

国外计算机科学经典教材

Java 语言的科学与艺术

(美) Eric S. Roberts 著

付勇 译

清华大学出版社

北京



Authorized translation from the English language edition, entitled *The Art & Science of Java*, 978-0-321-48612-7 by Eric S.Roberts, published by Pearson Education, Inc, publishing as Addison-Wesley, Copyright © 2008.

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

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and TSINGHUA UNIVERSITY PRESS Copyright © 2008.

北京市版权局著作权合同登记号 图字：01-2007-2032

本书封面贴有 Pearson Education(培生教育出版集团)防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

Java 语言的科学与艺术 / (美) 罗伯茨(Roberts, E.S.) 著；付勇 译。—北京：清华大学出版社，2009.1
(国外计算机科学经典教材)

书名原文：The Art & Science of Java

ISBN 978-7-302-18441-6

I. J... II. ①罗…②付… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2008)第 130802 号

责任编辑：王军 徐燕萍

装帧设计：孔祥丰

责任校对：成凤进

责任印制：何芊

出版发行：清华大学出版社

<http://www.tup.com.cn>

社 总 机：010-62770175

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zbliliang@tup.tsinghua.edu.cn

印 刷 者：北京密云胶印厂

装 订 者：三河市新茂装订有限公司

经 销：全国新华书店

开 本：185×260 印 张：80 字 数：768 千字

版 次：2009 年 1 月第 1 版 印 次：2009 年 1 月第 1 次印刷

印 数：1—4000

定 价：59.80 元

地 址：北京清华大学学研大厦 A 座

邮 编：100084

邮 购：010-62786544

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系
调换。联系电话：(010)62770177 转 3103 产品编号：020079-01

出版说明

近年来，我国的高等教育特别是计算机学科教育，进行了一系列大的调整和改革，亟需一批门类齐全、具有国际先进水平的计算机经典教材，以适应我国当前计算机科学的教学需要。通过使用国外优秀的计算机科学经典教材，可以了解并吸收国际先进的教学思想和教学方法，使我国的计算机科学教育能够跟上国际计算机教育发展的步伐，从而培养出更多具有国际水准的计算机专业人才，增强我国计算机产业的核心竞争力。为此，我们从国外多家知名的出版机构 Pearson、McGraw-Hill、John Wiley & Sons、Springer、Thomson 等精选、引进了这套“国外计算机科学经典教材”。

作为世界级的图书出版机构，Pearson、McGraw-Hill、John Wiley & Sons、Springer、Thomson 通过与世界级的计算机教育大师携手，每年都为全球的计算机高等教育奉献大量的优秀教材。清华大学出版社和这些世界知名的出版机构长期保持着紧密友好的合作关系，这次引进的“国外计算机科学经典教材”便全是出自上述这些出版机构。同时，为了组织该套教材的出版，我们在国内聘请了一批知名的专家和教授，成立了专门的教材编审委员会。

教材编审委员会的运作从教材的选题阶段即开始启动，各位委员根据国内外高等院校计算机科学及相关专业的现有课程体系，并结合各个专业的培养方向，从上述这些出版机构出版的计算机系列教材中精心挑选针对性强的题材，以保证该套教材的优秀性和领先性，避免出现“低质重复引进”或“高质消化不良”的现象。

为了保证出版质量，我们为该套教材配备了一批经验丰富的编辑、排版、校对人员，制定了更加严格的出版流程。本套教材的译者，全部由对应专业的高校教师或拥有相关经验的 IT 专家担任。每本教材的责编在翻译伊始，就定期不间断地与该书的译者进行交流与反馈。为了尽可能地保留与发扬教材原著的精华，在经过翻译、排版和传统的三审三校之后，我们还请编审委员或相关的专家教授对文稿进行审读，以最大程度地弥补和修正在前面一系列加工过程中对教材造成的误差和瑕疵。

由于时间紧迫和受全体制作人员自身能力所限，该套教材在出版过程中很可能还存在一些遗憾，欢迎广大师生来电来信批评指正。同时，也欢迎读者朋友积极向我们推荐各类优秀的国外计算机教材，共同为我国高等院校计算机教育事业贡献力量。

清华大学出版社

国外计算机科学经典教材

编审委员会

主任委员:

孙家广 清华大学教授

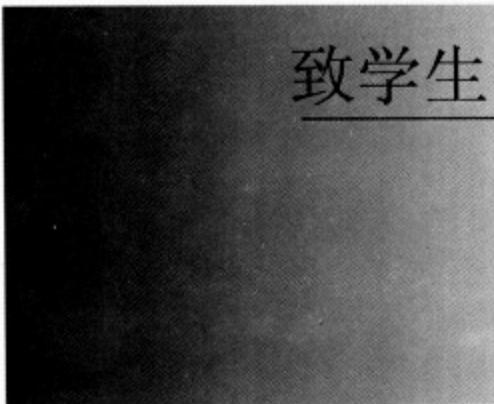
副主任委员:

周立柱 清华大学教授

委员（按姓氏笔画排序）：

王成山	天津大学教授
王 珊	中国人民大学教授
冯少荣	厦门大学教授
冯全源	西南交通大学教授
刘乐善	华中科技大学教授
刘腾红	中南财经政法大学教授
吉根林	南京师范大学教授
孙吉贵	吉林大学教授
阮秋琦	北京交通大学教授
何 晨	上海交通大学教授
吴百锋	复旦大学教授
李 彤	云南大学教授
沈钧毅	西安交通大学教授
邵志清	华东理工大学教授
陈 纯	浙江大学教授
陈 钟	北京大学教授
陈道蓄	南京大学教授
周伯生	北京航空航天大学教授
孟祥旭	山东大学教授
姚淑珍	北京航空航天大学教授
徐佩霞	中国科学技术大学教授
徐晓飞	哈尔滨工业大学教授
秦小麟	南京航空航天大学教授
钱培德	苏州大学教授
曹元大	北京理工大学教授
龚声蓉	苏州大学教授
谢希仁	中国人民解放军理工大学教授

致学生



恭喜你！当你拿起这本书的时候，就已经步入了计算机科学的殿堂——这一研究领域半个世纪以前还是一片空白，但现在已经发展成为我们这个时代最具活力的学科之一。

现在这个时代，计算机在人类活动的各个领域都创造了非凡的可能。今天，企业领导者能够管理遍布全球的企业，因为计算机让他们能够在任何地方以极快的速度传递数据。科学家现在能够解决很多问题，而在没有计算机进行必要的计算之前，这些问题超出了他们的能力范围。WWW提供了大量信息，它建立在一个充满活力的行业基础之上，而这个行业在十几年前根本不存在。电影摄制者可以使用计算机技术创建动画特征，而这些动画特征在迪斯尼时代简直不可想象。现代计算促进了许多领域发生变革：让生物学家能够排列人类基因，让经济学家能够模拟国际金融市场，让文学家能够确定非 Elizabethan 所写的原稿是否为 Shakespeare 所写。

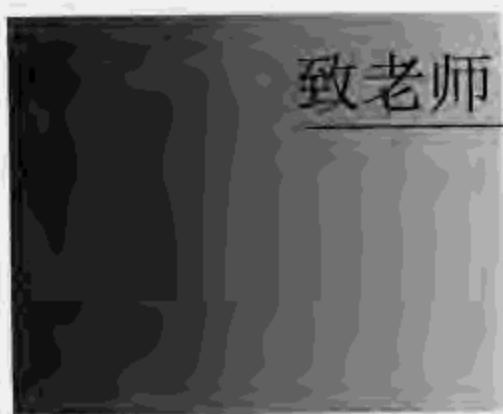
计算是一种意义深远的重要技术。随着计算产业不断发展，它提供的就业岗位比 20 世纪 90 年代 Internet 快速发展时提供的岗位还要多。但相对于本世纪我们将要经历的变化而言，现在看到的这些发展微不足道，今天还是学生的你们不久就会担负起引领这种发展的重任。不管将来选择什么领域，理解如何有效使用计算机都非常重要。

就像许多值得学习的技能一样，理解计算机的工作原理、学习如何控制它强大的能力也需要时间。你们不可能一蹴而就，必须从某一点开始。2500 年前，中国哲学家老子曾经说过“千里之行，始于足下”，本书就是一个开端。

然而，对于大多数同学而言，第一步是最艰难的。许多同学感到计算机过于强大，认为计算机科学超出了自己的理解范围。但是，学习编程基础不需要高等数学知识或全面理解电子学。对编程而言，最重要的是能否从问题的陈述中找到它的解决方案。因此，你必须能够从逻辑上思考。必须遵循必要的原则，以计算机可以理解的形式表达自己的逻辑。也许最重要的是，必须能够洞穿任务、成功完成，而不会被困难和挫折所吓倒。如果不断前进，你们就会发现，找到解决方案是多么令人欢欣鼓舞，这足以补偿一路上遇到的所有挫折。

祝愿你们在这条路上一路走好！

Eric Roberts
斯坦福大学
2007 年 1 月



本书适合普通学院或大学的初级编程课使用。书中介绍了传统 CS1 课程的内容，正如计算机协会(ACM)准备的 *Curriculum '78* 报告中定义的一样。同时也包含了最近的 *Computing Curriculum 2001* 报告计算机科学卷中指定为 CS101_o 或 CS111_o 课程的全部主题。

本书使用的方法与我 1995 年所著的 *The Art and Science of C* 一书使用的方法相同。这两本书都使用库来简化编程，从而让它们更适合于初学者。在 C 语言版中，事实证明这些库对于学生而言(不管是在斯坦福大学，还是在其他许多机构)都非常成功，本书使用 ACM Java 库来实现相同的目标。

自 1995 年首次发布以来，Java 编程语言作为一种教育语言变得日益重要，以至于现在成了初级计算课程的标准语言。在好的方面，Java 相对于早期教育语言具有更多优点，特别是它可以让学生编写高度交互式程序，这充分激发了他们的兴趣和想象。但 Java 比其他传统教育语言(例如 BASIC 和 Pascal)更复杂，老师和学生们在理解 Java 语言的结构时，复杂性就成为了最大的障碍。

为了解决初级老师在使用 Java 时会遇到的问题，2004 年 ACM 建立了 Java 工作组(Java Task Force)，并给出如下指示：

要从初级计算教育的角度讨论 Java 语言、API 和工具，要开发稳定的教育资源集合，以方便给一年级计算学生讲授 Java，让这些学生不至于被其复杂性所吓倒。

在接下来的两年时间内，Java 工作组开发了一组新库，这些库支持在初级层次使用 Java。在发行两个草案初稿进行了社会反馈后，2006 年夏天 Java 工作组发行了最终报告。在下面 Web 站点可以找到报告中描述的 ACM Java 库：

<http://jte.acm.org/>

除了 ACM Java 库本身之外，该 Web 站点还包含了大量演示程序、教学软件指南、各种程序包设计基本原理的详细讨论，以及确定下面几点为库最大优势的执行概要。ACM Java 库提供：

- 简单的面向对象程序模型。acm.program 程序包里定义的 Program 类为编写简单程序提供了易于使用的模型。除了隐藏静态 main 方法之外，Program 类及其标准子类提供了高度直观的面向对象的类层次结构示例。

11.8 ArrayList 类	354
11.9 小结	358
11.10 复习题	359
11.11 编程练习	360
第 12 章 搜索与排序	369
12.1 搜索	370
12.1.1 在整数数组中搜索	370
12.1.2 搜索表	370
12.1.3 折半搜索	373
12.1.4 搜索算法的相对效率	375
12.2 排序	376
12.2.1 给整数数组排序	376
12.2.2 选项排序算法	377
12.2.3 评估选项排序的效率	379
12.2.4 测量程序的运行时间	380
12.2.5 分析选项排序算法	381
12.2.6 基数排序算法	382
12.3 评估算法效率	384
12.3.1 big-O 表示法	385
12.3.2 big-O 的标准简化	385
12.3.3 选项排序计算的复杂性	385
12.3.4 根据代码结构预测计算 的复杂性	386
12.3.5 最坏情况与一般情况的 复杂性	387
12.4 使用数据文件	388
12.4.1 文件的概念	388
12.4.2 阅读 Java 中的文本文件	389
12.4.3 异常处理	391
12.4.4 倒转文件的程序	393
12.4.5 交互地选择文件	394
12.4.6 使用 Scanner 类	395
12.4.7 输出文件	397
12.5 小结	397
12.6 复习题	398
12.7 编程练习	400
第 13 章 数组与 ArrayList 类	405
13.1 ArrayList 类回顾	406
13.1.1 动态分配的能力	406
13.1.2 区分表示法和行为	408
13.2 HashMap 类	409
13.2.1 映射的简单示例	409
13.2.2 探讨可能的实现策略	410
13.2.3 实现 O(1) 性能	411
13.2.4 散列法思想	412
13.2.5 散列码	412
13.2.6 复制散列码	413
13.2.7 映射类的简单实现	413
13.2.8 实现 hashCode 方法	416
13.2.9 调整存储区的数量	418
13.3 Java 集合架构	418
13.3.1 Java 集合架构的结构	419
13.3.2 Collection 层次结构	420
13.3.3 区分行为和表示法	421
13.3.4 迭代器	422
13.3.5 Arrays 和 Collections 方法库	424
13.4 面向对象设计的原则	426
13.4.1 统一主题的重要性	427
13.4.2 简单和信息隐藏的原则	428
13.4.3 满足客户的需要	428
13.4.4 灵活性的好处	429
13.4.5 稳定的意义	429
13.5 小结	430
13.6 复习题	430
13.7 编程练习	431
第 14 章 展望	435
14.1 递归	436
14.1.1 递归的简单示例	436
14.1.2 Factorial 函数	437
14.1.3 信任的递归式跳跃	440
14.1.4 递归的范例	441
14.1.5 图形递归	442
14.1.6 递归式思考	444
14.2 并发	445
14.2.1 进程与线程	445
14.2.2 并发的简单示例	445
14.3 使用网络	448

有助于学生理解算法设计的重要性。

第 13 章介绍了 Java 集合架构，它在第二学期的编程课中也许更常见。然而，因为 Java 考虑到了这么多基本细节，所以教初学者如何使用这些类实际上非常合理，即使他们不理解其实现方式。Java 集合架构中唯一必不可少的类是 `ArrayList`，第 11 章已经提到过它。

第 14 章介绍了 4 个重要主题，这些主题有时会在初级编程课程中出现：递归、并发、网络和编程模式。斯坦福大学实行的是 4 学期制，因此在第二学期介绍这些主题。如果要在第一学期介绍递归，我强烈建议您尽早这样做，以便给学生提供更多时间来消化吸收这些材料。一种可能是在第 5 章之后开始讨论递归函数，在第 12 章的最后介绍递归算法。

补充资源

学生适用

本书的所有读者都可以在 Addison-Wesley 的 Web 站点(<http://www.aw.com/cssupport/>)获得下列几项资源：

- 书中所有示例程序的源代码文件。
- 彩色 PDF 版本的样本运行。
- 复习题的答案。

教师适用

认证教师可以在 Addison-Wesley 的教师资源中心(Instructor Resource Center，网址为 <http://www.aw.com/irc/>)获得下列几项资源：

- 书中所有示例程序的源代码文件。
- 彩色 PDF 版本的样本运行。
- 复习题的答案。
- 编程练习的解决方案。
- 基于 Applet 且包含程序示例动画的教学幻灯片。

ACM Java 库的用户适用

计算机协会(ACM)维护 Java 工作组开发了 ACM Java 库的 Web 站点(<http://jtf.acm.org/>)。该站点包括下列资源：

- ACM Java 库用法的执行概要。
- 可以以源代码及编译形式下载的 ACM 库的副本。
- 包含示例源代码的演示图库。
- 使用 ACM 库的初级指南。
- 设计基本原理的全面介绍。

目 录

第1章 前言	1		
1.1 计算简史	1	2.6.1 HelloProgram 示例回顾	29
1.2 计算机科学的含义	3	2.6.2 向 GObjects 发送消息	30
1.3 计算机硬件简介	4	2.6.3 GObject 类的层次结构	32
1.3.1 CPU	5	2.6.4 GRect 类	33
1.3.2 内存	5	2.6.5 GOval 类	36
1.3.3 辅助存储器	5	2.6.6 GLine 类	37
1.3.4 输入/输出(I/O)设备	5	2.7 小结	38
1.3.5 网络	5	2.8 复习题	39
1.4 算法	6	2.9 编程练习	40
1.5 编程过程的几个阶段	6		
1.5.1 创建和编辑程序	7	第3章 表达式	43
1.5.2 编译过程	7	3.1 原始数据类型	44
1.5.3 编程错误与调试	9	3.2 常量与变量	45
1.5.4 软件维护	9	3.2.1 常量	45
1.6 Java 和面向对象范例	10	3.2.2 变量	46
1.6.1 面向对象编程的历史	11	3.2.3 声明	47
1.6.2 Java 编程语言	11	3.2.4 命名常量	48
1.7 Java 和 WWW	14	3.3 运算符和操作数	48
1.8 小结	15	3.3.1 合并整数和浮点数	49
1.9 复习题	16	3.3.2 整数除法和余数运算符	50
第2章 编程示例	17	3.3.3 优先级	50
2.1 “Hello world” 程序	18	3.3.4 应用优先级规则	52
2.1.1 注释	19	3.3.5 类型转换	53
2.1.2 输入	19	3.4 赋值语句	55
2.1.3 主类	19	3.4.1 简写赋值运算符	57
2.2 编程过程的观点	21	3.4.2 递增运算符和递减运算符	58
2.3 两数相加的程序	22	3.5 布尔表达式	58
2.4 编程习语和模式	25	3.5.1 关系运算符	58
2.5 类和对象	26	3.5.2 逻辑运算符	59
2.5.1 类的层次结构	27	3.5.3 短路赋值	61
2.5.2 Program 类的层次结构	28	3.5.4 标志	62
2.6 图形程序	29	3.5.5 Boolean 计算示例	62
		3.6 设计改变	63
		3.6.1 可读性的重要性	63

3.6.2 使用命名常量支持程序维护	64	5.1.4 作为消息的方法调用	108
3.6.3 使用命名常量支持程序开发	64	5.2 编写自己的方法	109
3.7 小结	66	5.2.1 方法定义的格式	109
3.8 复习题	68	5.2.2 return 语句	110
3.9 编程练习	69	5.2.3 包含内部控制结构的方法	111
第4章 语句形式	73	5.2.4 返回非数学值的方法	112
4.1 Java 的语句类型	74	5.2.5 断言方法	115
4.1.1 简单语句	74	5.3 方法调用过程的技巧	116
4.1.2 复合语句	75	5.3.1 参数传递	117
4.1.3 控制语句	76	5.3.2 从其他方法内调用方法	120
4.2 控制语句和问题解决	76	5.4 分解	126
4.2.1 一般化 Add2Integers 程序	77	5.4.1 逐步细化	126
4.2.2 重复 N 次模式	78	5.4.2 指定参数	128
4.2.3 “读取到指定条件为止” 模式	79	5.4.3 自顶向下设计	129
4.3 if 语句	80	5.4.4 寻找共同特征	130
4.3.1 单行 if 语句	82	5.4.5 完成分解	131
4.3.2 多行 if 语句	82	5.5 算法方法	132
4.3.3 if-else 语句	83	5.5.1 “强力” 方法	132
4.3.4 缩联 if 语句	83	5.5.2 欧几里得算法	133
4.3.5 ?: 运算符	84	5.5.3 讨论欧几里得算法的正确性	133
4.4 switch 语句	85	5.5.4 两种算法的效率比较	134
4.5 while 语句	88	5.6 小结	135
4.5.1 使用 while 循环	88	5.7 复习题	135
4.5.2 无限循环	90	5.8 编程练习	136
4.5.3 解决“循环到中途” 问题	90	第6章 对象和类	141
4.6 for 语句	92	6.1 使用 RandomGenerator 类	142
4.6.1 for 和 while 之间的关系	94	6.1.1 伪随机数	142
4.6.2 在浮点数据中使用 for 语句	95	6.1.2 使用 RandomGenerator 类	143
4.6.3 嵌套 for 语句	96	6.1.3 随机数种子的作用	145
4.6.4 简单的图形动画	97	6.2 javadoc 文档系统	146
4.7 小结	99	6.3 定义自己的类	149
4.8 复习题	100	6.3.1 类定义的结构	149
4.9 编程练习	100	6.3.2 控制条目的可见性	150
第5章 方法	105	6.3.3 封装	150
5.1 方法概述	105	6.4 表示学生信息	150
5.1.1 作为降低复杂性机制的方法	106	6.4.1 声明实例变量	151
5.1.2 作为编程人员工具而不是		6.4.2 完成类定义	151
用户工具的方法	106	6.4.3 编写 javadoc 注释	154
5.1.3 作为表达式的方法调用	107	6.4.4 写构造函数	154

6.4.5	getters and setters	155	8.2.3	字符常量	205
6.4.6	toString 方法	155	8.2.4	Unicode 表示法的重要属性	205
6.4.7	定义类中的命名常量	156	8.2.5	特殊字符	206
6.4.8	使用 Student 类	156	8.2.6	字符算法	207
6.5	有理数	157	8.2.7	Character 类中有用的方法	208
6.6	扩展现有类	161	8.2.8	包含字符的控制语句	209
6.6.1	创建类表示实心三角形	161	8.3	作为抽象概念的字符串	210
6.6.2	继承构造函数的规则	165	8.3.1	从整体和简化论的观点 考察字符串	210
6.6.3	继承方法的规则	165	8.3.2	抽象类型的概念	210
6.7	小结	167	8.4	使用 String 类中的方法	211
6.8	复习题	168	8.4.1	确定字符串长度	212
6.9	编程练习	169	8.4.2	从字符串中选择字符	212
第 7 章	对象和内存	175	8.4.3	串联	213
7.1	内存结构	176	8.4.4	摘录部分字符串	215
7.1.1	位、字节和字	176	8.4.5	比较两个字符串	215
7.1.2	二进制和十六进制表示法	176	8.4.6	在字符串内搜索	216
7.1.3	内存地址	178	8.4.7	大小写字母转换	217
7.2	将内存分配给变量	179	8.5	字符串处理案例研究	218
7.2.1	Rational 类的内存图	180	8.5.1	应用自顶向下设计	218
7.2.2	无用单元收集	184	8.5.2	实现 translateLine	219
7.3	原始类型与对象	185	8.5.3	考虑空格和标点符号	220
7.3.1	参数传递	185	8.5.4	StringTokenizer 类	222
7.3.2	包装类	187	8.5.5	完成实现	223
7.3.3	装箱和拆箱	189	8.6	小结	226
7.4	链接对象	190	8.7	复习题	227
7.4.1	链接结构里的消息传递, Gondor 灯塔	191	8.8	编程练习	228
7.4.2	链接结构的内部表示法	193	第 9 章	面向对象图形	237
7.5	小结	194	9.1	acm.graphics 模型	238
7.6	复习题	195	9.2	acm.graphics 程序包的结构	238
7.7	编程练习	195	9.2.1	GCanvas 类	239
第 8 章	字符串和字符	199	9.2.2	Color 类的更多细节	241
8.1	枚举的原则	200	9.2.3	GPoint 类、GDimension 类和 GRectangle 类	241
8.1.1	在计算机内部表示枚举类型	200	9.2.4	GMath 类	242
8.1.2	将枚举类型作为整数表示	201	9.2.5	GObject 类及其子类	243
8.1.3	定义新的枚举类型	202	9.3	使用形状类	246
8.2	字符	203	9.3.1	GLabel 类	246
8.2.1	char 数据类型	203			
8.2.2	ASCII 和 Unicode 编码系统	203			

