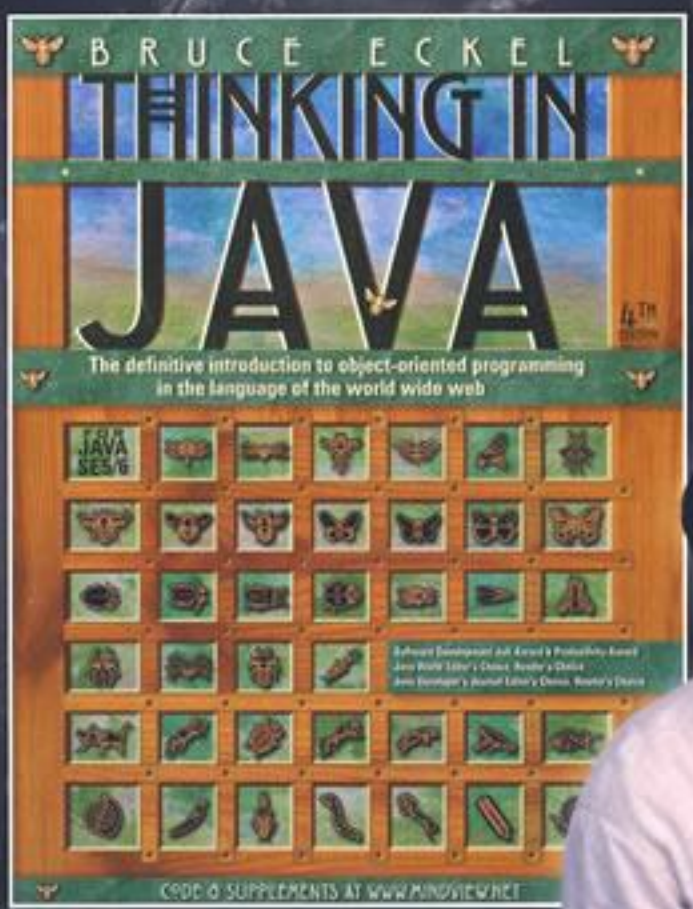
计 算 机 科 学 丛 书

Java编程思想

(美) Bruce Eckel 著 陈昊鹏 译

Thinking in Java

Fourth Edition



第4版

机械工业出版社
China Machine Press

《Thinking In Java》中文版

作者：Bruce Eckel
主页：http://www.BruceEckel.com
编译：Trans Bot
主页：http://member.netease.com/~transbot
致谢

——献给那些直到现在仍在孜孜不倦创造下一代计算机语言的人们！

指导您利用万维网的语言进行面向对象的程序设计
完整的正文、更新内容及程序代码可从http://www.bruceeckel.com下载
从Java的基本语法到它最高级的特性（网络编程、高级面向对象能力、多线程），《Thinking In Java》都能对您有所裨益。Bruce Eckel优美的行文以及短小、精悍的程序示例有助于您理解含义模糊的概念。
面向初学者和某种程度的专家
教授Java语言，而不是与平台有关的理论
覆盖Java 1.2的大多数重要方面：Swing和新集合
系统讲述Java的高级理论：网络编程、多线程处理、虚拟机性能以及同非Java代码的连接
320个有用的Java程序，15000行以上代码
解释面向对象基本理论，从继承到设计方案
来自与众不同的获奖作者Bruce Eckel
可通过万维网免费索取源码和持续更新的本书电子版
从www.BruceEckel.com获得配套CD（含15小时以上的合成语音授课）
读者如是说：“最好的Java参考书……绝对让人震惊”；“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SoftwareDevelopment Jolt Award最佳书籍大奖。作为一名有20经验的编程专家，曾教授过世界上许多地区的人进行对象编程。最开始涉及的领域是C++，现在也进军Java。他是C++标准协会有表决权的成员之一，曾就面向对象程序设计这一主题写过其他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”的学士和硕士学位。

读者的心声

> 比我看过的Java书好多了……非常全面，举例都恰到好处，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，肯定是一本学习Java的好书。（Anatoly Vorobey，TechnionUniversity，Haifa，以色列）。
>
> 是我见过的最好的编程指南，对任何语言都不外如是。（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。我以前真的有点儿不知所从的感觉（因为不是C程序员），但你的书浅显易懂，使我能很快掌握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，再也不用通过不断的试验和出错来建立概念模型了。希望不久能有机会参加您的讲座。（Randall R. Hawley，Automation Technician，Eli Lilly & Co）
>
> 我迄今为止看过的最好的计算机参考书。（Tom Holland）
>
> 这是我读过的关于程序设计的最好的一本书……第16章有关设计方案的内容是我这么久以来看过的

最有价值的。（Han Finci，助教，计算机科学学院，耶路撒冷希伯来大学，以色列）

有史以来最好的一本 Java 参考书。（Ravindra Pai，Oracle 公司 SUNOS 产品线）

这是关于 Java 的一本好书。非常不错，你干得太好了！书中涉及的深度真让人震惊。一旦正式出版，我肯定会买下它。我从 96 年十月就开始学习 Java 了。通过比较几本书，你的书可以纳入“必读”之列。这几个月来，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，并大大延伸了我已知的东西。甚至在会见承包商的时候，我都引用了书中的一些解释，它对我们的开发小组太有用了。通过询问组内成员我从书中学来的知识（比如数组和矢量的区别），可以判断他们对 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。再次感谢。（Richard Brooks，Java 顾问，Sun 专业服务公司，达拉斯市）

其他书讨论的都是 Java“是什么”（讲述语法和库），或者 Java“怎样用”（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：为什么象这样设计，为什么象这样工作，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“怎样用”的问题，但它的特色并在于此。这本书特别适合那些想追根溯源的人。（Robert S. Stephenson）

感谢您写出这么一本优秀的书，我对它越来越爱不释手。我的学生也喜欢它。（Chuck Iverson）

向你在《Thinking in Java》的工作致敬。这本书对因特网的未来进行了最恰当的揭示，我只是想对你说声“谢谢”，它非常有价值。（Patrick Barrell，Network Officer Mamco-QAF Mfg 公司）

市面上大多数 Java 书作为新手指南都是不错的。但它们的立意大多雷同，举的例子也有过多的重复。从未没见过象您这样的一本书，它和那些书完全是两码事。我认为它是迄今为止最好的一本参考书。请快些出版它！……另外，由于《Thinking in Java》都这么好了，我也赶快去买了一本《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++》一书的惊叹（那本书在我的书架上占有突出的位置）。今天，我很欣慰地看到你投向了 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，不象读那些干巴巴的参考书。你讲到了 Java 开发最重要、也最易忽略的方面：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 文档根本没有涉及那些方面。另外，这本书没有浪费读者的时间。程序员已经知道了一些基本的事实，你在这个基础上进行了很好的发挥。（Kai Engert，Innovative Software 公司，德国）

我是您的《Thinking in C++》的忠实读者。通读了您的 Java 书的电子版以后，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，我都想把《Thinking in Java》作为基本的指导教材使用。一般来说，我们会每次讨论书中的一章内容。（Mark Ertes）

衷心感谢你的书，它写得太好了。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）

我到现在只读了《Thinking in Java》的40页内容，但已对它留下了深刻的印象。这无疑是见过的最精彩的编程专业书……而且我本身就是一个作家，所以这点儿看法还是有些权威吧。我已订购了《Thinking in C++》，已经等得迫不及待了——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。以前看过的书似乎都有这方面的毛病，经常使我才提起的兴致消弥于无形。但看了你的书以后，感觉好多了。（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。我是在有了阅读《Thinking in C++》的经历以后，才来看这本《Thinking in Java》的，它确实没让我失望。（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参考收黯然失色。看来其他作者都应该向你看齐了。（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，并对以前我看过的一些Java书进行了比较。显然，只有你的书才能让我真正“入门”。现在，我已向我的许多朋友推荐了这本书，他们都对其作出了很高的评价。请接受我真诚的祝贺，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创作一份主要参考。你的目录做得相当不错，让人一目了然，很快就能找到自己需要的东西。更高兴的是，这本书没有写成一本改头换面的API字典，也没有把我们这些程序员看作傻瓜。（Grant Sayer，Java Components Group Leader，Ceedata Systems Pty有限公司，澳大利亚）

啧啧，一本可读性强、论据充分的Java书。外面有太多用词贫乏的Java书（也有几本好的），只有你的书是最好的。那些垃圾在你的书前面不值一提。（John Root，Web开发员，伦敦社会安全部）

我刚刚开始看《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。我是一名有经验的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“高产高质”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。谢谢你能让这本书通过互联网免费发行。如果不那样做，我根本不可能象现在这样有巨大的进步。但最令人高兴的是，你的书并没有成为一本官方Java手册，指出了Java一些不当的地方。你真是做了一件大好事。（Frederik Fix，Belgium）

我现在经常查阅你的书。大约两年前，当我想开始学习C++的时候，是《C++ Inside&Out》指导我游历C++的世界。它使我在这方面的技能大增，并找到了一个较好的职位。现在出于工作上的原因需要学习Java，又是《Thinking in Java》给我正确的指引。尽管现在可选择的书更多了，但我知道自己别无选择。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，真后悔没有早点儿发现它。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，是我的朋友告诉我网上能下载这本书。你干得真不错！（Jerry Nowlin，MTS，Lucent Technologies）

在我看过的 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送！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，那是关于C++的另一本好书），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 这样的语言编程打下了坚实的基础。我有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。顺便提一句，我是一个印尼畜牲，整天都喜欢和 Java 泡在一起！（Ray Frederick Djajadinata，Trisakti 大学学生，Indonesian Pork）

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 设计的新手，我要感谢你把即使最基本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！（Dan Cashmer，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，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，是一个顾问，兼软件工程师、教师／培训专家，最近辞职自己开了一间公司。所以见过不少的书。但是，正是这些书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）

必须认为你的《Thinking in Java》非常优秀！那正是我一直以来梦想的参考书。其中印象最深的是有关使用 Java 1.1 作软件设计时的一些优缺点分析。（DirkDuehr，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，你现在绝对能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 Software Architect 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，摘录的代码段也很有说服力。（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，只有你的书给人一种全新的视野。其他作者都该向你好好学习才是。（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 Ritchie，Java 研发组，KL Group 公司）

这真的是我看过的最好的一本 Java 书！（Daniel Eng）

我看过的最好的 Java 书！（Rich Hoffarth，Senior Architect，West Group）

感谢你这本出色的书籍，我好久都没有经历让人如此愉悦的阅读过程了。（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。这本书让学习变成了一件有趣的事情，我感觉满意，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》让整个自由世界都感受到了震憾！（Miko O'Sullivan，Idocs 公司总裁）

关于《Thinking in C++》：

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。”（Al Stevens，《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。这本书也是 C++的一本出色教材。”（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++》非常耐心和有技巧地讲述了关于 C++的各种问题，包括如何使用内联、索引、运算符过载以及动态对象。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。所有这些都精巧地编织在一起，成为 Eckel 独特的对象和程序设计思想。所有 C++开发者的书架上都应摆上这本书。如果你正在用 C++搞正式开发，这本书绝对有借鉴价值。”（Richard Hale Shaw，《PC Magazine》投稿编辑）。

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，就知道绝对不能就事论事，必须以人为中心。遗传对我们的生命有非常大的影响，但不能十分确定它能抹淡计算机革命——或至少信息革命——的影响。信息涉及人相互间的沟通：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关系，而其中最关键的就是通信。

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。“这样做还有谁来买它呢？”他们问。假如我是一个十分守旧的人，那么绝对不这样干。但我确实不想再沿原来的老路再写一本计算机参考书了。我不知道最终会发生什么事情，但的确认为这是我对一本书作出的最明智的一个决定。

至少有一件事是可以肯定的，人们开始向我发送纠错反馈。这是一个令人震惊的体验，因为读者会看到书中的每一个角落，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，往往发现自己确实有不当之处，而这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的一个可喜的反映，它使这本书显得的确与众不同。

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，开始意识到它的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：对一个准备解决的问题，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，由此决定了它们的主要设计目标就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。但从另一角度看，实际设计时肯定会有另一些问题浮现出来，需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让程序员头痛的。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），同时又要保证编程的效率。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额外的复杂性，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有关的紧密的联系。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。这段节省下来的时间可以产生更大的效益，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，其中任何每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（仅仅电子函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 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。我们认为认清真正的革命发生在哪里，因为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，甚至在星球之间。有人预言下一次大革命的发生就是由于足够多的人和足够多的相互连接造成的，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！

引言

同人类任何语言一样，Java 为我们提供了一种表达思想的方式。如操作得当，同其他方式相比，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显露无遗。

不可将 Java 简单想象成一系列特性的集合；如孤立地看，有些特性是没有任何意义的。只有在考虑“设计”、而非考虑简单的编码时，才可真正体会到 Java 的强大。为了按这种方式理解 Java，首先必须掌握它与编程的一些基本概念。本书讨论了编程问题、它们为何会成为问题以及 Java 用以解决它们的方法。所以，我对每一章的解释都建立在如何用语言解决一种特定类型的问题基础上。按这种方式，我希望引导您一步一步地进入 Java 的世界，使其最终成为您最自然的一种语言。

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的对应地方，然后自行演绎出答案。事实上，学习任何语言时，脑海里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

本书假定读者对编程多少有些熟悉。应已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。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并无什么区别。总之，只要能习惯基本的编程概念，就可顺利阅读本书。当然，C/C++程序员在阅读时能占到更多的便宜。但即使不熟悉 C，一样不要把自己排除在外（尽管以后的学习要付出更大的努力）。我会讲述面向对象编程的概念，以及 Java 的基本控制机制，所以不用担心自己会打不好基础。况且，您需要学习的第一类知识就会涉及到基本的流程控制语句。

尽管经常都会谈及 C 和 C++语言的一些特性，但并没有打算使它们成为内部参考，而是想帮助所有程序员都能正确地看待那两种语言。毕竟，Java 是从它们那里衍生出来的。我将试着尽可能地简化这些引用和参考，并合理地解释一名非 C/C++程序员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），我开始教授那种语言。程序设计语言的教授已成为我的专业。自 1989 年以来，我便在世界各地见过许多昏昏欲睡、满脸茫然以及困惑不解的面容。开始在室内面向较少的一组人授课以后，我从作业中发现了一些特别的问题。即使那些上课面带会心的微笑或者频频点头的学生，对许多问题也存在认识上的混淆。在过去几年间的“软件开发会议”上，由我主持 C++分组讨论会（现在变成了 Java 讨论会）。有的演讲人试图在很短的时间内向听众灌输过多的主题。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。这可能是由于问得太多了，但由于我是那些采取传统授课方式的人之一，所以很想使每个人都能跟上讲课进度。

