不写测试代码，便不能真正看到你的类“长得象什么样”。此外，通过测试代码的编写，通常还能发现类分析与设计时一些想不到的问题或者存在的一些局限。通过测试，也能从中体验到自己的类最后该怎么用。

(9) 所有软件设计问题都可以进一步简化，只要你在概念上再多“迂回包抄”一次。这是软件工程的基本规则[80]，也是“抽象”的基础——“抽象”是面向对象程序设计的主要特点。

(10) 每一次“迂回”都要有意义（与上一条相配合）。如果简单地说，就是“将通用的代码放到单独一个方法里”。假如你“迂回”了一下（通过抽象、封装等等进行），但却没有任何实际意义，那么结果反而会更糟，会使问题更加复杂化。

(11) 尽可能地细分类的用途。每个类都应该有一个（而且只能有一个）明确的用途。如果你的类或系统设计变得非常复杂，就需要将一个复杂的类拆分为一系列较简单的类：如果类过大，由于它要负责的事情太多，那么最终可能会崩溃。重新设计一个类时：

1. 假如是一条复杂的开关语句，那么考虑采用“多态”；

2. 假如是大量方法，它们牵涉到许多不同类型的操作，那么考虑使用几个类来分担任务。

3. 假如是大量成员变量，它们牵涉到许多不同的特征，那么考虑使用几个类来分担任务。

(12) 小心长参数列表。否则方法调用会变得很难读、写和维护。相反，应试着将方法移到一个更恰当的类里，并且（或者）将对象作为参数传递进去。

(13) 不要太多的重复。假如一部分代码在派生类的许多方法里经常重复使用，便将它们放到基类的一个方法里，然后从派生类方法里直接调用。这样不仅减少代码量，以后也更容易修改。有时，你会发现象这样的“通用代码”可为你的接口增加有价值的功能。

(14) 小心开关（switch）语句或链接的 if-else 从句。它们通常标志着“类型检查编码”，意思是你根据某些类型信息，来选择要执行的代码（最开始可能无法知道准确的类型是什么）。通常可用继承和多态来代替这种代码；一个多态的方法调用可帮你执行类型检查，而且可以使程序更可靠、更易扩展。

(15) 从设计的角度出发，找到要发生改变的东西，把它们同不变的东西分开。也就是说，你需要找出一个系统里希望改变、但又不希望以后被迫重新设计的元素，然后把那些元素单独封装到类里（可以用多个类）。在《Thinking in Patterns with Java》一书里，大家还可以学到这方面更多的知识——该书电子版可从 www.BruceEckel.com 免费下载。

(16) 不要用子类来扩展基本功能。假如一个接口元素对一个类来说是至关重要的，便应把它放到基类里，而不要在派生过程中添加。如果你当前正通过继承来添加方法，那么整个设计或许都应该重新考虑一下。

(17) 精简至上。先为一个类设计一个“最小化”的接口，它应该尽可能地小、尽可能

[80] 感谢 Andrew Koenig 提醒了我这一点。

地简单，能解决手边的问题便成！但是，千万不要在此时试图预测出这个类未来“可能”的所有用途。等类真正投入使用后，才可知道对接口进行扩展的正确方式。然而，一旦某个类投入正式使用后，便不能再轻易地“收缩”接口，否则便需要重新发布客户端代码。如果需要的是添加更多的方法，那么问题不大——这用不着重新发布代码——除非你必须重新编译。但即便新方法替代了老方法的功能，也不应该去碰原来的接口（如果愿意的话，可将功能合并到基础实施代码中）。如果想添加更多的参数，从而为一个现有的方法扩展它的接口，那么请创建一个重载的方法，再在其中增加新参数。只有这样，才不会干扰以前对原有方法进行的调用。

(18) 仔细检查你的类，确保它符合逻辑。特别注意核实基类和派生类之间是一种“属于”关系，而成员对象是“拥有”关系。

(19) 在继承和合成这两种方式间犹豫不决时，问自己是否需要向上强制转型成基类型。如果答案是否定的，那么不要再犹豫了，最好选择合成（成员对象），而不是用继承。选择“合成”，可以避免同时出现多个基类型；而选择“继承”，用户会认为自己必须进行向上强制转型。

(20) 如果值要变化，那么使用数据成员；如果行为要变化，那么使用方法重载。换言之，假如某个类使用了状态变量，同时又有方法根据那些变量来切换不同的行为，那么也许应该重新设计一下你的系统，准确反映出子类和重载方法内的行为差异。

(21) 小心重载。最起码，你不应该把一个方法设计成条件执行代码（根据一个参数值）。此时，应考虑创建两个或者更多的重载方法。

(22) 使用违例体系（层次结构）——最好从标准 Java 违例体系里最恰当的一个类处派生。随后，捕捉违例的人就可以捕捉住特定类型的违例，然后是基类型。如果添加新的、派生的违例，原有的客户端代码仍然会通过基类型捕捉违例。

(23) 简单集合有时便足够了。在一架飞机上，“乘客服务系统”由一系列分离的元素构成：座位、空调、电视等等。在飞机上，你需要创建许多类似的东西。那么，是不是应该制作私有成员，然后构建一个全新的接口呢？答案是否定的！在这种情况下，组件也属于公共接口的一部分，所以应该创建公共成员对象。那些对象有它们自己的私有实现方式，这样仍然是安全的。要注意尽管简单集合并非经常都能管用，但有的时候，它已经足够了！

(24) 不要光顾着自己，设身处地为客户程序员以及代码维护人员想想。你的类设计完毕后，它的用法应该是一目了然的。先预测一下将来可能会作出哪些改变，再来设计自己的类，使那些改变尽可能简单地进行。

(25) 谨惕“巨大对象综合症”。对一些习惯于程序化编程（Procedural Programming）思维、且初涉 OOP 领域的新手来说，他们往往喜欢先写一个这样的程序，再把它嵌入一个或两个巨大的对象里。除应用程序框架之外，对象表达的应该是你的程序里的概念，而非程序本身！

(26) 若迫不得已，非要进行一些令人“惨不忍睹”的编程，至少应该把那些代码放到一个类里。

(27) 若迫不得已，非要进行一些无法移植的编程，请对其进行“抽象”处理，把它的作用域限制在一个类内。利用这一级额外的“迂回”，可有效避免“不可移植”的特性蔓延至整个程序。

(28) 对象不该只用来简单地放一些数据。它们还应该具有经过良好设计的行为（有的时候，“数据对象”不在这个范围之内。但只有在一个通用容器无法使用的前提下，才明确用这种“数据对象”来封装和传输一组项目）。

(29) 在老类基础上创建新类时，最开始直接用“合成”。只有随着设计的深入，在觉得特别有必要的前提下，才应该考虑换成“继承”。假如在本来用“合成”可以搞定的前提下