9.3.2 GRect 类及其子类 (GRoundRect 和 G3DRect) 248	10.7.3 BorderLayout 布局管理器 305
9.3.3 GOval 类 249	10.7.4 FlowLayout 布局管理器 307
9.3.4 GLine 类 249	10.7.5 GridLayout 布局管理器 308
9.3.5 GArc 类 250	10.7.6 标准布局管理器的不足 309
9.3.6 GImage 类 253	10.8 使用 TableLayout 类 309
9.3.7 GPolygon 类 257	10.8.1 比较 GridLayout 与 TableLayout 310
9.4 创建复合对象 262	10.8.2 使用 TableLayout 创建 温度转换器 310
9.4.1 简单的 GCompound 示例 263	10.8.3 指定约束 312
9.4.2 GCompound 坐标系 265	10.8.4 使用 TableLayout 创建 简单的计算器 313
9.4.3 使用 GCompound 的对象分解 265	10.9 小结 319
9.4.4 嵌套 GCompound 对象 269	10.10 复习题 320
9.5 小结 270	10.11 编程练习 321
9.6 复习题 271	
9.7 编程练习 272	
第 10 章 事件驱动程序 279	第 11 章 数组与 ArrayList 类 327
10.1 Java 事件模型 280	11.1 数组简介 328
10.2 简单的事件驱动程序 281	11.1.1 数组声明 328
10.3 响应鼠标事件 283	11.1.2 数组选择 329
10.3.1 MouseListener 和 MouseListener 接口 283	11.1.3 简单数组的示例 330
10.3.2 重写侦听器方法 283	11.1.4 改变索引范围 331
10.3.3 画线程序 284	11.1.5 对象的数组 332
10.3.4 在画布上拖动对象 286	11.1.6 在图形程序中使用数组 332
10.4 响应键盘事件 288	11.1.7 ++ 和 -- 运算符的区别 334
10.5 创建简单的 GUI 289	11.2 数组的内部表示法 335
10.6 Swing 交互器层次结构 291	11.3 数组作为参数传递 337
10.6.1 JButton 类 292	11.4 使用数组制作表格 341
10.6.2 JToggleButton 类 293	11.5 数组初始化 343
10.6.3 JCheckBox 类 293	11.6 多维数组 344
10.6.4 JRadioButton 类和 ButtonGroup 类 295	11.6.1 将多维数组传递给方法 345
10.6.5 JSlider 类和 JLabel 类 296	11.6.2 初始化多维数组 346
10.6.6 JComboBox 类 297	11.7 图像处理 346
10.6.7 JTextField 类、IntField 类和 DoubleField 类 300	11.7.1 图像的表示方式 346
10.7 管理组件布局 304	11.7.2 使用 GImage 类操作图像 347
10.7.1 Java 窗口层次结构 304	11.7.3 位操作 348
10.7.2 布局管理器 305	11.7.4 使用位操作分解像素组件 350

11.8 ArrayList 类	354
11.9 小结	358
11.10 复习题	359
11.11 编程练习	360
第 12 章 搜索与排序	369
12.1 搜索	370
12.1.1 在整数数组中搜索	370
12.1.2 搜索表	370
12.1.3 折半搜索	373
12.1.4 搜索算法的相对效率	375
12.2 排序	376
12.2.1 给整数数组排序	376
12.2.2 选项排序算法	377
12.2.3 评估选项排序的效率	379
12.2.4 测量程序的运行时间	380
12.2.5 分析选项排序算法	381
12.2.6 基数排序算法	382
12.3 评估算法效率	384
12.3.1 big-O 表示法	385
12.3.2 big-O 的标准简化	385
12.3.3 选项排序计算的复杂性	385
12.3.4 根据代码结构预测计算 的复杂性	386
12.3.5 最坏情况与一般情况的 复杂性	387
12.4 使用数据文件	388
12.4.1 文件的概念	388
12.4.2 阅读 Java 中的文本文件	389
12.4.3 异常处理	391
12.4.4 倒转文件的程序	393
12.4.5 交互地选择文件	394
12.4.6 使用 Scanner 类	395
12.4.7 输出文件	397
12.5 小结	397
12.6 复习题	398
12.7 编程练习	400
第 13 章 数组与 ArrayList 类	405
13.1 ArrayList 类回顾	406
13.1.1 动态分配的能力	406
13.1.2 区分表示法和行为	408
13.2 HashMap 类	409
13.2.1 映射的简单示例	409
13.2.2 探讨可能的实现策略	410
13.2.3 实现 O(1) 性能	411
13.2.4 散列法思想	412
13.2.5 散列码	412
13.2.6 复制散列码	413
13.2.7 映射类的简单实现	413
13.2.8 实现 hashCode 方法	416
13.2.9 调整存储区的数量	418
13.3 Java 集合架构	418
13.3.1 Java 集合架构的结构	419
13.3.2 Collection 层次结构	420
13.3.3 区分行为和表示法	421
13.3.4 迭代器	422
13.3.5 Arrays 和 Collections 方法库	424
13.4 面向对象设计的原则	426
13.4.1 统一主题的重要性	427
13.4.2 简单和信息隐藏的原则	428
13.4.3 满足客户的需要	428
13.4.4 灵活性的好处	429
13.4.5 稳定的意义	429
13.5 小结	430
13.6 复习题	430
13.7 编程练习	431
第 14 章 展望	435
14.1 递归	436
14.1.1 递归的简单示例	436
14.1.2 Factorial 函数	437
14.1.3 信任的递归式跳跃	440
14.1.4 递归的范例	441
14.1.5 图形递归	442
14.1.6 递归式思考	444
14.2 并发	445
14.2.1 进程与线程	445
14.2.2 并发的简单示例	445
14.3 使用网络	448

14.4 编程模式	448	14.5 小结	453
14.4.1 模型/视图/控制器模式	449	14.6 复习题	454
14.4.2 说明性示例：用图表示电子 数据表的数据	449	14.7 编程练习	454



第 1 章

前 言

[The Analytical Engine offers] a new, a vast, and a powerful language. . . for the purposes of mankind.

—Augusta Ada Byron, Lady Lovelace, 1843



Augusta Ada Byron, Lady Lovelace(1815—1852)

Augusta Ada Byron 是英国著名诗人拜伦(Lord Byron)的女儿。当年有人鼓励她从事自己感兴趣的科学和数学专业，尽管那个时候几乎不允许妇女学习这些科目。17 岁时，Ada 遇到了 Charles Babbage。Charles Babbage 是一位杰出的英国科学家，他一生致力于设计实现数学计算功能的机器——尽管他未能完成这种机器的结构。但 Ada 坚信 Babbage 分析机的潜力，她记录了大量有关其设计的笔记，并编写了一些复杂的数学程序。这些程序使许多人把她当作第一位编程人员。为了纪念她，1980 年美国国防部将该编程语言命名为 Ada。

想想我们在 21 世纪初具有的优势，很难相信 1940 年竟然还没有计算机。而今天它们到处都是，用流行语来说(至少标题作者这么说)：我们生活在计算机时代。

1.1 计算简史

从某种意义上说，计算从古代就有了。许多早期数学都是为了解决现实中重要的计算问题，如监控牧群数量，计算小块土地面积，以及记录商业交易等。这些活动要求人们开发新的计算技术，有些情况下，还需要发明机器帮助计算，如算盘。算盘是一种简单的计算设备，由能够

在杆上滑动的珠子组成，已经在亚洲使用了几千年，很可能在公元前 2000 年就有了。

纵观其整个历史，计算的发展相对缓慢。1623 年，德国科学家 Wilhelm Schickard 发明了第一台为人所知的机械计算机，它能够自动执行简单的算术计算。虽然 Schickard 的机器在“三十年战争(1618—1648)”的破坏中消失于历史，但法国哲学家 Blaise Pascal 在 17 世纪 40 年代使用相同的技术构造了一台机械加法器，其仿制品现在陈列于法国巴黎国立工艺学院 (Conservatoire des Arts et Métiers)。1673 年，德国数学家 Gottfried Leibniz 开发了一种相当精密的设备，它能够进行加减乘除运算。所有这些设备都完全是机械的，没有引擎，也没有其他能量来源。操作员通过将金属轮设置到特殊位置来输入数字，运行过程中，转动这些轮子，可以设置机器的其他部分，改变输出显示。

工业革命时期，技术的快速发展使开发新的机械计算方法成为可能。蒸汽机已经提供了工厂和铁路所需的能量。是否有人能够用蒸汽机来驱动更加精密的计算机器(这些机器能够利用自身动力实现重要计算)呢？在那种情况下，问这种问题合情合理。然而，在取得进展之前，人们不得不提出这一问题，并开始寻找答案。必然的思想火花来自于英国数学家 Charles Babbage。在计算历史中，他是最值得关注的人之一。

Babbage 一生设计了两台不同的计算机器，分别命名为差分机和分析机，每台机器都对当时的计算机器进行了极大改进。他一生的遗憾是这两个项目都未能完成。他设计用来产生数学函数表的差分机，最终由瑞典发明家在 1854 年——提出最初设计 30 年后制造成功。分析机是 Babbage 一生的梦想，直到 1871 年他去世时仍未完成。即便如此，其设计仍然包含了现代计算机中的许多重要特征。更重要的是，Babbage 将分析机构想为一种多用途机器，依靠编程能够执行许多不同功能。在 Babbage 的设计中，分析机的运行由一段小孔控制，这些小孔被打在一张机器能够阅读的卡片上。改变小孔的样式，就可以改变机器的行为，执行不同类型的计算。

Augusta Ada Byron 是诗人拜伦(Lord Byron)与妻子 Annabella 的女儿。我们知道，Babbage 的许多工作都来源于她的著作。和大多数同时代的人一样，Ada 意识到了分析机的潜力，并成为了它的拥护者。她为该机器设计了一些复杂程序，也因此成为了第一位编程人员。20 世纪 70 年代，美国国防部为了纪念她在这方面做出的贡献，将自己的编程语言命名为 Ada。

Babbage 设计的有些方面确实影响了后来的计算历史，例如使用打孔卡来控制进程。法国发明家 Joseph Marie Jacquard 首先将这种思想作为织布机设备(后来被称为 Jacquard 织布机)的一部分，使织布过程自动化。1890 年，Herman Hollerith 使用打孔卡片使美国人口普查数据表格统计实现了自动化。为了让这种技术市场化，Hollerith 建立了一个公司，这就是后来的 IBM(国际商用机器)公司，它在 20 世纪大部分时间占据着计算机产业的主导地位。

Babbage 版的可编程计算机直到 20 世纪 40 年代才变为现实，电子学的出现使超越机械设备成为可能，之前都是机械设备在计算领域占主导地位。1939 年末，爱荷华州州立学院的 John Atanasoff 和他的学生 Clifford Barry 组装了第一台电子计算机原型。1942 年 5 月，他们完成了包含 300 个电子管的具有全面执行能力的机器。这台计算机能够解决线性方程这类小的系统问题，Atanasoff-Barry 计算机在设计方面做了一些调整，它可以执行更复杂的计算，但是这一项的工作被第二次世界大战中断了。

第一台大规模电子计算机是电子数字积分计算机(Electronic Numerical Integrator and Computer，简称 ENIAC)。1946 年，在宾夕法尼亚大学摩尔学校 J.Presper Eckert 和 John Mauchly 指导下完成的 ENIAC，包含了 18 000 多个电子管，占满了一个 30ft 长、50ft 宽的房间。ENIAC 通过将电线插入被称为配线架(patch panel)的小钉板状设备里进行编程。通过将配线架上不同

的插孔与电线连接起来，操作员可以控制 ENIAC 的行为。这种类型的编程需要具备机器内部运行的基本知识。事实证明，这种编程方法比 ENIAC 发明者想象的要困难得多。

也许现代计算技术最重大的突破发生于 1946 年。这一年，普林斯顿高级研究中心的 John von Neumann(冯·诺伊曼)提出可以用相同的方法表示程序和数据，并且这些程序和数据可以存储在相同的内存里。这一概念极大地简化了编程过程，几乎是所有现代计算机的基础。由于这方面的设计，据说现代计算机采用的都是冯·诺伊曼体系结构。

随着 ENIAC 的完成和冯·诺伊曼体系存储编程概念的发展，计算技术也快速发展起来。新系统、新概念被引入如此快速发展的系列产品中，将它们全部罗列出来毫无意义。根据基本技术，大多数历史学家将现代计算机的发展阶段分为如下 4 代：

- 第一代 第一代电子计算机使用电子管作为其内部电路的基础。这一计算时期从 1939 年 Atanasoff-Barry 原型开始。
- 第二代 1947 年晶体管的发明将计算机带入了新一代。晶体管是半导体设备，它的功能与电子管一样，但它比电子管小，需要少量电子能量。第一台使用晶体管的计算机是 1958 年推出的 IBM 7090。
- 第三代 虽然晶体管比电子管小，但是包含 100 000 或 1 000 000 个单个晶体管的计算机仍需要占用很大的空间。第三代计算机随着 1959 年集成电路(芯片)的发展而出现。集成电路(芯片)是一块小的硅晶片，它通过图像压印，包含了许多连接在一起的晶体管。第一台在结构上使用集成电路的计算机是 1964 年出现的 IBM 360。
- 第四代 第四代计算开始于 1975 年，将计算机的整个处理单元集成到单个硅芯片上。那一年，构建集成电路的技术使这一想法成为可能。该制作技术被称为大规模集成。由单个芯片组成的计算机处理器称为微处理器，今天大多数计算机都在使用这种处理器。

从历史的观点上说，第一代和第二代的早期计算机与现代计算机的前身同样重要，只是现在没有使用它们而已。它们是计算机科学中的恐龙：巨大、笨拙、智能低下，很快就灭绝。集成电路的发明者之一，Intel 公司的创始人 Robert Noyce 评论说，与 ENIAC 相比，典型的现代计算机“速度快 20 倍，内存大，可靠性强上千倍，消耗的能量相当于一个电灯泡，而不是一辆机车，体积是它的 1/30 000，成本是它的 1/10 000。”当然，计算机还有未来时代。

1.2 计算机科学的含义

成长于现代世界，这可能会让你明白计算机的含义。然而，本文更多关注的是计算机科学，而不是作为物理设备的计算机。

乍一看，计算机和科学好像是不可调和的一对词语。按其典型用法，科学是指对自然现象的研究：当人们说到生物科学或物理学时，我们可以理解，并觉得这种用法很恰当。计算机科学好像和这不一样。计算机是人造物品，这一点让我们勉强将计算机研究归类为科学。毕竟，现代技术也可以生产汽车，但我们没有讨论“汽车科学”。相反，我们说“汽车工程”或“汽车技术”。计算机为什么就不同呢？

要回答这个问题，关键是认识到计算机本身只是原因的一部分。今天可以在本地计算机商店购买到的物理机器只是计算机硬件的一个示例。它是有形的。您可以拿起它，把它带回家，放在桌子上。如果需要，可以把它作为制门器使用，尽管有点贵。但是，如果除了硬件之外什

么也没有，如果您拿到的机器和它下线时完全一样，那么作为制器可能是它能做的几件事之一。现代计算机是多用途机器，具有执行广泛任务的潜力。然而，要实现其潜力，必须对计算机进行编程。编程计算机的行为就是给它提供一组指令(程序)，这些指令指定解决问题所需的所有步骤。这些程序通常称为软件，只有软件和硬件相结合才可能进行计算。

相对于硬件而言，软件是抽象的、无形的实体。它是一系列简单的步骤和操作，表述为一种硬件可以解释的明确语言。讨论计算机科学时，主要关注计算机软件领域，更重要的是，关注更加抽象的问题解决域。问题解决是具有高度挑战性的活动，它需要创造性、技能和训练。在很大程度上，计算机科学被认为是解决问题的科学，其解决方案恰好与计算机有关。

这不是说计算机本身不重要。没有计算机时，人们只能解决相对简单的计算问题。最近 50 多年以来，计算机的存在使人们以一种及时高效的方式，解决日益困难和复杂的问题变成了可能。而且，随着基础技术的提高，解决更复杂的问题也是可能的。然而，由于要解决的问题越来越复杂，寻找有效解决方案策略的任务也越来越复杂。这样就要求解决问题的科学要随着计算技术的发展而发展。

1.3 计算机硬件简介

计算机由软件控制，也可以为每个单独的应用程序重新设计软件，计算机解决复杂问题的灵活性即来源于这一实际。考虑到软件体现了问题解决策略——这些策略是计算机科学的本质，因此本书专门讨论了计算软件方面的问题。虽然如此，本章花些时间简要讨论一下计算机硬件结构也很有必要。理由很简单：编程是一门在实践中学习的学科。只阅读本书，即使解决了纸上的所有练习，也不会成为编程人员。编程是一个需要动手的工作，这项工作需要与计算机配合。

要使用计算机，就要了解其硬件。要知道如何开机，如何使用键盘输入程序，如何执行已经写好的程序。很遗憾，各计算机系统执行这些操作所必须遵循的步骤差别很大。作为编写普通教科书的人，我无法给出特定系统的工作原理，相反，必须着重讨论通用规则，这些规则在用户可能使用的任何一台计算机上都是通用的。阅读这一部分的时候，用户应该看着自己的计算机，看看一般性讨论如何应用于自己的计算机。

今天使用的大多数计算机都由如图 1-1 所示的组件组成。被称为总线的通信信道将图中每个组件连接起来，它允许数据在各独立的单元间流动。单个组件在下面几节介绍。

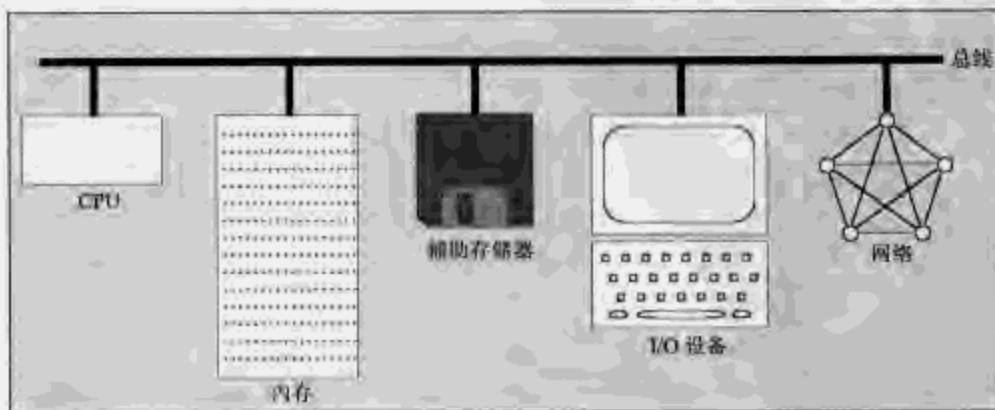


图 1-1 典型的计算机组件

1.3.1 CPU

CPU(中央处理器)是计算机的“大脑”。它执行实际计算，协调整个计算机的活动。程序决定CPU的行为，这些程序由存储在内存系统里的一系列编码指令组成。例如，一条指令可能命令计算机相加两个数字。另一条指令可能让一个字符出现在计算机屏幕上。通过执行适当序列的简单指令，CPU可以执行复杂任务。

现代计算机中，CPU由集成电路——一个微型硅芯片组成，该芯片压印集成了数百万个连接形成大型电路的晶体管，这些电路可以实现简单的算术运算和逻辑运算。

1.3.2 内存

计算机执行程序时，必须能够存储程序本身和计算所涉及到的数据。总的来说，任何能够存储和恢复信息的计算机硬件都是存储设备。程序主动运行时使用的存储设备构成其主存储器，通常称为内存。自从1946年冯·诺伊曼首次提出这一思想以来，计算机就使用同样的内存来存储组成程序的单个指令和计算中所使用的数据。

内存系统设计得非常有效，因此它们可以让CPU非常迅速地访问其内容。今天的计算机中，内存通常被扩建为一个特殊的集成电路芯片，该芯片称为随机存取存储器，简称RAM。随机存取存储器允许程序在任何时候使用任何存储单元的内容。

1.3.3 辅助存储器

程序运行时，计算机通常将活动数据保存在内存里，但是大多数主存设备存在不足，它们只是在计算机开机时才发挥作用。关闭计算机后，所有存储在主存里的信息都会丢失。要存储永久数据，需要使用一种存储设备，这种设备不需要电源就可以保存信息。这种设备就是辅助存储器。

今天的计算机使用了多种辅助存储设备。几乎每台买到的计算机都有硬盘，即使关闭计算机，硬盘也可以存储大量已有信息。创作或编辑程序通常在硬盘上进行。如果要将程序转移到另一台计算机上或为了保存而进行备份，可以将程序复制到可移动媒介——软盘、可写CD或内存条。

1.3.4 输入/输出(I/O)设备

对于有用的计算机而言，它必须有某种方式可以与外界用户进行交流。计算机输入通常由键盘上输入的字符组成，计算机输出通常显示在计算机显示屏上或打印机上。总的来说，执行输入和输出操作的硬件设备称为I/O设备。

不同计算机的I/O设备差异很大。除了标准的按字母顺序排列的键之外，计算机键盘的排列也各不相同，甚至是一些重要的键也使用不同的名称。例如，用于表示一行结束的键，有的键盘标记的是Return，有的则是Enter。有的计算机系统中，使用键盘上方或旁边的特殊功能键(它们提供简单编辑操作)就可以改变程序。而有些系统中，使用被称为鼠标来选择要改变的程序文本，也可以完成相同的任务。无论哪种情况，计算机都清楚当前输入位置，通常在屏幕上用闪烁的线或矩形(指针)来指示。

1.3.5 网络

图1-1中最后的组件是标记为网络的星形符号，表示到其他计算机群的连接，那些计算机连接在一起作为Internet的一部分。Internet是全球计算机的大集合，里面的计算机通过允许它们共享数据和程序的通信线路连接在一起。很多情况下，网络只是I/O结构的一部分。现代计

计算机的输入输出操作不仅仅局限于本地机器，而是扩展到了由 Internet 连接起来的广阔世界。记住这一点很有帮助。学习 Java 时，网络特别重要。因为 Java 作为一种编程语言，其成功之处就是紧随网络的发展，这一点将在本章稍后讨论。

1.4 算法

前面已经介绍了计算机系统的结构，下面继续讨论计算机科学。因为计算机科学是利用计算机解决问题的学科，所以要理解一个概念，它是计算机科学和问题解决抽象学科的基础，这个概念就是算法。“算法”一词来自于 19 世纪波斯数学家 Abu Ja'far Mohammed ibn Mūsā al-Khowārizmī 的名字，他在一本名为《*Kitab al-jabr w'al-muqabala*》的杂志上发表了一篇数学论文，其标题产生了英语单词“algebra”（代数学）。广义上说，可以将算法理解为解决问题的策略。然而，要理解计算机科学家如何使用这个术语，需要规范其直观理解，严格其定义。

要成为算法，解决方案方法必须符合 3 条基本要求。第一，算法必须以明确形式表示，这种形式可以让读者确切明白包含了哪些步骤。第二，算法内的步骤必须有效，即这些步骤有可能在实践中实现。例如，包含操作“用 r 乘以 it 的准确值”的方法就无效，因为不可能计算出 it 的准确值。第三，算法不能永远运行，必须在有限时间内显示答案。总之，算法必须：

- 有明确无误的定义。
- 有效，即其步骤必须可执行。
- 有限，即经过有限步运行之后可以终止。

稍后开始使用复杂算法时，这些属性就显得特别重要。这里，只要认为算法是抽象解决方案策略即可——这些策略最终会成为您所编写程序的核心。

很快您就会发现，算法——就像要解决的问题一样——变化非常复杂。有些问题很简单，大脑中马上就会闪现合适的算法，于是可以编写程序解决这样的问题，不会有太多麻烦。然而，随着问题变得越来越复杂，想出能合适地解决这些问题的算法就需要更多的思考。大多数情况下，几种不同的算法可以用来解决特殊问题。写出最终程序之前，需要考虑多种潜在解决方案。

1.5 编程过程的几个阶段

用计算机解决问题从概念上可以分为明显的两个步骤第一，需要开发一种解决问题的算法，或选择一种已有的算法。这一部分过程称为算法设计。第二步就是用编程语言将这种算法表述为计算机程序。这一过程称为编码。

随着编程的开始，编码过程——将算法转换为功能程序——似乎是该过程中更难的阶段。作为刚刚入门的编程人员，可以从具有简单解决方案的简单问题入手，算法设计阶段似乎不特别具有挑战性。然而，因为不熟悉语言及其规则，所以有时编码好像很难、很随意。我希望可以安心地说，学习更多编程过程之后，编码很快就会变得很容易。同时，随着请求解决的问题的复杂性不断增加，算法设计也会变得更难。

本书中介绍的新算法，最初通常用英语表示。虽然英语常常不够准确，但只要是在两个完全讲英语的人之间交流，它也是一种表达解决方案策略的合理语言。很明显，如果要给只会讲俄语的人表述自己的算法，英语就不是一个合适的选择。同样，要给计算机表达算法，英语也不是合适的选择。虽然计算机科学家研究这一问题已几十年，但让计算机理解英语、俄语或其他

他人类语言仍然超出了当前技术范围。要让计算机能够解释算法，需要将算法转换为编程语言。有许多编程语言，如 Fortran、BASIC、Pascal、Lisp、C 和 C++ 等。本书将学习如何使用编程语言 Java，它由 Sun Microsystems 于 1995 年开发，并已成为工业和计算机科学课程的标准。

1.5.1 创建和编辑程序

在大多数计算机系统上运行程序之前，需要输入程序文本并将它存放在文件里。文件是一个总名称，它集合了所有存储在计算机辅助存储器里的信息。每个文件必须有一个名称。名称通常分为两个部分，中间用句点隔开，如 **MyProgram.java**。创建文件时，需要决定根名称(名称里句号之前的部分)来说明文件所包含的内容。文件名中句点之后的部分表示文件的种类，称为扩展名。有些扩展名有预定的含义，例如，扩展名为 **java** 表示用 Java 语言所写的程序文件。包含程序文本的文件称为源文件。

通常，输入或改变文件内容的过程称为编辑文件。因为不同的计算机系统编辑过程大不相同，因此不可能用一种适用于所有硬件类型的方法描述该过程。在特定计算机系统上工作，需要学习创建新文件和编辑已有文件的方法。可以在与所使用的编程环境有关的手册或在线文档上找到这些信息。