有段时间，我编制了大量教学简报。经过不断的试验和修订（或称“反复”，这是在 Java 程序设计中非常有用的一项技术），最后成功地在一门课程中集成了从我的教学经验中总结出来的所有东西——我在很长一段时间里都在使用。其中由一系列离散的、易于消化的小步骤组成，而且每个小课程结束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）。

从每一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心，直到我最后认为它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，并按照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，而没有太大的可能参加此类专业研讨会。

3． 目标

就象我的前一本书《Thinking in C++》一样，这本书面向语言的教授进行了良好的结构与组织。特别地，我的目标是建立一套有序的机制，可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，可以在合理的时间内教完。随后是一些精心挑选的练习，可以在课堂上当即完成。

在这本书中，我想达到的目标总结如下：

(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前面的内容。
(2) 采用的示例尽可能简短。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。另一方面，这样做无疑会有些人会批评我采用了“不真实的例子”，但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等价的文档系统。而目前出版的与Java有关的几乎所有书籍都重复了这份文档。所以你要么已经拥有了它，要么需要下载。所以除非特别必要，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（电子版的东西更新也快）。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形性的概念。这一章会向大家提出一些对象创建的基本问题，比如构建器、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
本章将大家带到可以着手写自己的第一个Java程序的地方，所以必须对一些基本概念作出解释，其中包括对象“句柄”的概念；怎样创建一个对象；对基本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、自变量以及返回值；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，以及涉及造型、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
本章开始介绍构建器，它的作用是担保初始化的正确实现。对构建器的定义要涉及函数过载的概念（因为可能同时有几个构建器）。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：隐藏实现过程
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：类再生
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）。通过继承来重复使用原有的代码时（再生），一般需要保持“基础类”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：多形性
若由你自己来干，可能要花 9 个月的时间才能发现和理解多形性的问题，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础类对那个系列中的对象进行操作。通过 Java 的多形性概念，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必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。此外，Java 还通过“接口”提供了设置再生关系的第三种途径。这儿所谓的“接口”是对对象物理“接口”一种纯粹的抽象。一旦理解了多形性的概念，接口的含义就很容易解释了。本章也向大家介绍了 Java 1.1 的“内部类”。

(8) 第 8 章：对象的容纳
对一个非常简单的程序来说，它可能只拥有一个固定数量的对象，而且对象的“生存时间”或者“存在时间”是已知的。但是通常，我们的程序会在不定的时间创建新对象，只有在程序运行时才可了解到它们的详情。此外，除非进入运行期，否则无法知道所需对象的数量，甚至无法得知它们确切的类型。为解决这个常见的程序设计问题，我们需要拥有一种能力，可在任何时间、任何地点创建任何数量的对象。本章的宗旨便是探讨在使用对象的同时用来容纳它们的一些 Java 工具：从简单的数组到复杂的集合（数据结构），如 Vector 和 Hashtable 等。最后，我们还会深入讨论新型和改进过的 Java 1.2 集合库。

(9) 第 9 章：违例差错控制
Java 最基本的设计宗旨之一便是组织错误的代码不会真的运行起来。编译器会尽可能捕获问题。但某些情况下，除非进入运行期，否则问题是不会被发现的。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“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，违例发生在构建器中怎么办，以及违例控制器如何定位等等。

(10) 第 10 章：Java IO 系统
理论上，我们可将任何程序分割为三部分：输入、处理和输出。这意味着 IO（输入／输出）是所有程序最为

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”IO 和 Java 1.1 的“新”IO 将得到着重强调。除此之外，本节还要探讨如何获取一个对象、对其进行“流式”加工（使其能置入磁盘或通过网络传送）以及重新构建它等等。这些操作在 Java 的 1.1 版中都可以自动完成。另外，我们也要讨论 Java 1.1 的压缩库，它将用在 Java 的归档文件格式中（JAR）。

(11) 第 11 章：运行期类型鉴定
若只有指向基础类的一个句柄，Java 的运行期类型标鉴定（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，让 Java 的动态绑定机制（多形性）为那一类型实现正确的行为。但在某些场合下，对于只有一个基础句柄的对象，我们仍然特别有必要了解它的准确类型是什么。拥有这个资料后，通常可以更有效地执行一次特殊情况下的操作。本章将解释 RTTI 的用途、如何使用以及在适当的时候如何放弃它。此外，Java 1.1 的“反射”特性也会在这里得到介绍。

(12) 第 12 章：传递和返回对象
由于我们在 Java 中同对象沟通的唯一途径是“句柄”，所以将对象传递到一个函数里以及从那个函数返回一个对象的概念就显得非常有趣了。本章将解释在函数中进出时，什么才是为了管理对象需要了解的。同时也会讲述 String（字串）类的概念，它用一种不同的方式解决了同样的问题。

(13) 第 13 章：创建窗口和程序片
Java 配套提供了“抽象 Windows 工具包”（AWT）。这实际是一系列类的集合，能以一种可移植的形式解决视窗操纵问题。这些窗口化程序既可以程序片的形式出现，亦可作为独立的应用程序使用。本章将向大家介绍 AWT 以及网上程序片的创建过程。我们也会探讨 AWT 的优缺点以及 Java 1.1 在 GUI 方面的一些改进。同时，重要的“Java Beans”技术也会在这里得到强调。Java Beans 是创建“快速应用开发”（RAD）程序构造工具的重要基础。我们最后介绍的是 Java 1.2 的“Swing”库——它使 Java 的 UI 组件得到了显著的改善。

(14) 第 14 章：多线程
Java 提供了一套内建的机制，可提供对多个并发子任务的支持，我们称其为“线程”。这线程均在单一的程序内运行。除非机器安装了多个处理器，否则这就是多个子任务的唯一运行方式。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，但用户仍然可以毫无阻碍地按下一个按钮，或者键入一些文字。本章将对 Java 的多线程处理机制进行探讨，并介绍相关的语法。

(15) 第 15 章 网络编程
开始编写网络应用时，就会发现所有 Java 特性和库仿佛早已串联到了一起。本章将探讨如何通过因特网通信，以及 Java 用以辅助此类编程的一些类。此外，这里也展示了如何创建一个 Java 程序片，令其同一个“通用网关接口”（CGI）程序通信；揭示了如何用 C++编写 CGI 程序；也讲述了与 Java 1.1 的“Java 数据库连接”（JDBC）和“远程方法调用”（RMI）有关的问题。

(16) 第 16 章 设计范式
本章将讨论非常重要、但同时也是非传统的“范式”程序设计概念。大家会学习设计进展过程的一个例子。首先是最初的方案，然后经历各种程序逻辑，将方案不断改革为更恰当的设计。通过整个过程的学习，大家可体会到使设计思想逐渐变得清晰起来的一种途径。

(17) 第 17 章 项目
本章包括了一系列项目，它们要么以本书前面讲述的内容为基础，要么对以前各章进行了一番扩展。这些项目显然是书中最复杂的，它们有效演示了新技术和类库的应用。
有些主题似乎不太适合放到本书的核心位置，但我发现有必要在教学时讨论它们，这些主题都放入了本书的附录。

(18) 附录 A：使用非 Java 代码
对一个完全能够移植的 Java 程序，它肯定存在一些严重的缺陷：速度太慢，而且不能访问与具体平台有关的服务。若事先知道程序要在什么平台上使用，就可考虑将一些操作变成“固有方法”，从而显著加快执行速

度。这些“固有方法”实际是一些特殊的函数，以另一种程序设计语言写成（目前仅支持C/C++）。Java还可通过另一些途径提供对非Java代码的支持，其中包括CORBA。本附录将详细介绍这些特性，以便大家能创建一些简单的例子，同非Java代码打交道。

(19) 附录B：对比C++和Java
对一个C++程序员，他应该已经掌握了面向对象程序设计的基本概念，而且Java语法对他来说无疑是非常眼熟的。这一点是明显的，因为Java本身就是从C++衍生而来。但是，C++和Java之间的确存在一些显著的差异。这些差异意味着Java在C++基础上作出的重大改进。一旦理解了这些差异，就能理解为什么说Java是一种杰出的语言。这一附录便是为这个目的设立的，它讲述了使Java与C++明显有别的一些重要特性。

(20) 附录C：Java编程规则
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。

(21) 附录D：性能
通过这个附录的学习，大家可发现自己Java程序中存在的瓶颈，并可有效地改善执行速度。

(22) 附录E：关于垃圾收集的一些话
这个附录讲述了用于实现垃圾收集的操作和方法。

(23) 附录F：推荐读物
列出我感觉特别有用的一系列Java参考书。

6. 练习

为巩固对新知识的掌握，我发现简单的练习特别有用。所以读者在每一章结束时都能找到一系列练习。
大多数练习都很简单，在合理的时间内可以完成。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。有些练习要难些，他们是为那些有兴趣深入的读者准备的。大多数练习都可在较短时间内做完，有效地检测和加深您的知识。有些题目比较具有挑战性，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。

7. 多媒体CD-ROM

本书配套提供了一片多媒体CD-ROM，可单独购买及使用。它与其他计算机书籍的普通配套CD不同，那些CD通常仅包含了书中用到的源码（本书的源码可从www.BruceEckel.com免费下载）。本CD-ROM是一个独立的产品，包含了一周“Hads-OnJava”培训课程的全部内容。这是一个由Bruce Eckel讲授的、长度在15小时以上的课程，含500张以上的演示幻灯片。该课程建立在这本书的基础上，所以是非常理想的一个配套产品。
CD-ROM包含了本书的两个版本：
(1) 本书一个可打印的版本，与下载版完全一致。
(2) 为方便读者在屏幕上阅读和索引，CD-ROM提供了一个独特的超链接版本。这些超链接包括：
■230个章、节和小标题链接
■3600个索引链接
CD-ROM刻录了600MB以上的数据。我相信它已对所谓“物超所值”进行了崭新的定义。
CD-ROM包含了本书打印版的所有东西，另外还有来自五天快速入门课程的全部材料。我相信它建立了一个新的书刊品质评定标准。
若想单独购买此CD-ROM，只能从Web站点www.BruceEckel.com处直接订购。

8. 源代码

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下载。为保证大家获得的是最新版本，我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，都应检查正式站点，确定镜像版确实是最新的版本。可在课堂和其他教育场所发布这些代码。
版权的主要目标是保证源码得到正确的引用，并防止在未经许可的情况下，在印刷材料中发布代码。通常，

只要源码获得了正确的引用，则在大多数媒体中使用本书的示例都没有什么问题。
在每个源码文件中，都能发现下述版本声明文字：

```
//////////////////////////////////////////////
// Copyright (c) Bruce Eckel, 1998
// Source code file from 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 email all
// modified files with clearly commented changes to:
// Bruce@EckelObjects.com. (Please use the same
// address for non-code errors found in the book.)
//////////////////////////////////////////////
```

可在自己的开发项目中使用代码，并可在课堂上引用（包括学习材料）。但要确定版权声明在每个源文件中得到了保留。

9．编码样式

在本书正文中，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”）。

对于本书的示例，我采用了一种特定的编码样式。该样式得到了大多数 Java 开发环境的支持。该样式问世已有几年的时间，最早起源于 Bjarne Stroustrup 先生在《The C++ Programming Language》里采用的样式（Addison-Wesley 1991 年出版，第 2 版）。由于代码样式目前是个敏感问题，极易招致数小时的激烈辩论，所以我在这儿只想指出自己并不打算通过这些示例建立一种样式标准。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，所以读者完全可以根据自己的感觉选用了适合的编码样式。

本书的程序是由字处理程序包括在正文中的，它们直接取自编译好的文件。所以，本书印刷的代码文件应能正常工作，不会造成编译器错误。会造成编译错误的代码已经用注释//!标出。所以很容易发现，也很容易用自动方式进行测试。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 处出现）。

10．Java 版本

尽管我用几家厂商的 Java 开发平台对本书的代码进行了测试，但在判断代码行为是否正确时，却通常以 Sun 公司的 Java 开发平台为准。

当您读到本书时，Sun 应已发行了 Java 的三个重要版本：1.0，1.1 及 1.2（Sun 声称每 9 个月就会发布一个主要更新版本）。就我看，1.1 版对 Java 语言进行了显著改进，完全应标记成 2.0 版（由于 1.1 已作出了如此大的修改，真不敢想象 2.0 版会出现什么变化）。然而，它的 1.2 版看起来最终将 Java 推入了一个全盛时期，特别是其中考虑到了用户界面工具。

本书主要讨论了 1.0 和 1.1 版，1.2 版有部分内容涉及。但在有些时候，新方法明显优于老方法。此时，我会明显偏向于新方法，通常教给大家更好的方法，而完全忽略老方法。然而，有的新方法要以老方法为基础，所以不可避免地要从老方法入手。这一特点尤以 AWT 为甚，因为那儿不仅存在数量众多的老式 Java 1.0 代码，有的平台仍然只支持 Java 1.0。我会尽量指出哪些特性是哪个版本特有的。

大家会注意到我并未使用子版本号，比如 1.1.1。至本书完稿为止，Sun 公司发布的最后一个 1.0 版是 1.02；而 1.1 的最后版本是 1.1.5（Java 1.2 仍在做 β 测试）。在这本书中，我只会提到 Java 1.0，Java 1.1 及 Java 1.2，避免由于子版本编号过多造成的键入和印刷错误。

11．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

或发函至：

Bruce@EckelObjects.com

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

12．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 处找到）里指出有可能是错误的地方，填好我们提供的表单。将您推荐的纠错方法通过电子函件发给 Bruce@EckelObjects.com。经适当的核对与处理，Web 站点的电子版以及本书的下一个印刷版本会作出相应的改正。具体格式如下：

(1) 在主题行（Subject）写上“TIJ Correction”（去掉引号），以便您的函件进入对应的目录。

(2) 在函件正文，采用下述形式：

```
find: 在这里写一个单行字串，以便我们搜索错误所在的地方
Comment: 在这里可写多行批注正文，最好以“here's how I think it shoud read”开头
###
```