用了“继承”，你的设计就会变得无谓地复杂起来。

(30) 用继承和方法重载来表达行为差异，并用字段表达状态差异。一个例子是，你不应该继承不同的类来表达颜色，而应该用一个“color”字段。

(31) 小心“分歧”。两个表面不同的对象可能有一模一样的行动（或“职责”），而且很自然地，许多人会认为有必要将一个对象变成另一个对象的子类，目的仅仅是利用“继承”的好处。这便叫作“分歧”，但没有真正的理由需要强迫实现一个并不存在的超类/子类关系。一个更好的办法是创建一个通用基类，令其以派生类的形式，为两者都产生一个接口——尽管需要更多的代码，但仍然可以获得继承的好处，同时还有可能更准确地把握自己的整个设计。

(32) 小心继承时的“限制”。以住，我们认为最合理的设计是为继承的东西添加新功能；最糟糕的设计则是在继承期间删除旧功能，但同时又不添加新功能。但上述规则也有被打破的时候。如果你仍然需要使用一个老类库，那么更有效的一种做法是将一个老类限制到它的子类中，而不是重新调整整个分级结构，使自己的新类位于旧类之上。

(33) 利用设计范式来消除“裸功能”。也就是说，假若只需要为类创建一个对象，便用不着提前限制自己使用应用程序，加上一条“只生成其中一个”的注释，再把它封装成一个“独子”中。假如在主程序中用了大量散乱的代码来创建自己的对象，那么请寻找一种恰当的、专门用于创建的范式（比如一个 Factory 方法），将创建过程封装到其中。消除了这种“裸功能”之后，你的代码不仅更易理解、更易维护，也显得更加“健壮”。

(34) 警惕“分析瘫痪”。请记住，无论如何都要提前了解整个项目的状况，再去考察其中的细节。由于把握了全局，可快速认识自己未知的一些因素，防止由于一开始便考察细节而陷入“死逻辑”中。事实上，除非你真正“拥有”了一个方案，否则是没法子“认知”它的。Java 有其内建的防火墙，它们可让你大胆尝试，用不着害怕会造成伤害。谁都难免会犯错误；在类或类集里犯下的一点儿错误，不会破坏整个系统的完整性。

(35) 如果认为自己已进行了良好的分析、设计或者实施，那么请来一次全盘大检阅。这时可稍微换一下思维角度。试试邀请一些外来人士——并不一定是专家，但可以是来自本公司其他部门的人。请他们用完全新鲜的眼光考察你的工作，看看是否能找出你一度熟视无睹的问题。采取这种方式，往往能在最适合修改的阶段找出一些关键性问题，避免由于产品发行后才发现问题而造成的金钱及精力方面的损失。

C.2 实　　现

(36) 通常，你应遵守 Sun 制订的编码规范。规范详情可见 java.sun.com/docs/codeconv/index.html（我尽了最大可能让本书内的代码符合这一规范）。尽管要多付出一些劳动，但却是非常值得的。假如还是顽固地坚持使用自己一直习惯使用的“另类”编码样式，那么你的读者就会觉得“头大”。但是，不管你决定采用何种样式，都要保证贯穿整个程序项目，都坚持使用这种样式。另外，还有一个免费的 Java 代码自动格式化工具可供使用：home.wtal.de/software-solutions/jindent。

(37) 无论采用哪种编码样式，只有整个开发队伍都坚持采用它，才会真正有用（如果你的整个公司都采用它，那么效果会更好）。也就是说，在员工相互修改对方的代码时，不应该出现重新熟悉对方的编码样式的过程。一切都应该是自然而然的。标准统一之后，便可省下分析代码时消耗的大量精力，将更多的注意力放在代码的含义上。

(38) 大小写规则要统一。类名首字母应该大写。字段、方法以及对象（引用）的首字母应小写。对于所有标识符来说，其中包含的所有单词都应紧靠在一起，而且要大写中间所有单词的首字母。例如：

ThisIsAClassName

thisIsAMethodOrFieldName

若在定义中出现了常数初始化模块，则应大写 static final 原始数据类型标识符中的所有字母。这样便可标志出它们属于编译时间常数。

Java 封装（Package）属于一种特殊情况：它们全都是小写字母，即便中间的单词亦是如此。对于域扩展名来说，比如 com、org、net 或者 edu 等等，全部都应小写（这也是 Java 1.1 和 Java 2 的区别之一）。

(39) 不要创建自己的“装饰过的”私有数据成员名称。这通常表现为前置的下划线和字符。匈牙利记号法（Hungarian Notation）是其中最糟糕的例子。在这种记号法中，我们附加额外的字符，来表示数据类型、用途、位置等等——结果就是好象正在用某种汇编语言写程序，而且编译器根本没有提供额外的帮助。这类记号方式令人混淆、难于阅读，使人感觉十分“不爽”。正确的做法应该是，让类和封装自己为你规定名字的作用域。

(40) 对于一个常规用途的类，应该采取“正宗形式”加以创建。也就是说，你需要在其中包括对 equals()、hashCode()、toString()以及 clone()（实现 Cloneable）的定义，同时还要实现 Comparable 和 Serializable。

(41) 针对那些要读写 private 字段的方法，为其采用 JavaBeans 的“get”、“set”和“is”命名规范——即使当前并不想自己需要制作一个 JavaBean。这样一来，用户不仅可以把你的类当作一个 Bean 方便地使用，而且由于这类方法的名称已经统一，所以用户们也能更容易地理解它们。

(42) 针对你创建的每个类，都考虑包括进去一个 static public test()，在加上对那个类进行测试的代码。在项目中使用一个类时，并不需要删除它原先的测试代码。而且假如你以后进行了某种改动，那么也可以方便地重新测试一下。参照这些测试代码，你也能对类的使用方式做到心中有数。

(43) 有时需要继承才能访问基类的“受保护”成员。这便带来了同时存在多个基类型的问题。假如不需要向上强制转型，那么首先派生一个新类，以执行受保护的访问。然后在需要用到那个新类的任何一个类内，把新类变成后者的一个成员对象，而不是采用“继承”的方式。

(44) 尽量避免用 final 来提高效率。只有当程序运行时，你感觉不够快，而且分析工具告诉你一个方法调用已成为“瓶颈”，那么才应考虑使用 final。

(45) 假如两个类通过某种功能性方式（比如容器和继承器）相互联系在一起，那么请试着将一个类变成另一个类的内部类。这样做不仅强调了类和类之间的联系，也允许类名在单独一个封装内复用（通过把它封装到另一个类里）。Java 容器库为了达到这个目的，做法是在每个容器类里定义一个内部 Iterator 类，从而为容器赋予了一个通用的接口。还有另一个理由也促使我们使用内部类——需要将其作为 private（私有的）实现的一部分。不过在这种情况下，内部类的好处在于可将具体的实现隐藏起来——而不是在于象上面说的那样，突出类和类的联系，以及防止命名空间受到损害等等。

(46) 任何时候只要发现类与类之间结合得非常紧密，就需要考虑采用内部类，以便为代码的编写和维护带来一定的方便。使用内部类后，不会破坏类和类原有的那种联系。相反，它会使这种联系更加明显，而且更加方便。