1.5.2 编译过程

创建源文件以后，下一步就是将程序转换为计算机可以理解的形式。Java、C 和 C++ 都是计算机科学家所说的高级语言。设计这些语言是为了让编程人员更方便地表达算法，而不需要他们深入理解基本软件究竟如何执行这些算法。一般来说，高级语言具有独立的区分单个计算机结构的特定特性。然而，每个计算机系统内部只能理解专用于那一类硬件的低级语言，这种语言称为机器语言。例如，Apple Macintosh 和 Windows 计算机使用不同的内部机器语言，即使它们都能执行用高级语言编写的程序。

要让用高级语言编写的程序可以在不同的计算机系统上运行，有两种基本策略。典型方法是使用被称为编译器的程序，将所写的程序转换为适合计算机运行的低级机器语言。使用这种方法，具有不同硬件结构的计算机就需要不同的转换器。例如，如果为 Macintosh 编写了 C 语言程序，就需要一个编译器将 C 语言程序转换为 Macintosh 的机器语言程序。如果要在 Window 计算机上运行相同的程序，就需要一个不同的编译器，因为基本硬件使用的是不同的机器语言。

第二种方法是将程序转换为独立于基本结构的中间语言。对于所有特定的机器结构，这些中间语言程序的运行都由解释程序执行，由解释程序在计算机上执行中间语言。例如，如果中间语言程序包含将两个数字相加的指令，解释程序通过执行在基本硬件上执行加法所必需的所有指令来完成这项任务。然而，相对于编译器而言，解释程序实际没有产生机器语言指令来执行原始程序指定的特定加法运算。相反，解释程序通过执行有相同效果的指令模仿了加法运算。

现代 Java 系统使用混合策略，结合了编译器和解释程序的功能。在最初编译阶段，Java 将程序翻译为普通的独立于基本硬件的中间语言。然后，一种称为 Java 虚拟机(简称 JVM)的程序会解释中间语言，并执行机器的中间语言。运行 Java 虚拟机的程序通常将中间代码块编译为内部机器语言。因此，Java 通常可以达到传统解释程序不能达到的效果。

在典型的编译器系统中，编译器将您所写的源文件翻译到包含该计算机系统相关指令的第二个文件(称为目标文件)中。然后，该目标文件与其他目标文件相结合，通常包括被称为库的预定义目标文件。该目标文件包含用于不同普通操作的机器语言指令。这些组合文件共同生成一个可以在系统上运行的可执行文件。将所有单个目标文件组合为一个可执行文件的过程称为

链接。图 1-2 显示的是整个编译过程。

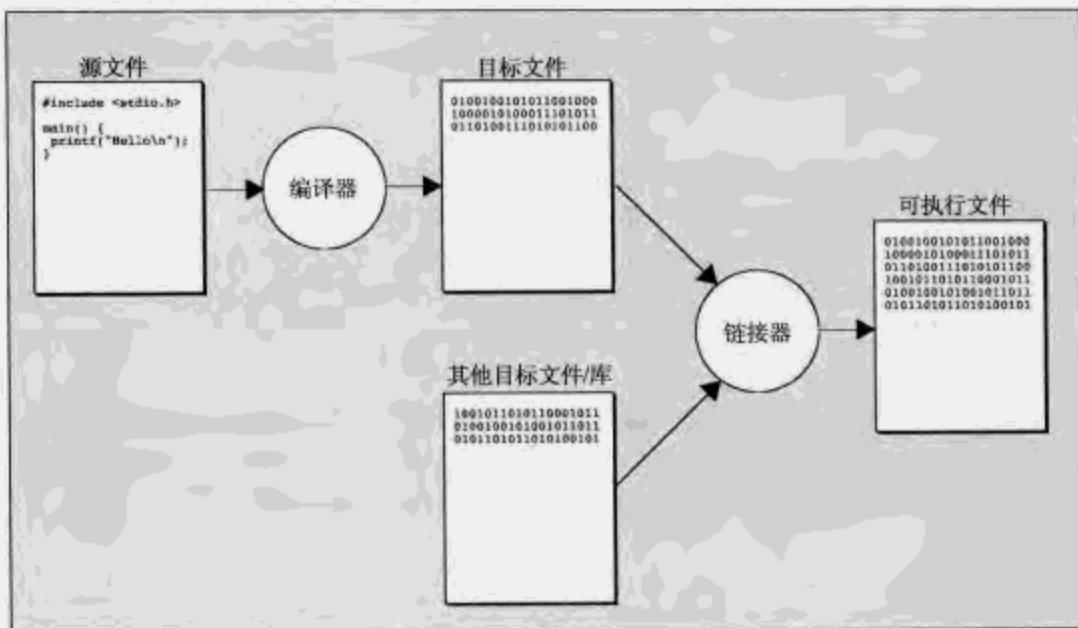


图 1-2 典型编译过程的几个阶段

在 Java 中，过程稍微详细一些。如前所述，Java 产生中间代码，这些代码存储在被称为类文件的文件里。这些类文件与其他类文件和库结合，生成中间程序的完整版本。该程序版本的常见格式是单独文件经过压缩的集合，称为 JAR 存档文件。然后通过 Java 虚拟机来解释，并在计算机上显示输出，过程如图 1-3 所示。

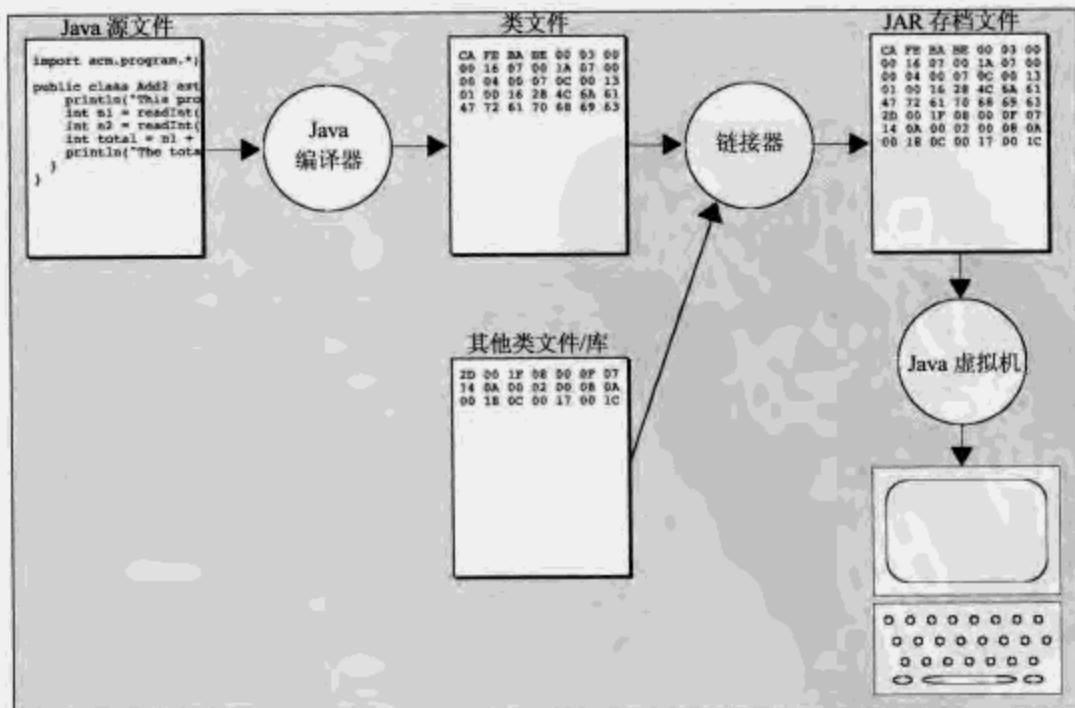


图 1-3 运行 Java 程序的阶段

1.5.3 编程错误与调试

除了翻译功能之外，编译器还可以执行另一项重要功能。和人类语言一样，编程语言有它自己的词汇和语法。语法是一套规则，它决定不同语言部分如何结合在一起。这些语法规则可以判定哪些语句结构比较好，哪些不好。例如，英语中说“we goes”就不对，因为主语和动词之间数的形式不一致。每种编程语言都有自己的语法，它决定该语言中程序的各个元素如何组织在一起。

编译程序时，编译器首先会检查程序的语法是否正确。如果违反了语法规则，编译器会显示一条错误信息。由于违反这些语法规则而产生的错误称为语法错误。只要收到编译器指示语法错误的信息，都必须重新编辑程序来更正错误。

虽然语法错误令人沮丧，尤其是对新的编程人员而言，但它们不是沮丧的最大来源。通常，如果编写的程序不能正确运行，往往不是语法问题，而是逻辑问题。许多情况下，完美的逻辑程序不知何故就会出现不正确的答案或根本就不能产生答案。检查程序，发现在逻辑方面出现了错误——编程人员将这种类型的错误称为故障(bug)。查找和改正这种错误的过程称为调试，它是编程的重要组成部分。

故障具有很强的隐蔽性，这很容易让人灰心。您绝对相信算法是正确的，但会后来发现它不能执行一些前面忽略的案例。或者，您知道程序的某个部分需要进一步修改，只是后来忘了。或者，您可能犯了一个似乎很愚蠢的错误，以至于您不相信有人会犯下如此严重的错误。

不要紧张。您在一个优秀的公司之中，甚至最优秀的编程人员也有此经历。实际上，所有编程人员都犯过逻辑错误，您也一样。算法需要技巧，通常您会发现并没有好好使用它。

从某种程度上说，发现自己易错的缺点是每一个编程人员必经的重要过程。20世纪60年代，计算机科学家前辈 Maurice Wilkes 描述其经历时写道：

不知何故，在摩尔学校及毕业之后，有人总认为编程过程中没有特别的困难。我总记得那个时刻，它让我渐渐明白，我未来生活中很大一部分时间都要花在查找自己程序里的错误上。

优秀编程人员与普通编程人员之间的区别不是他们可以完全避免故障，而是他们努力将已完成代码里的故障数量减少到最小。设计好一个算法并将它转换为语法正确的程序后，工作并没有完成，明白这一点至关重要。基本上，程序里总会有个别故障。编程人员的任务就是找出故障并修复它。完成一个之后，应该继续找出下一个故障并修复它。要对自己的程序总是有所怀疑，要尽可能彻底地测试它们。

1.5.4 软件维护

软件开发最令人惊讶的问题之一是程序需要维护。实际上，软件开发研究显示，对于大多数程序而言，软件发行之后，因为维护软件而给编程人员支付的费用占总成本的 80%~90%。然而，在软件方面，要准确理解维护是什么意思有点困难。乍一听，这个概念有点古怪。如果用在小轿车或大桥上，什么东西坏了就需要维护——金属锈蚀、某块机械连接因过度使用而报废，或者什么东西因事故而被打碎等。这些情况都不适用于软件。代码本身不会锈蚀。一遍又一遍地重复使用相同的程序丝毫不降低其功能。偶然误用的确会产生不良后果，但不会损害程序本身；即使损害了程序本身，它也可以从备份副本中恢复。在那种环境中，维护是什么意思呢？

两个主要原因使软件需要维护。第一，即使经过大量测试，有时甚至实际使用多年，原始代码中也仍然存在故障。因此，当出现始料未及的情况时，以前隐匿的故障就会导致程序失效。所以，调试是程序维护的重要部分。但它不是最重要的部分。更重要的部分，特别是按照它在程序维护总成本贡献方面的大小来说，是功能强化。编写程序就是为了使用；它们通常比其他方法更快、成本更低地执行客户需要完成的任务。同时，程序很可能不会执行客户的任何要求。程序使用一段时间之后，客户会想，如果程序能够做其他事，或者以不同的方式做，或者以一种更有用的方式显示数据，或者运行得更快一点，或者能扩展容量，或者有一些更简单但更吸引人的功能(贸易中通常称为额外特征)，那就太好了。由于软件具有极大的灵活性，因此供应商有办法解决这些问题。无论哪种情况——修复故障或添加功能——必须有人参与进来，查看程序，弄明白发生了什么，做一些必需的修改，检验这些修改是否可行，然后发行新版本。这个过程很艰难，很消耗时间，代价很高，也会出错。

程序维护十分困难，一部分原因是许多编程人员编写程序时没有长远打算。对他们而言，程序能够运行似乎就足够了，然后他们就转而做其他的事情。编写程序以便其他人可以理解、维护这些程序的学科称为软件工程。我们鼓励编写表现良好工程风格的程序。

编写程序时，想象一下，两年之后其他人再看到它们时会有何感想。您的程序还有意义吗？程序会告诉新读者您要做什么吗？改变它容易吗？它看起来是不是晦涩难懂？如果设身处地为未来维护人员(许多公司新的编程人员，您很可能就是那个角色)想一下，这会让你明白为什么好的风格至关重要。

许多编程新手很迷惑，因为没有一套准确规则可以遵循，从而确保好的编程风格。好的软件工程不是过程食谱。相反，它是一门技术，还带有一点艺术性质。实践最重要。一个人要想写出好的程序，就要动手写，并且要阅读其他人的程序，就像一个人要成为一个小说家一样。好的编程需要有针对未来维护人员的训练——这些训练不能抄近道或被遗忘，不能匆忙完成。好的编程风格需要开发一种审美感——意思就是程序应该可读，表示正确。

1.6 Java 和面向对象范例

正如本章前面所述，本文使用 Java 编程语言来阐明编程和计算机科学的一般概念。为什么要用 Java 呢？主要原因是 Java 鼓励编程人员思考编程过程。

过去十多年，计算机科学和编程都经历了某种革命。和大多数革命一样——要么是政治剧变要么是概念重构，正如 1962 年 Thomas Kuhn 在他的书 *The Structure of Scientific Revolutions* 里所描写的那样——一种思想的出现推动了这种改变，这种思想对现有的理论构架(或范例)提出了挑战。起初，两种观点互不相让。曾经有段时间，旧秩序占据着主导地位。但是随着时间的推移，新思想的力量和声望不断增长，直至取代了旧秩序，Kuhn 称之为范例转换。今天的编程世界中，旧秩序以过程范例为代表，在过程范例中，程序由在数据上运行的过程和功能的集合组成。挑战者是面向对象范例。在面向对象范例中，程序被视为“对象”的集合，数据和相关操作被封装在综合单元里。许多传统语言，包括 Fortran、Pascal 和 C，体现的都是过程范例。面向对象范例最著名的代表是 Smalltalk、C++ 和 Java。

虽然面向对象语言以牺牲过程语言为代价得到了普及，但是认为面向对象和过程范例互相排斥则是错误的。不同编程范例之间与其说是互相竞争的，不如说是互补的。面向对象的范例

和过程范例及其他重要范例，例如嵌入 LISP 和 Scheme 中的功能性编程风格，实际中都有很多重要的应用程序。即使在单个应用程序的情况下，也很可能发现使用了多个方法。作为编程人员，必须掌握许多不同的范例，以便使用最适合于当前任务的概念模型。在本书中，可以在面向对象的范例方面打下坚实的基础，同时也会学到过程范例的一些知识。

1.6.1 面向对象编程的历史

面向对象编程的思想并不是很新奇。第一种面向对象的语言是 SIMULA，它是一种编码模拟语言，由斯堪的纳维亚(北欧一地区)计算机科学家 Ole-Johan Dahl、Björn Myhrhaug 和 Kristen Nygaard 于 20 世纪 60 年代设计。由于其远远领先于时代的设计，SIMULA 首先提出了许多概念，这些概念后来成为编程中的通用概念，包括抽象数据的概念和许多现代面向对象范例。实际上，用来描述面向对象系统的大多数术语都来源于 SIMULA 最初版及后续版 SIMULA 67 的原始报告。

然而，许多年来，SIMULA 大部分时间都被束之高阁。很少有人注意到它，也许能听到它的唯一地方就是在编程语言设计课上。计算界公认的第一种面向对象语言是 Smalltalk，由 Xerox 帕洛阿尔托研究中心(通常称为 Xerox PARC)于 20 世纪 70 年代后期开发。Smalltalk 的目的，Adele Goldberg 和 David Robson 所著的 *Smalltalk-80: The Language and Its Implementation* 一书中说，就是让广大读者可以理解编程。因此，Smalltalk 是 Xerox PARC 成果的一部分，这些成果催生了许多现代用户界面技术的开发，这些技术现在是个人计算机的标准。

尽管 Smalltalk 有许多吸引人的特征和简化编程过程的高度交互式用户环境，但它从未取得广泛的商业成就。业界总体上对面向对象编程感兴趣，在其中心思想与编程语言 C 的变量合并之后，C 语言已经成为了行业标准。尽管在设计基于 C 语言的面向对象语言方面有一些类似努力，但最成功的是 C++，它是 20 世纪 80 年代初由 Bjarne Stroustrup 在 AT&T 贝尔实验室设计的。通过让面向对象技术与现有的 C 代码结合，C++ 让许多编程人员能够以一种渐进的、革命性的方式采用面向对象的范例。

1.6.2 Java 编程语言

面向对象编程历史中的最近一项是 Sun 微系统公司 James Gosling 领导的编程人员团队对 Java 的开发。1991 年，Sun 公司开始这个项目(它最终创造了 Java)时，其目标是设计一种能够适用于编写嵌入客户电子设备微处理器的语言。如果项目专注于这个目标，Java 就不可能发展到现在这个程度。计算案例通常这样，为了适应不断变化的产业环境，Java 项目的方向也在其开发阶段改变了。导致改变的主要原因是 20 世纪 90 年代早期出现的 Internet 的显著增长，特别是 World Wide Web(简称 WWW)的出现。WWW 是全球计算机用户提供的互连资源的空前集合。1993 年，当人们对 Web 的兴趣空前高涨时，Sun 公司将 Java 重新设计为一个可以编写高度交互式、基于 Web 应用程序的工具。这个决定被证明正合时宜。自从 1995 年 5 月发布 Java 以来，它在理论计算领域和商业计算领域都产生了强烈影响。在此过程中，面向对象编程已经被牢牢确立为计算产业的中心范例。

要感受 Java 的力量，看一看图 1-4 就会有所了解，图 1-4 是一篇关于原始 Java 设计的优秀论文的摘要，这篇论文是 James Gosling 和 Henry McGilton 在 1996 年写的。论文中，作者用一长串形容词来描述 Java：简单的、面向对象的、熟悉的、充满活力的、安全的、结构中立的、可移植的、高性能的、解释的、线程的和动态的。图 1-4 的讨论会让您明白这些专门用语的意思。随着对 Java 和计算机科学的不断学习，就更能理解这些特征的重要性。

JAVA™编程语言的设计目标

计算环境(必须部署软件)的本质推动着 Java™编程语言的设计需求。

Internet 和 WWW 的飞速增长让我们以全新的方式看待软件开发和发布。要在电子商务和分布式的世界上生存, Java 技术就必须在异构、分布式网络中的多平台上确保开发出安全、高性能和颇具活力的应用程序。

在异构网络里多平台上的操作, 使二进制分布、发行、升级、补丁等传统配置失效。要在这种复杂的环境中生存下来, Java 编程语言必须是结构中性的、可移植的和动态适应的。

满足这些需要的系统很简单, 因此许多开发人员可以很容易地对它进行编程; 熟悉的, 以致现在的开发人员很容易就能学会 Java 编程语言: 面向对象的, 要利用现代软件开发方法论, 适应分布式客户端—服务器端应用程序; 多线程的, 为了应用程序的高性能, 这些应用程序需要执行多个并发活动, 如多媒体; 解释的, 为了最大方便和动态性能。

上面的要求是一些专门用语的集合, 因此在继续讨论之前, 我们研究其中一些术语及各自的优点。

简单的、面向对象的和熟悉的

Java 编程语言的基本特征之一就是简单, 编程人员不需要大量的训练就可以进行编程, 并与目前软件实践相协调。很快就可以掌握 Java 技术的基本概念; 编程人员从一开始就会高产。

Java 编程语言完全是面向对象的。对象技术在经过 30 年的酝酿之后最终找到了通向编程主流的途径。分布式、基于系统的客户端—服务器端需求符合基于对象软件的封装、消息传递范例。要在日益复杂的、基于网络的环境里发挥作用, 编程系统必须采用面向对象的概念。Java 技术提供了一个整洁有效的面向对象的开发平台。

使用 Java 编程语言的编程人员可以访问经过测试的对象的现有库, 这些库可以提供多种功能性, 范围可以从通过 I/O 和网络界面获得的基本数据类型一直到图形用户界面工具箱。可以扩展这些库来提供新的行为。

即使 C++ 作为一种执行语言被拒绝, 让 Java 编程语言尽可能看起来和 C++一样, 也会使它成为一种熟悉的语言。同时删除了 C++ 不必要的复杂性。让 Java 编程语言保留面向对象的许多特征, “形式和感觉”都像 C++, 这样编程人员就可以很方便地转移到 Java 平台, 很快就会变得多产。

健壮的和安全的

设计 Java 编程语言就是为了创建高度可靠的软件。它提供广泛的编译时检查, 然后是二级运行时检查。语言特征指导编程人员养成可靠的编程习惯。

内存管理模型也非常简单: 用新的运算符创建对象。没有明显的编程人员定义的指针数据类型, 没有指针算法和自动无用单元收集。这种简单的内存管理模型减少了困扰 C 和 C++ 编程人员的编程错误的整个类。可以开发 Java 代码, 在产品代码装运之前, 相信系统很快就会发现许多错误, 相信主要问题不是隐蔽的。

图 1-4 Java 开发人员在语言设计上的洞见力

Java 技术在分布式环境中运行，这意味着安全性极为重要。Java 技术的安全特征被设计到语言和运行系统之中，它让您可以构造从外部无法入侵的应用程序。网络环境中，用 Java 编程语言写的应用程序是安全的，可以防止入侵，这些入侵来自未授权代码企图到达幕后创建病毒或入侵文件系统。

体系结构中立的和可移植的

Java 技术支持应用程序。这些应用程序会被部署到异构网络环境。在那种环境中，应用程序必须能够在不同硬件结构上执行。在这些硬件平台上，应用程序必须在多种操作系统上执行，并能够与多编程语言界面互操作。为了适应操作系统的多样性，Java CompilerTM 产品生成了字节码——一种结构中立的中间格式，可以将代码有效传递到多种硬件和软件平台。Java 技术解释的特征解决了二进制分布问题和版本问题：相同的 Java 编程语言位元码可以在任何平台上运行。

结构中立性只是真正便携式系统的一部分。Java 技术通过严格基本的语言定义，让便携性又进了一步。Java 技术奠定了基础，指定了其基本数据类型的大小，以及算子的行为。程序在每个平台上都是相同的——在所有硬件和软件结构中所有数据类型都兼容。

Java 技术的结构中立性和便携式语言平台被称为 Java 虚拟机。它是抽象机器的规范，Java 编程语言编译器可以为它生成代码。Java 虚拟机为专用硬件和软件平台的特殊执行提供了虚拟机的具体实现。Java 虚拟机主要基于 POSIX 界面规范——便携式系统界面的行业标准定义。在新结构上执行 Java 虚拟机是一项相对简单的任务，只要目标平台满足基本要求就行，如支持多线程。

高性能的

性能总是一个需要考虑的因素。Java 平台通过采用一种配置取得了较高性能，通过这种配置，解释程序可以全速运行，而不需要检查运行时的环境。自动无用单元收集器作为一种低优先权的后台线程运行，确保内存存在需要时可用的可能性提高，这样会产生更好的性能。可以设计需要大量运算能力的应用程序，这样可以按要求用本地机器代码编写计算密集的部分，并通过界面与 Java 平台连接。通常，用户认为交互式应用程序反应迅速，即使它们需要进行解释。

解释的、线程的和动态的

Java 解释程序可以直接在任何计算机上执行 Java 字节码，这些计算机通过端口连接解释程序和运行时系统。在解释平台(如 Java 技术系统)中，程序的链接阶段是简单的、递增的和轻量级的。受益于更加快速的开发周期——原型、实验，相对于传统重型编译、链接和测试周期而言，快速开发是正常情况。

现代基于网络的应用程序，如适用于 WWW 的 HotJavaTM Browser，通常需要同时做几件事。使用 HotJava Browser 的用户可以同时运行几部动画片，同时可以下载图像、滚动页面。Java 技术的多线程功能提供创建包含活动的多个同时线程应用程序的方法。多线程为终端用户提供了高度的交互性。

Java 平台支持语言级的多线程和精确的同步原语：语言库提供线程类，运行时系统提供监控器和条件锁定原语。而且，在库级别，Java 技术的高级系统库线程是安全的，库提供的功能是可用的，不会与执行的多个并发线程相冲突。

图 1-4 (续)

Java 编译程序在其编译时的静态检查是严格的，在链接阶段，语言和运行时系统是动态的。只有需要时才会链接类。按照不同资源甚至是网络资源的要求，可以将新代码模块链接进来。在 HotJava 浏览器和类似应用程序中，可以从任何地方加载交互式可执行代码，这样就可以进行应用程序的透明更新。结果促进了在线服务的持续发展：它们可以保持创新和新鲜，吸引更多客户；刺激 Internet 上电子商务的增长。

——摘自 James Gosling, Henry McGilton.