其中，“###”指出批注正文的结束。这样一来，我自己设计的一个纠错工具就能对原始正文来一次“搜索”，而您建议的纠错方法会在随后的一个窗口中弹出。

若希望在本书的下一版添加什么内容，或对书中的练习题有什么意见，也欢迎您指出。我们感谢您的所有意见。

13．封面设计

《Thinking in Java》一书封面的创作灵感来源于 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。当时正是机器革命产生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，回归自然的初衷是整个运动的核心。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，封面向大家展示了一个收集箱，自然学者可能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）。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）。

14．致谢

首先，感谢 Doyle Street Cohousing Community（道尔街住房社区）容忍我花两年的时间来写这本书（其实他们一直都在容忍我的“胡做非为”）。非常感谢 Kevin 和 Sonda Donovan，是他们把科罗拉多 Crested Butte 市这个风景优美的地方租给我，使我整个夏天都能安心写作。感谢 Crested Butte 友好的居民；以及 Rocky Mountain Biological Laboratory（岩石山生物实验室），他们的工作人员总是面带微笑。

这是我第一次找代理人出书，但却绝没有后悔。谢谢“摩尔文学代理公司”的 Claudette Moore 小姐。是她强大的信心与毅力使我最终梦想成真。

我的头两本书是与 Osborne/McGraw-Hill 出版社的编辑 Jeff Pepper 合作出版的。Jeff 又在正确的地方和正确的时间出现在了 Prentice-Hall 出版社，是他为了清除了所有可能遇到的障碍，也使我感受了一次愉快的出书经历。谢谢你，Jeff——你对我非常重要。

要特别感谢 Gen Kiyooka 和他的 Digigami 公司，我用的 Web 服务器就是他们提供的；也要感谢 Scott Callaway，服务器是由他负责维护的。在我学习 Web 的过程中，一个服务器无疑是相当有价值的帮助。

谢谢 Cay Horstmann（《Core Java》一书的副编辑，Prentice Hall 于 1997 年出版）、D'Arcy Smith（Symantec 公司）和 Paul Tyma（《Java Primer Plus》一书的副编辑，The Waite Group 于 1996 年出版），感谢他们帮助我澄清语言方面的一些概念。

感谢那些在“Java 软件开发会议”上我的 Java 小组发言的同志们，以及我教授过的那些学生，他们提出的问题使我的教案愈发成熟起来。

特别感谢 Larry 和 Tina O'Brien，是他们将这本书和我的教学内容制成一张教学 CD-ROM（关于这方面的问题，http://www.BruceEckel.com 有更多的答案）。

有许多人送来了纠错报告，我真的很感激所有这些朋友，但特别要对下面这些人说声谢谢：Kevin Raulerson（发现了多处重大错误），Bob Resendes（发现的错误令人难以置信），John Pinto，Joe Dante，Joe Sharp，David Combs（许多语法和表达不清的地方），Dr. Robert Stephenson，Franklin Chen，Zev Griner，David Karr，Leander A. Stroschein，Steve Clark，Charles A. Lee，AustinMaher，Dennis P. Roth，Roque Oliveira，Douglas Dunn，Dejan Ristic，NeilGalarneau，David B. Malkovsky，Steve Wilkinson，以及其他许多热心读者。

为了使这本书在欧洲发行，Prof. Ir. Marc Meurrens 进行了大量工作。

有一些技术人员曾走进我的生活，他们后来都和我成了朋友。最不寻常的是他们全是素食主义者，平时喜欢练习瑜珈功，以及另一些形式的精神训练。我在练习了以后，觉得对我保持精力的旺盛非常有好处。他们是 Kraig Brockschmidt，GenKiyooka 和 Andrea provaglio，是这些朋友帮我了解了 Java 和程序设计在意大利

的情况。
显然，在Delphi上的一些经验使我更容易理解Java，因为它们有许多概念和语言设计决定是相通的。我的Delphi朋友提供了许多帮助，使我能够洞察一些不易为人注意的编程环境。他们是Marco Cantu（另一个意大利人——难道会说拉丁语的人在学习Java时有得天独厚的优势？）、Neil Rubenking（他最喜欢瑜珈／素食／禅道，但也非常喜欢计算机）以及Zack Urlocker（是我游历世界时碰面次数最多的一位同志）。
我的朋友Richard Hale Shaw（以及Kim）的一些意见和支持发挥了非常关键的作用。Richard和我花了数月的时间将教学内容合并到一起，并探讨如何使学生感受到最完美的学习体验。也要感谢KoAnn Vikoren，Eric Eaurot，DeborahSommers，Julie Shaw，Nicole Freeman，Cindy Blair，Barbara Hanscome，Regina Ridley，Alex Dunne以及MFI其他可敬的成员。
书籍设计、封面设计以及封面照片是由我的朋友Daniel Will-Harris制作的。他是一位著名的作家和设计家（http://www.WillHarris.com），在初中的时候就已显露出了过人的数学天赋。但是，小样是由我制作的，所以录入错误都是我的。我是用Microsoft Word 97 for Windows来写这本书，并用它生成小样。正文字体采用的是Bitstream Carmina；标题采用Bitstream Calligraph 421（www.bitstream.com）；每章开头的符号采用的是来自P22的Leonardo Extras（http://www.p22.com）；封面字体采用ITC Rennie Marckintosh。
感谢为我提供编译器程序的一些著名公司：Borland，Microsoft，Symantec，Sybase/Powersoft/Watcom以及Sun。
特别感谢我的老师和我所有的学生（他们也是我的老师），其中最有趣的一位写作老师是Gabrielle Rico（《Writing the Natural Way》一书的作者，Putnam于1983年出版）。
曾向我提供过支持的朋友包括（当然还不止）：Andrew Binstock，SteveSinofsky，JD Hildebrandt，Tom Keffer，Brian McElhinney，Brinkley Barr，《Midnight Engineering》杂志社的Bill Gates，Larry Constantine和LucyLockwood，Greg Perry，Dan Putterman，Christi Westphal，Gene Wang，DaveMayer，David Intersimone，Andrea Rosenfield，Claire Sawyers，另一些意大利朋友（Laura Fallai，Corrado，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，KevinDonovan和Sonda Eastlack，Chester和Shannon Andersen，Joe Lordi，Dave和Brenda Bartlett，David Lee，Rentschlers，Sudeks，Dick，Patty和Lee Eckel，Lynn和Todd以及他们的家人。最后，当然还有我的爸爸和妈妈。

Table of Contents

第 1 章 对象入门

“为什么面向对象的编程会在软件开发领域造成如此震憾的影响？”

面向对象编程（OOP）具有多方面的吸引力。对管理人员，它实现了更快和更廉价的开发与维护过程。对分析与设计人员，建模处理变得更加简单，能生成清晰、易于维护的设计方案。对程序员，对象模型显得如此高雅和浅显。此外，面向对象工具以及库的巨大威力使编程成为一项更使人愉悦的任务。每个人都可从中获益，至少表面如此。

如果说它有缺点，那就是掌握它需付出的代价。思考对象的时候，需要采用形象思维，而不是程序化的思维。与程序化设计相比，对象的设计过程更具挑战性——特别是在尝试创建可重复使用（可再生）的对象时。过去，那些初涉面向对象编程领域的人都必须进行一项令人痛苦的选择：

(1) 选择一种诸如 Smalltalk 的语言，“出师”前必须掌握一个巨型的库。

(2) 选择几乎根本没有库的 C++（注释①），然后深入学习这种语言，直至能自行编写对象库。

①：幸运的是，这一情况已有明显改观。现在有第三方库以及标准的 C++库供选用。

事实上，很难很好地设计出对象——从而很难设计好任何东西。因此，只有数量相当少的“专家”能设计出最好的对象，然后让其他人享用。对于成功的 OOP 语言，它们不仅集成了这种语言的语法以及一个编译程序（编译器），而且还有一个成功的开发环境，其中包含设计优良、易于使用的库。所以，大多数程序员的首要任务就是用现有的对象解决自己的应用问题。本章的目标就是向大家揭示出面向对象编程的概念，并证明它有多么简单。

本章将向大家解释 Java 的多项设计思想，并从概念上解释面向对象的程序设计。但要注意在阅读完本章后，并不能立即编写出全功能的 Java 程序。所有详细的说明和示例会在本书的其他章节慢慢道来。

1.1 抽象的进步

所有编程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在此基础上则跨出了一大步，程序员可利用一些工具表达问题空间内的元素。由于这种表达非常普遍，所以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。

Alan Kay 总结了 Smalltalk 的五大基本特征。这是第一种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。可将对象想象成一种新型变量；它保存着数据，但可要求它对自身进行操作。理论上讲，可从要解决的问题身上提出所有概念性的组件，然后在程序中将其表达为一个对象。

(2) 程序是一大堆对象的组合；通过消息传递，各对象知道自己该做些什么。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。或者说，通过封装现有对象，可制作出新型对象。所以，尽管对象的概念非常简单，但在程序中却可达到任意高的复杂程度。
(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。

一些语言设计者认为面向对象的程序设计本身并不足以方便解决所有形式的程序问题，提倡将不同的方法组合成“多形程序设计语言”（注释②）。

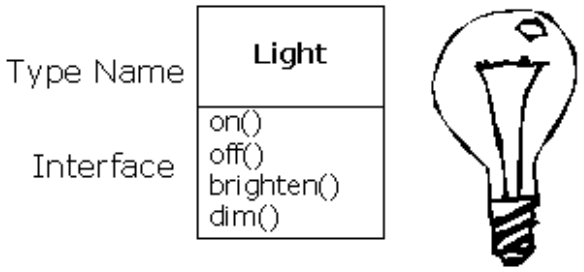
②：参见Timothy Budd编著的《Multiparadigm Programming in Leda》，Addison-Wesley 1995年出版。

1.2 对象的接口

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和行为。在Simula-67中，首次用到了class这个关键字，它为程序引入了一个全新的类型（clas和type通常可互换使用；注释③）。

③：有些人进行了进一步的区分，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。

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。
建好一个类后，可根据情况生成许多对象。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，令其做一些实际的事情，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();
```

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包括打开（on）、关闭（off）、变得更明亮（brighten）或者变得更暗淡（dim）。通过简单地声明一个名字（lt），我们为 Light 对象创建了一个“句柄”。然后用 new 关键字新建类型为 Light 的一个对象。再用等号将其赋给句柄。为了向对象发送一条消息，我们列出句柄名（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。

1.3 实现方案的隐藏

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；注释④）。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”，以便快速开发符合自己要求的应用。而对类创建者来说，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），其他所有细节都隐藏起来。为什么要这样做？隐藏之后，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，可确保它们不会对其他人造成影响。

④：感谢我的朋友 Scott Meyers，是他帮我起了这个名字。

“接口”（Interface）规定了可对一个特定的对象发出哪些请求。然而，必须在某个地方存在着一些代码，以便满足这些请求。这些代码与那些隐藏起来的数据便叫作“隐藏的实现”。站在程式化程序编写（Procedural Programming）的角度，整个问题并不显得复杂。一种类型含有与每种可能的请求关联起来的函数。一旦向对象发出一个特定的请求，就会调用那个函数。我们通常将这个过程总结为向对象“发送一条消息”（提出一个请求）。对象的职责就是决定如何对这条消息作出反应（执行相应的代码）。
对于任何关系，重要一点是让牵连到的所有成员都遵守相同的规则。创建一个库时，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），或者用您的库构建一个更大的库。
若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，但倘若未进行访问控制，就没有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（明确）关键字以及一个隐式（暗示）关键字来设置类边界：public，private，protected 以及暗示性的 friendly。若未明确指定其他关键字，则默认为后者。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。“friendly”（友好的）涉及“包装”或“封装”（Package）的概念——即 Java 用来构建库的方法。若某样东西是“友好的”，意味着它只能在这个包装的范围内使用（所以这一访问级别有时也叫作“包装访问”）。“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。

1.4 方案的重复使用

创建并测试好一个类后，它应（从理想的角度）代表一个有用的代码单位。但并不象许多人希望的那样，这种重复使用的能力并不容易实现；它要求较多的经验以及洞察力，这样才能设计出一个好的方案，才有可能重复使用。

许多人认为代码或设计方案的重复使用是面向对象的程序设计提供的最伟大的一种杠杆。

为重复使用一个类，最简单的办法是仅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”。新类可由任意数量和类型的其他对象构成。无论如何，只要新类达到了设计要求即可。这个概念叫作“组织”——在现有类的基础上组织一个新类。有时，我们也将组织称作“包含”关系，比如“一辆车包含了一个变速箱”。

对象的组织具有极大的灵活性。新类的“成员对象”通常设为“私有”（Private），使用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。也可以在“运行期”更改成员，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。

由于继承的重要性，所以在面向对象的程序设计中，它经常被重点强调。作为新加入这一领域的程序员，或许早已先入为主地认为“继承应当随处可见”。沿这种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“组织”对象；这样做显得更加简单和灵活。利用对象的组织，我们的设计可保持清爽。一旦需要用到继承，就会明显意识到这一点。

1.5 继承：重新使用接口

就其本身来说，对象的概念可为我们带来极大的便利。它在概念上允许我们将各式各样数据和功能封装到一起。这样便可恰当表达“问题空间”的概念，不用刻意遵照基础机器的表达方式。在程序设计语言中，这些概念则反映为具体的数据类型（使用 class 关键字）。

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，情况就显得理想多了。“继承”正是针对这个目标而设计的。但继承并不完全等价于克隆。在继承过程中，若原始类（正式名称叫作基础类、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 Java 语言中，继承是通过 extends 关键字实现的

使用继承时，相当于创建了一个新类。这个新类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，它复制了基础类的接口。也就是说，可向基础类的对象发送的所有消息亦可原样发给衍生类的对象。根据可以发送的消息，我们能知道类的类型。这意味着衍生类具有与基础类相同的类型！为真正理解面向对象程序设计的含义，首先必须认识到这种类型的等价关系。

由于基础类和衍生类具有相同的接口，所以那个接口必须进行特殊的设计。也就是说，对象接收到一条特定的消息后，必须有一个“方法”能够执行。若只是简单地继承一个类，并不做其他任何事情，来自基础类接口的方法就会直接照搬到衍生类。这意味着衍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。

有两种做法可将新得的衍生类与原来的基础类区分开。第一种做法十分简单：为衍生类添加新函数（功能）。这些新函数并非基础类接口的一部分。进行这种处理时，一般都是意识到基础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础类是否真的需要这些额外的函数。

1.5.1 改善基础类

尽管 extends 关键字暗示着我们要为接口“扩展”新功能，但实情并非肯定如此。为区分我们的新类，第二个办法是改变基础类一个现有函数的行为。我们将其称作“改善”那个函数。

为改善一个函数，只需为衍生类的函数建立一个新定义即可。我们的目标是：“尽管使用的函数接口未变，但它的新版本具有不同的表现”。

1.5.2 等价与类似关系

针对继承可能会产生这样的一个争论：继承只能改善原基础类的函数吗？若答案是肯定的，则衍生类型就是与基础类完全相同的类型，因为都拥有完全相同的接口。这样造成的结果就是：我们完全能够将衍生类的一个对象换成基础类的一个对象！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，我们通常认为基础类和衍生类之间存在一种“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。

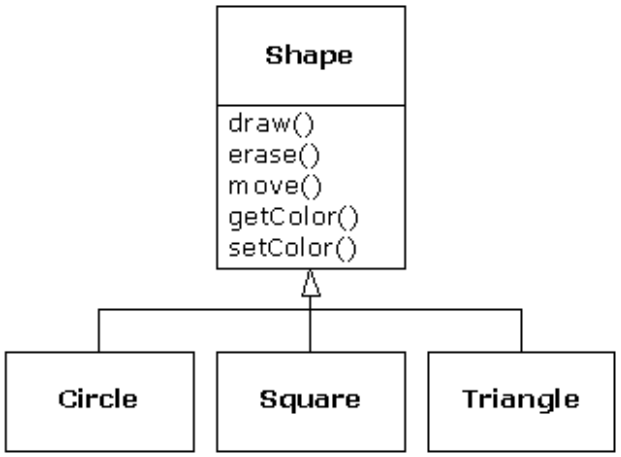
但在许多时候，我们必须为衍生类型加入新的接口元素。所以不仅扩展了接口，也创建了一种新类型。这种新类型仍可替换成基础类型，但这种替换并不是完美的，因为不可在基础类里访问新函数。我们将其称作“类似”关系；新类型拥有旧类型的接口，但也包含了其他函数，所以不能说它们是完全等价的。举个例子来说，让我们考虑一下制冷机的情况。假定我们的房间连好了用于制冷的各种控制器；也就是说，我们已拥有必要的“接口”来控制制冷。现在假设机器出了故障，我们把它换成一台新型的冷、热两用空调，冬天和夏天均可使用。冷、热空调“类似”制冷机，但能做更多的事情。由于我们的房间只安装了控制制冷的设备，所以它们只限于同新机器的制冷部分打交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

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，但您会发现在某些情况下，仍然有明显的理由需要在衍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 多形对象的互换使用

通常，继承最终会以创建一系列类收场，所有类都建立在统一的接口基础上。我们用一幅颠倒的树形图来阐明这一点（注释⑤）：

⑤：这儿采用了“统一记号法”，本书将主要采用这种方法。



对这样的一系列类，我们要进行的一项重要处理就是将衍生类的对象当作基础类的一个对象对待。这一点是非常重要的，因为它意味着我们只需编写单一的代码，令其忽略类型的特定细节，只与基础类打交道。这样一来，那些代码就可与类型信息分开。所以更易编写，也更易理解。此外，若通过继承增添了一种新类型，如“三角形”，那么我们为“几何形状”新类型编写的代码会象在旧类型里一样良好地工作。所以说程序具备了“扩展能力”，具有“扩展性”。

以上面的例子为基础，假设我们用 Java 写了这样一个函数：

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