(47) 过早的优化只会适得其反。特别要注意的是，在首次构建系统的时候，不要试着写固有方法，不要把某些方法变成 final，也不要优化代码的执行效率。在这个时候，你唯一的目的就是证明自己的设计可行。可行之后，再来考虑效率问题。

(48) 使一个作用域尽可能地小，使对象的可见性尽可能地小，生存时间也尽可能地短。这样做可避免由于不慎，在一个错误的场合下使用了一个对象；而且避免了将那些很难发现的 Bug 深深地隐藏起来。例如，假定你有一个容器，并有相应的代码对那个容器进行“遍

历”。假如复制那些代码，把它应用于某个新容器，那么极有可能不慎使用老容器的大小作为新容器的上边界。但是，假如老容器超出作用域，错误也会在编译时被捕捉到。

(49) 使用标准 Java 库提供的容器。熟悉了那些容器的用法后，可以显著提高编程效率。为序列用 ArrayList；为 Set 用 HashSet；为关联数组使用 HashMap；而为堆栈和队列使用 LinkedList（注意不要为堆栈使用 Stack）。

(50) 要想程序健壮，每个组件都必须健壮。在你创建的每个类中，使用由 Java 提供的所有工具——访问控制、违例、类型检查等等。只有这样，在你构建自己的系统时，才可放心地、安全地转移到下一级抽象。

(51) 编译时间错误比运行时间错误好。任何错误都是越提早发现越好。如果在产生一个违例的时候，便能使错误得到控制，那么便是最理想的结果。只要有了足够的信息对一个违例进行控制，一个控制器就应该及时地“切入”，捉住那个违例，尽快完成对它的处理。心中要有一个树形分级结构的大致形状，所有违例都最好是在当前这一级（当前的“分权”上）完成得以控制；当时，假如确实无法解决问题，就应该尽快重新“掷”出那个违例，让上一级去控制。

(52) 小心过长的方法定义。所有方法都应该是精简的、可以发挥某种功能的一个单元，它应该描述并实现了整个类接口的一个独立部分。假如方法过长和过于复杂，那么维护起来就显得非常困难，而且要花更大的代价。假如发现了这样的一个方法，至少应该把它分割成几个小方法。另外，你恐怕也得考虑一下创建新类。另外，即便是小方法，它们在你的类中也应该能够重复使用（尽管有的方法必须那样“大”，但仍然不应该同时负责多项操作。无论如何，它的功能应该是单一的）。

(53)　让一切东西都尽可能地“私有化”（private）。一旦库的某一部分“公共化”了（比如一个方法、类或者一个字段等等），那么便永远不能把它拿出来了。若强行拿出，就可能破坏其他人现有的代码，使他们不得不重新编写和设计。但是，假如只公布那些必须公布的，就可以放心大胆地、自由地改变其他任何东西。而且由于设计本身通常是在不断进步的，所以对你来说，这种“自由”显得非常可贵。这样一来，实现的改变就不会对派生类造成多大的影响。而且在多线程环境中，“私有化”显得尤其重要——只有 private 字段才能在非同步使用的情况下受到保护。

(54) 不要吝啬为程序加上注释，并用 javadoc 注释文档语法，产生你的程序文档。不过，注释也不能乱加，它们应该反映出代码的确切含义，令人能够一目了然。注意由于 Java 类和方法名本身的完善，许多注释都不是特别必要的。

(55) 避免使用“魔法数”，亦即用硬编码形式强行加到代码里的数字。这些数字很难与代码很好地配合。如以后需要修改它，无疑会成为一场噩梦，因为根本不知道“100”到底是指“数组大小”还是其他全然不同的东西。相反，我们应创建一个常数，并为其使用具有说服力的描述性名称，并在整个程序中都采用常数标识符。这样可使程序更易理解以及更易维护。

(56) 创建构造函数时，请注意违例。在最理想的情况下，构造函数做的一切事情都不会产生违例。而在次理想的情况下，所有类都通过“合成”产生，并从健壮的类继承，所以即使有一个违例产生，它们也不需要进行任何清除。否则的话，你必须在一个 finally 从句内清除合成的类。假如构造函数必须失败，那么最正确的做法是产生一个违例，使调用者不至于继续盲目操作——误以为对象已正确创建。

(57) 若对象已拿给客户程序员使用，但你的类又需要进行任何清除，那么请将清除代码放到一个定义良好的方法里——用一个象“cleanup()”这样的名字，清楚地指出它的功用。除此之外，还要在类里放一个 boolean 标记，指出对象是否已被清除，以便 finalize()能正确检查“死亡条件”——参见第 4 章（4.3.3 节）。

(58) finalize()的职责只能是检查一个对象的“死亡条件”，以便于程序调试（参见第 4 章）。在特殊情况下，可能需要强行回收内存，因为垃圾收集器暂时不能回收它。由于垃圾收集器或许不会针对你的对象进行调用，所以不能用 finalize()来执行必要的清除工作。考虑到这个原因，你必须创建自己的“清除”方法。在类的 finalize()方法中，务必保证对象已被清除；如果还没有，则必须产生一个自 RuntimeException 派生出来的类，从而指示一个程序错误。最后，在你正式决定采用这种方案之前，先要保证 finalize()在你的系统上能正常工作（也许要调用一下 System.gc()，查实系统是否支持这种行为）。

(59) 在一个特定的作用域内，假如一个对象必须被清除（而不是当作垃圾“收”走），可采用下述方法：初始化对象；如成功，而立即进入一个 try 块，用一个 finally 从句执行清除。

(60) 继承期间覆盖 finalize()时，记着调用 super.finalize()。但是，假如 Object 是紧接在前面的一个超类，便没必要这样做。在你覆盖的 finalize()中，对 suyper.finalize()的调用应当是采取的最后一个行动，而不应该是第一个。这样可保证在你需要的时候，基类组件仍然有效。

(61) 为对象创建固定大小的容器时，把它们传输到一个数组里。特别地，假如你打算从一个方法返回该容器，那么更应该这样做。这样一来，我们不仅可从数组的编译时间类型检查中获益，而且对数组的接收者来说也有好处——因为在需要使用它们的时候，可能并不需要对数组中的对象进行强制转型。要注意的是，容器库 java.util.Collection 的基类有两个 toArray()方法都可做到这一点。

(62) 接口比抽象类好。如果已知某样东西将来会成为一个基类，那么第一个反应就是把它变成一个“接口”。只有在不得不使用方法定义或成员变量的前提下，才可考虑把它变成一个“抽象”类。“接口”指出客户想做什么，而一个“类”把重点放在实现细节上。

(63) 在构造函数内，尽可能将对象设置成正确的状态。主动避免调用其他方法（final 方法除外），因为那些方法可能会被其他人覆盖，从而在构建期间导致不可预见的后果（详见第 7 章）。类似地，构造函数越简单，产生违例或带来问题的机会就越小。