White Paper: The Java Language Environment

图 1-4 (续)

1.7 Java 和 WWW

从某种程序上说，Java 作为语言的最初成功与 20 世纪 90 年代初计算机网络的刺激密不可分。从 1969 年 ARPANET(当今 Internet 的前身)最初的 4 个节点连成一线起，计算机网络到那时已经存在了 20 年。整个 20 世纪 90 年代，推动 Internet 技术空前繁荣的，与其说是网络本身，还不如说是 WWW 的发明，通过点击交互式链接，它允许用户从一个文档移动到另一个文档。

包含交互式链接的文档称为超文本——这一术语是 1965 年由 Ted Nelson 创造的，他计划创建一个综合的文档集合，这个集合与今天的 WWW 十分类似。然而，基本概念还要早些，第一位总统科学顾问(Presidential Science Advisor)Vannevar Bush 在 1945 年提出了相同的思想。然而，这种分布式超文本系统的思想直到 1989 年才成功实行。当时位于日内瓦的欧洲粒子物理研究所(CERN)的 Tim Berners-Lee 提出创建一个他称为 WWW 的资料档案库。1991 年，CERN 的执行者完成了第一个浏览器，它是用来显示 Web 文档的程序，通过这种方式，用户可以很容易跟踪内部链接访问 Web 的其他部分。

CERN 取得成果的新闻传播到物理领域其他研究人员耳中之后，更多团队便开始研究浏览器。这些产品中，最成功的是 Mosaic 浏览器，它是位于美国伊利诺斯州 Champaign 的国家超级电脑应用中心(NCSA)开发的。1993 年，Mosaic 浏览器出现后，对 Web 的兴趣也爆发了。执行 WWW 资料档案库的计算机系统的数量从 1993 年的约 500 个猛增到 2003 年的 35 000 000 多个。对 Internet 上 Web 的热情也激发了大量的商业兴趣，催生了一些新公司，发布了一些商业 Web 浏览器，如 Apple 公司的 Safari、Netscape 公司的 Navigator 及 Microsoft 公司的 Internet Explorer。

WWW 上可用文档的数量飞速增长，因为 Internet 用户可以方便地创建新的文档并把它们添加到 Web。如果要将新文档添加到 Web，要做的就是在装有称为 Web 服务器程序的系统上创建文件。Web 服务器允许外部用户访问系统上的文件。服务器输出的单个文件称为 Web 页面。Web 页面通常是由一种称为 HTML 的语言写的，HTML 是超文本链接标识语言的简称。HTML 文档由文本、格式化信息和到 Web 上其他页面的链接组成。每个页面都由统一资源定位符(简称 URL)标识，它让 Web 浏览器可以在现有页面的海洋中找到页面。WWW 的 URL 以前缀“<http://>”开始，紧接着描述的是 Internet 路径，通过该路径可以到达想访问的页面。

Java 特别有趣的一点是，虚拟机并不总是运行于存储程序的同一台计算机上。Java 设计目标之一是让语言在网络上能够运行良好。这种设计目标特别有趣的结果是，Java 支持创建 applet(一种 Java 小程序)，它是一种运行于网络浏览器上下文中的程序。运行 applet 的过程如图 1-5 所示。

applet 作者采取的步骤

1. Web 页面作者为作为 applet 运行的程序写代码。

```
HelloProgram.java
/* File: HelloProgram.java */
import java.awt.*;
import java.applet.*;

public class HelloProgram extends Applet {
    public void run() {
        add(new Label("Hello, world", 10, 20));
    }
}
```

2. applet 作者使用 Java 编译器生成一个文件，这个文件包含 applet 的中间代码。

```
HelloProgram.jar
C8 F8 DA D8 00 03 00 2D 00 1F 00 00 00 0F 07 C8 00
00 16 07 00 1A 07 00 14 B8 00 02 00 00 00 5F
00 04 00 01 0C 09 13 00 18 0C 00 17 00 1C 72 A4
01 00 16 28 4C 8A 61 76 61 2F 81 77 74 2F 00 FF
47 72 81 70 68 69 63 73 3B 29 34 01 00 04 9E 00
```

3. applet 作者发布一个 HTMLWeb 页面，该 Web 页面包含对编译过的 applet 的引用。

```
HelloProgram.html
<html>
<title>Graphic Hello Applet</title>
<applet archive="HelloProgram.jar"
       code="HelloProgram.class"
       width=300 height=150>
</applet>
</html>
```

applet 用户采取的步骤

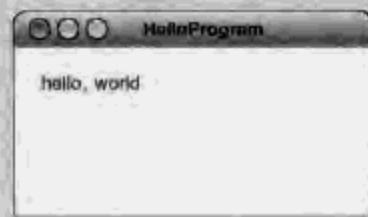
4. 用户在 Web 浏览器上输入 applet 页面的 URL。

5. 浏览器读取、解释 Web 页面的 HTML 源。

6. HTML 源文件中 applet 标记的出现让浏览器从网络上下载编译过的 applet。

7. 浏览器中的检查器程序检查 applet 中间代码，确保它没有违反用户系统的安全。

8. 浏览器程序中的 Java 解释程序运行编译过的 applet，这样会在用户的控制台上产生用户期望的显示结果。



1.8 小结

本章的目的是为学习计算机科学和编程(该过程将从第 2 章正式开始)打好基础。本章着重介绍了编程过程主要包含的内容，以及它如何延伸到计算机科学更大的范围。

本章介绍的重点包括以下内容。

- 计算机系统的物理组件——看得见、摸得着的部分——硬件。然而，计算机硬件要发挥作用，必须事先指定一系列指令(程序)告诉硬件要做什么。这些程序称为软件。
- 计算机科学与其说是计算机的科学，还不如说是使用计算机解决问题的科学。
- 使用计算机解决问题的策略称为算法。要成为算法，策略必须有明确无误的定义，必须有效、有限。
- 程序一般使用高级语言编写，人类读者可以理解所设计的这种高级语言。
- 执行用高级语言写的程序有两种基本策略：编译和解释。编译器将程序转换为特殊计算机系统的机器语言，因此，允许程序直接在该计算机的硬件上运行。解释程序通过读取程序结构，执行必需的操作来模拟该过程。

- Java 在程序执行方面使用混合策略。Java 编译器将程序转换为中间语言，然后解释该中间语言。中间语言像机器语言一样在假想计算机(称为 Java 虚拟机)上执行。
- 编程语言有一套语法规则，它决定程序结构是否正确。编译器使用这些语法规则来检查程序，当违反规则时，它就会报告语法错误。
- 编程错误最严重的类型是，语法正确，但是程序产生了不正确的结果或者根本没有结果。这种类型的错误是一种逻辑错误，它阻止程序正确解决问题，称为故障。发现和修复故障的过程称为调试。
- 大多数程序必须定期更新，以便更正故障或适应应用程序要求的变化。这个过程称为软件维护。设计程序以方便维护是软件工程的重要部分。
- 本文使用 Java 编程语言来阐明编程过程。区别 Java 和其前身语言的基本特征就是，Java 是一种面向对象语言。它可以将数据和所有相关操作封装到概念上统一的实体(对象)里。Java 在 20 世纪 90 年代“Internet 爆发”的时期开始设计，目的就是在网络环境中能运行良好。特别是，Java 可以在 web 浏览器的上下文中运行程序。以这种方式运行的程序称为 applet。

1.9 复习题

1. 即使 Babbage 的两台机器都没有完成，但是分析机的设计引入了一种新思想。这种思想成为了现代计算的中心。区分分析机和早期差分机的重要特征是什么？
2. 通常谁被认为是第一位编程人员？
3. 冯·诺伊曼结构的核心概念是什么？
4. 硬件和软件之间有何不同？
5. 传统科学关注抽象理论还是宇宙本质——不是人造物品。构成计算机科学核心的抽象概念是什么？
6. 算法必须满足的 3 个标准是什么？
7. 算法设计和编码之间有何差别？哪种活动通常更难一些？
8. 术语“高级语言”是什么意思？作为本文基础使用的高级语言是什么？
9. 解释程序和编译器有何不同？
10. 源文件和目标文件之间的关系如何？作为编程人员，哪些文件可以直接使用？
11. 语法错误和故障之间有何不同？
12. 判断题：优秀的编程人员从来不会在其程序中引入故障。
13. 判断题：编写程序的主要成本是程序的开发；一旦程序投入使用，编程成本就可忽略不计。
14. 术语“软件维护”是什么意思？
15. 编写程序时，遵循好的软件工程规则为什么很重要？
16. 面向对象和过程范例有什么主要不同点？
17. 在 Web 浏览器的控制下运行 applet 有哪些步骤？运行 Java applet 与运行 Java 应用程序的方式有何不同？

第 2 章

编程示例

Example is always more efficacious than precept.

—Samuel Johnson, Rasselas, 1759



Grace Murray Hopper(1906-1992)

Grace Murray Hopper 在 Vassar 大学学习数学和物理，后来在耶鲁大学取得了数学博士学位。第二次世界大战期间，Hopper 参加了美国海军，被分配到哈佛大学的法规计算局(Bureau of Ordnance Computation)，在那里她与计算先驱 Howard Aiken 一起工作。Hopper 是最早的 Mark I 数字计算机编程人员之一。Mark I 数字计算机是最早能够执行复杂计算的机器之一。Hopper 为早期计算做出了贡献，是 COBOL 语言开发的主要人员之一。COBOL 语言被广泛应用于商业编程应用程序。1985 年，Hopper 成为了第一位晋升为海军上将的女性。Grace Murray Hopper 一生都被认为是计算机科学领域最著名的成功妇女的典型。为了纪念她的贡献，现在每两年召开一次的“计算界妇女庆典(Celebration of Women in Computing,)”就是以她的名字命名的。

本书的目的是讲述编程基础。照这样做下去，您就会非常熟悉 Java 编程语言，但是该语言的细节不是重点。编程是用计算机解决问题的科学，本书的大部分内容与 Java 的具体细节无关。即使这样，最终您也必须掌握许多细节，以便程序可以最大限度利用 Java 提供的各种工具。

从一个编程新手的角度来看，既要理解编程的抽象概念，又要理解特殊编程语言的具体细节。这是一件两难的事情，没有明显开始的地方。要学习编程，就要编写一些相当复杂的程序。要用 Java 来写这些程序，就必须足够了解这种语言并会使用合适的工具。但是如果用全部精力学习 Java，很可能不会学到更多的普通编程问题(事实上应该学到更多)。而且，Java 是为专家设计的，不是为入门级编程人员设计的。如果没有理解编程的相关问题就想掌握 Java，就会

遇到许多细节问题，从而阻碍学习。

因为对您来说重要的是，在掌握复杂编程之前明白什么是编程。本章从全面介绍一些简单程序入手。查看这些程序时，要尽量从总体上理解发生了什么，而不是关心细节，第3章和第4章将学习这些细节。本章的主要目的是帮助您建立编程和解决问题的直观认识。从长远来看这非常重要。

2.1 “Hello world” 程序

C 语言是历史上最成功的编程语言之一。Java 是由 C 语言发展而来的语言集合的一部分。Brian Kernighan 和 Dennis Ritchie 所著的《C 程序设计语言》一书被认为是 C 语言的定义文档。在那本中，作者在第1章的第1页提出了如下忠告：

学习新的编程语言的唯一方法就是用它写程序。所有语言要写的第一段程序都一样：

打印单词

`hello, world`

这是一个很大的障碍：要克服它就必须能够在某个地方创建程序文本，成功地编译它，加载它，运行它，并知道在哪里输出。掌握这些机械的细节之后，剩下的就比较容易。

忠告下面就是“hello world”程序的4行文本，它是所有 C 编程人员共享的。当然，Java 与 C 不同，但是基本忠告还是一样的：所写的第一段程序必须尽量简单，确保可以掌握编程过程的技巧。

同时，现在是 21 世纪，20 世纪 70 年代早期适用的程序与今天使用的程序是不一样的，记住这一点很重要。那时的机械式电传打字机和控制台——原始的连接到键盘并基于字符的显示屏——已经被更复杂的硬件所取代了。能打印一连串单词不再像当时那样令人兴奋。今天，输出很可能定向到屏幕上的图形窗口。很幸运，Java 程序正好非常简单。“hello world”程序的 Java 版如图 2-1 所示。

如图 2-1 所示，HelloProgram 分为 3 个独立部分：程序注释、输入和主类。虽然 HelloProgram 结构非常简单，但它是一个典型程序，接下来的几章都会看到这样的程序。可以将它作为组织 Java 程序的一种模式。

```
/*
 * File: HelloProgram.java
 *
 * This program displays "hello, world" on the screen.
 * It is inspired by the first program in Brian
 * Kernighan and Dennis Ritchie's classic book,
 * The C Programming Language.
 */

import acm.graphics.*;
import acm.program.*;

public class HelloProgram extends GraphicsProgram {
    public void run() {
        add(new GLabel("hello, world", 100, 75));
    }
}
```



图 2-1 “hello,world” 程序

2.1.1 注释

HelloProgram的第一部分是英语注释，它是编译器会忽略的简单程序文本。在Java中，注释有两种形式。第一种形式由包含在标记“/*”和“*/”之间的文本组成，即使文本延续了几行。第二种形式(本书没有使用)由符号“//”引入，一直延续到行的结束。HelloProgram程序中，注释以第一行的“/*”开始，以几行后的“*/”结束。

注释是为人而写，不是为计算机而写。它们的目的是将程序的信息传达给其他编程人员。

Java编译器将程序转换为机器可以执行的形式时，会完全忽略注释。

本书中，每条程序都以称为程序注释的特殊注释开始，它描述了程序的整体运行情况。注释包括程序文件名和描述程序运行的一两个句子。本例中，程序注释为程序的原始思想提供了信任。注释也可以描述程序特别复杂的部分，指出使用对象，对如何改变程序行为提供建议，或者提供其他编程人员想知道的其他附加信息。像HelloProgram这样简单的程序，通常不需要扩充的注释。然而，随着程序越来越复杂，您会发现，与代码一起包含有用的注释是让其他人理解程序——或者，暂时没有看它后来回头再看这段程序时——能够明白自己意图的最好方法之一。

2.1.2 输入

HelloProgram的第二部分由下面的行组成

```
import acm.graphics.*;
import acm.program.*;
```

这些行说明程序使用了两个库程序包。库程序包是其他编程人员写的工具的集合，这些工具可以执行特殊操作。HelloProgram使用的库是一个图形库和一个程序库，每个库都出自计算机协会(ACM)程序包的集合。程序包名称结尾的星号说明，相关程序包里的所有组件都应该输入。本书中的每个程序都会至少输入acm.program程序包，所有使用图形的程序都输入acm.graphics，说明大多数程序需要在程序注释之后紧接着包含这些行。有些程序也会使用附加程序包，但每个程序都必须包含import行。

写程序时，可以使用这些程序包提供的工具，从而免去亲自写这些工具的麻烦。库对于编程来说至关重要。写更复杂的程序时，通常会依赖几个重要的程序包。

2.1.3 主类

HelloProgram.java文件的最后一部分由下面几行组成：

```
public class HelloProgram extends GraphicsProgram {
    public void run() {
        add(new GLabel("hello, world", 100, 75));
    }
}
```

这5行是Java中类定义的第一个示例。类是基本单元(Java程序被划分到这些单元中)，它为创建个人对象建立了模板。诚然，这里的定义相对含糊一些，在2.5节中将对该定义进行提炼。

要明白程序的定义，最有用的办法就是将它按层次结构分类。类的定义——和Java中其他许多结构一样——由两部分组成：整体定义Java定义的某些本质特征的标题行和填充细节

的主体。阅读程序定义时，单独想想标题行和主体通常很有用。例如，下面是 HelloProgram 类的示意图，图中用一个框取代了类的主体。

```
public class HelloProgram extends GraphicsProgram {
    //类定义的主体
}
```

对于 HelloProgram 类采用这种方法，可以着重看标题行。理解标题行之后，就可以浏览主体，体会细节。

在类定义中，标题行提供了有关类特征的重要信息。HelloProgram 类标题行中的第一个单词是 public，这表示其他程序可以访问该类。所有程序都定义为公共类，因为其他有些应用程序——要么是 Web 浏览器要么是用来测试程序的编程环境——必须能够启动这些程序。标题行的第二个单词是 class，它告诉 Java 此行是类定义的开始。标题行后面的单词 public、class 和 extends 在 Java 中都有特殊的意思。这样的单词称为关键字。

此标题行中的 extends 关键字表示 HelloProgram 是 GraphicsProgram 的子类，它是 acm.program 程序包中定义的程序类型之一。2.6 节概述了 GraphicsProgram 类的特殊性能，并在第 9 章进行了详细定义。HelloProgram 是 GraphicsProgram，因此，GraphicsProgram 能做的，HelloProgram 都可以做。这里理解这一点就够了，详细情况稍后讨论。

HelloProgram 类定义的主体包括一个单独定义，如下所示：

```
public void run() {
    add(new GLabel("hello, world", 100, 75));
}
```

该定义是 Java 方法的示例。Java 方法就是一系列收集在一起并命名的程序步骤。这种方法的名称是 run，如其标题行所示。大括号之间方法执行的步骤，称为语句。语句共同构成方法的主体。HelloProgram.java 示例中的方法 run 只有一条语句，但是方法通常包含连续执行的几条语句，以第一条语句开始，延续到主体内的最后一条语句。

方法 run 在使用 acm.program 程序包的程序中发挥了特殊作用。无论什么时候运行 Java 程序，计算机总是执行包含在主类 run 方法主体中的语句。HelloProgram 中，run 的主体由单个语句组成。

```
add(new GLabel("hello, world", 100, 75));
```

该语句使用了库程序包中的两种工具。第一个是 GLabel 类，它来源于 acm.graphics。看到的行部分是

```
new GLabel("hello, world", 100, 75)
```

称为构造函数(它用于创建新的对象)的 Java 表达的示例。这里，构造函数创建了一个新的包含文本“hello, world”的 GLabel 对象，该文本开始于 x 和 y 坐标分别为 100 和 75 的点。(此时，虽然您还不能确切理解这些坐标的意思，但这不妨碍对程序的大概了解 2.6.4 小节将介绍 Java 坐标模型)。语句格式如下：

```
add(新生成的标签)
```

它采用了新 GLabel 并将它添加到对象列表，这些对象是当前显示的一部分。结果程序在图形窗口生成如图 2-2 所示图像。



图 2-2 "hello world" 程序产生的图像

2.2 编程过程的观点

本章重点不是详细理解 Java 的工作原理，而是让我们对一些简单程序有一个良好的全面认识——心理学家通常称为“完全形态”。只要将 HelloProgram.java 文件输入计算机，然后试验它，就可以了解 Java 的许多问题，实际上这是本章后面的第一个练习。例如，很容易就可以将程序显示的文本从“Hello, world”改变为一些更有趣的东西：将数字 100 和 70 改为其他的数字，可以将文本放在图形窗口中的不同位置。在此过程中，您会发现 Java 坐标系的一些有趣特征，例如原点是左上角而不是左下角(传统几何学中原点在左下角)。这样，如果将 y 坐标从 75 改为 100，GLabel 会向屏幕下方移动。也可以对用于表述这些坐标值的单元有个直观感觉，即使不确切知道选择这些单元的理由。所有要做的就是尝试一些不同的值，看看它们在屏幕上如何移动。完全可以将多个 GLabel 放在相同的图形窗口。如果想在窗口的不同位置包括第二个 GLLabel，只要将一行添加到与第一个看起来非常想象的程序里就行了，尽管可能要指定不同的文本串和不同的 x 和 y 坐标。

重要的是记住，从试验中会学到很多东西。正如 Brian Kernighan 和 Dennis Ritchie 所说“学习新的编程语言的唯一方法就是用它写程序”。写的程序越多，研究的程序越多，学到的编程方法就越多。

当然，有时需要学习 Java 语句的细节，以便理解每条语句的运行方法。然而，详细分析不是思考程序的唯一有效方法。有时，退一步从整体看待程序会很有帮助。程序分解的、一步一步的检查是简化法。从更全面的观点看，如果将程序视为一个完整实体(十分关注其整体运行)，就是采用了一种更全面的观点，这些观点允许从不同的角度查看程序。

简化论是哲学原理。该原理说，通过理解对象的组成部分就可以很好地理解对象的整体。与之对立的是整体论，它认为整体通常不仅仅是其部分的总和。学习如何写程序，必须学习从这些观点分析程序。如果只专注于大的方面，就不可能理解问题解决所需的工具；然而，如果只专注于细节，就总是会只见树木而不见森林。

刚开始学习编程，最好的方法通常是在这两种观点间轮换。采用整体观点有助于提高对编程过程的直观认识，让您可以从程序中退出来说“我知道程序要做什么。”另一方面，要练习编写程序，必须充分采用简化论观点，以便理解这些程序如何放在一起。

2.3 两数相加的程序

如果有许多示例，那么通过示例学习程序就比较容易。图 2-3 显示的是一种不同类型的程序，它要求用户输入两个整数(这是一个数学术语，这种数只有整数部分，没有小数部分)，将这两个整数相加，然后显示他们的和。

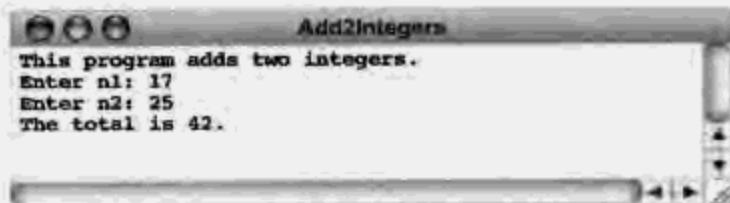


图 2-3 相加两个整数

图 2-4 中的程序介绍了几个编程概念，这些概念不是 HelloProgram 的一部分。首先，Add2Integers 是不同类型的程序，如其标题所示：

```
public class Add2Integers extends ConsoleProgram
```

程序扩充 ConsoleProgram 而不是 GraphicsProgram，这说明它访问了不同的实用工具集。ConsoleProgram 类支持用户以一种传统的基于文本的风格进行交互。ConsoleProgram 可以要求用户从键盘输入，在计算机屏幕上的窗口中显示信息；由于历史的原因，这种键盘/显示的结合称为控制台。图 2-3 说明了与 ConsoleProgram 的交互，显示运行 Add2Integers 程序后可以看到的输出。

```
/*
 * File: Add2Integers.java
 *
 * This program adds two integers and prints their sum.
 */
import acm.program.*;
public class Add2Integers extends ConsoleProgram {
    public void run() {
        println("This program adds two integers.");
        int n1 = readInt("Enter n1: ");
        int n2 = readInt("Enter n2: ");
        int total = n1 + n2;
        println("The total is " + total + ".");
    }
}
```

图 2-4 相加两个整数的程序

这种显示程序输出的图称为样本运行。虽然不可能看到本书使用的黑白打印效果，但是用户输入的内容在屏幕上用蓝色显示出来了。

如果从整体来看该程序，完全可以确定程序要做什么，即使不理解单个语句的细节。如果没有别的，程序注释和 run 方法的第一行都是固定样本。但即使没有这些标志，许多编程新手

在理解代码功能上也基本没有障碍。一般来说，读程序比写程序容易。阅读完许多小说之后再写小说就容易一些，同样，如果花时间阅读一些设计良好的程序，模仿其结构和风格，写出好的程序也会容易些。特殊语句的意思——就像小说中不熟悉的单词——通常从上下文就能理解。开始下几段详细讨论之前，仔细查看 Add2Integers 的每一行，看看自己能理解多少。

run 方法的第一行是

```
println("This program adds two integers.");
```

println 方法(打印行 print in 的缩写)用于在 ConsoleProgram 类显示的窗口上显示信息。圆括号里的值告诉 println 方法应该显示什么，这样的值称为参数。包含文本

```
This program adds two integers.
```

的双引号输出时不会显示。但是 Java 用它来说明，引号之间的字符是文本数据的实例，称为字符串。因此，如果接着产生几个 println 调用，输出中的每个字符串会以单独行的形式出现。

为用户创建一行输出时，有时分行打印行会比较方便。ConsoleProgram 类提供了一种称为 print 的方法，除了不换行之外，它与 println 是一样的。print 方法可以打印行的一部分，随后以调用 println 结束。因为本节稍后讨论的串联运算符也可以很容易联合一行里的几个值，所以 print 方法在本书中很少出现。