```
}
```

这个函数可与任何“几何形状”（Shape）通信，所以完全独立于它要描绘（draw）和删除（erase）的任何特定类型的对象。如果我们在其他一些程序里使用 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()的调用会自动良好地工作，无论对象的具体类型是什么。
这实际是一个非常有用的编程技巧。请考虑下面这行代码：
doStuff(c);
此时，一个 Circle（圆）句柄传递给一个本来期待 Shape（形状）句柄的函数。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，不会造成错误。
我们将这种把衍生类型当作它的基本类型处理的过程叫作“Upcasting”（上溯造型）。其中，“cast”（造型）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——即基础类位于顶部，而衍生类在下方展开。所以，根据基础类进行造型就是一个从上面继承的过程，即“Upcasting”。
在面向对象的程序里，通常都要用到上溯造型技术。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()；请自己采取那个行动，并自己去控制所有的细节吧。”

1.6.1 动态绑定

在 doStuff()的代码里，最让人吃惊的是尽管我们没作出任何特殊指示，采取的操作也是完全正确和恰当的。我们知道，为 Circle 调用 draw()时执行的代码与为一个 Square 或 Line 调用 draw()时执行的代码是不同的。但在将 draw()消息发给一个匿名 Shape 时，根据 Shape 句柄当时连接的实际类型，会相应地采取正确的操作。这当然令人惊讶，因为当 Java 编译器为 doStuff()编译代码时，它并不知道自己要操作的准确类型是什么。尽管我们确实可以保证最终会为 Shape 调用 erase()，为 Shape 调用 draw()，但并不能保证为特定的 Circle，Square 或者 Line 调用什么。然而最后采取的操作同样是正确的，这是怎么做到的呢？
将一条消息发给对象时，如果并不知道对方的具体类型是什么，但采取的行动同样是正确的，这种情况就叫作“多形性”（Polymorphism）。对面向对象的程序设计语言来说，它们用以实现多形性的方法叫作“动态绑定”。编译器和运行期系统会负责对所有细节的控制；我们只需知道会发生什么事情，而且更重要的是，如何利用它帮助自己设计程序。
有些语言要求我们用一个特殊的关键字来允许动态绑定。在 C++中，这个关键字是 virtual。在 Java 中，我们则完全不必记住添加一个关键字，因为函数的动态绑定是自动进行的。所以在将一条消息发给对象时，我们完全可以肯定对象会采取正确的行动，即使其中涉及上溯造型之类的处理。

1.6.2 抽象的基础类和接口

设计程序时，我们经常都希望基础类只为自己的衍生类提供一个接口。也就是说，我们不想其他任何人实际创建基础类的一个对象，只对上溯造型成它，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，它完全禁止了所有的函数定义。“接口”是一种相当有效和常用的工具。另外如果自己愿意，亦可将多个接口都合并到一起（不能从多个普通 class 或 abstract class 中继承）。

1.7 对象的创建和存在时间

从技术角度说，OOP（面向对象程序设计）只是涉及抽象的数据类型、继承以及多形性，但另一些问题也可能显得非常重要。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，除非进入运行期，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。
C++允许我们决定是在写程序时创建对象，还是在运行期间创建，这种控制方法更加灵活。大家或许认为既然它如此灵活，那么无论如何都应在内存堆里创建对象，而不是在堆栈中创建。但还要考虑另外一个问题，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，编译器会判断对象的持续时间有多长，到时会自动“破坏”或者“清除”它。程序员可用两种方法来破坏一个对象：用程序化的方式决定何时破坏对象，或者利用由运行环境提供的一种“垃圾收集器”特性，自动寻找那些不再使用的对象，并将其清除。当然，垃圾收集器显得方便得多，但要求所有应用程序都必须容忍垃圾收集器的存在，并能默许随垃圾收集带来的额外开销。但这并不符合C++语言的设计宗旨，所以未能包括到 C++里。但 Java 确实提供了一个垃圾收集器（Smalltalk 也有这样的设计；尽管 Delphi 默认为没有垃圾收集器，但可选择安装；而 C++亦可使用一些由其他公司开发的垃圾收集产品）。
本节剩下的部分将讨论操纵对象时要考虑的另一些因素。

1.7.1 集合与继承器

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？事先上根本无法提前知道，除非进入运行期。
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。用于解决特定问题的新型对象容纳了指向其他对象的句柄。当然，也可以用数组来做同样的事情，那是大多数语言都具有的一种功能。但不能只看到这一点。这种新对象通常叫作“集合”（亦叫作一个“容器”，但 AWT 在不同的场合应用了这个术语，所以本书将一直沿用“集合”的称呼。在需要的时候，集合会自动扩充自己，以便适应我们在其中置入的任何东西。所以我们事先不必知道要在一个集合里容下多少东西。只需创建一个集合，以后的工作让它自己负责好了。
幸运的是，设计优良的 OOP 语言都配套提供了一系列集合。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）提供集合。Smalltalk 提供了一套非常完整的集合。而 Java 也用自己的标准库提供了集合。在某些库中，一个常规集合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集合。例如，可以用一个矢量统一对所有元素的访问方式；一个链接列表则用于保证所有元素的插入统一。所以我们能根据自己的需要选择适当的类型。其中包括集、队列、散列表、树、堆栈等等。
所有集合都提供了相应的读写功能。将某样东西置入集合时，采用的方式是十分明显的。有一个叫作“推”（Push）、“添加”（Add）或其他类似名字的函数用于做这件事情。但将数据从集合中取出的时候，方式却并不总是那么明显。如果是一个数组形式的实体，比如一个矢量（Vector），那么也许能用索引运算符或函数。但在许多情况下，这样做往往会无功而返。此外，单选定函数的功能是非常有限的。如果想对集合中的一系列元素进行操纵或比较，而不是仅仅面向一个，这时又该怎么办呢？
办法就是使用一个“继续器”（Iterator），它属于一种对象，负责选择集合内的元素，并把它们提供给继承器的用户。作为一个类，它也提供了一级抽象。利用这一级抽象，可将集合细节与用于访问那个集合的代码隔离开。通过继承器的作用，集合被抽象成一个简单的序列。继承器允许我们遍历那个序列，同时毋需关心基础结构是什么——换言之，不管它是一个矢量、一个链接列表、一个堆栈，还是其他什么东西。这样一来，我们就可以灵活地改变基础数据，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）提供的是一个标准继承器，名为 Enumeration（枚举），为它的所有集合类提供服务。Java 1.2 新增一个更复杂的集合库，其中包含了一个名为 Iterator 的继承器，可以做比老式的 Enumeration 更多的事情。
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。有两方面的原因促使我们需要对集合作出选择。首先，集合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集（Set）或列表的不同。利用这个特征，我们解决问题时便有更大的灵活性。
其次，不同的集合在进行特定操作时往往有不同的效率。最好的例子便是矢量（Vector）和列表（List）的区别。它们都属于简单的序列，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，需要的开销却是完全不同的。对矢量内的元素进行的随机访问（存取）是一种常时操作；无论我们选择的选择是什么，需要的时间量都是相同的。但在一个链接列表中，若想到处移动，并随机挑选一个元素，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，用列表就比用矢量划算得多。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，我们可以先从一个列表开始。最后调整性能的时候，再根据情况把它换成矢量。由于抽象是通过继承器进行的，所以能在两者方便地切换，对代码的影响则显得微不足道。
最后，记住集合只是一个用来放置对象的储藏所。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）。如果在一个编程环境中工作，它由于其他因素（比如在 Windows 下运行，或者由垃圾收集器带来了开销）产生了内在的开销，那么矢量和链接列表之间在系统开销上的差异就或许不是一个大问题。我们可能只需要一种类型的序列。甚至可以想象有一个“完美”的集合抽象，它能根据自己的使用方式自动改变基层的实现方式。

1.7.2 单根结构

在面向对象的程序设计中，由于 C++的引入而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础类继承。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，而且这个终级基础类的名字很简单，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本类型。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。但假期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。需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）都可以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。一个单根结构，加上所有对象都在内存堆中创建，可以极大简化参数的传递（这在 C++里是一个复杂的概念）。
利用单根结构，我们可以更方便地实现一个垃圾收集器。与此有关的必要支持可安装于基础类中，而垃圾收集器可将适当的消息发给系统内的任何对象。如果没有这种单根结构，而且系统通过一个句柄来操纵对象，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。
由于运行期的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，比如违例控制；而且也能在程序设计时获得更大的灵活性。
但大家也可能产生疑问，既然你把好处说得这么天花乱坠，为什么C++没有采用单根结构呢？事实上，这是早期在效率与控制上权衡的一种结果。单根结构会带来程序设计上的一些限制。而且更重要的是，它加大了新程序与原有C代码兼容的难度。尽管这些限制仅在特定的场合会真的造成问题，但为了获得最大的灵活程度，C++最终决定放弃采用单根结构这一做法。而Java不存在上述的问题，它是全新设计的一种语言，不必与现有的语言保持所谓的“向后兼容”。所以很自然地，与其他大多数面向对象的程序设计语言一样，单根结构在Java的设计方案中很快就落实下来。

1.7.3 集合库与方便使用集合

由于集合是我们经常都要用到的一种工具，所以一个集合库是十分必要的，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集合，将其插入自己的程序。Java提供了这样的一个库，尽管它在Java 1.0和1.1中都显得非常有限（Java 1.2的集合库则无疑是一个杰作）。

1．下溯造型与模板／通用性
为了使这些集合能够重复使用，或者“再生”，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”！所以容纳了Object的一个集合实际可以容纳任何东西。这使我们对它的重复使用变得非常简便。
为使用这样的一个集合，只需添加指向它的对象句柄即可，以后可以通过句柄重新使用对象。但由于集合只能容纳Object，所以在我们向集合里添加对象句柄时，它会上溯造型成Object，这样便丢失了它的身份或者标识信息。再次使用它的时候，会得到一个Object句柄，而非指向我们早先置入的那个类型的句柄。所以怎样才能归还它的本来面貌，调用早先置入集合的那个对象的有用接口呢？
在这里，我们再次用到了造型（Cast）。但这一次不是在分级结构中上溯造型成一种更“通用”的类型。而是下溯造型成一种更“特殊”的类型。这种造型方法叫作“下溯造型”（Downcasting）。举个例子来说，我们知道在上溯造型的时候，Circle（圆）属于Shape（几何形状）的一种类型，所以上溯造型是安全的。但我们不知道一个Object到底是Circle还是Shape，所以很难保证下溯造型的安全进行，除非确切地知道自己要操作的是什么。
但这也不是绝对危险的，因为假如下溯造型成错误的东西，会得到我们称为“违例”（Exception）的一种运行期错误。我们稍后即会对此进行解释。但在从一个集合提取对象句柄时，必须用某种方式准确地记住它们是什么，以保证下溯造型的正确进行。
下溯造型和运行期检查都要求花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，我们能不能创建一个“智能”集合，令其知道自己容纳的类型呢？这样做可消除下溯造型的必要以及潜在的错误。答案是肯定的，我们可以采用“参数化类型”，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集合，编译器可对那个集合进行定制，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，这部分是C++没有单根结构的缘故。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，因为由于使用的是单根结构，所以使用它显得有些笨拙。但这并不能保证以后的版本不会实现，因为“generic”这个词已被Java“保留到将来实现”（在Ada语言中，“generic”被用来实现它的模板）。Java采取的这种关键字保留机制其实经常让人摸不着头脑，很难断定以后会发生什么事情。

1.7.4 清除时的困境：由谁负责清除？

每个对象都要求资源才能“生存”，其中最令人注目的资源是内存。如果不再需要使用一个对象，就必须将其清除，以便释放这些资源，以便其他对象使用。如果要解决的是非常简单的问题，如何清除对象这个问题并不显得很突出：我们创建对象，在需要的时候调用它，然后将其清除或者“破坏”。但在另一方面，我们平时遇到的问题往往要比这复杂得多。
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。
但事情并没有这么简单，可能还需要另一套系统来记录与飞机有关的数据。当然，和控制器的主要功能不同，这些数据的重要性可能一开始并不显露出来。例如，这条记录反映的可能是离开机场的所有小飞机的飞行计划。所以我们得到了由小飞机组成的另一个集合。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集合。然后在系统空闲时期，需对这个集合中的对象进行一些后台处理。
问题现在显得更复杂了：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，所以问题会变得异常复杂（注释⑥）。

⑥：注意这一点只对内存堆里创建的对象成立（用 new 命令创建的）。但在另一方面，对这儿描述的问题以及其他所有常见的编程问题来说，都要求对象在内存堆里创建。

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单个根类 Object 继承的事实，而且由于我们只能在内存堆中以一种方式创建对象，所以 Java 的编程要比 C++的编程简单得多。我们只需要作出少量的抉择，即可克服原先存在的大量障碍。

1．垃圾收集器对效率及灵活性的影响
既然这是如此好的一种手段，为什么在C++里没有得到充分的发挥呢？我们当然要为这种编程的方便性付出一定的代价，代价就是运行期的开销。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（但不具有在运行期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。如果总是从同一个基础类继承，并使所有函数调用都具有“同质多形”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，存在着一种不连贯的因素。所以在某些特殊的场合，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要这方面的要求——注释⑦）。

⑦：根据本书一些技术性读者的反馈，有一个现成的实时 Java 系统（www.newmonics.com）确实能够保证垃圾收集器的效能。

C++语言的设计者曾经向C 程序员发出请求（而且做得非常成功），不要希望在可以使用 C 的任何地方，向语言里加入可能对C++的速度或使用造成影响的任何特性。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 无疑都应是我们的首选。

1.8 违例控制：解决错误

从最古老的程序设计语言开始，错误控制一直都是设计者们需要解决的一个大问题。由于很难设计出一套完美的错误控制方案，许多语言干脆将问题简单地忽略掉，将其转嫁给库设计人员。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强制标准。如果程序员不够警惕——若比较匆忙，这几乎是肯定会发生的——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设计语言中，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。这样可保证程序的连贯性，使错误控制变得更加容易。
注意违例控制并不属于一种面向对象的特性，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。早在面向对象语言问世以前，违例控制就已经存在了。

1.9 多线程

在计算机编程中，一个基本的概念就是同时对多个任务加以控制。许多程序设计问题都要求程序能够停下手头的工作，改为处理其他一些问题，再返回主进程。可以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。尽管这是一种有用的方法，但编出的程序很难移植，由此造成了另一类的代价高昂问题。

有些时候，中断对那些实时性很强的任务来说是很有必要的。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。利用线程，用户可按下一个按钮，然后程序会立即作出响应，而不是让用户等待程序完成了当前任务以后才开始响应。

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，毋需作出任何特殊的调校。

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，就会遇到一个问题。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。所以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，使其他线程可以接着使用同样的资源。

Java 的多线程机制已内建到语言中，这使一个可能较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，需要使用 synchronized 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检查这把锁。

1.10 永久性

创建一个对象后，只要我们需要，它就会一直存在下去。但在程序结束运行时，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，对象仍然在那里，里面保留的信息仍然是程序上一次运行时的那些信息。当然，可以将信息写入一个文件或者数据库，从而达到相同的效果。但尽管可将所有东西都看作一个对象，如果能将对象声明成“永久性”，并令其为我们照看其他所有细节，无疑也是一件相当方便的事情。

Java 1.1 提供了对“有限永久性”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，是由于我们仍然需要明确发出调用，进行对象的保存和取回工作。这些工作不能自动进行。在 Java 未来的版本中，对“永久性”的支持有望更加全面。

1.11 Java 和因特网

既然 Java 不过另一种类型的程序设计语言，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？如果您来自一个传统的程序设计背景，那么答案在刚开始的时候并不是很明显。Java 除了可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计算领域。

1．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”（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！

2．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”，它是对传统的“服务器端编程”的一个非常重要的拓展。

1.11.2 客户端编程（注释⑧）

Web 最初采用的“服务器－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。

⑧：本节内容改编自某位作者的一篇文章。那篇文章最早出现在位于 www.mainspring.com 的 Mainspring 上。本节的采用已征得了对方的同意。

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。

1．插件
朝客户端编程迈进的时候，最重要的一个问题就是插件的设计。利用插件，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）。

2．脚本编制语言
插件造成了脚本编制语言的爆炸性增长。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 也许是目常用的，它得到的支持也最全面。无论 NetscapeNavigator，Microsoft Internet Explorer，还是 Opera，目前都提供了对 JavaScript 的支持。除此以外，市面上讲述 JavaScript 的书籍也要比讲述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，如果你已经有 Visual Basic 或者 Tcl/Tk 的深厚功底，当然用它们要简单得多，起码可以避免学习新语言的烦恼（解决 Web 方面的问题就已经够让人头痛了）。

3．Java
如果说一种脚本编制语言能解决 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“程序片”（Applet）巧妙地解决了客户端编程的问题。
程序片（或“小应用程序”）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，程序片代码会自动下载回来（这和网页中的图片差不多）。激活程序片后，它会执行一个程序。程序片的一个优点体现在：通过程序片，一旦用户需要客户软件，软件就可从服务器自动下载回来。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 程序片的另一个优点是它采用编译好的形式，所以客户端看不到源码。当然在另一方面，反编译 Java 程序片也并不是件难事，而且代码的隐藏一般并不是个重要的问题。大家要注意另外两个重要的问题。正如本书以前会讲到的那样，编译好的 Java 程序片可能包含了许多模块，所以要多次“命中”（访问）服务器以便下载（在Java 1.1 中，这个问题得到了有效的改善——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。不管你平时听别人怎么说，Java 都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，因为它们可能已经能够满足你的需要，不必经历学习 Java 的艰苦过程。

4．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。

5．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”里在它的程序片中运行，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，在程序片装载时会检查所有有嫌疑的指令。特别地，程序片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）。我们通常认为程序片是安全的。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“签了名的程序片”，它用公共密钥加密算法验证程序片确实来自它所声称的地方。当然在通过验证后，签了名的一个程序片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，可让一个程序片“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（以及它当时的责任有多大）？

6．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程序片之间的差异，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 程序片、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，服务器端编程是用 Perl 和 CGI 脚本进行的，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）。

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程序片有关的。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。如果将视线从程序片身上转开（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 提供了许多有用的特性，使我们能在较短的时间里创建出比用从前的程序设计语言更健壮的程序。
但要注意任何东西都不是十全十美的，我们为此也要付出一些代价。其中最明显的是执行速度放慢了（尽管

可对此进行多方面的调整）。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，能充分解决的问题也变得越来越多。

1.12 分析和设计

面向对象的范式是思考程序设计时一种新的、而且全然不同的方式，许多人最开始都会在如何构造一个项目上皱起了眉头。事实上，我们可以作出一个“好”的设计，它能充分利用 OOP 提供的所有优点。
有关 OOP 分析与设计的书籍大多数都不尽如人意。其中的大多数书都充斥着莫名其妙的话语、笨拙的笔调以及许多听起来似乎很重要的声明（注释⑨）。我认为这种书最好压缩到一章左右的空间，至多写成一本非常薄的书。具有讽刺意味的是，那些特别专注于复杂事物管理的人往往在写一些浅显、明白的书上面大费周章！如果不能说得简单和直接，一定没多少人喜欢看这方面的内容。毕竟，OOP 的全部宗旨就是让软件开发的过程变得更加容易。尽管这可能影响了那些喜欢解决复杂问题的人的生计，但为什么不从一开始就把事情弄得简单些呢？因此，希望我能从开始就为大家打下一个良好的基础，尽可能用几个段落来说清楚分析与设计的问题。

⑨：最好的入门书仍然是 Grady Booch 的《Object-Oriented Design withApplications，第 2 版本》，Wiely & Sons 于 1996 年出版。这本书讲得很有深度，而且通俗易懂，尽管他的记号方法对大多数设计来说都显得不必要地复杂。

1.12.1 不要迷失

在整个开发过程中，最重要的事情就是：不要将自己迷失！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。但无论多么有限，某些形式的处理总是有益的，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！
也就是说，假如你正在考察一种特殊的方法，其中包含了大量细节，并推荐了许多步骤和文档，那么仍然很难正确判断自己该在何时停止。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？（需要将什么消息发给每一个对象？）
在确定了对象和它们的接口后，便可着手编写一个程序。出于对多方面原因的考虑，可能还需要比这更多的说明及文档，但要求掌握的资料绝对不能比这还少。
整个过程可划分为四个阶段，阶段 0 刚刚开始采用某些形式的结构。

1.12.2 阶段 0：拟出一个计划

第一步是决定在后面的过程中采取哪些步骤。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：有些人甚至没有进入阶段 1，便忙忙慌慌地开始编写代码。如果你的计划本来就是“直接开始开始编码”，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）。但最低程度也应同意自己该有个计划。
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“完成工作”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。
座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，发现结构要比小说简单得多，所以不需要考虑太多这方面的问题。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，即使你的计划就是直接开始写程序，仍然需要经历以下的阶段，同时向自己提出一些特定的问题。

1.12.3 阶段 1：要制作什么？

在上一代程序设计中（即“过程化或程序化设计”），这个阶段称为“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。但当时的初衷是好的。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”。系统规格则表示“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（即使客户就在本公司内部工作，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。可能还会面临另一些限制，需要把它们扩充成为更大的文档。

我们特别要注意将重点放在这一阶段的核心问题上，不要纠缠于细枝末节。这个核心问题就是：决定采用什么系统。对这个问题，最有价值的工具就是一个名为“使用条件”的集合。对那些采用“假如……，系统该怎样做？”形式的问题，这便是最有说服力的回答。例如，“假如客户需要提取一张现金支票，但当时又没有这么多的现金储备，那么自动取款机该怎样反应？”对这个问题，“使用条件”可以指示自动取款机在那种“条件”下的正确操作。

应尽可能总结出自己系统的一套完整的“使用条件”或者“应用场合”。一旦完成这个工作，就相当于摸清了想让系统完成的核心任务。由于将重点放在“使用条件”上，一个很好的效果就是它们总能让你放精力放在最关键的东西上，并防止自己分心于对完成任务关系不大的其他事情上面。也就是说，只要掌握了一套完整的“使用条件”，就可以对自己的系统作出清晰的描述，并转移到下一个阶段。在这一阶段，也有可能无法完全掌握系统日后的各种应用场合，但这也没有关系。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”，否则也容易产生挫败感和焦燥情绪。

在这一阶段，最好用几个简单的段落对自己的系统作出描述，然后围绕它们再进行扩充，添加一些“名词”和“动词”。“名词”自然成为对象，而“动词”自然成为要整合到对象接口中的“方法”。只要亲自试着做一做，就会发现这是多么有用的一个工具；有些时候，它能帮助你完成绝大多数的工作。

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，所以可能已经感觉到了它大概会花多长的时间来完成。此时要考虑多方面的因素：如果估计出一个较长的日程，那么公司也许决定不再继续下去；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，最好从一开始就草拟出一份“诚实”的时间表，以后再进行一些暂时难以作出的决策。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。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！

1.12.4 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的。此时可考虑采用一种特殊的图表工具：“统一建模语言”（UML）。请到 http://www.rational.com 去下载一份 UML 规格书。作为第 1 阶段中的描述工具，UML 也是很有帮助的。此外，还可用它在第 2 阶段中处理一些图表（如流程图）。当然并非一定要使用 UML，但它对你会很有帮助，特别是在希望描绘一张详尽的图表，让许多人在一起研究的时候。除 UML 外，还可选择对对象以及它们的接口进行文字化描述（就象我在《Thinking in C++》里说的那样，但这种方法非常原始，发挥的作用亦较有限。

我曾有一次非常成功的咨询经历，那时涉及到一小组人的初始设计。他们以前还没有构建过 OOP（面向对象程序设计）项目，将对象画在白板上面。我们谈到各对象相互间该如何沟通（通信），并删除了其中的一部分，以及替换了另一部分对象。这个小组（他们知道这个项目的目的是什么）实际上已经制订出了设计方案；他们自己“拥有”了设计，而不是让设计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，并从小组中得到反馈，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！

作出了对对象以及它们的接口的说明后，就完成了第 2 阶段的工作。当然，这些工作可能并不完全。有些工作可能要等到进入阶段 3 才能得知。但这已经足够了。我们真正需要关心的是最终找出所有的对象。能早些发现当然好，但 OOP 提供了足够完美的结构，以后再找出它们也不迟。

1.12.5 阶段 3：开始创建

读这本书的可能是程序员，现在进入的正是你可能最感兴趣的阶段。由于手头上有一个计划——无论它有多么简要，而且在正式编码前掌握了正确的设计结构，所以会发现接下去的工作比一开始就埋头写程序要简单得多。而这正是我们想达到的目的。让代码做到我们想做的事情，这是所有程序项目最终的目标。但切不要急功冒进，否则只有得不偿失。根据我的经验，最后先拿出一套较为全面的方案，使其尽可能设想周全，能满足尽可能多的要求。给我的感觉，编程更象一门艺术，不能只是作为技术活来看待。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。

构建好系统，并令其运行起来后，必须进行实际检验，以前做的那些需求分析和系统规格便可派上用场了。全面地考察自己的程序，确定提出的所有要求均已满足。现在一切似乎都该结束了？是吗？

1.12.6 阶段 4：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”。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。

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（注释⑩）。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（“设计范式”在这里可起到很大的帮助作用。有关它的讨论，请参考本书第 16 章）。

构建一套系统时，“校订”几乎是不可避免的。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，检查它是否自己希望的，使自己的思想不断趋向成熟。

反复的“校订”同“递增开发”有关密不可分的关系。递增开发意味着先从系统的核心入手，将其作为一个框架实现，以后要在这个框架的基础上逐渐建立起系统剩余的部分。随后，将准备提供的各种功能（特性）一个接一个地加入其中。这里最考验技巧的是架设起一个能方便扩充所有目标特性的一个框架（对这个问题，大家可参考第 16 章的论述）。这样做的好处在于一旦令核心框架运作起来，要加入的每一项特性就象它自身内的一个小项目，而非大项目的一部分。此外，开发或维护阶段合成的新特性可以更方便地加入。OOP 之所以提供了对递增开发的支持，是由于假如程序设计得好，每一次递增都可以成为完善的对象或者对象组。

⑩：这有点类似“快速造型”。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。快速造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。

1.12.7 计划的回报

如果没有仔细拟定的设计图，当然不可能建起一所房子。如建立的是一所狗舍，尽管设计图可以不必那么详尽，但仍然需要一些草图，以做到心中有数。软件开发则完全不同，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在意——似乎需要把自己的大多数时间花在编写文档上，而没有多少时间来编程（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，一些形式的计划会极大改善你的项目。请记住：根据估计，没有计划的50%以上的项目都会失败！

1.13 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 程序片，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 的简单易用就能有效地缩短开发时间。

最大的问题是速度。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。此时要用到人们称为“刚好及时”（Just-In Time，或 JIT）的编译器，甚至考虑更低级的代码编译器（写作本书时，也有两款问世）。当然，低级代码编译器会使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）。

在本书附录，大家可找到与 Java／C++比较．对 Java 现状的观察以及编码规则有关的内容。

第 2 章　一切都是对象

“尽管以 C++为基础，但 Java 是一种更纯粹的面向对象程序设计语言”。
无论 C++还是 Java 都属于杂合语言。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风格；之所以说 C++是一种杂合语言，是因为它支持与 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，正式用它设计之前，必须先将自己的思想转入一个面向对象的世界（除非早已习惯了这个世界的思维方式）。只有做好这个准备工作，与其他 OOP 语言相比，才能体会到 Java 的易学易用。在本章，我们将探讨 Java 程序的基本组件，并体会为什么说 Java 乃至 Java 程序内的一切都是对象。

2.1　用句柄操纵对象

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）？
所有这些在 Java 里都得到了简化，任何东西都可看作对象。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，但操纵的标识符实际是指向一个对象的“句柄”（Handle）。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（句柄）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（句柄），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，只是由于拥有一个句柄，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 句柄：