(64) 为避免编程时遇到麻烦请保证在自己类路径指到的任何地方，每个名字都仅对应一个未封装的类。否则，编译器可能会先找到完全同名的另一个类，并报告毫无意义的出错消息。若怀疑自己碰到了类路径问题，请试试在类路径的每一个起点，搜索一下同名的.class 文件。理想情况下，所有类都应该放到封装里。

(65) 当心不慎重载。如试图覆盖一个基类方法，但由于拼写错误，所以最后实际是新增了一个方法，而不是覆盖一个原有的方法。然而，这样做在表面上又是完全合法的，所以无论编译器还是运行时间系统都不会报错。不管你如何努力地寻找错误，代码就是不能工作——这是令人多么尴尬的一件事情！

(66) 当心过早优化。先让它工作起来，再考虑加快速度——但若非必需，而且经过证实，某个代码段确实存在性能瓶颈，否则也用不着进行什么优化。这里可考虑使用性能分析工具，不要仅凭“感觉”。性能调节的隐含代价是相当高的，因为即使“优化”了性能，代码的可读性也会大打折扣，而且会变得不易维护。

(67) 请记住，阅读代码的时间比写代码的时间多得多。思路清晰的设计可获得易于理解的程序，但那些注释、详细解释以及一些恰到好处的例子，才是潜在价值真正不可估量的东西。不管对你，还是对后来的人，它们都是相当重要的。如对此仍有怀疑，那么请试着从联机 Java 文档中找一找自己需要的信息。经过那样的挫折后，或许就能够把你说服了。

附录 D　参考资料

D.1　软　　件

首先是来自 java.sun.com 的 JDK。即便你决定采用由其他厂商提供的一个开发环境，手上拥有 JDK 总是不错的——在你怀疑是一个编译器错误的时候，“正宗”的 JDK 可以解决你心中的疑难。JDK 就是我们的试金石；即使其中真的有一个 Bug，那么也必然是一个非常“著名”的 Bug。

其次是 HTML Java 文档，同样来自 java.sun.com。迄今为止，我还没有发现市面上有一本讲述标准 Java 库的参考书是没有过时的，或者没有“短斤少两”的。最新、最全面的参考资料非这份文档莫属！尽管由 Sun 提供的这份 HTML 文档存在这样或那样的小错误，而且有时也显得过于简洁，但最起码有一点它是能够保证的：涵括了所有类和所有方法，无一缺失！有的人刚开始可能不会很喜欢阅读屏幕上的东西，觉得还是手拿一本书来得方便。但是，联机文档仍然有它的价值。等你用浏览器打开这份 HTML 文档之后，便会立即感受到这一点。最起码，你可以获得一个全面的印象。当然，假如看电子版真的让你头痛，那么还是看书好了。书上出现一个解决不了的问题后，再来参照联机文档。

D.2　书　　籍

《Thinking in Java，第一版》。本书配套光盘提供了该书的电子版，采用完全索引式结构，全彩色语法标注，HTML 格式。另外，也可以从 www.BruceEckel.com 下载。书中包括一些较老的材料，以及在第二版中已经没必要再重复的一些内容。

《Core Java 2》，由 Horstmann 和 Cornell 合著。第 1 卷是“基本原理”，由 Prentice-Hall 于 1999 年出版；第 2 卷是“高级特性”，于 2000 年出版。庞大、包容面极广的一本参考书。在我碰到问题的时候，它往往是我寻求答案的第一个地方。不过，建议你在完成了《Thinking in Java》的阅读，需要加深学习的时候，再去阅读该书。

《Java in a Nutshell:A Desktop Quick Reference，第二版》，David Flanagan 著，O'Reilly & Assoc 于 1997 年出版。对 Java 联机文档的一个简要总结。就个人来说，我更喜欢从 java.sun.com 处在线阅读，特别是在它们更新得如此快的情况下。然而，许多人仍然喜欢印刷出来的书本，所以这本书仍然有它的用处。而且书中也提供了比联机文档更多的讨论。

《The Java Class Libraries:An Annotated Reference》，Patrick Chan 和 Rosanna Lee 合著，Addison-Wesley 于 1997 年出版。这本书是前一本书的“加厚”版本。它为我们心目中理想的联机参考资料提供了榜样：应该向读者提供足够多的说明，使其简单易用。《Thinking in Java》的一名技术审定员说道：“如果我只能有一本 Java 书，那么选的肯定是它。”不过我可没有他那么激动。尽管说明够多，但也太大、太贵，而且例子的质量并不能令我满意。但在遇到麻烦的时候，该书还是很有参考价值的。而且与《Java in a Nutshell》相比，它看起来有更大的深度（当然也有更多的文字）。

《Java Network Programming》，Elliote Rusty Harold 著，O'Reilly 出版社于 1997 年出版。在阅读这本书之前，我可以说根本不理解 Java 有关网络的问题。后来，我还发现他的 Web 站点“Cafe au Lait”，这是一个令人激动的、很有个性以及经常更新的去处，涉及大量有价值的 Java 开发资源，而且并不偏袒任何一家 Java 厂商。由于几乎每天更新，所以在这里能

看到与 Java 有关的大量新闻。站点地址是：http://metalab.unc.edu/javafaq/。

《JDBC Database Access with Java》，Hamilton、Cattell 和 Fisher 合著，Addison-Wesley 出版社于 1997 年出版。如果对 SQL 和数据库一无所知，这本书就可以作为一个相当好的起点。它也对 API 进行了详尽的解释，并提供了 API 的一个“注释参考”（同样地，这是我们心目中理想的联机参考文档的样子）。与“Java 系列”（由 JavaSoft 授权的唯一一套丛书）的其他所有书籍一样，这本书的缺点也是进行了过份的渲染，只说 Java 的好话——在这一系列书籍里找不到任何不利于 Java 的地方。

《Java Programming with CORBA》，Andreas Vogel 和 Keith Duddy 合著，Jonh Wiley & Sons 于 1997 年出版。针对三种主要的 Java ORB（Visbroker、Orbix 和 oe），本书分别用大量代码实例进行了详尽阐述。

《Design Patterns》，由 Gamma、Helm、Johnson 和 Vlissides 合著，Addison-Wesley 于 1995 年出版。这是在编程领域发动了一场范式（Pattern）革命的经典书籍。

《Practical Algorithms for Programmers》，Binstock 和 Rex 合著，Addison-Wesley 于 1995 年出版。专门讲“算法”的一本书。其中的所有算法都是用 C 描述的，所以它们很容易就能转换到 Java 里面。每种算法都有详尽的解释。