程序中首行的目的不是告诉其他编程人员程序要做什么；这项功能由文件开头的程序注释实现。对 println 的首次调用告诉用户程序要做什么。今天使用计算机的大多数人都不是编程人员，希望这些用户通过查看程序代码来确定它要做什么，这不太合乎情理。程序本身必须明确此目的。

run 方法的下一行如下所示。

```
int n1 = readInt("Enter n1: ");
```

从整体来看，本行的目的相当清楚，假如看到的都说明它可能输出第一个整数值。然而，如果采取简化论的观点，这一行代码则引入了几个新概念。其中，最重要的是变量，很容易将它当作一些数据(编写程序时不知道这些数据的值)的占位符。写一段程序让两个整数相加，根本不需要知道用户要相加的两个整数是什么。程序运行时，用户会输入这些整数。因此，可以在程序中引用仍未指明的值，创建一个变量来保存要记住的每个值，给它命名，当需要引用该变量包含的值时，使用名称就可以了。通常选择变量名，这样将来阅读程序的编程人员就很方便地知道每个变量的用法。Add2Integers 程序中，变量 n1 和 n2 表示要相加的两个整数，变量 total 表示和。

在 Java 中引入新的变量时，必须声明该变量。为此，需要提供声明——它是一行代码，指定了变量名称，并告诉编译器该变量包含什么类型的数据。Java 中，用于存储整数数据的类型称为 int。声明形式

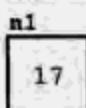
```
int n1 = [值];
```

引入了一个新的称为 n1 的整数变量，它的值由标记为这里，表达式为

```
readInt("Enter n1: ")
```

就像第一行的 `println` 示例一样，这个表达式调用了 `ConsoleProgram` 中的 `readInt` 方法。`readInt` 方法从在屏幕上显示参数开始，因此用户知道下一步要做什么。这种告诉用户输入什么内容的字符串通常称为提示符。然而，不像 `println` 那样，`readInt` 方法没有返回到下一行的开头，而是在提示符后等待用户输入整数。用户输入完整数，按下 `Return` 或 `Enter` 键，作为 `readInt` 方法的结果，整数被传回 `run` 方法。编程术语中，我们说 `readInt` 返回了用户输入的值。

在纸上描述程序的运算，编程人员通常使用框图来表示指定给变量的值。如果考查本节前面的样本 `run`，会发现用户为了适应第一个输入请求，输入了值“17”。这说明赋值语句在变量 `n1` 里存储值 17，画一个框，命名为框 `n1`，然后在框里面写上 17 表示它的值，如下所示。



`run` 方法中的第三行几乎完全与第二行一样，它读取变量 `n2` 的值。如果用户适应提示符输入 25，就可以更新框图来显示新变量，如下所示。



`run` 方法的下一行是

```
int total = n1 + n2;
```

该语句声明变量 `total`，将它的值指派在等号右边，即：

```
n1 = n2
```

这一段代码是称为表达式的基本编程结构的示例，表达式表示运算的结果。第 3 章会更加正式地定义表达式的结构，但理解 Java 表达式的意思却很容易，因为它们中许多都很像传统数学中的表达式。

`Add2Integers` 程序中，目的是相加存储在变量 `n1` 和 `n2` 里的值。为此，可以使用`+`运算符，在小学算术中您就理解这个运算符。然而，计算和还不够。还需要将和存储在变量里，以便后面可以引用它。语句

```
int total = n1 + n2;
```

将每个运算都结合起来了。它计算变量 `n1` 和 `n2` 的和，然后引入变量 `total` 来保存该值。`run` 方法的最后一条语句是

```
println("The total is " + total + ",");
```

它完成显示计算结果的任务。这条语句看起来很像程序中的第一条语句，它也是对 `println` 方法的调用。然而，这时有一个新变化。该语句转到 `println` 参数值而不是采用单个字符串参数。

```
"The total is " + total + ","
```

和前面语句中的 `n1+n2` 表达式一样，这个表达式使用`+`运算符来连接单个值。本语句中，至少`+`应用的有些值是字符串而不是传统加法定义的数值。Java 中，将`+`运算符应用到字符串数据来

重新解释运算符，意思是将字符串首尾相连地相加在一起起来联合它们的字符。这种运算称为串联。如果表达式的有些部分不是字符串，在应用串联运算符之前，Java 就会将它们转换为标准的字符串表示法。最后一条语句 `println` 的作用就是显示 `total` 的值(在将它与告诉用户输出值的字符串串联起来之后)。在样本运行里可以看到语句的结果。

虽然 `Add2Integers` 只能与整数运行，Java 可以与其他许多类型的数据运行。例如，只要改变变量类型和输入方法的名称，就可以改变程序，让它相加两个实数，如图 2-5 所示。

```
/*
 * File: Add2Doubles.java
 *
 * This program adds two double-precision floating-point numbers
 * and prints their sum.
 */
import acm.program.*;
public class Add2Doubles extends ConsoleProgram {
    public void run() {
        println("This program adds two numbers.");
        double n1 = readDouble("Enter n1: ");
        double n2 = readDouble("Enter n2: ");
        double total = n1 + n2;
        println("The total is " + total + ".");
    }
}
```

图 2-5 相加两个双精度数的程序

大多数编程语言中同，包含小数的数称为浮点数，在数学中类似于实数。Java 中最常见的浮点数类型是 `double` 类型，它是双精度浮点的简称。如果要在程序中存储浮点值，必须声明 `double` 类型的变量，和前面必须声明 `int` 类型的变量来写 `Add2Integers` 一样。程序中唯一的不同是，它要求用户通过调用 `readDouble` 而不是 `readInt` 输入，但程序的基本模式没有改变。

2.4 编程习语和模式

程序通常遵循指定的模式，这些模式可以应用于新的应用程序，在学习编程的过程中，认识到这一点很重要。前面几节，没有要求深入研究如何将 `Add2Integers` 程序改变为 `Add2Doubles`：所有要做的是在这里或那里改变一些单词，指出程序应该使用 `double` 类型而不是 `int` 类型。可以很容易地创建一个 `Add3Doubles` 程序，只要添加下列行

```
double n3 = readDouble("Enter n3: ");
```

然后将 `total` 运算改变为读取即可。

```
double total = n1 + n2 + n3;
```

如果用简化的方法来看这些行，确实可行。要详细了解这些行，需要多方面了解 `readDouble` 方法和 Java 执行运算的方式。如果从整体考查这些语句，那么这些细节都无关紧要。将这些语句作为执行特殊运算的术语模式使用是很有效的。例如，如果要读取用户输入的浮点值，记

住下面的模式就很有用：

```
double variable = readDouble("prompt");
```

所有要填写的就是变量名和希望用户查看的提示字符串。因此，模式可用作所有运算的模板(这些运算要求读取用户的取浮点值)。读取整数值的模式几乎都一样：

```
int variable = readInt("prompt");
```

知道了这些模式之后，就不需要记住那些细节了。

这一类模式是人类历史上的重要工具。在书写出现之前，历史和宗教作为口头传统的一部分被一代一代地传下来。荷马的《伊利亚特》和《奥德赛》、印度的吠陀梵语文学、古斯堪的那维亚语神话、让非洲传统在几个世纪的奴隶制度下得以保留下来的说道和歌曲——这些都是口头传统的例子。这些著作都有习惯用语模式化重复的特征，这让歌唱者、传教士和讲故事的人容易记住它们。这种重复的模式提供了记忆提示，这些记忆提示有助于记住和改变那些又长又复杂的故事。

总体上来看，Java 规则和 Java 库提供的许多其他特征组成了一段又长又复杂的故事，这个故事的细节太多了，以至于不可能全部记住它们。即使这样，编写程序时，您仍会注意到许多公式化结构会重复出现，就像口头传统的规则一样。如果学会识别这些公式，并将它们当作概念单元，就会发现，Java 中要记住的编程比您想象的要少。编程过程中，这种公式称为习语或编程模式。要有效地编写程序，必须学习如何将这些编程模式应用到当前任务。最终，不需要故意专注过程，也应该能够编写程序。作为解决方案策略的一部分，一个大致想法会出现在您的脑海里：编写程序时，会自动将这种想法转换为合适的模式。

2.5 类和对象

继续讨论第 3 章和第 4 章中的表达式和语句的细节之前，先引入一个更高级的概念(如本章示例所示)很有必要。我们所看到的程序——HelloProgram、Add2Integers 和 Add2Doubles——都定义为 Java 类。而且，这些类都定义为 acm.program 程序包提供的现有类的扩充。HelloProgram 是 GraphicsProgram 的扩充，另外两个是 ConsoleProgram 的扩充。定义新类作为现有类的扩充时，通常将新类称为原始类的子类。因此，HelloProgram 是 GraphicsProgram 的子类。相反，GraphicsProgram 是 HelloProgram 的超类。

Java 中类的概念是最重要的思想之一。从本质上来说，类是可扩充的模板，这种模板指定对象特殊风格的结构。每个对象都是特殊类的实例，它可以轮流作为不同对象的模板。如果要用 Java 创建对象，首先必须定义这些对象所属的类，然后构造单个对象(这些对象是类的实例)。

从概念上讲，对象是一个封装状态和行为的综合实体。对象的状态由一组属性组成，这些属性适用于该对象，并可随着时间而更改。例如，对象可能表现为它在空间上的位置、颜色、名称及其他许多特征。对象的行为指对象对内部事件或其他对象发出的请求做出响应的方式。在面向对象编程的语言中，对象中触发特殊行为的通用词语称为消息。Java 中，发送消息相当于调用与对象(消息发送给该对象)相关的方法，这就有可能将消息和方法调用看作同一概念的不同名称。

2.5.1 类的层次结构

Java 中的类形成了层次结构。这些层次结构在结构上与许多熟悉的分类结构类似，例如生物界的组织分类。生物界分类最初由瑞典植物学家 Carl Linnaeus 于 18 世纪提出，其部分层次结构如图 2-6 所示。图的最上面是所有生物的全体类别。该类别再细分为几个界，然后依次分解为门、纲、目、科、属和种。在图 2-6 所示的分类树中，最后两级——属和种，它们一起组成了特殊种类生物的学名——*Iridomyrmex purpureus*，一种红蚂蚁的名称。世界上的单个红蚂蚁相当于编程语言中的对象。因此，每个个体都是物种 *purpureus* 的实例。然而，依据这种层次结构，个体也是 *Iridomyrmex* 属、Insecta 纲和 Arthropoda 门的实例。当然，动物和生物是一样的。而且，每只红蚂蚁都具有符合每个祖先类别的特征。例如，红蚂蚁有 6 条腿，这就是 Insecta 类具有的特征之一。

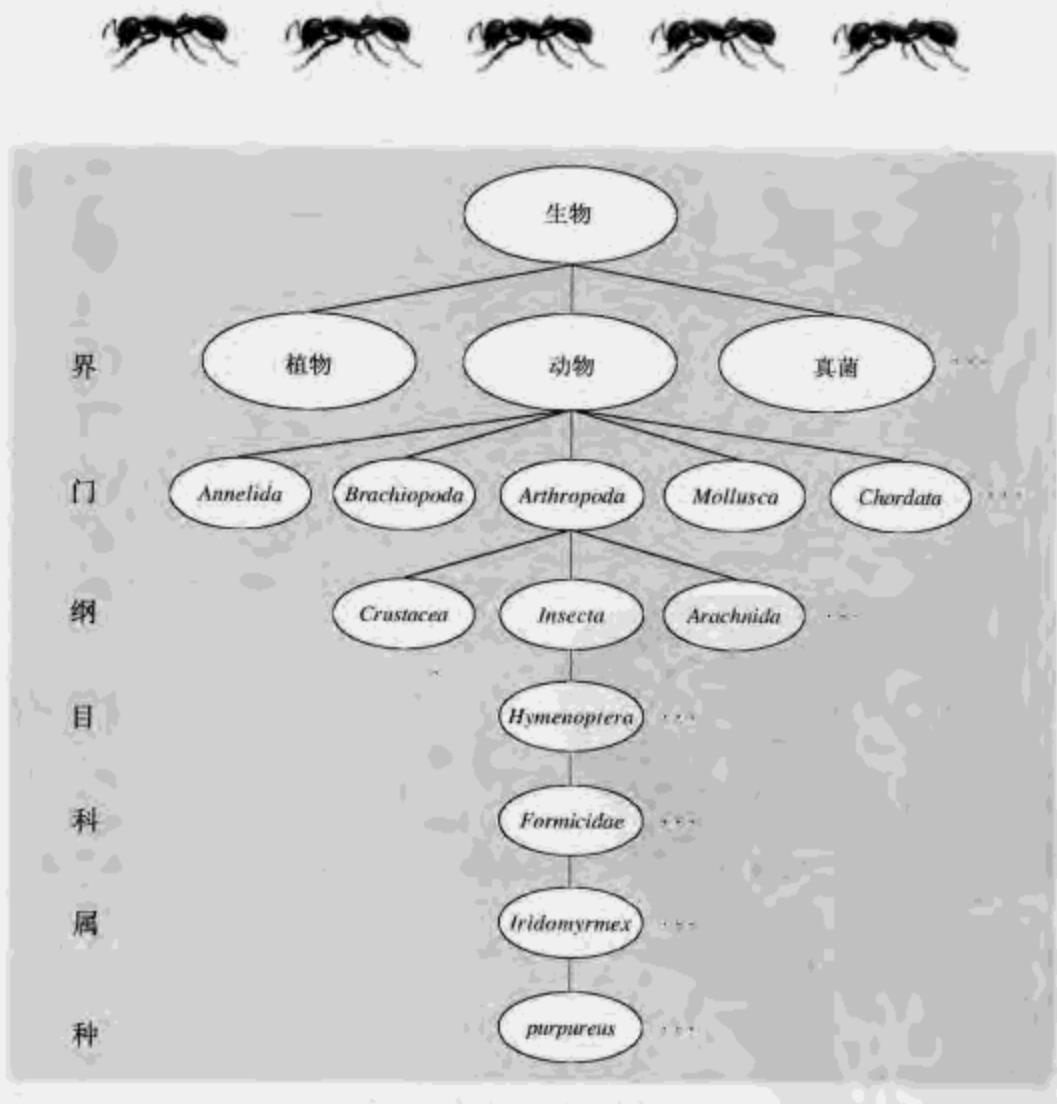


图 2-6 生物分类层次结构级别

生物比喻说明了 Java 中类的基本特性之一。所有特殊类的实例也是其所有子类的实例。因此，依据定义，所有 HelloProgram 类的实例也是 GraphicsProgram 类的实例，而且，每个 HelloProgram 类的实例自动获得 GraphicsProgram 的公共行为，这种呈现超类行为的属性称为继承。

2.5.2 Program 类的层次结构

由 acm.program 定义的类形成了一种层次结构，这种层次结构比您到目前为止看到的层次结构要复杂。这种层次结构如图 2-7 所示。看到的每个类——GraphicsProgram 和 ConsoleProgram——都是高级类 Program 的子类。Program 类是 JApplet 类的子类，JApplet 类本身也是标准的 Java 类(称为 Applet)的专门子类。图中显示，设计的程序的每个实例(如图 2-1 中所示的 HelloProgram 类的实例)，同时也是 GraphicsProgram、Program、JApplet 和 Applet 的实例。applet 程序可以在 Web 浏览器上运行它。通过继承，所有的 Applet 类、Program 类和 GraphicsProgram 类共享这个属性。

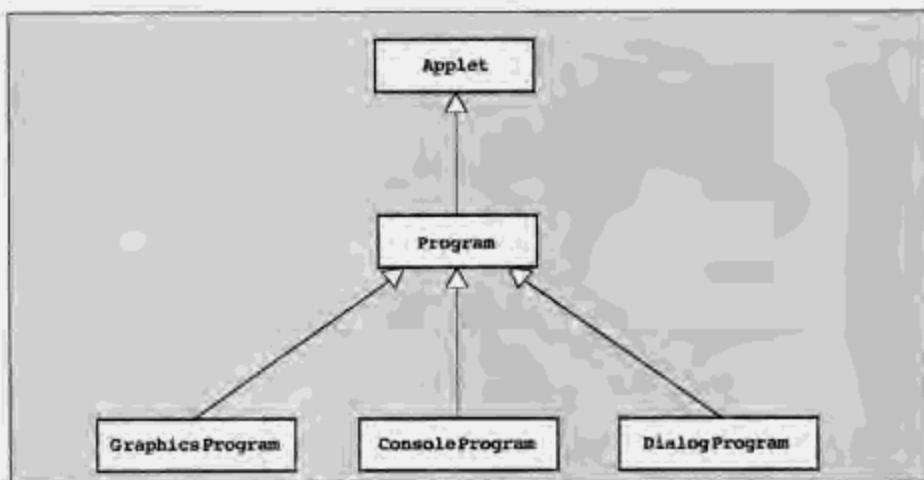


图 2-7 程序类层次结构

图 2-7 也显示出除了已经看到的两个子类之外，还有另一个 Program 子类。DialogProgram 子类的整体组织与 ConsoleProgram 类非常类似。特别是，它完全共享一组相同的方法，这些方法实际上都由 Program 类指定。不同点是这些方法有不同的解释。ConsoleProgram 类中，对 println 和 readInt 的方法调用指定在每个 ConsoleProgram 类显示的基于文本窗口控制之下的用户交互。DialogProgram 类中，对相同方法的调用指定通过程序运行时弹出的交互对话框的用户交互。

如果改变如图 2-4 所示类的标题行，它会显示

```
public class Add2Integers extends DialogProgram {
```

程序仍然会将两个数相加，但是交互的风格截然不同。运行新版程序会产生一系列对话框，如图 2-8 所示。

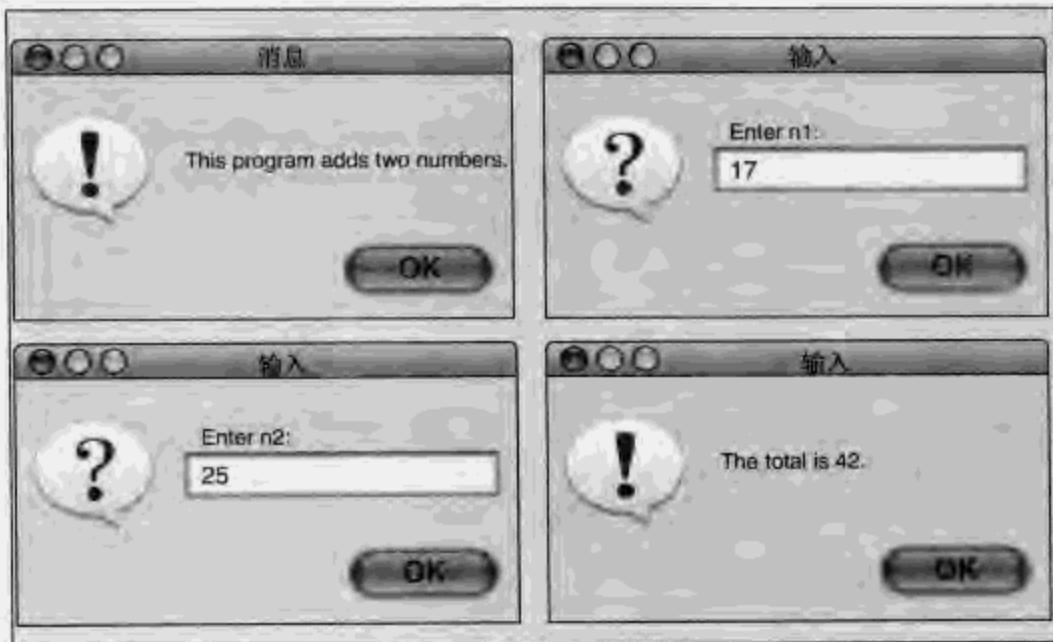


图 2-8 运行 Add2Integers 对话版时出现的对话框

ConsoleProgram 和 DialogProgram 类分别定义了自己版本的 `println` 和 `readInt`，因此，它们以一种适合于该类的风格运行。重新定义现有方法，以便其运行与超类定义的执行不同，这种行为称为重写该方法。

2.6 图形程序

在接下来的几章您会发现，ConsoleProgram 类为阐明新的编程概念提供了有用的架构。很多情况下，在 ConsoleProgram 类中说明这些概念要比在 GraphicsProgram 类中容易些，因为使用图形的程序比基于控制台的程序更为复杂。另一方面，图形程序通常比控制台程序更令人兴奋。对许多同学而言，图形程序更有趣，这一事实至少减少了它们涉及的额外复杂性。

凑巧的是，对 GraphicsProgram 类使用整体的方法也是可能的。就像无需掌握基本细节就可以编写简单的 ConsoleProgram 一样，只要学习一些简单概念，就可以创建 GraphicsProgram。本章余下的部分将用示例告诉您如何编写一些简单的 GraphicsProgram 类程序。这样，在详细讨论 `acm.graphics` 程序包(见第 9 章)之前，就可以开始创建一些有趣的应用程序。

2.6.1 HelloProgram 示例回顾

理解 GraphicsProgram 结构的第一步，就是要更仔细考查图 2-1 中的 HelloProgram 示例。该程序的 `run` 方法如下：

```
public void run() {
    add(new GLabel("hello, world", 100, 75));
}
```

从 2.1 节的讨论中可知，如何将调用解释为组成方法主体的 `add`。特别是，`GLabel` 中

acm.graphics 程序包中定义的类，该程序包可以在图形窗口显示文本字符串。我们也知道，关键字 new 让程序可以构造新的类的对象，将它初始化并在位置(100, 75)上显示字符串“hello, world”。

本示例中，GLabel 构建函数产生的对象立即被传送给 add 方法，以便让它显示在窗口。然而，既然已经知道了变量，就可以在程序中首先做一些似乎不太重要的改变。这种改变就是将 GLabel 对象传送给 add 方法之前，先将它存储在变量里。做了这个改变之后，run 方法如下所示：

```
public void run() {
    GLabel msg = new GLabel("hello, world", 100, 75);
    add(msg);
}
```

依据图形窗口中显示的内容，程序像以前一样运行。唯一的区别在于实现：包含字符串“hello, world”的 GLabel 对象除了显示在屏幕上之外，现在还存储在变量 msg 里。这种改变——其实很简单——使您可能写很多更有趣的程序。

2.6.2 向 GObjects 发送消息

将 GLabel 对象存储在变量里的好处是可以给对象发送消息。因为变量给出了指定对象的方法。Java 中，这种消息表现为方法调用，这些调用指定对象(消息发送给该对象)，称为接收方。具有明确接收方的方法调用，其一般语法模式是

```
receiver.name(arguments)
```

其中，*receiver* 是消息发送的对象，*name* 是响应特殊消息的方法的名称，*arguments* 是一列——很可能是空的——值，它给接收方提供所有做出合适响应所必需的附加信息。

在 acm.graphics 程序包中，可以使用这样的消息来改变图形窗口中显示的图形对象的外观。例如，改变 HelloProgram 显示的字母大小。程序的原始版本显示“hello, world”的字母相对于屏幕尺寸而言太小了。要让输出更有视觉冲击力，可能想给 GLabel 发送一条消息，让它使显示的文本更大一些。

在 Java 中，改变标签外观大小的方法是改变标签显示的字体 font。您多半已经从其他计算机应用程序中熟悉了字体，对字体决定字符显示的风格有了直观的认识。更正式地说，字体是编码，它将字符对应到显示在屏幕上的图像。Java 字体由 3 部分组成：字体族名(可能是 Times 或 Helvetica，但更可能是本节稍后定义的标准字体族名之一)、风格(如粗体或斜体)和磅值(指出字符大小的一个整数。这些字符使用标准打印机单元的点，通常等于 1/72inch)。

要改变 GLabel 对象的字体，可以给它发送一条 setFont 消息，以字符串的形式指定新的字体族名、风格和磅值。例如，改变 run 方法来指定 24 点 Helvetica 字体，如下所示：

```
public void run() {
    GLabel msg = new GLabel("hello, world", 100, 75);
    msg.setFont("Helvetica-24");
    add(msg);
}
```

行 msg.setFont("Helvetica-24"); 是 Java 用来给对象发送消息所用语法的简单示例。本示例中，接收方是存储在变量 msg 里的 GLabel 对象，执行消息的方法名称是 setFont。

参数由字符串“Helvetica-24”组成。正如所料，GLabel 对象对消息做出了响应，将其字体改变为 Helvetica，磅值为 24。

对 HelloProgram 类做了这个改变之后，显示如图 2-9 所示。



图 2-9 hello word 程序的输出结果

这虽然更有可读性，但也许不是那么有趣。

从使用单词处理器中可以知道，试验不同的字体可能很有意思。例如，在我的笔记本电脑上，有一个名为 Lucida Blackletter 的字体，它产生一种固定风格的手稿，让人想起中世纪流行的手写稿风格。因此如果将此程序中的 setFont 调用改变为

```
msg.setFont("Lucida Blackletter-28");
```

输出变成图 2-10 所示。



图 2-10 hello word 程序的输出结果

这当然比前面的显示更具有想象力。然而问题是，其他运行程序的人可能看不到相同的输出。只有他们使用的计算机也安装了 Lucida Blackletter 字体，输出才会是这种形式。字体列表中没有 Lucida Blackletter 的计算机，Java 会用默认字体代替，这种默认字体很可能有完全不同的外观。