```
String s;
```

但这里创建的只是句柄，并不是对象。若此时向 s 发送一条消息，就会获得一个错误（运行期）。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：创建一个句柄时，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　所有对象都必须创建

创建句柄时，我们希望它同一个新对象连接。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）并非唯一的类型。Java 配套提供了数量众多的现成类型。对我们来讲，最重要的就是记住能自行创建类型。事实上，这应是 Java 程序设计的一项基本操作，是继续本书后余部分学习的基础。

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：
(1) 寄存器。这是最快的保存区域，因为它位于和其他所有保存方式不同的地方：处理器内部。然而，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踪迹。
(2) 堆栈。驻留于常规 RAM（随机访问存储器）区域，但可通过它的“堆栈指针”获得处理的直接支持。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——特别是对象句柄，但 Java 对象并不放到其中。

(3) 堆。一种常规用途的内存池（也在 RAM 区域），其中保存了 Java 对象。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停留多长的时间。因此，用堆保存数据时会得到更大的灵活性。要求创建一个对象时，只需用 new 命令编制相关的代码即可。执行这些代码时，会在堆里自动进行数据的保存。当然，为达到这种灵活性，必然会付出一定的代价：在堆里分配存储空间时会花掉更长的时间！
(4) 静态存储。这儿的“静态”（Static）是指“位于固定位置”（尽管也在 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，则程序不运行时仍可存在，并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“固定对象”。对于流式对象，对象会变成字节流，通常会发给另一台机器。而对于固定对象，对象保存在磁盘中。即使程序中止运行，它们仍可保持自己的状态不变。对于这些类型的数据存储，一个特别有用的技巧就是它们能存在于其他媒体中。一旦需要，甚至能将它们恢复成普通的、基于 RAM 的对象。Java 1.1 提供了对 Lightweight persistence 的支持。未来的版本甚至可能提供更完整的方案。

2.2.2 特殊情况：主要类型

有一系列类需特别对待；可将它们想象成“基本”、“主要”或者“主”（Primitive）类型，进行程序设计时要频繁用到它们。之所以要特别对待，是由于用 new 创建对象（特别是小的、简单的变量）并不是非常有效，因为 new 将对象置于“堆”里。对于这些类型，Java 采纳了与 C 和 C++相同的方法。也就是说，不是用 new 创建变量，而是创建一个并非句柄的“自动”变量。这个变量容纳了具体的值，并置于堆栈中，能够更高效地存取。
Java 决定了每种主要类型的大小。就象在大多数语言里那样，这些大小并不随着机器结构的变化而变化。这种大小的不可更改正是 Java 程序具有很强移植能力的原因之一。

主类型 大小 最小值 最大值 封装器类型

boolean 1 位 - - Boolean
char 16 位 Unicode 0 Unicode 2 的 16 次方-1 Character
byte 8 位 -128 +127 Byte（注释①）
short 16 位 -2 的 15 次方 +2 的 15 次方-1 Short（注释①）
int 32 位 -2 的 31 次方 +2 的 31 次方-1 Integer
long 64 位 -2 的 63 次方 +2 的 63 次方-1 Long
float 32 位 IEEE754 IEEE754 Float
double 64 位 IEEE754 IEEE754 Double
Void - - - Void（注释①）

①：到 Java 1.1 才有，1.0 版没有。

数值类型全都是有符号（正负号）的，所以不必费劲寻找没有符号的类型。
主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要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');
```