D.2.1 分析和设计

《Extreme Programming Explained》，Kent Beck 著，Addison-Wesley 于 2000 年出版。我非常喜欢这本书！是的，我平常倾向于采用传统方式来解决问题，但总觉得应该有一种全然不同的、好得多的程序开发之道。而“极度编程”（ExtremeProgramming，简称“XP”）和我心中想的正好完全合拍。唯一对我有类似影响的一本书是《PeopleWare》（后文有它的介绍），它主要谈论工作环境和企业文化的问题。当然，《Extreme Programming Explained》谈论的是编程，它提出的“极度编程”是一种非常令人着迷的编程思想，其间涉及到许多新的理论和概念。甚至更进一步，对于一个特定的问题，这种思想认为只要你不花太多的时间在上面，而且希望把它抛开，那么问题自然会迎刃而解（你会注意到这本书的封面并没有“UML 认证”标志）。看了这本书之后，我可以说以后判断是否要为一家公司工作，完全取决于他们的人是否在用“极度编程”。紧凑的结构、小巧的章节、容易阅读的排版风格、充满智慧的话语、令人激动的思考，一本好书的全部要素，在这里都得到了精采的再现。看着这本书，你会想象自己正是这样的一种气氛下工作，同时视野得到了极大的开拓，好象进入了一个全新的世界。

《UML Distilled，第二版》，Martin Fowler 著，Addison-Wesley 于 2000 年出版。首次接触 UML 时，会觉得它“讨厌透顶”，因为有太多的图表和细节需要自己熟悉。但按照 Fowler 的说法，大部分这样的东西都可以忽略不计，所以他大笔一挥，砍掉了许多细枝末节，向你展现出来的只是其中最本质的东西。对大多数项目来说，都只需要知道少数几个制图工具就可以了。Fowler 最终的目的是让你得到一个好设计，而不是让你和那些细枝末节纠缠不休。不错的一本书，短小精悍。如果你需要掌握 UML，那么就把它作为一个起点吧！

《UML Toolkit》，Hans-Erik Eriksson 和 Magnus Penker 合著，Jonh Wiley &Sons 于 1997 年出版。解释 UML 以及如何使用它，并用 Java 提供了实际案例供你参考。它的配套光盘提供了 Java 代码以及 Rational Rose 的一个简化版。本书对 UML 进行了非常出色的描述，并解释了如何用它构建真正的系统。

《The Unified Software Development Process》，由 Ivar Jacobsen、GradyBooch 和 James Rumbaugh 合著，Addison-Wesley 于 1999 年出版。刚开始，我已经做好了嫌恶这本书的一切准备。光看书名，会觉得它是一本无聊透顶的、充满说教的参考书。但翻开书页，我却完全惊呆了——书中只有一小部分包含了解释，感觉一般。但其他大多数内容不仅清晰易懂，而

且令人愉快。而且最难能可贵的是，它讲的内容颇为实用。它不是 Extreme Programming（关于测试，它讲得没有 XP 清楚），但它仍然是 UML 的一部分——即使你不能接受 XP，大多数人都超越了“UML 不错”这一境界（无论他们实际的 UML 经验如何），所以你也可能接受它。我认为这是 UML 的一本“旗舰”参考书。在你读完 Fowler 的《UML Distilled》之后，可以再来读这本书，加深对一些细节之处的学习。

在你正式选择任何方法之前，都有必要先从这里获得一些公正的看法。其实，人们很容易接受一个方法，同时没有理解自己想从它那里获得什么，也不知道它会为你提供什么。这是什么原因呢？因为其他人正在用它——这似乎就是一个决定性的因素。不过，这实际是一个人的心理问题：假如人们相信什么能解决自己的问题，就会主动地去尝试它（这是实验，应该支持）；但假如发现它解决不了自己的问题，那就不得了了，他们会夸张自己付出的努力，然后郑重宣告自己的“重大发现”（这是否定，不应该支持）。这里的假定是有其他人和你同舟共济，你并非独自一个人——即使根本不知这艘船驶向何方（或者是沉没？）。

当然，这里并非暗示你所有方法学理论都一无是处，只是指出你应该尽一切可能，坚守自己的心理防线，让自己停留在“实验模式”下（“它没用？没关系，让我们再试试其他方法”），而不是急切地进入“盲目模式”，立即去否定什么东西（“不，那不是真正的问题。所有事情都 OK，我们不需要改变。”）我认为在后面提到的一系列书的帮助下，可以为你提供这方面必要的工具。在你正式决定采用一种方法之前，务必先读一读它们。

《Software Creativity》，Robert Glass 著，Prentice-Hall 于 1995 年出版。这是我见过的、从整个方法学理论的角度进行全盘透视的最好的一本书。其中实际包含了一系列由 Glass 本人编写或者从别人那里收集来的（P.J. Plauger 是撰稿人之一）随笔和评论，反映了他在这个主题上经年的思考及研究。所有文章都非常精彩，长度适中，以说明问题为准，不会在你面前絮絮叨叨，纠缠不休。不过，他要说的一系列问题也并不是凭空捏造的；而是经过长时间的摸索和思考，最后总结出来的“经验之谈”。所有程序员和管理者在陷入方法学的泥沼之前，都应该先读一下这本书。

《Software Runaways:Monumental Software Disasters》，Robert Glass 著，Prentice-Hall 于 1997 年出版。这本书的亮点在于，它谈到了我们通常没有勇气谈论的东西：许多项目不仅失败，而且失败得莫名其妙！我们中的许多人仍然认为：“那怎么可能发生在我头上呢？”，或者“那种事下一次一定不会发生了”。这可是一种危险的想法。只有事先做足了灾难预防的措施，才能将以后的损失减轻到最低程度。

《Peopleware，第二版》，Tom Demarco 和 Timothy Lister 著，Dorset House 于 1999 年出版。尽管作者有软件开发的出身，但这本书主要讲的是涉及“人”的东西，包括项目和开发队伍等等。书中突出了人和人的要求，而不是技术和技术的要求。书中认为，人的工作环境不应该象“典型程序员的窝”那样一团糟，而应该令人舒适、富有效率。但是，也不能过于刻板，规定这、规定那，否则也发挥不出他们的工作热情。非常“人性化”的一本书，极易引起我们这些程序员的共鸣。

《Complexity》，M. Mitchell Waldrop 著，Simon & Schuster 于 1992 年出版。这本书是跨行业合作的一个结晶，书中的言论来自不同领域的科学家。有一年，这些人会聚新墨西哥州的圣塔弗市，讨论仅凭自己领域的力量所无法解决的现实问题，比如经济学解决不了股市问题；生物学解决不了生命的早期型态问题；……等等。通过联合物理、经济、化学、数学、计算机科学、社会学以及其他学科的精英，一种崭新的多学科方案正在酝酿中。但更重要的是，对超复杂现实问题的一种全新思维方式正在逐渐浮现：不要试图用一个等式来精确归纳所有行为，首先应该“观察”，然后寻找一种“范式”，最后尽一切可能，尝试完善那种范式，令其投入实用，最终解决那些复杂问题（例如，书中讲到了目前正在浮现的遗传学算法问题。生命密码的破解，是一个超复杂的问题，所以有必要用这种全新的方法学理论去解决）。我