如果对编写能够在不同计算机上运行的可移植程序感兴趣，就最好避免这种非传统字体。实际上，要想让程序尽可能可移植，最好坚持使用下面的字体族名。

Serif 一种传统的报纸风格的字体，这种字体的字符在顶部和底部都有一条短线（称为衬线），它们引导眼睛将单词读取为单个单元。最常见的衬线字体是 Times。

SansSerif	一种没有衬线的更简单、更朴实的字体风格，最常见的是 Helvetica。
Monospaced	一种打字机风格的字体，其字符宽度相同。最常见的等宽字体是 Courier。
Dialog	一种依赖系统的字体，用于对话框中的输出文本。
DialogInput	一种依赖系统的字体，用于对话框中的用户输入。

本书剩下的几章只使用这些通用字体名。

还想改变什么呢？要生成一个更具有想象力的显示效果，可以通过给 GLabel 对象发送 setColor 消息来改变文本颜色。Java 中虽然可以使用多种颜色，但是 Color 类定义了一组标准颜色，在很多情况下，有这些颜色就足够了。这些预定义的颜色名称如下：

Color.BLACK	Color.RED	Color.BLUE
Color.DARK_GRAY	Color.YELLOW	Color.MAGENTA
Color.GRAY	Color.GREEN	Color.ORANGE
Color.LIGHT_GRAY	Color.CYAN	Color.PINK
Color.WHITE		

例如，要将 GLabel 对象改变为红色，只要在 run 方法中再添加一行代码即可，如下所示：

```
public void run() {
    GLabel msg = new GLabel("hello, world", 100, 75);
    msg.setFont("Helvetica-24");
    msg.setColor(Color.RED);
    add(msg);
}
```

称为 java.awt 的程序包中定义了定义标准颜色名称的 Color 类。该程序包执行抽象窗口工具箱(简称 AWT)，该工具箱是 Java 本身图形库的基础。因此，如果要在程序中使用任何颜色名称，都必须在程序开头用下面的代码行输入程序包：

```
import java.awt.*;
```

2.6.3 GObject 类的层次结构

GLabel 类是 acm.graphics 程序包定义的几个类之一。acm.graphics 层次结构的一小部分如图 2-11 所示。图顶部是 GObject 类，它表示图形对象的通用类，这些图形对象可以显示在图形窗口中。下一级层次结构由 4 个 GObject 子类组成，每个子类表示一个要显示的图形对象的具体类型。已经见过 GLabel 类，它用于显示文本字符串。GRect、GOval 和 GLine 类分别用于显示矩形、椭圆和直线。

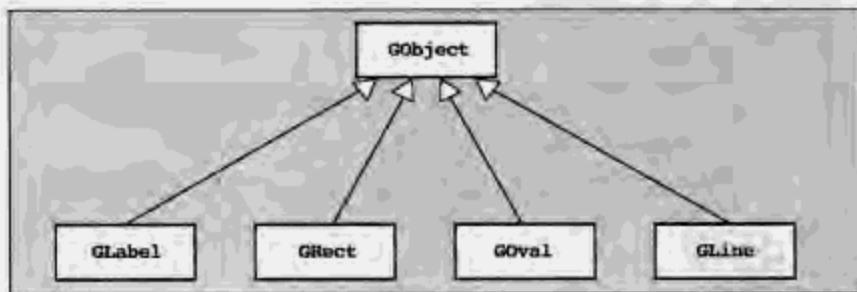


图 2-11 部分 GObject 层次结构

就像图 2-7 中 Program 层次结构中的类一样, 图 2-11 中的类也是子类和继承的有用例证。在 GObject 级定义所有与每个图形对象相同的行为, 从这一级开始, 每个子类自动继承。例如, 因为所有图形对象都可以上色, 所以将 setColor 方法定义为 GObject 高级定义的一部分。2.6.2 小节结尾的程序可以使用 setColor 方法来改变 GLabel 对象的颜色, 因为 GLabel 从其超类那里继承了 setColor 方法。相反, 只为 GLabel 类定义 setFont 方法。毕竟, 字体只对涉及字符的结构有意义。因为将字体设置为矩形、椭圆或直线都没有意义, 所以只为 GLabel 类定义 setFont。

图 2-12 包含一些用于 GLabel、GRect、GOval 和 GLine 类的有用方法。虽然每个类包含的方法不只这里列出的这些, 但是刚开始有这些设置应该足够了, 特别是如果有一些样本程序作为模型。

构造函数

new GLabel(string, x, y)	创建新的包含指定字串的 GLabel 对象, 该字串从点(x, y)开始
new GRect(x, y, width, height)	创建新的有指定尺寸的 GRect 对象。该尺寸左上角位于点(x, y)
new GOval(x, y, width, height)	创建新的 GOval 对象, 其大小设置为与有相同参数的 GRect 内部相切
new GLine(x1, y1, x2, y2)	创建新的连接点(x1, y1)和(x2, y2)的 GLine 对象

所有图形对象的公用方法

object.setColor(color)	将对象的颜色设置为 color, 它是 java.awt 里的一个普通颜色名
object.setLocation(x, y)	将对象的位置改变为点(x, y)
object.move(dx, dy)	给 x 坐标增加 dx, 给 y 坐标增加 dy 来移动对象

GRect 和 GOval 专用的方法

object.setFilled(fill)	设置对象是否填充(true 表示填充, false 表示不填充)
object.setFillColor(color)	设置用于填充对象内部的颜色, 它可能与边界的颜色不同

GLabel 专用的方法

label.setFont(string)	将字体设置为 string 指定的 label, 它给出了字体族名、风格和磅值)
------------------------------	--

图 2-12 用 acm.graphics 程序包中一些有用的方法

2.6.4 GRect 类

GRect 类用于表示可以在图形窗口中显示的矩形框。使用所有 GObject 子类, 将 GRect 对象显示在屏幕上, 该过程由下面几步完成:

- (1) 使用 GRect 构造函数, 生成一个新的 GRect 类实例。很多情况下, 需要将对象存储在变量中, 便于后面引用它。
- (2) 给对象发送消息, 这些消息是确保对象正确显示所必需的。对于 GRect 类的对象而言, 最有用的方法是 setColor(在前面的 GLabel 示例中就已经知道这种方法)和 setFilled(它决定矩形是应该以实体颜色框出现还是以外在轮廓出现)。

(3) 使用 GraphicsProgram 类中的 add 方法，将新的矩形添加到图形窗口显示的图形对象中。

图 2-13 包含的简单程序称为 GRectExample，其输出如图 2-14 所示。

```
/*
 * File: GRectExample.java
 *
 * This program creates a new GRect object, sets it to be filled,
 * colors it red, and then displays it on the screen. The GRect
 * is 125 pixels wide by 60 pixels high, with its upper left
 * corner at the point (100, 50).
 */

import acm.graphics.*;
import acm.program.*;
import java.awt.*;

public class GRectExample extends GraphicsProgram {

    public void run() {
        GRect rect = new GRect(100, 50, 125, 60);
        rect.setFilled(true);
        rect.setColor(Color.RED);
        add(rect);
    }
}
```

图 2-13 显示红色矩形的程序

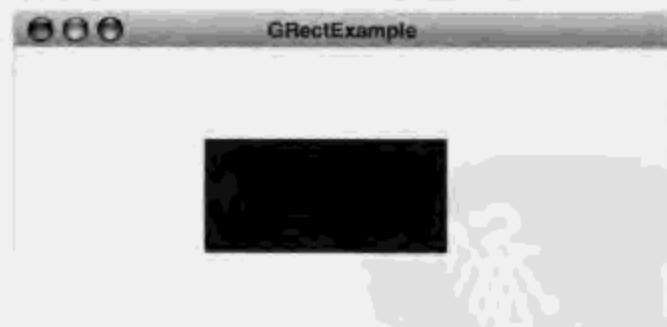


图 2-14 GRectExample 程序的输出结果

如果仔细审核 run 方法中的语句，即使有些细节仍然不清楚，但几乎肯定会遵循程序的逻辑。在 4 条语句中，run 方法创建一个新的 GRect 实例，设置为填充，颜色设置为红色，然后将它添加到图形窗口。

然而，代码有几个细节需要进一步解释说明。第一，setFilled 方法的参数如图 2-12 所示，它属于两个值中之一：true 或 false。这些值(称为布尔值)是极为重要的 Java 类型的实例，在第 3 章将进一步学习。这里，按照这两个值的常规英语解释理解它们就足够了。调用

```
rect.setFilled(true);
```

表示实际上应该填充矩形。相反，调用

```
rect.setFilled(false);
```

表示不应该填充它，只留轮廓线。默认时，GRect 构造函数创建的对象都没有填充。因此，如果完全让 GRectExample 程序不执行该语句，输出将会如图 2-15 所示。



图 2-15 GRectExample 程序的输出结果

该程序值得另外讨论的第二个方面是 GRect 构造函数。现在的情况是，构造函数的参数——数字 100、125 和 60——似乎有点难以理解，特别是如果用程序注释来解释它们。要理解这些参数的意义，必须知道构造函数需要什么值。随着对 Java 不断熟悉，最终会学会通过查看定义，或最好查看类的在线文档来理解每个参数的意义。这里，可以在图 2-12 中找到需要的文档，它指出 GRect 构造函数的参数是什么，从左到右依次为，对象位置的 x 坐标，对应的 y 坐标，对象的宽度和高度。

到目前为止一切顺利。知道 GRect 构造函数每个参数的意义，这会有助于了解矩形如何显示在屏幕上，但是也必须更好理解 Java 如何解释这些参数的值。特别是，必须知道 Java 坐标系的如下事实：

- 在 Java 中，位置(0, 0)在图形窗口的左上角。该点称为原点。因此，图形对象给出的位置通常指定其左上角的坐标。
- 与传统的笛卡尔坐标系相比，向下移动时，y 坐标的值会增加。坐标的值从左到右增加，和平常一样。
- 坐标和距离以相应的单点单元表达，这些单点构成显示，称为像素。

因此，构造函数调用

```
new GRect(100, 50, 125, 60)
```

指定了一个矩形，在图形窗口，其左上角距右 100 像素，左上角距下方 50 像素。两个主要参数指定了矩形的大小：该矩形宽为 125 个像素单元，高为 60 像素。绘出矩形相对于窗口的几何形状，如图 2-16 所示。

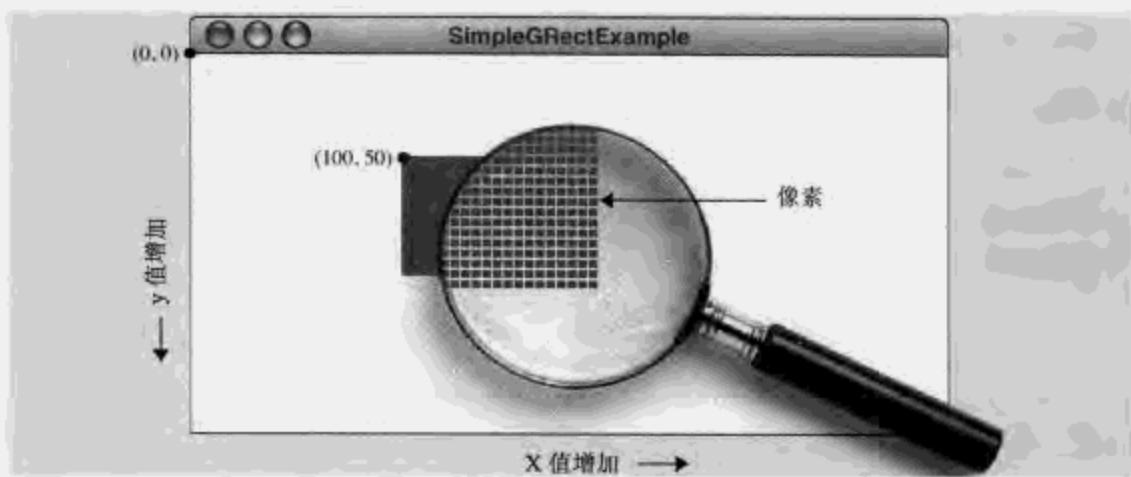


图 2-16 Java 坐标系

2.6.5 GOval 类

顾名思义，GOval 类用于在图形窗口中显示椭圆。在结构上，GOval 类与 GRect 类很相似。两个类的构造函数采用相同的参数，两个类也会对相同的方法产生响应。不同点是两个类在屏幕上生成的图形不同。GRect 类显示矩形，其位置和大小由参数值 x、y、width 和 height 确定。GOval 类显示椭圆，它的边界刚好与矩形的边界相切。

GRect 和 GOval 类的关系用示例更容易说明。图 2-17 中的 GRectPlusGOval 程序从前面的 GRectExample 程序中提取代码，并通过添加有相同坐标和尺寸的 GOval 来扩充它。结果输出如图 2-18 所示。

```
/*
 * File: GRectPlusGOval.java
 *
 * This program creates a GRect and a GOval using the same
 * parameters. The GRect is colored red; the GOval is outlined
 * in black but filled in green. The example illustrates that
 * the GOval fills the boundary set by the enclosing rectangle.
 */

import acm.graphics.*;
import acm.program.*;
import java.awt.*;

public class GRectPlusGOval extends GraphicsProgram {
    public void run() {
        GRect rect = new GRect(100, 50, 125, 60);
        rect.setFilled(true);
        rect.setFillColor(Color.RED);
        add(rect);
        GOval oval = new GOval(100, 50, 125, 60);
        oval.setFilled(true);
        oval.setFillColor(Color.GREEN);
        add(oval);
    }
}
```

图 2-17 显示红色矩形和绿色椭圆的程序

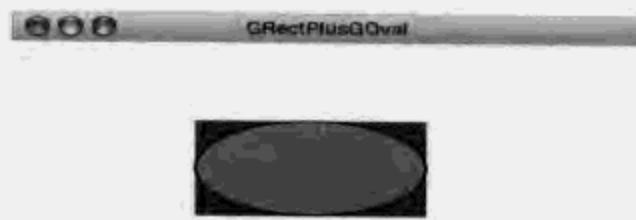


图 2-18 GRectPlusGOval 程序的输出结果

本示例中要注意的两件事：第一，绿色 GOval 扩充，以便其边界——轮廓为黑色，因为程序只改变 GOval 的填充颜色——与矩形的边界相切。第二，在 GRect 之后添加的 GOval 隐藏了矩形在椭圆边界下面的部分。如果以相反的顺序添加这些图形，就只能看到 GRect，因为整个 GOval 都会被覆盖在 GRect 的边界下面。

2.6.6 GLine 类

很多情况下，GLine 是图 2-11 中类图最简单的 GObject 子类。GLine 构造函数有 4 个参数，分别是两个端点的 x 坐标和 y 坐标。因为填充线条或设置字体没有意义，所以 GLine 类不包括像 setFilled 或 setFont 这样的方法。在看到的方法中，唯一能应用于 GLine 实例的是 setColor，它可以改变线条的颜色。

图 2-19 是一个名为 TicTacToeBoard 的程序，它可以画出用于玩 Tic-Tac-Toe 游戏的#形线，如图 2-20 所示。

```
/*
 * File: TicTacToeBoard.java
 *
 * This program draws a Tic-Tac-Toe board as an illustration
 * of the GLine class. This version uses explicit coordinate
 * values which makes the program difficult to extend or
 * maintain. In Chapter 3, you will learn how to use constants
 * and expressions to calculate these coordinate values.
 */

import acm.graphics.*;
import acm.program.*;

public class TicTacToeBoard extends GraphicsProgram {
    public void run() {
        add(new GLine(30, 60, 120, 60));
        add(new GLine(30, 90, 120, 90));
        add(new GLine(60, 30, 60, 120));
        add(new GLine(90, 30, 90, 120));
    }
}
```

图 2-19 显示 TicTacToeBoard 的程序

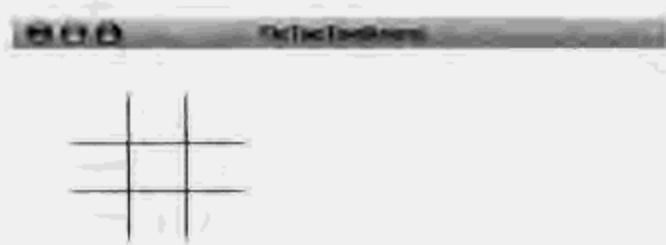


图 2-20 TicTacToeBoard 程序的输出结果

该程序的 run 方法只给图片添加了 4 个 GLine 对象。第一个连接点(30, 60)和(120, 60)，第二个连接点(30, 90)和(120, 90)，等等。前两行的 y 坐标相同，因此是水平的；后两行的 x 坐标是相同的，因此是垂直的。

该程序的存在问题是明确指定所有的坐标值。使用明确的坐标值使程序难以编写、阅读和维护。例如，怎样知道第一条水平线的端点应该是(30, 60)和(120, 60)？最简单的方法就是在方格纸上画一个要产生的图片，然后用直尺量出坐标值和尺寸值。所有这些都很单调乏味，很容易出错。真正要做的是用一组较小的参数来计算这些值，这些参数共同定义了图形的整体尺寸。例如，在 Tic-Tac-Toe 案例中，知道水平直线应该将垂直直线分成 3 份，反之亦然。因此，只要将图形分成 3 份，就可以计算出交叉点的坐标。然而要这样做，需要理解如何使用 Java 表达式指定这种运算，这将在第 3 章学习。

2.7 小结

本章有机会查看了几个完整的 Java 程序，对其整体结构和工作方法有了一定的了解。这些程序的细节稍后讨论：这里，首要目标应该是用整体的观点着重研究编程过程。即便这样，利用目前提供的编程示例基础，应该能够编写包含如下操作的简单程序。

- 调用方法 `readInt` 和 `readDouble`，读取用户输入的数值。在 `ConsoleProgram` 中，用户在程序创建的基于文本的窗口中输入数据；在 `DialogProgram` 中，用户在交互式对话框中输入数据。
- 调用 `println` 方法，为用户显示信息。
- 通过创建 `GRect`、`GOval`、`GLine` 和 `GLabel` 类实例生成图形程序。

关于编程，本章介绍的重点是：

- 规范编写的程序包含注释，它用英语解释程序在做什么。
- 大多数程序使用提供工具的程序包，编程人员不需要从头重新创建这些工具。本章的程序使用了两个程序包：`acm.program` 和 `acm.graphics`。后续章将介绍其他程序包。
- 在程序顶部包括程序包的 `import` 行就可以访问程序包。
- 本书中的 Java 程序通常由类定义组成，此类定义扩充 `acm.program` 程序包中的一个类。这个类称为主类。
- 每个主类都包含一个称为 `run` 的方法。执行程序时，按顺序执行 `run` 主体里的语句。
- 类的作用相当于模板，用于创建代表类里单个实例的对象。一个类可以产生许多对象，这些对象都是该类的实例。然而，每个对象都是一个特定类的实例。

- 指定关键字 new(随后是对类构造函数的调用), 可以创建新的实例。
- 类形成反映 extends(扩充)关系的层次结构。如果类 A 扩充类 B, 那么 A 是 B 的子类, B 是 A 的超类。
- 子类继承其超类的行为。
- acm.program 里的 Program 类有 3 个已定义的子类: GraphicsProgram、ConsoleProgram 和 DialogProgram。
- acm.graphics 中的 GObject 类有许多有用的子类。第 9 章之前, 虽然没有机会了解细节, 但是可以使用 GLabel、GRect、GOval 和 GLine 类来创建简单的图片。
- 在 Java 等面向对象语言中, 影响对象的标准方式是向它发送消息。这种消息使用方法调用执行, 形式如下:

```
receiver.name(arguments)
```

- 所有 GObject 子类都会响应 setColor 方法, setColor 方法可以改变对象显示的颜色。因为这种行为每个子类都可以共享, 因此应该在 GObject 类级定义这种行为。
- 一些 GObject 子类定义专门用于该子类的方法。例如, GLabel 类定义方法 setFont, 它改变标签显示的字体。GRect 和 GOval 类定义方法 setFilled, 它决定图形填充为实心颜色还是显示为轮廓。
- Java 使用坐标系, 其原点在图形窗口的左上角; 向下移动时, y 坐标值增加。所有坐标和距离都以像素的形式表达, 像素是填充屏幕表面的单个点。

2.8 复习题

1. 本章每个程序开头显示的注释的目的是什么?
2. 库程序包有什么作用?
3. Java 程序在 acm.program 程序包的控制下开始时, 执行方法的名称是什么?
4. 编程中单词“参数”指什么? 参数的目的是什么?
5. 描述 println 方法的功能。名称结尾的字母 ln 有什么重要意义?
6. readInt 方法的目的是什么? 在程序中如何使用它?
7. 本章描述了 Java 程序中+运算符的两种用法。这两种用法是什么? Java 如何决定使用哪种解释?
8. 描述哲学术语整体论和简化论的不同。这些概念对于编程而言为什么很重要?
9. 类和对象之间有何不同?
10. 给出术语子类、超类和继承的定义。
11. 什么 Java 关键字与构造函数的使用有关?
12. ConsoleProgram 和 DialogProgram 之间有何不同?
13. 判断: 向 Java 对象发送消息的过程通常通过调用对象里的方法完成。
14. Java 中, 如何指定消息发送的对象?
15. 本章描述的 4 个 GObject 子类是什么?
16. 这些子类中哪个会响应 setFilled 方法? 哪个会响应 setFont 方法?
17. 除这里列出的 4 个子类之外, 第 9 章还将学习另外几个 GObject 子类。这些子类会响

应方法 `setColor` 吗？

18. Java 坐标系与传统的笛卡尔坐标系有哪些不同？

2.9 编程练习

1. 正确输入本章出现的 `HelloProgram.java` 程序，并运行它。改变消息，让它显示为“`I love Java`”。在右下角添加您的名字作为签名。

2. 下面的程序除了一些输入提示符之外，没有给用户提供注释或说明：

```
import acm.program.*;

public class MyProgram extends ConsoleProgram {
    public void run() {
        double b = readDouble("Enter b: ");
        double h = readDouble("Enter h: ");
        double a = (b * h) / 2;
        println("a = " + a);
    }
}
```

仔细阅读程序，理解它在做什么。它计算的结果是什么？重写程序，以便让用户和将来修改程序的编程人员更容易理解。

3. 扩充图 2-4 所示的 `Add2Integers` 程序，让它对 3 个整数进行相加。

4. 编写 `GraphicsProgram` 程序，让它生成下面图 2-21 所示的房子图片。这个房子是个尖屋顶，有两扇窗和一个带圆形把手的门。



图 2-21 房子图片

使用本章的每个图形练习，您应该可以为程序选择坐标值，这些程序产生所显示图的合理近似值。第 3 章我们将学习计算坐标值的方法。例如，房子在图形窗口居中，门居中于房子框架，窗户水平居中于门和两堵墙之间。

5. 编写 `GraphicsProgram` 程序，使其画出图 2-22 所示的机器人脸图片。

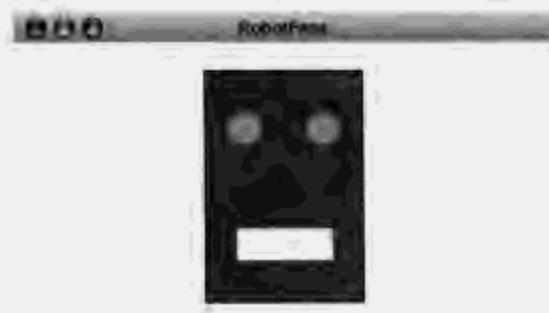


图 2-22 机器人脸

眼睛是橙色的，鼻子是黑色的，嘴是白色的。脸部填充为灰色，但边框是黑色的。

6. 编写 GraphicsProgram 程序，使其画出图 2-23 所示箭靶，它恰巧也是大降价的徽标。



图 2-23 箭靶

内环和外环都填充为红色。

7. 编写一段 GraphicsProgram 程序，使其画出组成奥运会标志的 5 个环(蓝色、黄色、黑色、绿色和红色)，如图 2-24 所示。

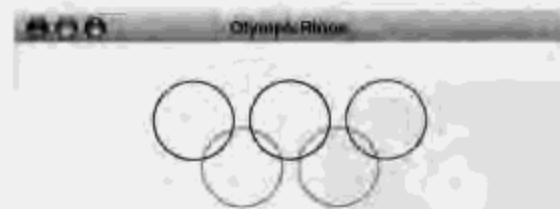


图 2-24 奥运会五环

8. 在许多输出设备上，前面练习中的奥运会标识有的显示效果不好，因为黄色圆圈(比绿色稍浅一些)在白色背景的窗口上显示不出来。部分原因是 Java 画的 GOval 类的轮廓只有 1 个像素宽，在浅颜色上显示就不是很清楚。如果边界有 3 个像素宽，那么就很容易看见这些环，如图 2-25 所示。

如果 GOval 类包含一种方法能够设置边界宽度，改变起来就很容易。但是，还没有这种方法。稍微考虑一下这个问题，看看能否找到一种策略，只使用目前学到的工具来创建一个 3 像素宽的显示图片。

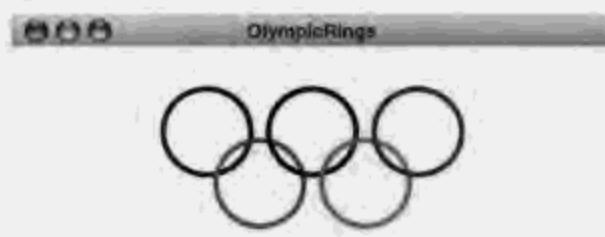


图 2-25 奥运会五环

9. 编写 GraphicsProgram 程序，使其画出彩虹图片，如图 2-26 所示。

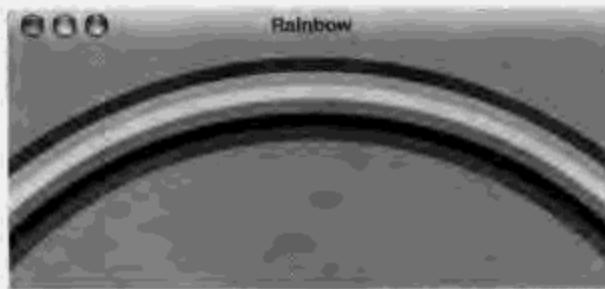


图 2-26 彩虹

从上至下，彩虹的 6 条条纹分别是红色、橙色、黄色、绿色、蓝色和洋红色；天空是可爱的青色。

乍一看，问题好像是要求画弓形。事实上，使用圆就可以创建彩虹条纹。这怎么可能呢？想一想这个问题可以拓宽您的思维——运用字面的和比喻的判断。每个圆的公共中心都在窗口底部以下的某个地方，圆的直径比屏幕要宽，GraphicsProgram 只显示窗口中实际出现的图形部分。减少图片可见区域的过程称为剪辑。



第3章

表达式

[The Analytical Engine offers] a new, a vast, and a powerful language... for the purposes of mankind.