也可以直接使用：

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

这样做的原因将在以后的章节里解释。

1. 高精度数字
Java 1.1 增加了两个类，用于进行高精度的计算：BigInteger 和 BigDecimal。尽管它们大致可以划分为“封装器”类型，但两者都没有对应的“主类型”。

这两个类都有自己特殊的“方法”，对应于我们针对主类型执行的操作。也就是说，能对int或float做的事情，对BigInteger和BigDecimal一样可以做。只是必须使用方法调用，不能使用运算符。此外，由于牵涉更多，所以运算速度会慢一些。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选用的构建器和方法，请自行参考联机帮助文档。

2.2.3 Java的数组

几乎所有程序设计语言都支持数组。在C和C++里使用数组是非常危险的，因为那些数组只是内存块。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（属于常规编程错误），会产生不可预测的后果（注释②）。

②：在C++里，应尽量不要使用数组，换用标准模板库（Standard TemplateLibrary）里更安全的容器。

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里重复。一个Java可以保证被初始化，而且不可在它的范围之外访问。由于系统自动进行范围检查，所以必然要付出一些代价：针对每个数组，以及在运行期间对索引的校验，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。为此付出少许代价是值得的。
创建对象数组时，实际创建的是一个句柄数组。而且每个句柄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，就知道该句柄并未指向一个对象。正式使用前，必须为每个句柄都分配一个对象。若试图使用依然为null的一个句柄，就会在运行期报告问题。因此，典型的数组错误在Java里就得到了避免。
也可以创建主类型数组。同样地，编译器能够担保对它的初始化，因为会将那个数组的内存划分成零。
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，那么何时进行？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，将阐示Java如何帮助我们完成所有清除工作，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名字，作用域同时决定了它的“可见性”以及“存在时间”。在C，C++和Java里，作用域是由花括号的位置决定的。参考下面这个例子：

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

作为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可使用。
在上面的例子中，缩进排版使Java代码更易辨读。由于Java是一种形式自由的语言，所以额外的空格、制表位以及回车都不会对结果程序造成影响。
注意尽管在C和C++里是合法的，但在Java里不能象下面这样书写代码：

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

编译器会认为变量 x 已被定义。所以 C 和 C++能将一个变量“隐藏”在一个更大的作用域里。但这种做法在 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 */
```

那么句柄 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句柄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句柄。
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。这个编程问题在 C 和 C++里特别突出。看来在 C++里遇到的麻烦最大：由于不能从语言获得任何帮助，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，并最终造成程序的“凝固”呢。在 C++里，这个问题最令程序员头痛。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，以便能由新对象使用。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造成的“内存溢出”。

2.4 新建数据类型：类

如果说一切东西都是对象，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：“我准备告诉你对象一种新类型的外观”。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，应该跟随新数据类型的名称。例如：

```
class ATypeName {/*类主体置于这里}
```

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 ATypeName 里，类主体只由一条注释构成（星号和斜杠以及其中的内容，本章后面还会详细讲述），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 里的全部工作就是定义类、制作那些类的对象以及将消息发给那些对象），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（通过它的句柄与其通信），可以为任何类型。它也可以是主类型（并不是句柄）之一。如果是指向对象的一个句柄，则必须初始化那个句柄，用一种名为“构建器”（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，则可在类定义位置直接初始化（正如后面会看到的那样，句柄亦可在定义位置初始化）。

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，首先要写上对象句柄的名字，再跟随一个点号（句点），再跟随对象内部成员的名字。即“对象句柄.成员”。例如：

```
d.i = 47;
d.f = 1.1f;
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）。为正确理解工作原理，首先必须知道“自变量”和“返回值”的概念。我们马上就会详加解释。

1．主成员的默认值

若某个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，也可以保证它们获得一个默认值。

主类型 默认值

```
Boolean false
Char '\u0000'(null)
byte (byte)0
short (short)0
int 0
long 0L
float 0.0f
double 0.0d
```

一旦将变量作为类成员使用，就要特别注意由 Java 分配的默认值。这样做可保证主类型的成员变量肯定得到了初始化（C++不具备这一功能），可有效遏止多种相关的编程错误。

然而，这种保证却并不适用于“局部”变量——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。我们责任是在正式使用 x 前分配一个适当的值。如果忘记，就会得到一条编译期错误，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++编译器会对变量未初始化发出警告，但在 Java 里却是错误。

2.5 方法、自变量和返回值

迄今为止，我们一直用“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”。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的消息。通过本节的学习，大家会知道方法的定义有多么简单！

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

返回类型 方法名(/* 自变量列表*/) {/* 方法主体 */}

返回类型是指调用方法之后返回的数值类型。显然，方法名的作用是对具体的方法进行标识和引用。自变量列表列出了想传递给方法的信息类型和名称。
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（注释③），而且那个对象必须能够执行那个方法调用。若试图为一个对象调用错误的方法，就会在编译期得到一条出错消息。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(自变量1，自变量2，自变量3...)。举个例子来说，假设我们有一个方法名叫 f()，它没有自变量，返回的是类型为 int 的一个值。那么，假设有一个名为 a 的对象，可为其调用方法 f()，则代码如下：
int x = a.f();
返回值的类型必须兼容 x 的类型。
象这样调用一个方法的行动通常叫作“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”。

③：正如马上就要学到的那样，“静态”方法可针对类调用，毋需一个对象。

2.5.1 自变量列表

自变量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，我们必须在自变量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，我们实际传递的是“句柄”（注释④）。然而，句柄的类型必须正确。倘若希望自变量是一个“字串”，那么传递的必须是一个字串。

④：对于前面提及的“特殊”数据类型 boolean，char，byte，short，int，long，，float 以及 double 来说是一个例外。但在传递对象时，通常都是指传递指向对象的句柄。

下面让我们考虑将一个字串作为自变量使用的方法。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：

```
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）。自变量的类型为 String，而且叫作 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 的方法之一。该方法返回的是一个字串里的字符数。
通过上面的例子，也可以了解 return 关键字的运用。它主要做两件事情。首先，它意味着“离开方法，我已完工了”。其次，假设方法生成了一个值，则那个值紧接在 return 语句的后面。在这种情况下，返回值是通过计算表达式“s.length()*2”而产生的。
可按自己的愿望返回任意类型，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 2.718; }
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返回。但假设已指定了一种非 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

为自己的自变量使用，而且将消息发给那些对象。这种说法大体正确，但通过以后的学习，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，只需理解消息传送就足够了。

2.6 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

在所有程序设计语言里，一个不可避免的问题是对名字或名称的控制。假设您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。此时，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 语言里特别突出。因为程序未提供很好的名字管理方法。C++的类（即 Java 类的基础）嵌套使用类里的函数，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，所以仍然难以避免冲突。为解决这个问题，C++用额外的关键字引入了“命名空间”的概念。

由于采用全新的机制，所以 Java 能完全避免这些问题。为了给一个库生成明确的名字，采用了与 Internet 域名类似的名字。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。由于我的域名是 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 1.2 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（当然，一个文件里的类名必须是唯一的）。所以不必学习特殊的语言知识来解决这个问题——语言本身已帮我们照顾到这一点。

2.6.2 使用其他组件

一旦要在自己的程序里使用一个预先定义好的类，编译器就必须知道如何找到它。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仍未得到定义。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？您或许认为编译器应该足够“联盟”，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，但那个类的定义位于多个文件里。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、纠缠不清的情况。为达到这个目的，要用 import 关键字准确告诉 Java 编译器我们希望的类是什么。import 的作用是指示编译器导入一个“包”——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，它们是与编译器配套提供的。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.Vector;
```

它的作用是告诉编译器我们想使用 Java 的 Vector 类。然而，util 包含了数量众多的类，我们有时希望使用其中的几个，同时不想全部明确地声明它们。为达到这个目的，可使用“*”通配符。如下所示：

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

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

通常，我们创建类时会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。

但在两种特殊的情形下，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——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。所以尽管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）。
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，您会在其他一些 Java 书刊里发现这样的称呼。
为了将数据成员或方法设为static，只需在定义前置和这个关键字即可。例如，下述代码能生成一个 static 数据成员，并对其初始化：

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

现在，尽管我们制作了两个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 都有同样的值 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（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，++运算符会使变量增值。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。静态方法的定义是类似的：

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

从中可看出，StaticFun 的方法 incr()使静态数据 i 增值。通过对象，可用典型的方法调用 incr()：

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

或者，由于 incr()是一种静态方法，所以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，就会明确改变数据的创建方式（一个类一个成员，以及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。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入口方法 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，让我们正式编一个程序（注释⑤）。它能打印出与当前运行的系统有关的资料，并利用了来自 Java 标准库的 System 对象的多种方法。注意这里引入了一种额外的注释样式：“ // ”。它表示到本行结束前的所有内容都是注释：