相信，对于这种形式的“思维”来说，它对于复杂软件项目的开发也会颇有帮助。

D.2.2 Python

《Learning Python》，Mark Lutz 和 David Ascher 合著，O'Reilly 于 1999 年出版。Python 目前已成为我非常喜欢的一种语言，它是对 Java 的一种非常不错的补充。这本程序员参考书包括了对 JPython 的简要介绍，教你如何在一个程序中同时使用 Java 和 Python（JPython 解释器已编译成纯 Java 字节码，所以你实际并不需要做太多的工作）。两种语言结合后，可做出许多从前不敢想象的事情。

D.2.3 我的著作一览

按出版时间排列，现在并不是所有书都能从书店里买到了：

《Computer Interfacing with Pascal & C》。1988 年，我自己出版的一本书。现在只有从 www.BruceEckel.com 下载它的电子版。在当时写作它的年代，CP/M 正如日中天，DOS 只是一个看似没多大前途的“暴发户”。这其实是讲电子控制的一本书，我用高级语言控制计算机的并口，以便驱动各种电子设备。书中内容改编自我之前为《Micro Cornucopia》杂志写的专栏，那是我开始专栏文章写作生涯的第一本、也是最好的一本杂志。唉，Micro C 如昙花一现——在 Internet 问世之前的很长一段时间里，它便已经消声匿迹了。这本书的制作和出版为我带来了非常好的“感觉”。

《Using C++》，Osborne/McGraw-Hill 于 1989 年出版。我写作的首批 C++书籍之一。该书已经绝版，已被它的第二版取代，并更名为《C++ Inside & Out》。

《C++ Inside & Out》，Osborne/McGraw-Hill 于 1993 年出版。前面说过，这实际是《Using C++》的第二版。尽管这本书讲的 C++都十分准确，但那是 1992 年的事情。后来，随着语言本身的发展，我又计划用《Thinking in C++》来代替它。可在 www.BruceEckel.com 了解这本书的详情并下载源码。

《Thinking in C++，第一版》，Prentice-Hall 于 1995 年出版。

《Thinking in C++，第二版，第一卷》，Prentice-Hall 于 2000 年出版，可从 www.BruceEckel.com 下载。

《Black Belt C++, the Master's Collection》，我是这本书的编辑，M&TBooks 于 1994 年出版。这是一本“大杂烩”的书籍，包含了由许多 C++权威写作的章节，素材来自他们在“软件开发者大会”上的讲演（我是那个大会的主席）。不过，这本书的封面实在……所以我当时便下定决心，以后完全由我来控制所有的封面设计！

《Thinking in Java，第一版》，Prentice-Hall 于 1998 年出版。它荣获了《软件开发》杂志的“创新奖”、《Java Developer's Journal》的“最佳书籍编辑选择奖”以及《JavaWorld》的“读者选择奖”。电子版可从 www.BruceEckel.com 下载。

指导您利用万维网的语言，进行面向对象的程序设计

本书全文、修订内容及程序代码，可从 http://www.bruceeckel.com 下载

本书荣获：

- 2000 年由《JavaWorld》颁发的“读者选择奖”
- 1999 年由《Java Developer's Journal》颁发的“最佳书籍编辑选择奖”
- 1999 年由《Software Development Magazine》颁发的“创新奖”

从 Java 的基本语法到它最高级的特性（分布式计算、高级面向对象功能、多线程处理），本书都能对您有所裨益。Bruce Eckel 优美的行文，以及短小、精悍的示范程序，有助于您正确理解那些含义模糊的概念。

- 本书仅适用于 Java 2！
- 随配套光盘，奉送完整的“Thinking in C:Java 基础”多媒体课程
- 面向初学者和某种程度的专家
- 本书第一版（适用于 Java 1）随光盘全文奉送，亦可直接从 www.BruceEckel.com 下载
- 重点是 Java 语言本身，内容并不依赖某种具体的操作系统平台
- 透彻讲述基础知识，例证重要高级主题
- 300 多个能直接运行的 Java 程序，共计 15000 多行代码
- 配套光盘奉送源代码，亦可从网上下载
- 解释面向对象的基本理论，并强调与 Java 的关系
- 本书内容在线定期更新
- 配套光盘含有 15 个小时的现场授课，亦可在网上收听或收视
- 现场讨论会将定期举行，详情见 www.BruceEckel.com

读者如是说：“最好的 Java 参考书……绝对令人震惊”；“购买 Java 参考书最明智的选择”；“我见过的最棒的编程指南”。

Bruce Eckel 也是《Thinking in C++》的作者，该书曾获 1995 年“Software Development Jolt Award”最佳书籍大奖。作为一名有 20 年经验的编程专家，自 1986 年起，曾向世界许多国家、地区的学生教授过对象编程课程。最开始涉及的领域是 C++，现在大举进军 Java。“C++标准委员会”有表决权的成员之一，曾就“面向对象的程序设计”这一主题连续写过其他 5 本书，发表过 150 多篇文章，并是多家计算机杂志的专栏作家，其中包括《Web Techniques》的 Java 专栏。曾出席过 C++和 Java 的“软件开发者会议”，并分获“应用物理”与“计算机工程”的学士及硕士学位。

读 者 之 声

比我看过的其他 Java 书好多了……非常全面，举例恰到好处，全书布局显得颇具“智慧”。和其他许多 Java 书籍相比，我觉得它更成熟、连贯、更有说服力、更严谨。总之，写得非常好，是一本学习 Java 语言的好书！（Anatoly Vorobey，TechnionUniversity，Haifa，以色列）。

是我见过的最好的编程指南，其他任何语言的参考书都比不上。（Joakim Ziegler，FIX 系统管理员）

感谢你写出如此优秀的一本 Java 参考书。（Dr. Gavin Pillay，Registrar，King Edward VII Hospital，南非）

再次感谢您这本令人震惊的书。刚接触 Java 时，真有点儿不知所从的感觉（因为不是 C 程序员）。但是，你的书浅显易懂，使我能很快掌握了 Java——差不多就是阅读的速度吧。能从头掌握基本原理和概念的感觉真好，再也不用通过不断试验和频繁的出错来建立正确的概念模型了。希望不久便能有机会参加您的讲座。（Randall R. Hawley，Automation Technician，Eli Lilly & Co）

我迄今见过的最好的计算机语言参考书。（Tom Holland）

这是在编程语言领域，我读过的最好的一本 Java 参考书。（Ravindra Pai，Oracle 公司 SUNOS 产品系列）

这是关于 Java 的一本好书。非常不错，你写得太好了！书中涉及的深度真让人着迷和震惊。一旦正式出版，我肯定会买下它。我从 96 年 10 月便开始学 Java 了。通过比较几本书，你的书可纳入“必读”之列。几个月来，我一直在搞一个完全用 Java 写的产品。你的书巩固了我一些薄弱的地方，并大大延伸了我已有的知识。甚至在会见承包商的时候，我都会引用了书中的一些解释。它对我们的开发小组也非常有用。拿书中的一些知识来考察小组成员（比如数组和矢量的区别），甚至能判断出他们对 Java 的掌握有多深。（Steve Wilkinson，MCI 通信公司资深专家）