—Augusta Ada Byron, Lady Lovelace, 1843



George Boole(1815—1864)

虽然 George Boole 主要通过自学成材，从未获得过正规大学的学位，但是他取得了巨大成就，也因此被任命为爱尔兰科克郡皇后大学的数学教授，并当选英国皇家学会会员。他最具影响力的著作是 1854 年出版的名为 *An Investigation into the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities* 一书。这本书介绍了一种现在称为 Boolean(布尔)代数学的逻辑体系，它是本章讲述的 boolean 数据类型的基础。

第 2 章已经介绍了几个简单程序——Add2Integers 和 Add2Doubles，它们执行代数运算。第 2 章的目的就是让您了解什么是程序类型。本章的目的就是介绍一些基本规则，这些规则允许用户编写指定运算的 Java 程序。在学习过程中，要将对细节的简化理解和对程序的整体理解结合起来。其中每个程序，实际计算都由下面的代码指定

```
n1 + n2
```

很显然，它说明计算机应该相加存储在变量 n1 和 n2 里的值。变量与+运算符的结合就是表达式

的示例。表达式是用编程语言对产生期望结果所需要的操作顺序的符号表达。为了使编程对人类而言相对容易，这些表达式具有句法形式，这些句法形式与传统数学公式里使用的表达法非常相似。的确，第一种包含表达式语法的编程语言FORTRAN，其名称就是词语公式转换(formula translation)的缩写，因为这样的语言被广泛认为是机器语言编程的主要优势。

和大多数语言一样，Java 的表达式也由项和运算符组成。项——例如第 2 章表达式中的 n1 或 n2——表示单个数据值。运算符(例如+)是表示计算操作的字符(有时是短的字符序列)。在表达式中，项一定是下面几种之一。

- 常量。作为程序一部分出现的明确的数据值称为常量，如 0 或 3.14159 这样的数字就是常量。
- 变量。变量是数据的占位符，这些占位符在程序执行过程中可以改变。
- 方法调用。通过调用其他方法(很可能来自库程序包)生成的值，这些方法可以将数据值返回到原始表达式。例如，在 Add2Integers 程序中，方法 readInt 用于读取每个输入值；所以方法调用 readInt 是术语的项。第 5 章将详细讨论方法调用。
- 圆括号里的表达式。圆括号可用来表示表达式里的分组，和在数学里一样。从编译器的角度而言，圆括号里的子表达式也是项，所以将它当成单元对待。

程序运行时，完成表达式里每个指定操作的过程称为赋值。计算机为 Java 表达式赋值时，它会将每个运算符应用于周围项所代表的值。所有运算符都被赋值以后，剩下的就是表示运算结果的单个数据值。例如，表达式

```
n1 = n2
```

赋值过程就是读取变量 n1 和 n2 的值，然后将它们相加。赋值结果就是和。

3.1 原始数据类型

讨论表达式之前，了解一下表达式使用的数据类型会很有帮助。本书中首先会用到数据对象，它们是类的代表。这些对象和类表示面向对象范例(Java 建立在它之上)的定义特征。然而，表达式使用的是较简单的数据类型，至少在 Java 中，这些数据类型处于对象层次结构之外。这些数据类型被定义为语言的一部分，都是原始数据类型，用这些原始数据类型可以创建更复杂的对象。

要在各种应用程序中都有用，程序必须能够存储多个不同类型的数据。如第 2 章所示，可以编写像 Add2Integers 这样使用整数的程序；也可以编写像 Add2Doubles 这样的程序，它可以使用户有小数部分的数，如 1.5 或 3.1415926。使用字处理程序时，单个数据值是字符，这些字符随后组成较大的单元，如单词、句子和段落。随着程序越来越复杂，就要逐步使用以不同方法构建的信息集成。所有这些不同类的信息组成了数据。

无论什么时候使用数据——可能是整数或者是有小数部分的数，或者是字符——Java 编译器都需要知道其数据类型。从整体上说，两种属性定义数据类型：一组值，或者域；一组操作。域就是一组值，这些值是某种类型的元素。例如，int 类型的域包括所有整数(-2、-1、0、1、2)直到 Java 语言定义的极限为止。对于字符数据，域是显示在键盘上或者可以在终端屏幕上显示的一组符号。操作集包含一些已有工具可以处理的类型的值。例如，考察两个整数，可能将它们相乘或相除。另一方面，如果考察文本，很难想象乘除这样的运算有什么意义。相反，我们希望运用这样的操作：比较两个单词，看看字母顺序是否正确；或者在屏幕上显示消息。因此，操作必须与域的元素相适应，域和操作这两者一起定义数据类型。

Java 定义了 8 种原始数据类型, 如图 3-1 所示。前 4 种类型——byte、short、int 和 long——表示有不同最小值和最大值的整数, 这些最小值和最大值反映了存储这些整数的存储单元的容量。接下来的两种类型——float 和 double——表示不同动态范围内的浮点数。除了少数情况下库方法要求使用其他类型之外, 本文都使用 int 和 double 作为标准数字类型。char 类型用于表示字符数据, 将在第 8 章介绍。boolean 类型(第 2 章简要讨论过)对于编程而言至关重要, 本章稍后将重点介绍。

类型	域	常见运算符
byte	范围为 128~127 的 8 位整数	算术运算符 + 加 * 乘 - 减 / 除 % 取余
short	范围为 -32 768~32 767 的 16 位整数	关系运算符 == 等于 != 不等于 < 小于 <= 小于或等于 > 大于 >= 大于或等于
int	范围为 -2 147 483 648~2 147 483 647 的 32 位整数	除 % 之外的算术运算符
long	范围为 -9 223 372 036 854 775 808~9 223 372 036 854 775 807 的 64 位整数	关系运算符
float	范围为 $\pm 1.4 \times 10^{-45}$ ~ $\pm 3.4 \times 10^{38}$ 的 32 位浮点数	逻辑运算符 与 或 非
double	范围为 $\pm 4.39 \times 10^{-322}$ ~ $\pm 1.7976 \times 10^{308}$ 的 64 位浮点数	
char	使用 Unicode 的 16 位编码字符	
boolean	值为 true 和 false	

图 3-1 Java 的原始数据类型

除了图 3-1 所示的原始数据类型之外, Java 中通常将 String 类型当成原始类型, 虽然它实际上是程序包 java.lang 里定义的类。把它当成原始类型的理由之一是, String 嵌入 Java 语言里的方法与原始类型一样。例如, Java 为字符串常量指定特定语法, 这与指定 int、double 或 boolean 是相同的。将 String 作为原始类型一个更重要的原因是, 这样做有利于用更全面的方式看待字符串。在第 8 章会发现, String 类定义了执行各种有用操作的许多方法。虽然理解这些方法的细节、知道使用它们的方式很重要, 但是字符串值与整数值非常相似(除了这两种类型的域不同之外), 这样想可能会便于理解。可以将变量声明为 int 类型, 并给它指定一个整数值; 同样, 也可以将变量声明为 String 类型, 并给它指定一个字符串值。只有需要使用 String 类提供的方法时, 才会有 String 是类还是原始类型这样的区别。

3.2 常量与变量

表达式的项中最简单的两种类型是常量和变量。当需要使用明确的值, 并且这个值在程序执行过程中不会改变时, 通常使用常量。变量是数据的占位符, 它在执行过程中会发生改变, 接下来将介绍如何书写常量和变量, 详细讨论 Java 使用这些规则的方法。

3.2.1 常量

写数学公式时, 公式中有些符号通常表示未知值, 而其他的符号则是代表已知值的常量。

例如，要计算给定半径(r)的圆的周长(C)，考察该数学公式：

$$C = 2\pi r$$

要将公式转换为表达式，需要用变量来表示半径和周长。这些变量随着数据的改变而改变。然而，值 2 和 π 是常量，它们有明确的值，不会改变。值 2 是整数常量，值 π 是实数常量，它是一个浮点型近似值，如 3.14 159 265 358 979 323 846。因为常量是构造表达式的重要组成部分，所以学会书写每种基本数据类型的常量值就非常重要。

- 整数常量 要将整数常量作为程序的一部分或输入数据，只要写出组成数的数字就行了。如果整数是负数，就要在数字前加上负号，和数学中一样。中间绝对不要用逗号。因此，值一千万应该写作“1000000”，而不是“1,000,000”。
- 浮点常量 Java 中，浮点常量用小数点表示。因此，如果程序中出现 2.0，它在内部表示浮点值；如果编程人员写“2”，这个值则表示整数。浮点值也可以用科学计数的特殊编程人员风格来写，其中的值表示为浮点值乘以 10 的整数幂。要用这种风格写数字，先用标准计数法写一个浮点数，紧接着写字母 E 和整数指数，前面可以加上正号或负号。例如，光速(用米每秒表示)大概是

$$2.9979 \times 10^8$$

Java 中可写成：

$$2.9979e+8$$

其中 E 代表“乘以 10 次幂”。

- Boolean 常量 只有两种 boolean 类型的常量：Java 关键字 true 和 false。也有字符常量，这将在第 8 章介绍。然而，还需要知道另一个其他形式的常量，虽然它不是原始类型：
- 字符串常量 将组成字符串的字符封装在双引号内，这样在 Java 中就写出了字符串常量。例如，本书中使用的第一个示例就是第 2 章 HelloProgram 示例中的字符串

`"hello, world"`

这个字符串由双引号内显示的字符组成，包括字母、逗号和空格。引号不是字符串的一部分，它只是用来标记字符串的开始和结束。写字符串常量有其他一些规则，这些规则允许在字符串内包括特殊字符(例如引号)。第 8 章将详细讨论这些规则。

3.2.2 变量

变量是值的占位符，它有 3 个重要属性：名称、值和类型。要理解这些属性之间的关系，最简单的方法是将变量看作一个外部贴有标签的方框。变量名称写在标签上，用来分辨不同的方框。如果有 3 个方框(或变量)，使用名称就可以引用特定的那个。变量值相当于方框里的内容。方框标签上的名称是固定的，但是可以任意改变方框的值。变量的类型表示什么类型的值可以存储在方框里。例如，如果设计一个方框用来保存 int 类型的值，那么 String 类型的值就不能保存在这个方框里。

Java 中变量或其他类型元素(例如类和方法)的名称称为标识符。构建标识符必须符合如下

规则：

- 标识符必须以字母或下划线字符(_)开始。在 Java 中，标识符中区分大小写字母，因此，名称 ABC、Abc 和 abc 是 3 个单独的标识符。
- 标识符中的其他所有字符必须是字母、数字或下划线。不允许有空格或其他特殊字符。标识符长度不限。
- 标识符不能是下面的预定字，这些预定字在 Java 中被定义用作特殊目的。

abstract	else	interface	super
boolean	extends	long	switch
break	false	native	synchronized
byte	final	new	this
case	finally	null	throw
catch	float	package	throws
char	for	private	transient
class	goto	protected	true
const	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while
double	int	strictfp	

另外，好的编程风格还要求符合另外两条规则，即：

- 标识符应该将变量、类或方法的目的明确地告诉读者。虽然友元的名称、感叹词等按照其他规则可能是合法的，但它们对提高程序的可读性完全毫无帮助。
- 标识符应该符合已成为 Java 标准的约定，特别是变量名称应该以小写字母开始，类名称应该以大写字母开始。因此，nl 作为变量名就比较合适，HelloProgram 作为类名称就很合适。名称中出现的每个附加英语单词通常是以大写字母写的，用来提高可读性，例如变量名称 numberOfStudents。用作常量的名称(3.2.4 小节将讨论此问题)只能使用大写字母和带下划线的单个单词，如 PLANCKS_CONSTANT。

3.2.3 声明

正如第 2 章 Add2Integers 程序所述，将变量引入程序时，必须明确指定每个变量的数据类型，这个过程称为声明变量。声明变量的语法如右边的方框所示。行

```
type identifier = expression;
```

是语法模板的示例。斜体字表示选项，在这些选项中可以填充符合模板规范的所有内容。例如，写赋值语句时，可以声明变量的类型，在等号左边使用任意标识符，在等号右边写任意表达式。模板中的黑体选项——这里是等号和分号——是固定的。因此，编写声明要按照这样的顺序：先写类型名称，然后依次是变量名称、等号、表达式和分号。

声明的语法

```
type identifier = expression;
identifier(标识符)指变量的名称, expression(表达式)指定初始值。
```

本书引入新的句法结构时，都会伴随有一个定义其结构的语法框，例如本节开头详细讨论

声明结构的语法框。语法框汇集了 Java 语法规则概要，可用作随身参考。

可以在程序的几个不同部分声明变量。到目前为止所看到的变量都在方法的主体内声明。(到目前为止，方法都是 `run`，但是在任何方法中声明变量都是合法的。)在方法里声明的变量称为局部变量，因为只是那种方法可用，其他部分则不可用。然而，可以在类的定义内而又在所有方法之外声明变量。这样定义的变量称为实例变量(简称 ivars)，作为每个对象的一部分而存储。使用实例变量时要小心，在学习更多类结构(将在第 6 章讨论)之前最好避免使用。

除了局部变量和实例变量以外，Java 支持称为类变量的第三种变量声明风格。顾名思义，类变量在类级别进行定义而不是在特定方法或对象级别进行定义。类变量与实例变量的声明方法很类似；唯一不同之处是类变量将关键字 `static` 作为声明的一部分。使用类变量时也要注意，但是它们又是引入常量名称的合适工具，3.2.4 小节将讨论这个问题。

3.2.4 命名常量

编写程序时，会发现在程序中经常多次使用相同的常量。例如，如果执行涉及到圆的几何运算，就经常会出现常量 π 。而且，如果这种运算要求高精度，实际上可能要求所有数字满足 `double` 类型的值，这意味着要用值 3.14 159 265 358 979 323 846 进行运算。重复写这个常量非常乏味，如果每次都手工输入而不是剪切和粘贴，则很可能出错。如果给常量定义一个名称，在程序中任何地方都可以通过名称引用它，就会好得多。当然，只要声明它为局部变量就可以了，如下所示：

```
double pi = 3.14159265358979323846;
```

但只能在定义它的方法内使用。更好的策略是声明它为类变量，如下所示：

```
private static final double PI = 3.14159265358979323846;
```

每个声明开头的关键字都提供了一些有关声明本质的信息。`private` 关键字表示该常量只能在定义它的类里使用。将常量声明为 `public` 通常更有意义，但是编程实践说明，应该让所有声明都保持私有(声明 `private`)，除非有强制性理由要求不这么做。`static` 关键字表示声明引入了类变量而不是实例变量。`final` 关键字声明，变量初始化后值不会改变，进而确保值不变。毕竟，改变 π 的值不合适。声明的其余部分由类型、名称和值组成，和前面一样。唯一不同是，完全用大写字母写名称与 Java 的命名计划是一致的。

使用命名常量有几个好处：首先，描述性常量名称通常让程序更易于阅读。更重要的是，使用常量可以极大简化维护程序代码涉及到的问题。3.6 节将详细讨论此过程。

3.3 运算符和操作数

表达式中，连接单个项的符号运算符表示实际的运算步骤。要定义的最简单的运算符是那些用于算术运算表达式的运算符，算术运算表达式使用标准的算术运算符。适用于所有数字数据类型的算术运算符有：

- 加
- 减(或负数，如果它的左边没有值)
- 乘

/ 除

连接两个较小的表达式，一个在运算符的左边，一个在右边，这样这些运算符就形成了新的表达式。运算符应用的这些补充表达式(或者子表达式)称为它的操作数。例如，表达式中

$x + 3$

加号运算符(+)的操作数是子表达式 x 和 3。操作数通常是单个项，但也可能是更复杂的表达式。例如，表达式

$(2 * x) + (3 * y)$

中，加的操作数是子表达式 $(2 * x)$ 和 $(3 * y)$ 。

和常规数学里一样，运算符 - 可用于两种形式。它位于两个操作数之间时，表示减，例如 $x - y$ ；当它的左边没有操作数时，表示负数，例如 $-x$ 表示值 x 的负数。以这种方式使用时，- 运算符称为一元运算符，因为它应用于单个操作数。其他运算符(包括 -，表示减法时)称为二进制运算符，因为它们应用于一对操作数。

只要将加、减、乘和除加入到您的指令表，运用这些运算符就有编写能编写出一些程序，这些程序可以计算出比 Add2Doubles 程序更有趣、更有用的结果。例如，有了这些运算符的更大集合，就有可能写出一段程序，将以英寸(in)作为单位的长度转换为以厘米(cm)作为单位的相同长度。所有真正需要知道的就是 1(in)等于 2.54(cm)；从第 2 章的 Add2Doubles 示例选取一些行，然后将它们以合适的方式放在一起，这样可以构建程序的剩余部分。图 3-2 显示了最终结果。

```
/*
 * File: InchesToCentimeters.java
 *
 * This program converts inches to centimeters.
 */
import acm.program.*;

public class InchesToCentimeters extends ConsoleProgram {
    public void run() {
        println("This program converts inches to centimeters.");
        double inches = readDouble("Enter value in inches: ");
        double cm = inches * CENTIMETERS_PER_INCH;
        println(inches + "in = " + cm + "cm");
    }
    /* Private constants */
    private static final double CENTIMETERS_PER_INCH = 2.54;
}
```

图 3-2 将英寸(in)转换为厘米(cm)的程序

3.3.1 合并整数和浮点数

Java 中，int 类型与 double 类型的值可以自由组合。如果将二进制运算符应用于两个 int 类型的值，结果也是 int 类型；如果有一个或两个操作数是 double 类型，那么，结果一定是 double 类型。因此，表达式的值

`n * 1`

是 int 类型，如果变量 n 声明为 int 类型。另一方面，表达式

`n + 1.5`

一定是 double 类型。这种约定可以确保运算结果尽可能精确。例如，在表达式 `n + 1.5` 中，如果结果使用整数算法就无法表示出 .5。

3.3.2 整数除法和余数运算符

从 3.3.1 小节可以看到，将算术运算符应用于整数操作数总是产生整数结果。关于除法，则会发生一种有趣的情况。如果写如下所示的表达式

`9 / 4`

Java 规则指定该操作的结果必须是整数，因为两个操作数都是 int 类型。程序给表达式赋值时，它用 9 除以 4，舍去余数。因此，表达式的值是 2，而不是 2.25。从数学上说，如果要计算正确的结果，就至少有一个操作数必须是浮点数(double 类型)。例如，下面 3 个表达式

`9.0 / 4``9 / 4.0``9.0 / 4.0`

每一个都产生浮点数 2.25。只有当两个操作数都是 int 类型时，才会舍去余数。

计算余数有另外一个算术运算符，Java 中用百分比符号(%)表示。% 运算符要求两个操作数都是 int 类型。用第一个操作数除以第二个操作数，返回余数。例如，下面这个表达式的值是 1，

`9 % 4`

因为 9 除以 4，剩余 1。下面是其他一些% 运算符的示例：

`0 % 4=0``1 % 4=1``4 % 4=0``19 % 4= 3``20 % 4= 0``2001 % 4= 1`

事实证明，在大量编程应用程序中，/ 和 % 运算符非常有用。例如，% 运算符通常用于测试一个数能否被另一个数除尽。例如，要确定整数 n 能否被 3 除尽，只要检查表达式 `n % 3` 的结果是不是 0 就可以了。

注意，当% 的一个或两个操作数为负数时，要知道结果是什么就不是那么容易了。Java 确实定义了那些情况下的结果，但是所开发的定义与机器体系结构的工作方式相一致，遗憾的是，这与数学家的期望不相符。为了防止混淆，本文避免与负数操作数一起使用%，在您自己的代码里采取相同的约定也很有必要。

3.3.3 优先级

如果表达式有多个运算符，应用这些运算符的先后顺序就成了一个重要问题。在 Java 中，总是可以用圆括号包含单个子表达式来指定顺序，这和传统数学中一样。例如，下面这个表达式中的圆括号

$(2 * x) + (3 * y)$

表示, Java 应该在做加法之前执行所有乘法运算。但如果没有圆括号, 情况又会怎样呢? 假设表达式如下所示:

$2 * x + 3 * y$

Java 编译器如何确定应用单个运算的顺序呢?