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

public class Property {
  public static void main(String[] args) {
    System.out.println(new Date());
    Properties p = System.getProperties();
```

```
    p.list(System.out);
    System.out.println("--- Memory Usage:");
    Runtime rt = Runtime.getRuntime();
    System.out.println("Total Memory = "
                       + rt.totalMemory()
                       + " Free Memory = "
                       + rt.freeMemory());
  }
}
```

⑤：在某些编程环境里，程序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 {
Thread.currentThread().sleep(5 * 1000);
} catch(InterruptedException e) {}
}
```

它的作用是暂停输出 5 秒钟。这段代码涉及的一些概念要到本书后面才会讲到。所以目前不必深究，只知道它是让程序暂停的一个技巧便可。

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所有额外的类。注意我们说它们是“额外”的，因为一个特殊的类库会自动导入每个 Java 文件：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://www.java.sun.com 下载，或用其他方式安装了 Java 文档，请立即下载）。在 packages.html 文件里，可找到 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 默认进入每个 Java 代码文件，所以这些类在任何时候都可直接使用。在这个列表里，可发现 System 和 Runtime，我们在 Property.java 里用到了它们。java.lang 里没有列出 Date 类，所以必须导入另一个类库才能使用它。如果不清楚一个特定的类在哪个类库里，或者想检视所有的类，可在 Java 用户文档里选择“Class Hierarchy”（类分级结构）。在 Web 浏览器中，虽然要花不短的时间来建立这个结构，但可清楚找到与 Java 配套提供的每一个类。随后，可用浏览器的“查找”（Find）功能搜索关键字“Date”。经这样处理后，可发现我们的搜索目标以 java.util.Date 的形式列出。我们终于知道它位于 util 库里，所以必须导入 java.util.*；否则便不能使用 Date。

观察 packages.html 文档最开头的部分（我已将其设为自己的默认起始页），请选择 java.lang，再选 System。这时可看到 System 类有几个字段。若选择 out，就可知道它是一个 static PrintStream 对象。由于它是“静态”的，所以不需要我们创建任何东西。out 对象肯定是 3，所以只需直接用它即可。我们能对这个 out 对象做的事情由它的类型决定：PrintStream。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，就可看到能够为 PrintStream 调用的所有方法。方法的数量不少，本书后面会详细介绍。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，形式如下：

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

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()的自变量是包含了 String 对象的一个数组。args 不会在本程序中用到，但需要在这个地方列出，因为它们保存了在命令行调用的自变量。

程序的第一行非常有趣：

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

请观察它的自变量：创建 Date 对象唯一的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，我们没太大的必要关心“清除”的细节。

第二行调用了 System.getProperties()。若用 Web 浏览器查看联机用户文档，就可知道 getProperties()是

System 类的一个 static 方法。由于它是“静态”的，所以不必创建任何对象便可调用该方法。无论是否存在该类的一个对象，static 方法随时都可使用。调用 getProperties()时，它会将系统属性作为 Properties 类的一个对象生成（注意 Properties 是“属性”的意思）。随后的的句柄保存在一个名为 p 的 Properties 句柄里。在第三行，大家可看到 Properties 对象有一个名为 list()的方法，它将自己的全部内容都发给一个我们作为自变量传递的 PrintStream 对象。
main()的第四和第六行是典型的打印语句。注意为了打印多个 String 值，用加号（+）分隔它们即可。然而，也要在这里注意一些奇怪的事情。在 String 对象中使用时，加号并不代表真正的“相加”。处理字串时，我们通常不必考虑“+”的任何特殊含义。但是，Java 的 String 类要受一种名为“运算符过载”的机制的制约。也就是说，只有在随同 String 对象使用时，加号才会产生与其他任何地方不同的表现。对于字串，它的意思是“连接这两个字串”。
但事情到此并未结束。请观察下述语句：

```
System.out.println("Total Memory = "
+ rt.totalMemory()
+ " Free Memory = "
+ rt.freeMemory());
```

其中，totalMemory()和 freeMemory()返回的是数值，并非 String 对象。如果将一个数值“加”到一个字串身上，会发生什么情况呢？同我们一样，编译器也会意识到这个问题，并魔术般地调用一个方法，将那个数值（int，float 等等）转换成字串。经这样处理后，它们当然能利用加号“加”到一起。这种“自动类型转换”亦划入“运算符过载”处理的范畴。
许多 Java 著作都在热烈地辩论“运算符过载”（C++的一项特性）是否有用。目前就是反对它的一个好例子！然而，这最多只能算编译器（程序）的问题，而且只是对 String 对象而言。对于自己编写的任何源代码，都不可能使运算符“过载”。
通过为 Runtime 类调用 getRuntime()方法，main()的第五行创建了一个 Runtime 对象。返回的则是指向一个 Runtime 对象的句柄。而且，我们不必关心它是一个静态对象，还是由 new 命令创建的一个对象。这是由于我们不必为清除工作负责，可以大模大样地使用对象。正如显示的那样，Runtime 可告诉我们与内存使用有关的信息。

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++继承而来的。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。注意许多程序员在连续注释内容的每一行都用一个“*”开头，所以经常能看到象下面这样的内容：

```
/* 这是
* 一段注释，
* 它跨越了多个行
*/
```

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

```
/* 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，表示这一行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“*”（只需按同样的键两次），而且不必在注释结尾时加一个结束标记。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

对于 Java 语言，最体贴的一项设计就是它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个文件。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，并按有价值的形式将其展现出来。这些都是 Java 必须做到的。

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，所以我们能够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。

2.8.2 具体语法

所有 javadoc 命令都只能出现于“/**”注释中。但和平常一样，注释结束于一个“*/”。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（但前导的“*”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”（详见 5 章）成员的注释会被忽略，我们看不到任何输出（也可以用 -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，这是客户程序员的希望。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，用户会非常熟悉这种格式，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 HTML

javadoc 将 HTML 命令传递给最终生成的 HTML 文档。这便使我们能够充分利用 HTML 的巨大威力。当然，我们的最终动机是格式化代码，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 Web 文档里那样运用 HTML，对普通文本进行格式化，使其更具条理、更加美观：

```
/**
```

```
* 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，我们给出的标题会与之冲撞。
所有类型的注释文档——类、变量和方法——都支持嵌入 HTML。

2.8.4 @see：引用其他类

所有三种类型的注释文档都可包含@see 标记，它允许我们引用其他类里的文档。对于这个标记，javadoc 会生成相应的 HTML，将其直接链接到其他文档。格式如下：

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

每一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。注意 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

随同嵌入 HTML 和@see 引用，类文档还可以包括用于版本信息以及作者姓名的标记。类文档亦可用于“接口”目的（本书后面会详细解释）。

1. @version
格式如下：

```
@version 版本信息
```

其中，“版本信息”代表任何适合作为版本说明的资料。若在 javadoc 命令行使用了“-version”标记，就会从生成的 HTML 文档里提取出版本信息。

2. @author
格式如下：

```
@author 作者信息
```

其中，“作者信息”包括您的姓名、电子函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，就会专门从生成的 HTML 文档里提取出作者信息。
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。全部作者信息会一起存入最终 HTML 代码的单独一个段落里。

2.8.6 变量文档标记

变量文档只能包括嵌入的 HTML 以及@see 引用。

2.8.7 方法文档标记

除嵌入 HTML 和@see 引用之外，方法还允许使用针对参数、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”代表一些可延续到后续行内的说明文字。一旦遇到一个新文档标记，就认为前一个说明结束。可使用任意数量的说明，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. @exception
有关“违例”（Exception）的详细情况，我们会在第 9 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”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：
@exception 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）告诉我们为什么这种特殊类型的违例会在方法调用中出现。

4. @deprecated
这是 Java 1.1 的新特性。该标记用于指出一些旧功能已由改进过的新功能取代。该标记的作用是建议用户不必再使用一种特定的功能，因为未来改版时可能摒弃这一功能。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：

```
//: Property.java
import java.util.*;

/** The first Thinking in Java example program.
 * Lists system information on current machine.
 * @author Bruce Eckel
 * @author http://www.BruceEckel.com
 * @version 1.0
*/
public class Property {
  /** 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(new Date());
    Properties p = System.getProperties();
    p.list(System.out);
    System.out.println("--- Memory Usage:");
    Runtime rt = Runtime.getRuntime();
    System.out.println("Total Memory = "
                       + rt.totalMemory()
                       + " Free Memory = "
                       + rt.freeMemory());
  }
```

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

第一行：
//: Property.java
采用了我自己的方法：将一个“:”作为特殊的记号，指出这是包含了源文件名字的一个注释行。最后一行也用这样的一条注释结尾，它标志着源代码清单的结束。这样一来，可将代码从本书的正文中方便地提取出来，并用一个编译器检查。这方面的细节在第 17 章讲述。

2.9 编码样式

一个非正式的 Java 编程标准是大写一个类名的首字母。若类名由几个单词构成，那么把它们紧靠到一起（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 { // ...}
对于其他几乎所有内容：方法、字段（成员变量）以及对象句柄名称，可接受的样式与类样式差不多，只是标识符的第一个字母采用小写。例如：

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

当然，要注意用户也必须键入所有这些长名字，而且不能输错。

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

(1) 参照本章的第一个例子，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，并置入自变量列表——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。
(2) 写一个程序，打印出从命令行获取的三个自变量。
(3) 找出 Property.java 第二个版本的代码，这是一个简单的注释文档示例。请对文件执行 javadoc，并在自己的 Web 浏览器里观看结果。
(4) 以练习(1)的程序为基础，向其中加入注释文档。利用 javadoc，将这个注释文档提取为一个 HTML 文件，并用 Web 浏览器观看。

第 3 章 控制程序流程

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

在 Java 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。

3.1 使用 Java 运算符

运算符以一个或多个自变量为基础，可生成一个新值。自变量采用与原始方法调用不同的一种形式，但效果是相同的。根据以前写程序的经验，运算符的常规概念应该不难理解。
加号（+）、减号和负号（-）、乘号（*）、除号（/）以及等号（=）的用法与其他所有编程语言都是类似的。
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，一个运算符可改变运算对象的值，这叫作“副作用”（Side Effect）。运算符最常见的用途就是修改自己的运算对象，从而产生副作用。但要注意生成的值亦可由没有副作用的运算符生成。
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（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，把它复制到左边”。右边的值可以是任何常数、变量或者表达式，只要能产生一个值就行。但左边的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，可将一个常数赋给一个变量（A=4;），但不可将任何东西赋给一个常数（比如不能 4=A）。
对主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句柄，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句柄。所以倘若“从一个对象到另一个对象”赋值，实际就是将句柄从一个地方复制到另一个地方。这意味着假若为对象使用“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。
这里有一些题外话。在后面，大家在代码示例里看到的第一个语句将是“package 03”使用的“package”语句，它代表本书第 3 章。本书每一章的第一个代码清单都会包含象这样的一个“package”（封装、打包、包裹）语句，它的作用是为那一章剩余的代码建立章节编号。在第 17 章，大家会看到第 3 章的所有代码清单（除那些有不同封装名称的以外）都会自动置入一个名为 c03 的子目录里；第 4 章的代码置入 c04；以此类推。所有这些都是通过第 17 章展示的 CodePackage.java 程序实现的；“封装”的基本概念会在第 5 章进行详尽的解释。就目前来说，大家只需记住象“package 03”这样的形式只是用于为某一章的代码清单建立相应的子目录。
为运行程序，必须保证在 classpath 里包含了我们安装本书源码文件的根目录（那个目录里包含了 c02，c03c，c04 等等子目录）。

对于 Java 后续的版本（1.1.4 和更高版本），如果您的 main()用 package 语句封装到一个文件里，那么必须在程序名前面指定完整的包裹名称，否则不能运行程序。在这种情况下，命令行是：
java c03.Assignment
运行位于一个“包裹”里的程序时，随时都要注意这方面的问题。
下面是例子：

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

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 值都赋予了一个不同的值。随后，将 n2 赋给 n1，而且 n1 发生改变。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。但由于我们已赋予了一个句柄，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 都包含了相同的句柄，它指向相同的对象（最初的句柄位于 n1 内部，指向容纳了值 9 的一个对象。在赋值过程中，那个句柄实际已经丢失；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。但您很快就会意识到，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，所以留待第 12 章详细论述，那一章是专门讨论别名的。其时，大家也会注意到对象的赋值会产生一些令人震惊的效果。

1．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。

```
//: PassObject.java
// Passing objects to methods can be a bit tricky
```

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

在许多程序设计语言中，f()方法表面上似乎要在方法的作用域内制作自己的自变量Letter y 的一个副本。但同样地，实际传递的是一个句柄。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
```

别名和它的对策是非常复杂的一个问题。尽管必须等至第 12 章才可获得所有答案，但从现在开始就应加以重视，以便提早发现它的缺点。

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。其中包括加号（+）、减号（-）、除号（/）、乘号（*）以及模数（%，从整数除法中获得余数）。整数除法会直接砍掉小数，而不是进位。

Java 也用一种简写形式进行运算，并同时进行赋值操作。这是由等号前的一个运算符标记的，而且对于语言中的所有运算符都是固定的。例如，为了将4 加到变量x，并将结果赋给x，可用：x+=4。

下面这个例子展示了算术运算符的各种用法：

```
//: 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String；pInt()先打印一个String，再打印一个int；而pFlt()先打印一个String，再打印一个float。当然，它们最终都要用System.out.println()结尾。
为生成数字，程序首先会创建一个Random（随机）对象。由于自变量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。

1．一元加、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，再生成值。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：

```
//: 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“超载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，形成一种更精简的语言。正如大家会在这本书中学到的那样，Java的许多地方都得到了简化，所以Java的学习比C++更容易。

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。

1．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：

```
//: 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，句柄却是不同的，而==和!=比较的正好就是对象句柄。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。下面举例说明如何使用：

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

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！假设您创建了自己的类，就象下面这样：

```
//: 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()的默认行为是比较句柄。所以除非在自己的新类中改变了 equals()，否则不可能表现出我们希望的行为。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够避免一些“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，而非它们的句柄。

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）——以自变量的逻辑关系为基础。下面这个例子向大家展示了如何使用关系和逻辑运算符。

```
//: 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
```

注意若在预计为String 值的地方使用，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 以外的其他任何主数据类型。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。

1．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：