好书！我见过的最好的一本 Java 教材。（Jeff Sinclair，软件工程师，Kestral Computing 公司）

感谢你的《Thinking in Java》。终于有人能突破传统的计算机参考书模式，进入一个更全面、更深入的境界。我读过许多书，只有你的和 Patrick Winston 的书才在我心目中占据了一个位置。我已向客户郑重推荐这本书。再次对你的工作表示感谢！（Richard Brooks，Java 顾问，Sun 专业服务公司，达拉斯市）

其他书说的都是“Java 是什么”（讲语法和库函数），或者“Java 怎么用”（列举编程实例）。《Thinking in Java》显然与众不同，是我所知唯一一本解释“Java 为什么”的书，包括：为什么要这样设计、为什么要这样工作、为什么有时不能工作、为什么比 C++好、为什么没有 C++好，等等。尽管这本书也很好讲述了“是什么”和“怎么用”的问题，但它真正的特色并在于此。这本书特别适合那些想追根溯源的人。（Robert S. Stephenson）

感谢您出了如此优秀的一本书，我越来越爱不释手。我的学生们也喜欢它。（Chuck Iverson）

向你在《Thinking in Java》一书上的工作致敬。这本书对因特网的未来进行了最恰当的揭示，我这里只想对你说声“谢谢”，它非常有价值。（Patrick Barrell，Network Officer Mamco-QAF Mfg 公司）

市面上大多数 Java 参考书作为“新手指南”来看都是不错的。但它们的立意大多雷同，举的例子也有过于重复之嫌。从未见过象您这样的一本书，和那些书完全是两码事。我认为它是迄今为止最好的参考书。请快些出版吧！……另外，既然《Thinking in Java》都这么好，我还赶快去买了一本《Thinking in C++》。（George Laframboise，LightWorx 技术咨询公司）

从前给你写过信，主要是表达对《Thinking in C++》一书的赞叹（那本书在我的书架上一直占有醒目位置）。今天，我很欣慰地看到你转向了 Java 领域，并有幸拜读了最新的《Thinking in Java》电子版。看过之后，我不得不说：“服了！”内容非常精彩，有很强的说服力，不象一些干巴巴的参考书。你讲到了 Java 开发中最重要、但也最易忽略的一个方面：基本原理！（Sean Brady）

你举的例子都非常浅显，很容易理解。Java 的许多重要细节都照顾到了，而单薄的 Java 开发文档根本没涉及那些方面。另外，这本书并没有浪费读者们的时间。程序员已经知道了一些基本的事实，你只是在这个基础上进行了很好的发挥。（Kai Engert，Innovative Software 公司，德国）

我是《Thinking in C++》的忠实读者。通读了您的 Java 书电子版后，发现您在这两本书上有同样高级别的写作水平。谢谢！（Peter R. Neuwald）

写得非常好的一本 Java 书……我认为您的工作简直可以说“伟大”。我是芝加哥地区 Java 特别兴趣组的头儿，已在最近的几次聚会上推荐了您的书和 Web 站点。以后每个月开 SIG 会的时候，我都会把《Thinking in Java》作为基本教材使用。一般来说，我们会每次讨论书内的一章内容。（Mark Ertes）

衷心感谢你的书，它写得太棒了。我已把它推荐给自己的用户和 Ph.D.学生。（Hugues Leroy//Irisa-Inria Rennes France，Head of Scientific Computingand Industrial Tranfert）

我现在只读了《Thinking in Java》前 40 页的内容，但它已给我留下了深刻印象。这无疑是我见过的最精彩的一本编程专业参考书……而我本身就是一名作家，所以这点儿眼光还是有的吧！我已订购了《Thinking in C++》，已等得迫不及待了——我是一名编程新手，最怕的就是散乱无章的学习线索。所以必须在这里向您的出色工作表示敬意。从前看过的书似乎都有这方面的毛病，仅翻了几页，便使我才提起的兴致消弥无踪。但看了你的书以后，却越来越有兴趣，不至于再出现失去劲头的情况。（Glenn Becker，EducationalTheatre Association）

谢谢您这本出色的书。在终于认识了 Java 与 C++之间纠缠不清的一些事实后，我真的要非常感谢这本书。对您的书非常满意！（Felix Bizaoui，Twin OaksIndustries，Louisa，Va）

恭喜你写出这么好的一本书。我是在看了《Thinking in C++》一书之后，再来读这本《Thinking in Java》的，它果然没让我失望！（Jaco van derMerwe，软件专家，DataFusion Systems 有限公司，Stellenbosch，南非）

这是我看过的最好的 Java 书之一。（E.E. Pritchard，资深软件工程师，英国剑桥动画系统有限公司）

你的东东让其他 Java 参考书黯然失色，看来其他作者都该向你看齐了。（Brett g Porter，资深程序员，Art & Logic）

我花了一、两个星期的时间来看你的书，并对我从前看过的一些 Java 书进行了比较。显然，只有你的书才能让我真正“入门”。现在，我已向许多朋友推荐了这本书，他们都对其作出了很高评价。请接受我最真诚的祝贺，并希望她早些正式出版。（Rama Krishna Bhupathi，软件工程师，TCSI 公司，圣琼斯）

这是一本充满智慧的书，与简单的参考书有着截然不同的风格。它现在已成了我进行 Java 编程的一份主要参考。特别是你的目录做得非常不错，让人一目了然，很快便能找到需要的东西。更让人高兴的是，这本书并没有变成一本改头换面的 API 字典，也没把我们这

些程序员当菜鸟看待。（Grant Sayer，Java Components Group Leader，Ceedata Systems Pty 有限公司，澳大利亚）

嗬嗬，一本可读性极强、论据非常充分的 Java 书。外面有太多内容贫瘠的 Java 书（当然也有几本稍好的），只有你的书才是最好的。那些垃圾在这本书面前不值一提。（John Root，Web 开发员，伦敦社会安全部）

我刚开始看《Thinking in Java》。我希望它能有更大的突破，因为《Thinking in C++》写得实在太好了。我是一名 C++程序员，事先看那本书对学习 Java 很有帮助。但我在 Java 上的经验不够，希望这本新书能让我满意。您真是一名“高产优质”作者。（Kevin K. Lewis，ObjectSpace 公司技术员）

我认为这是本好书。从这本书中，我学到了与 Java 有关的所有知识。谢谢你让这本书透过互联网免费发行。通过这本书，我在 Java 编程上已有了巨大进步。最令人高兴的是，你的书并没有变成一本官方 Java 手册，而是非常技巧地指出了 Java 一些不足的地方。你可真是做了一件大善事！（Frederik Fix, 比利时）

我现在经常查阅你的书。约两年前，在我想学习 C++的时候，是《C++ Inside&Out》带领我游历 C++的世界。它使我在这方面的技能大增，并因此找到了一个好职位。现在，由于工作方面的原因，又要开始学 Java 了。这个时候，又是《Thinking in Java》给我正确的指引。尽管现在可供选择的书相当多，但我知道自己已别无选择。很奇妙，不是吗？现在看这本书的时候，我居然有一种重新认识自己的感觉。衷心感谢你，我现在的理解又比以前深入多了。（Anand Kumar S.，软件工程师，Computervision 公司，印度）