```
//: 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");
  }
} ///:~
```

每次测试都会比较自变量，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数据类型中的单个“比特”，即二进制位。按位运算符会对两个自变量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的设计初衷是嵌入电视顶置盒内，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；它只对一个自变量进行操作（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。

按位运算符可与等号（=）联合使用，以便合并运算及赋值：&=，|=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”。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，防止移动超过long值里现成的位数。但在进行“无符号”右移位时，也可能遇到一个问题。若对byte或short值进行右移位运算，得到的可能不是正确的结果（Java 1.0和Java 1.1特别突出）。它们会自动转换成int类型，并进行右移位。但“零扩展”不会发生，所以在那些情况下会得到-1的结果。可用下面这个例子检测自己的实现方案：

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

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，再将得到的结果赋回左边的值。
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，以及它们的效果：

```
//: 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。目前，可暂时忽略它们具体的实现方案。
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，以便在同一行里罗列多种信息。
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，使大家能体会它们的情况。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：

```
-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 的补值”。

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。但它确实属于运算符的一种，因为它最终也会生成一个值。这与本章后一节要讲述的普通 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”，而且它的结果成为最终由运算符产生的值。
当然，也可以换用普通的 if-else 语句（在后面介绍），但三元运算符更加简洁。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;
return i * 10;
}
```

但第二种形式更易理解，而且不要求更多的录入。所以在挑选三元运算符时，请务必权衡一下利弊。

3.1.10 逗号运算符

在C和C++里，逗号不仅作为函数自变量列表的分隔符使用，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“运算符过载”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，与C++的另外一些限制结合，运算符过载成为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，尽管运算符过载在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过载运算符。
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。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编译程序会将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，如果使用下述语句：

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

那么早期版本的Java就会提示出错（以后的版本能将x转换成一个字串）。因此，如果想通过“加号”连接字串（使用Java的早期版本），请务必保证第一个元素是字串（或加上引号的一系列字符，编译能将其识别成一个字串）。

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。即使对一个表达式如何计算有丝毫不确定的因素，都容易混淆括号的用法。这个问题在Java里仍然存在。
在C和C++中，一个特别常见的错误如下：

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

程序的意图是测试是否“相等”（==），而不是进行赋值操作。在C和C++中，若y是一个非零值，那么这种赋值的结果肯定是true。这样使可能得到一个无限循环。在Java里，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，而且不会从一个int数值中转换得来。所以在编译时，系统就会提示出现错误，有效地阻止我们进一步运行程序。所以这个缺点在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时候是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并不属于的类型。

3.1.13 造型运算符

“造型”（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。通过造型，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。
为进行一次造型，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;
}
```

正如您看到的那样，既可对一个数值进行造型处理，亦可对一个变量进行造型处理。但在这儿展示的两种情况下，造型均是多余的，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，仍然可以设置一个造型，提醒自己留意，也使程序更清楚。在其他情况下，造型只有在代码编译时才显出重要性。
在 C 和 C++中，造型有时会让人头痛。在 Java 里，造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造型，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，请明确造型。”而对于“放大转换”（Widening conversion），则不必进行明确造型，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“造型”为其他任何一种主类型，但布尔值（bollean）要除外，后者根本不允许进行任何造型处理。“类”不允许进行造型。为了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造型到一个类型“家族”里；例如，“橡树”可造型为“树”；反之亦然。但对于其他外来类型，如“岩石”，则不能造型为“树”）。

1. 字面值
最开始的时候，若在一个程序里插入“字面值”（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”。方法是用与字面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。

```
//: 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“缩小造型”。这样一来，我们就可清楚获知自己已超载了边界。
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“字面”表示方法。
字面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（注释①）。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”。

①：John Kirkham 这样写道：“我最早于 1962 年在一部 IBM 1620 机器上使用 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。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。”

注意如果编译器能够正确地识别类型，就不必使用尾随字符。对于下述语句：
```
long n3 = 200;
```
它并不存在含混不清的地方，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，告诉我们须用一个“造型”将 double 转换成 float。

2．转型
大家会发现假若对主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“造型”。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。正如您可能已经想到的那样，移植是令C和C++程序员颇为头痛的一个问题。
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 + - + + - [[其余的]]
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，上表并不是完美的助记方法；但对于其他运算来说，它确实很管用。

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数据类型。从根本上说，它是同一个例子反反复复地执行，只是使用了不同的主数据类型。文件编译时不会报错，因为那些会导致错误的行已用//!变成了注释内容。

```
//: 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“造型”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，却不必进行造型处理，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。如果对两个足够大的 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：

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

输出结果如下：

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

而且不会从编译器那里收到出错提示，运行时也不会出现异常反应。爪哇咖啡（Java）确实是很好的东西，但却没有“那么”好！
对于 char，byte 或者 short，混合赋值并不需要造型。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，将造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造型变为其他主类型。同样地，当造型成一种较小的类型时，必须留意“缩小转换”的后果。否则会在造型过程中不知不觉地丢失信息。

3.2 执行控制

Java 使用了 C 的全部控制语句，所以假期您以前用 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，涉及的关键字包括 if-else、while、do-while、for 以及一个名为 switch 的选择语句。然而，Java 并不支持非常有害的 goto（它仍是解决某些特殊问题的权宜之计）。仍然可以进行象 goto 那样的跳转，但比典型的 goto 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，下面这个 test()方法可告诉我们猜测的一个数字位于目标数字之上、之下还是相等：

```
static int test(int testval) {
  int result = 0;
  if(testval > target)
    result = -1;
  else if(testval < target)
    result = +1;
  else
    result = 0; // match
  return result;
}
```

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。

1. 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：

```
static int test2(int testval) {
  if(testval > target)
    return -1;
  if(testval < target)
    return +1;
  return 0; // match
}
```

不必加上else，因为方法在遇到 return 后便不再继续。

3.2.3 反复

while，do-while 和 for 控制着循环，有时将其划分为“反复语句”。除非用于控制反复的布尔表达式得到“假”的结果，否则语句会重复执行下去。while 循环的格式如下：

```
while(布尔表达式)
语句
```

在循环刚开始时，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：

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

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 肯定会至少执行一次；也就是说，至少会将其中的语句“过一遍”——即便表达式第一次便计算为 false。而在 while 循环结构中，若条件第一次就为 false，那么其中的语句根本不会执行。在实际应用中，while 比 do-while 更常用一些。

3.2.5 for

for 循环在第一次反复之前要进行初始化。随后，它会进行条件测试，而且在每一次反复的时候，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)
语句
```

无论初始表达式，布尔表达式，还是步进，都可以置空。每次反复前，都要测试一下布尔表达式。若获得的结果是 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：

```
//: 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
      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 的作用域是由 for 控制的表达式。
以于象 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。所以在编译器创建一个块的时候，它可以为那些变量分配空间。而在 Java 和 C++中，则可在整个块的范围内分散变量声明，在真正需要的地方才加以定义。这样便可形成更自然的编码风格，也更易理解。
可在 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++)
 /* body of for loop */;
```

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

1. 逗号运算符
早在第 1 章，我们已提到了逗号运算符——注意不是逗号分隔符；后者用于分隔函数的不同自变量。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种能力，下面则是另一个例子：

```
//: 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
```

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

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，不执行循环中剩余的语句。而 continue 则停止执行当前的反复，然后退回循环起始和，开始新的反复。
下面这个程序向大家展示了 break 和 continue 在 for 和 while 循环中的例子：

```
//: 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 到达 74，break 语句就会中断循环。通常，只有在不知道中断条件何时满足时，才需象这样使用 break。只要 i 不能被 9 整除，continue 语句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，则将值显示出来。
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 continue 移回循环顶部，同时不完成剩余的内容（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(;;)看作同一回事。所以具体选用哪个取决于自己的编程习惯。

1．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 ”。若阅读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许多跳转。然而，goto 是在源码的级别跳转的，所以招致了不好的声誉。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代码的流程呢？随着 Edsger Dijkstra 著名的“ Goto 有害 ”论的问世，goto 便从此失宠。
事实上，真正的问题并不在于使用 goto，而在于 goto 的滥用。而且在一些少见的情况下，goto 是组织控制流程的最佳手段。
尽管 goto 仍是 Java 的一个保留字，但并未在语言中得到正式使用；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，就象下面这样：

```
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：

```
//: 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.
          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，就没有办法在一个内部循环里找到出外部循环的路径。这是由于break本身只能中断最内层的循环（对于continue同样如此）。
当然，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：

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

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，并移离当前标签的末尾。
(4) 带标签的break 会中断当前循环，并移离由那个标签指示的循环的末尾。
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 里唯一需要用到标签的地方就是拥有嵌套循环，而且想中断或继续多个嵌套级别的时候。
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到处传递程序的控制权。由此也引出了一个有趣的问题：通过限制语句的能力，反而能使一项语言特性更加有用。

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）。若没有发现相符的，就执行 default 语句。
在上面的定义中，大家会注意到每个 case 均以一个 break 结尾。这样可使执行流程跳转至 switch 主体的末尾。这是构建 switch 语句的一种传统方式，但 break 是可选的。若省略 break，会继续执行后面的 case 语句的代码，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，也许能够善加利用。注意最后的 default 语句没有 break，因为执行流程已到了 break 的跳转目的地。当然，如果考虑到编程风格方面的原因，完全可以在 default 语句的末尾放置一个 break，尽管它并没有任何实际的用处。
switch 语句是实现多路选择的一种易行方式（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，假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：

```
//: 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，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，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。也应注意将 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。

1．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，必须无将'a'转换成一个 double。利用一个“造型”，double 结果会转换回 char。

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

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

输出结果如下：

```
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 值造型成整数值后，总是将小数部分“砍掉”，不作任何进位处理。

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：

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

public class RandomBounds {
  static void usage() {
    System.err.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。若想到 0 和 1 之间有 2 的 128 次方不同的双精度小数，所以如果全部产生这些数字，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。或者用数字语言表达，输出值范围是 [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循环等等。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

(1) 写一个程序，打印出 1 到 100 间的整数。

(2) 修改练习(1)，在值为 47 时用一个 break 退出程序。亦可换成 return 试试。

(3) 创建一个 switch 语句，为每一种 case 都显示一条消息。并将 switch 置入一个 for 循环里，令其尝试每一种 case。在每个 case 后面都放置一个 break，并对其进行测试。然后，删除 break，看看会有什么情况出现。

第 4 章 初始化和清除

“随着计算机的进步，‘不安全’的程序设计已成为造成编程代价高昂的罪魁祸首之一。”

“初始化”和“清除”是这些安全问题的其中两个。许多 C 程序的错误都是由于程序员忘记初始化一个变量造成的。对于现成的库，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）用尽的后果。
C++为我们引入了“构建器”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 用构建器自动初始化

对于方法的创建，可将其想象成为自己写的每个类都调用一次 initialize()。这个名字提醒我们在使用对象之前，应首先进行这样的调用。但不幸的是，这也意味着用户必须记住调用方法。在 Java 中，由于提供了名为“构建器”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。若某个类有一个构建器，那么在创建对象时，Java 会自动调用那个构建器——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。存在两方面的问题。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建器，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：构建器的名字与类名相同。这样一来，可保证象这样的一个方法会在初始化期间自动调用。
下面是带有构建器的一个简单的类（若执行这个程序有问题，请参考第 3 章的“赋值”小节）。

```
//: SimpleConstructor.java
// Demonstration of a simple constructor
package c04;

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，并调用构建器。这样可保证在我们经手之前，对象得到正确的初始化。
请注意所有方法首字母小写的编码规则并不适用于构建器。这是由于构建器的名字必须与类名完全相同！
和其他任何方法一样，构建器也能使用自变量，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，以便构建器使用自己的自变量。如下所示：

```
class Rock {
  Rock(int i) {
```

```
    System.out.println(
      "Creating Rock number " + i);
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock(i);
  }
}
```

利用构建器的自变量，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 有一个构建器，它用一个整数自变量标记树的高度，那么就可以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)是我们唯一的构建器，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。
构建器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——那些方法在概念上独立于定义内容。在 Java 中，定义和初始化属于统一的概念——两者缺一不可。
构建器属于一种较特殊的方法类型，因为它没有返回值。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。构建器则不同，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。

4.2 方法过载

在任何程序设计语言中，一项重要的特性就是名字的运用。我们创建一个对象时，会分配到一个保存区域的名字。方法名代表的是一种具体的行动。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——目的是与读者沟通。
我们用名字引用或描述所有对象与方法。若名字选得好，可使自己及其他人更易理解自己的代码。
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“过载”。我们说“洗衬衫”、“洗车”以及“洗狗”。但若强制象下面这样说，就显得很愚蠢：“衬衫洗 衬衫”、“车洗 车”以及“狗洗 狗”。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，仍然可以推断出含义。我们不需要独一无二的标识符——可从具体的语境中推论出含义。
大多数程序设计语言（特别是 C）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 里，另一项因素强迫方法名出现过载情况：构建器。由于构建器的名字由类名决定，所以只能有一个构建器名称。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，我们需要两个构建器，一个没有自变量（默认构建器），另一个将字串作为自变量——用于初始化对象的那个文件的名字。由于都是构建器，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自变量类型使用，“方法过载”是非常关键的一项措施。同时，尽管方法过载是构建器必需的，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过载构建器和过载的原始方法：

```
//: 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 既可创建成一颗种子，不含任何自变量；亦可创建成生长在苗圃中的植物。为支持这种创建，共使用了两个构建器，一个没有自变量（我们把没有自变量的构建器称作“默认构建器”，注释①），另一个采用现成的高度。

①：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建器——“无参数构建器”（no-arg constructors）。但“默认构建器”这个称呼已使用了许多年，所以我选择了它。

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 自变量；而假设没有其他话可说，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，方法过载允许我们为两者使用相同的名字。

4.2.1 区分过载方法

若方法有同样的名字，Java 怎样知道我们指的哪一个方法呢？这里有一个简单的规则：每个过载的方法都必须采取独一无二的自变量类型列表。

若稍微思考几秒钟，就会想到这样一个问题：除根据自变量的类型，程序员如何区分两个同名方法的差异呢？

即使自变量的顺序也足够我们区分两个方法（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）：

```
//: 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()方法有完全一致的自变量，但顺序不同，可据此区分它们。

4.2.2 主类型的过载

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。涉及过载问题时，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过载的方法时发生的情况：

```
//: 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;
```