你的书给人一种“鹤立鸡群”的感觉。（Peter Robinson，剑桥大学计算机实验室）

这是我看过的最好的一本 Java 参考书。现在想起来，能找到这样的一本书简直是幸运。谢谢！（Chuck Peterson，因特网产品线主管，IVIS International 公司）

这本书太棒了！它已是我看过的第三本 Java 书了，真后悔没早点儿发现它。前两本书都没坚持看完，但我已决心看完这一本。不妨告诉你，当时我是想寻找关于内部类使用的一些资料，是我的朋友告诉我能从网上下载这本书。你真是个好人呐！（Jerry Nowlin，MTS，Lucent Technologies）

在我看过的共计 6 本 Java 书中，你的《Thinking in Java》是最好和最有用的。（Michael Van Waas，Ph.D，TMR Associates 公司总裁）

我很想对《Thinking in Java》说声谢谢。这是一本多么出色的书——并不单指它在网上免费发行！作为一名学生，我认为你的书有不可估量的价值（我有《C++ Inside&Out》的拷贝，那是讲 C++的一本好书），因为它不仅教我怎样做，而且解释了为什么。这当然为我用 C++或 Java 这样的语言打下了坚实基础。我的许多朋友都象我一样热爱编程，在向他们推荐了这本书后，反映都非常好，他们的看法同我一样。再次感谢您！顺便说一句，我是一个印尼人，整天都喜欢同 Java 泡在一起！（Ray Frederick Djajadinata，Trisakti 大学学生）

你把这本书放在网上，引起了相当程度的轰动，我对你的做法表示真诚的感谢与支持！（Shane LeBouthillier，加拿大艾伯特大学计算机工程系学生）

告诉你吧，我是多么热烈地盼望读到你每个月的专栏！作为 OOP（面向对象编程）的新手，我惊叹于你能把即便最复杂的概念都讲得那么透彻和全面。我已下载了你的书，但我保证会在它正式出版后另行购买。感谢你提供的所有帮助！（DanCashmer，B.C.Ziegler & Co.）

祝贺你完成了一件伟大的作品。我现在下载的是《Thinking in Java》的 PDF 版。这本书还没有读完呢，便迫不及待地跑到书店去找你的《Thinking in C++》。我在计算机界干了 8 年，是一名顾问兼软件工程师、教师 / 培训专家，最近辞职自己开了一间公司。以前见过不少书，但是，正是这些书才使我的女朋友称我为“书呆子”！并不是我概念掌握得不深入——只是由于现在的发展太快，使我短期内不能适应新技术。但这两本书都给了我很大的启示，

它与以前接触过或买过的计算机参考书都大不相同。写作风格很棒，每个新概念都讲得很好，书中充满了“智慧”。（Simon Goland，simonsez@smartt.com，Simon Says Consulting 公司）

必须说，你的《Thinking in Java》非常优秀！那正是我一直以来梦想的参考书。其中印象最深的是用 Java 作软件设计时的一些优缺点分析。（Dirk Duehr，Lexikon Verlag，Bertelsmann AG，德国）

谢谢您写出两本空前绝后的书：《Thinking in Java》和《Thinking in C++》。它们使我在面向对象的程序设计上跨出了一大步。（Donald Lawson，DCLEnterprises）

谢谢你花时间写出一本真正有用的 Java 参考书，你现在绝对可以为自己的工作感到骄傲了。（Dominic Turner，GEAC Support）

这是我见过的最好的一本 Java 书。（Jean-Yves MENGANT，Chief SoftwareArchitect NAT-SYSTEM，法国巴黎）

《Thinking in Java》无论在覆盖的范围还是讲述方法上都有独到之处。看懂这本书非常容易，摘录的代码也很有说服力。（Ron Chan，Ph.D，Expert Choice 公司，Pittsburgh PA）

你的书太棒了！我看过许多编程书刊，只有你的书才能赋予人一种全新的视野。其他作者都该好好向你学习才是。（Ningjian Wang，信息系统工程师，The Vangurad Group）

《Thinking in Java》是一本出色的、可读性极强的书，我已向我的学生推荐阅读。（Dr. Pual Gorman，计算机科学系，Otago 大学，Dunedin 市，新西兰）

在我看过的书中，你的书最有品味，不象有的书那样粗制滥造。任何搞软件开发的人都不应错过。（Jose Suriol，Scylax 公司）

感谢您免费提供这本书，它是我看过（或翻过:P）的最好的一本。（JeffLapchinsky，Net Results Technologies 公司程序员）

这本书简明扼要，看起来不仅毫不费力，而且象是一种享受。（Keith Itchie，Java 研发组，KL Group 公司）

这真是我看过的最好的一本 Java 参考书！（Daniel Eng）

我看过的最好的 Java 书！（Rich Hoffarth，Senior Architect，West Group）

感谢你这本出色的参考书，我好久都没有如此愉快的阅读体验了。（Fred Trimble，Actium 公司）

你的写作能准确把握轻重缓急，并能成功抓住细节，让本来枯燥的学习过程变成了一件轻松有趣的事情，我觉得满意，非常满意！谢谢你这本出色的学习教程。（Rajesh Rau，软件顾问）

《Thinking in Java》让整个免费世界都感受到了震憾！（Miko O'Sullivan，Idocs 公司总裁）

关于《C++编程思想》

荣获 1995 年由《软件开发》杂志评选的“最佳书籍”奖！

“这本书可算一个完美的典型。把它放到自己的书架上绝对不会后悔。关于 IO 数据流的那部分内容包含了迄今为止我看过的最全面、最容易理解的文字。”（AlStevens，《道伯博士》杂志投稿编辑）

“Eckel 的书是唯一一本清楚解释了面向对象程序设计基础问题的书。从这本书中，你能深刻体会到 C++的各种优缺点。”（Andrew Binstock，《Unix Review》编辑）”

“Bruce 用他对 C++深刻的洞察力震惊了我们，《Thinking in C++》无疑是各种伟大思想的出色组合。如果想得到各种困难的 C++问题的答案，请购买这本杰出的参考书”（Gary Entsminger，《对象之道》的作者）

“《Thinking in C++》非常有耐心和技巧性地讲述了关于 C++的各种问题，包括如何使用内联、索引、运算符重载以及动态对象。另外还包括一些高级主题，比如模板的正确使用、违例和多重继承等。所有这些东西都精巧编织在一起，成为 Eckel 独特的对象及程序设计思想。所有 C++开发者的书架上都应该摆上一本。如果你在用 C++搞开发，这本书绝对有借鉴价值。”（Richard Hale Shaw，《PC Magazine》编辑）。

致谢

献给那些至今仍在孜孜不倦创造下一代计算机语言的人们。