在 Java 及其他许多编程语言中, 一组符合标准数学用法的排序规则决定了运算的顺序。它们称为优先级规则。对于算术表达式而言, 规则有:

- (1) Java 编译器首先应用所有一元负数运算符(左边没有操作数的负号)。
- (2) 编译器应用乘法运算符(*、/和%)。如果两个这样的运算符应用于同一个操作数, 最左边的最先执行。
- (3) 应用加法运算符(+和-), 如果这一优先级的两个运算符应用于相同的操作数, Java 从最左边的一个开始。

因此, 在表达式

$2 * x + 3 * y$

中, 即使没有圆括号, 乘法运算符也最先执行。使用圆括号可以让顺序更清楚, 但是这里不要求使用, 因为运算预定的顺序与传统数学的优先假设是一致的。如果要首先执行加法, 就必须用圆括号明确表示出来, 如下所示:

$2 * (x + 3) * y$

只有当两个运算符“竞争”单个操作数时, 才会应用优先级规则。例如, 下面的表达式

$2 * x + 3 * y$

中, 运算符*和+竞争操作数 x。优先级规则规定首先执行*运算符, 因为乘法比加法有更高的优先级。同样, 考察值 3 的两个操作数, 也可以确定先执行*运算符, 理由和前面一样。在运算符实际没有竞争相同的操作数的情况下, 像这个表达式中的两个乘法运算一样, Java 按从左至右的顺序计算。因此, Java 会先执行 2 乘以 x, 再执行 3 乘以 y。

优先级规则可以产生完全不同的表达式赋值结果。例如, 考察表达式

$10 - 5 - 2$

因为 Java 的优先级规则规定最左边的首先执行, 所以运算会执行, 好像表达式和下面一样:

$(10 - 5) - 2$

其结果是值 3。如果想用其他顺序先执行减法, 就必须使用明确的圆括号:

$10 - (5 - 2)$

这种情况下, 运算结果是 7。

有许多情况需要使用圆括号来得到想要的结果。例如, 假设不是用 Add2Doubles 这样的方法相加两个浮点数, 而是求它们的平均值, 程序十分类似, 如图 3-3 所示。

```

/*
 * File: Average2Doubles.java
 *
 * This program averages two double-precision floating-point numbers.
 */

import acm.program.*;

public class Average2Doubles extends ConsoleProgram {
    public void run() {
        println("This program averages two numbers.");
        double n1 = readDouble("Enter n1: ");
        double n2 = readDouble("Enter n2: ");
        double average = (n1 + n2) / 2;
        println("The average is " + average + ".");
    }
}

```

图 3-3 求两个双精度数平均值的程序

注意，下面这条语句中的圆括号必不可少：

```
double average = (n1 + n2) / 2;
```

它用来确保先执行加法，后执行除法。如果没有用圆括号，Java 的优先级规则会规定先执行除法，结果可就是数学表达式

$$\frac{n1+n2}{2}$$

而不是

$$\frac{n1}{2} + n2$$

3.3.4 应用优先级规则

要实际说明优先级规则，请考察表达式：

```
8 * (7 - 6 + 5) % (4 + 3 / 2) - 1
```

假设您就是计算机。如何为该表达式赋值呢？

第一步是按照从左至右的顺序求出圆括号内子表达式的值。要计算 $(7-6+5)$ 的值，从 7 减去 6 等于 1，然后加上 5 等于 6。这样，给求出第一个子表达式的值以后，剩下的就是

$8 * \boxed{6} \% (4 + 3 / 2) - 1$

其中方框说明值是前面计算出的表达式的结果。

然后继续计算第二个括号内子表达式的值。此时，必须先做除法，因为除法和乘法优先于加法。因此，第一步应该是计算 3 除以 2，结果是 1(别忘了整数除法舍去余数)。然后相加 4 和 1，等于 5。此时，表达式如下所示。

8 * 6 % 5 - 1

在该计算中, Java 的优先级规则规定按顺序先执行乘法和余数运算, 然后做减法。6 乘以 8 等于 48, 48 除以 5 余数是 3。最后一步是减去 1, 整个表达式的结果是 2。

3.3.5 类型转换

我们已经知道, 可以合并 Java 程序里不同数字类型的值。这样做的时候, Java 使用自动类型转换这一过程来处理这种情况。在此过程中, 编译器将一种类型的值转换为另一种相容类型的值作为计算的隐式部分。例如, 使用算术运算符结合整数和浮点值时, 应用操作之前, Java 自动将整数转换为数学上相等的 double 类型。因此, 如果写表达式:

`1 + 2.3`

在将两个值相加之前, Java 会将整数 1 转换为浮点数 1.0。

在 Java 中, 当赋给变量限制性更强的类型的值时, 也会使用自动类型转换。例如, 如果编写声明:

```
double total = 0;
```

在将整数 0 赋值给变量 total 之前, Java 将它转换为 double 类型。有些编程语言(有些编程人员)会坚持将该语句写成:

```
double total = 0.0;
```

从其结果来看, 该语句的意思与前面的语句一样。另一方面, 对于数学家而言值 0 和 0.0 是不一样的。写值 0 表示值确实是 0, 因为整数是精确的。然而, 当 0.0 出现在统计或数学文本中时, 通常的解释是它表示一个和 0 很接近的数, 但是其精确度只到小数点之后一位有效数。为避免这样的混淆, 本文使用整数表示精确度, 即便是在浮点上下文中也如此。

然而, 在 Java 中使用 double 类型的值作为 int 类型变量的初始值是不合法的。使用称为类型强制转换的语法结构可以不受此限制, 它由括号里所需的类型(后面紧接着要转换的值)组成。因此, 如果写声明:

```
int n = (int) 1.9999;
```

在将 1.9999 作为 n 的初始值之前, Java 会将它转换为整数。然而, 您一定会奇怪, 它选择的整数是 1 而不是 2, 将值从浮点型转换为整数表示——在类型强制转换和整数除法中都会发生——会舍去所有小数。这种形式的转换称为切断。

作为切断如何有用的示例, 假设有人要求您写一段程序, 将用厘米(cm)表示的公制距离转换为英语单位——图 3-2 中 InchesToCentimeters.java 程序的反向计算。如果所有需要的就是英寸(in)数量, 程序的主体和前面很相似, 计算中唯一不同是除以而不是乘以 CENTIMETERS_PER_INCH。

然而, 假设您的老板要求显示的答案不仅仅是英寸总数, 而是要求显示英尺整数加上余下的英寸数。要计算整个英尺数, 可以用英寸总数除以 12, 并舍去余数。要计算余下的英寸数, 可以用 12 乘以英尺数, 然后用英寸总数减去前面得到的数。整个程序如图 3-4 所示。

```

/*
 * File: CentimetersToFeetAndInches.java
 *
 * This program converts centimeters to an integral number of feet
 * and any remaining inches.
 */

import acm.program.*;

public class CentimetersToFeetAndInches extends ConsoleProgram {
    public void run() {
        println("This program converts centimeters to feet and inches.");
        double cm = readDouble("Enter value in centimeters: ");
        double totalInches = cm / CENTIMETERS_PER_INCH;
        int feet = (int) (totalInches / INCHES_PER_FOOT);
        double inches = totalInches - INCHES_PER_FOOT * feet;
        println(cm + "cm = " + feet + "ft + " + inches + "in");
    }

    /* Private constants */
    private static final int INCHES_PER_FOOT = 12;
    private static final double CENTIMETERS_PER_INCH = 2.54;
}

```

图 3-4 将厘米转换为英尺和英寸的程序

声明

```
int feet = (int) (totalInches / INCHES_PER_FOOT);
```

舍去了余数，因为运算符(int)将结果强制转换为 int。表达式中(totalInches / INCHES_PER_FOOT)的括号是必需的，因为类型强制转换有更高的优先级。如果没有用这些圆括号，Java 会先将 totalInches 转换为整数，然后用结果除以 INCHES_PER_FOOT。

有些强制转换过程中，需要指定一个类型转换，即使没有应用自动转换规则。例如，假设已经声明两个整数变量 num 和 den，要计算它们的数学商(包括小数)，然后将结果赋值给新声明的名为 quotient 的 double 变量。不能写成：

```
double quotient = num / den;
```

因为 num 和 den 都是整数。当除法运算符应用于这两个整数时，舍去了余数。要避免这个问题，在执行除法之前必须至少将一个值转换为 double 类型。例如，可以转换分数的分母，如下所示。

```
double quotient = num / (double) den;
```

现在分母是 double 类型，执行除法时使用浮点算法，保留了小数。同样，也可以转换分子，如下所示：

```
double quotient = (double) num / den;
```

这条语句有相同的效果，但因为类型强制转换的优先级比除法要高，所以先进行类型转换。

3.4 赋值语句

Java 中，可以用两种方法中的任意一种给变量赋值。到目前为止所看到的程序，每个变量的值都被设置为其声明的一部分。然而，更复杂的程序通常在初始声明之后改变变量的值。因为变量最大的好处就是在程序生命周期内可以改变它们的值。要给变量赋一个新值，需要使用赋值语句，如下面语法框中所示。该语法与声明中使用的语法非常相似，唯一不同是没有类型名称。

赋值语句的语法

```
variable=expression;
```

其中，**variable** 是要设置的变量，**expression** 指定新的值。

如第 2 章所述，画框图有助于形象化程序中变量的作用。当声明一个局部变量为方法定义的一部分时，可以画一个新的方框来保存它的值，用变量名称标记方框。例如，假设方法使用下面的声明引入了 3 个新变量——两个是 int 类型，分别命名为 n1 和 n2；一个是 String 类型，命名为 msg。

```
int n1 = 17;  
int n2 = 0;  
String msg = "Hello";
```

可以给每个变量画一个方框，用这种绘图方法来表示变量，如图 3-5 所示。

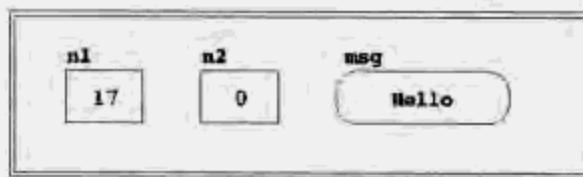


图 3-5 3 个变量的框图

本书中，包围所有变量的双线边界表示在相同的方法里定义这些变量。编程中，变量及相关方法特定调用的集合称为堆栈架构，这将在第 7 章讨论。这里选择为 String 变量画一个不同尺寸的方框，用来突出表示它保存不同类型的值。

赋值语句改变了当前堆栈架构里变量的值。因此，如果程序执行语句

```
n2 = 25;
```

就在名为 n2 的方框里写上 25，这样可以在图 3-6 中表示这个语句。

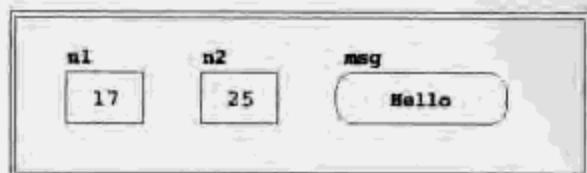


图 3-6 n2 为 25 的框图

同样，可以表示语句的结果

```
msg = "Welcome";
```

如图 3-7 所示。

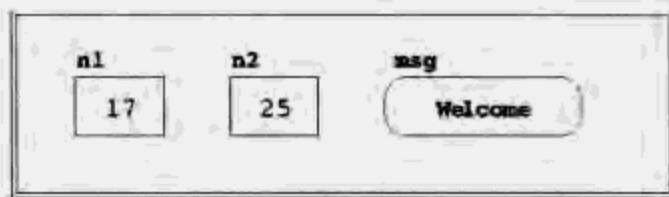


图 3-7 msg 为 welcome 的框图

变量只能保存合适类型的值，记住这一点很重要。例如，如果写如下语句

```
msg = 17;
```

在程序中，Java 编译器会将该语句标记为错误，因为变量 msg 被声明成了 String 类型。

框图说明的最重要的属性是每个变量只能保存一个值。给一个变量赋值以后，它保持这个值，除非再给它指派一个新的值。如果将一个变量的值指派给另一个变量，这个变量的值不会消失。因此，赋值语句

```
n2 = n1;
```

中，改变 n2，n1 保持不变，如图 3-8 所示。

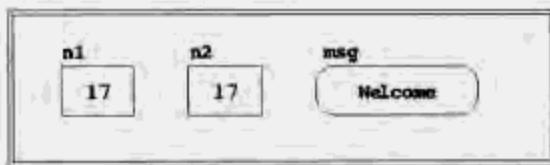


图 3-8 n2 改变，n1 不变

给变量指派新值会擦去它以前的内容。因此，语句

```
msg = "Aloha!";
```

将框图改变为

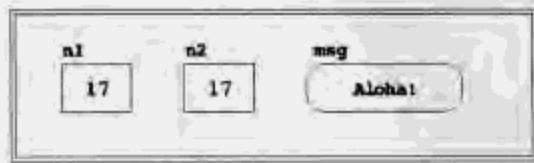


图 3-9 msg 改为 "Aloha!"

变量 msg 前面的值丢失了。

赋值是一个主动操作，不是等式的数学陈述，认识到这一点尤为重要。数学方程

```
x = x + 1
```

没有意义。没有 x 值能够使 x 等于 $x+1$ 。相应的赋值语句

```
x = x + 1;
```

则具有完全合理的意义并且非常有用。许多赋值语句中，等号右边表达式的值存储在左边出现的变量里。所以，该语句的结果是用以前的值取代 x 的值，换句话说，就是在 x 的基础上加 1。这种结构在编程中十分常见，为此 Java 定义了特殊域，将在 3.4.1 小节介绍。

3.4.1 简写赋值运算符

随着对编程越来越熟悉，会发现实际运用中有些赋值语句会经常出现。虽然的确看到许多赋值语句，在这些赋值语句中表达式的结果被赋给计算中没有什么作用的变量，但是许多赋值语句可以改变已有变量的值。例如，会发现通常需要改变变量——给它的当前值添加一些数，减去一些数，或执行类似操作。例如，如果要编写结算支票簿的程序，就可以使用下面的赋值语句：

```
balance = balance + deposit;
```

该语句将变量 $deposit$ 里的值与变量 $balance$ 的当前值相加，结果保存在 $balance$ 变量里。在更通俗的英语中，该语句的结果可以概括为“将 $deposit$ 和 $balance$ 相加”。然而，即使语句

```
balance = balance + deposit;
```

可以产生所需的结果，但是 Java 编程人员通常不会写这样的语句。对变量执行某个操作，然后将结果存储到相同的变量，这样的语句在编程中经常出现。因此 C(Java 表达式结构即来源于它)语言的设计者给为此概括了一种习惯简写。对于每个二进制运算符 op 而言，语句

```
variable = variable op expression;
```

可以用下面的语句取代

```
variable op = expression;
```

运算符与用于赋值形式的等号 (=) 的结合称为简写赋值运算符。

使用加法的简写赋值运算符，所以下列语句

```
balance = balance + deposit;
```

更常见的形式是

```
balance += deposit;
```

英语中，它的意思就是“将 $deposit$ 和 $balance$ 相加”。

因为相同的简写应用于 Java 中的所有二进制运算符，因此可以从 $balance$ 的值中减去 $surcharge$ ，如下所示。

```
balance -= surcharge;
```

或者用 x 的值除以 10，如下所示。

```
x /= 10;
```

或者用下面的表达式让 salary 的值加倍。

```
salary *= 2;
```

3.4.2 递增运算符和递减运算符

除了简写赋值运算符之外，Java 还为两个特别的常用编程运算符提供了进一步缩写，这两个运算符就是给变量加 1 或减 1。给变量加 1 称为递增该变量，给变量减 1 称为递减该变量。为了用一种非常简洁的形式表示这些运算符，Java 使用了运算符++和--。例如，语句

```
x++;
```

与下面的语句有相同的最终结果。

```
x += 1;
```

它本身就是下面这条语句的缩写

```
x = x + 1;
```

同样，

```
y--;
```

与

```
y -= 1;
```

或者

```
y = y - 1;
```

有相同的结果。

++和--运算符在 Java 程序中经常出现，实际上它们的运算比这里介绍的要复杂得多。然而，要理解这些运算符在第 4 章介绍的语句形式里如何应用，这里给出了的几种形式就足够了。

3.5 布尔表达式

在解决问题的过程中，通常需要让程序测试特定条件。这些条件会影响程序的后续行为。如果特定条件是正确的，就希望程序采取一种行动；如果条件不对，希望它采取其他的行动。在 Java 中，通过创建一些值要么对要么错的表达式，可以表示这种条件，这些表达式称为布尔(Boolean)表达式，以数学家 George Boole 命名，他开发了代数方法用于处理这种类型的数据。Java 中，Boolean 值使用原始类型 boolean 表示，它有由两个值组成的域：true 和 false。Java 定义了一些与 boolean 值一起使用的运算符。这些运算符属于两大类——关系运算符和逻辑运算符，下两节将讨论它们。

3.5.1 关系运算符

关系运算符用于比较两个值。Java 定义了 6 种关系运算符，分为两个优先级类。测试两个

量之间排序关系的运算符有：

> 大于	\geq 大于或等于
< 小于	\leq 小于或等于

这些运算符的优先级层次结构低于算术运算符，依次紧接着的是：

= 等于	\neq 不等于
------	------------

这些运算符用于测试等于和不等于。

如果编写测试是否相等的程序，使用 $=$ 运算符要非常小心，它由两个等号组成。单个等号是赋值运算符。由于双等号不是标准数学的一部分，因此，用单个等号取代双等号是十分常见的错误。很幸运，Java 编译器在编译程序时通常会发现这些错误，因为在它出现的上下文中赋值是不合法的。

关系运算符只能用于比较原子数据值——不是由较小要素部分创建的数据值。例如，整数、浮点数、Boolean 值和字符都是原子数据，因为它们不能分解为更小的部分。另一方面，字符串不是原子数据，因为它们由单个字符组成。因此，可以使用关系运算符比较两个 int 类型、double 类型、char 类型甚至是 boolean 类型本身的数据值，但不能比较 String 类型的数据值。第 8 章将学习比较字符串的方法。

常见错误

编写程序测试是否相等时，要确保使用 $=$ 运算符而不是 $=$ 运算符。 $=$ 运算符用于赋值。

3.5.2 逻辑运算符

关系运算符比较所有类型的原子值，产生 Boolean 结果。除了这些关系运算符之外，Java 定义了 3 种运算符，它们可以获得 Boolean 操作数，并将这些操作数合并形成其他 Boolean 值。

$!$ 逻辑非(如果下面的操作数是 false，那么它就是 true)

$\&\&$ 逻辑与(如果两个操作数都是 true，它也是 true)

$\|$ 逻辑或(如果有任何一个操作数都是 true，它也是 true)

这些运算符称为逻辑运算符，这里以优先级降序的顺序排列。

运算符 $\&\&$ 、 $\|$ 和 $!$ 与英语单词“和、或、否”非常相似。即使这样，但说到逻辑，英语就显得不是那么严密，记住这一点很重要。要避免这种不严密，用一种更正式、更精确的方式思考这些运算符通常会有所帮助。逻辑学家用真值表来定义这些运算符。真值表显示的是 Boolean 表达式的值随着其操作数值的改变而改变的情况。例如，假设 Boolean 值为 p 和 q，那么 $\&\&$ 运算符的真值表如表 3-1 所示。

表 3-1 $\&\&$ 运算符的真值表

p	q	$p \&\& q$
false	false	false
false	true	false
true	false	false
true	true	true

假设 Boolean 变量 p 和 q 的单个值如前两列所示，那么表的最后一列表示 Boolean 表达式 $p \&& q$ 值。因此，真值表的第一行表示，p 是 false，q 是 false，表达式 $p \&& q$ 的值也是 false。||的真值表如表 3-2 所示。

表 3-2 ||运算符的真值表

p	q	$p \parallel q$
false	false	false
false	true	true
true	false	true
true	true	true

即使||运算符相当于英语单词“或”，它不表示“一个或另一个”（英语中通常是这个意思），而是表示“一个或两个”，这是它的数学含义。

!运算符的真值表很简单，如表 3-3 所示。

表 3-3 !运算符的真值表

p	!p
false	true
true	false

如果要确定更复杂的逻辑表达式的操作方法，可以将它分解成这些原始操作，再为表达式的单个操作构建真值表。

大多数情况下，逻辑表达式不至于复杂到要用真值表来计算它们。可能导致混淆的唯一常见情况是，! 或 != 运算符与 && 或 || 同时出现。当英语中提到“否”时（与使用!和 != 运算符的情况一样），意思对人类而言很清楚的语句通常与数学逻辑不相符。应该特别小心，避免出错。

常见错误

使用&&和 || 运算符进行包含! 和 != 运算符的关系测试时要特别注意。英语对于逻辑而言可能有点不明确，但编程却要求精确。

例如，假设要在程序中表达“x 既不等于 2 也不等于 3”这个意思。对于这种条件测试的英文版，新的编程人员很可能如下编码表达式：

~~x != 2 || x != 3~~



本书使用故障符号（如上所示）来标记包含故意错误的代码。这种情况下问题是，代码的非正式英语转换与它在 Java 中的解释不对应。如果从数学的观点看待这种条件测试就会发现，如果①x 不等于 2 或②x 不等于 3，表达式就是 true。不管 x 是什么值，其中一条语句一定是 true，因为如果 x 等于 2，它也不可能等于 3。反过来也一样。要解决这个问题，需要深化对英语表

达式的理解，以便更清楚地陈述条件。也就是，当“不是 x 等于 2 或 x 等于 3 这种情况”时，条件是 true。可以直接将这个表达式转换为 Java，如下所示。

```
!(x == 2 || x == 3)
```

但是结果表达式有点难懂。真正要问的问题是，下面两个条件是否都是 true：

- x 不等于 2，并且
- x 不等于 3。

如果用这种形式思考这个问题，可以写出如下测试：

```
x != 2 && x != 3
```

这种简化从数学逻辑方面更具体地说明了下面这种更普遍的关系：

$$\neg(p \vee q) \text{ 等于 } \neg p \wedge \neg q$$

对于所有逻辑表达式 p 和 q 而言，这种转换规则及其相称性对等物

$$\neg(p \wedge q) \text{ 等于 } \neg p \vee \neg q$$

称为 De Morgan 法。忘了要应用这些规则，而是依赖英语风格的逻辑，这是出现编程错误的主要原因。

另一个常见错误是连接几个关系测试时忘记使用合适的逻辑连接词。数学中，通常会看到下面这种形式的表达式：

```
0 < x < 10
```

这种表达式在数学中有意义，但在 Java 中就没有意义。为了测试变量 x 是否大于 0 且小于 10，需要明确表示条件，如下所示。

```
0 < x && x < 10
```

常见错误

要测试一个数是否在特定范围内，仅仅合并关系运算符是不够的，就像数学里的约定一样。条件的两个部分必须使用 `&&` 明确地写出来，如下所示。

```
(0 < x) && (x < 10)
```

3.5.3 短路赋值

Java 解释 `&&` 和 `||` 运算符的方法不同于解释其他许多编程语言的方法。例如，在编程语言 Pascal 中，给这些运算符(写作 AND 和 OR)赋值要求给条件的两半赋值，即使结果在过程中间就可以确定。Java 的设计者采用不同的方法，对于编程人员而言，这种方法通常更方便。

Java 给下面这种形式的表达式赋值时，

```
exp1 && exp2
```

或

```
exp1 || exp2
```

单个子表达式总是从左至右赋值，只要结果确定了，赋值即结束。例如，如果 `exp1` 在带有`&&`的表达式中是 `false`，那么就没有必要给 `exp2` 赋值了，因为最终答案总是 `false`。同样，在使用`||`的示例中，如果第一个操作数是 `true`，那么也没有必要给第二个操作数赋值。只要知道了答案，赋值即停止，这种类型的赋值称为短路赋值。

短路赋值的主要好处是它允许一种条件控制另一种条件的执行。许多情况下，复合条件的第二部分只有在第一部分出现后才有意义。例如，假设要表达复合条件①整数 `x` 的值非零且② `y` 可被 `x` 整除。Java 中可以这样写条件测试：

```
(x != 0) && (y % x == 0)
```

因为只在当 `x` 非零时才会给表达式 `y % x` 赋值。Pascal 中对应的表达式不能生成所需的结果，因为总是会给 Pascal 条件的两个部分赋值。因此，如果 `x` 等于 0，包含该表达式的 Pascal 程序将通过除以 0 结束，即使好像有条件测试检验条件。防止复合条件后续部分出现赋值错误的条件称为防护，例如前面示例中的如下条件测试。

```
(x != 0)
```

3.5.4 标记

Boolean 类型的变量称为标记。例如，使用下面的声明来声明变量

```
boolean done = false;
```

变量 `done` 就是标记，它最初设置为 `false`。可以使用该变量记录是否完成了操作的某个阶段。就像可以给其他变量赋值一样，也可以给标记指派新的值。例如，可以将 `done` 的值设置为 `true`，如下所示。

```
done = true;
```

更重要的是，可以将具有 Boolean 值的任何表达式赋给 Boolean 变量。例如，假设程序的逻辑说明，如果变量 `itemsRemaining` 的值变成了 0，就表示已经完成了操作的某个阶段。要将 `done` 设置为合适的值，可以这样写

```
done = (itemsRemaining == 0);
```

该表达式中的圆括号不是必需的，但通常用来强调将条件测试的结果赋给变量这一事实。上面的语句说明，“计算(`itemsRemaining = 0`)的值，结果要么是 `true` 要么是 `false`，再将该结果存储在变量 `done` 里。”

3.5.5 Boolean 计算示例

即使在学习第 4 章介绍的语句形式之前不可能在程序里使用 Boolean 表达式，先看一看 Boolean 计算的实际示例也会很有用。假设编写的程序与日期有关，需要确定给定的年份是否是闰年。虽然我们知道闰年每 4 年出现一次，但是天文学上却不是这么简单。因为地球围绕太阳公转一周所用的时间是 $365\frac{1}{4}$ 天。每 4 年增加一天，这样有助于使历法与太阳同步，但还是有点不精确。要确保一年的开始不会慢慢偏离季节，就要求用于闰年的规则更复杂。除了以 00

结尾的年份以外，闰年每4年一次。以00结尾的年份只有能被400除尽才是闰年。因此，即使1900可被4除尽，但它不是闰年；而2000是闰年，因为它可以被400除尽。因此，闰年必须满足下列两个条件之一：

- 可以被4除尽，但不能被100除尽。
- 可以被400除尽。

虽然这种规则似乎比大多数时候所说的每4年就有一个闰年的方法要复杂，但是作为Boolean表达式在Java中编写正确规则的代码却相对容易。如果年份包含在变量y里，依据该年份是否是闰年，下给面的表达式赋值为true或false。

```
((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0)
```

虽然根据Java的优先级规则，实际上没有要求在表达式中使用圆括号，但是使用圆括号可以让长的Boolean表达式便于阅读。如果得到该表达式的结果，并将它存储在称为isLeapYear的标记里，那么就可以在程序的其他地方测试标记，以确定isLeapYear条件是否属实。

3.6 设计改变

到目前为止所看到的程序都非常简单，编程风格问题似乎并不重要。在这种情况下，很容易知道大多数程序是否包含了良好的注释，是否选择了合适的变量名称，或者在精确数值的地方是否使用了命名常量。然而，正是由于这些程序非常简单，所以最重要的是开发软件工程技能。如果一直等到程序变得复杂(以至于好的编程风格至关重要)才关注编程风格，就不能磨练您减少这种复杂性的策略。如果及早养成好的软件编写习惯，就很容易实现到复杂程序的飞跃。

好的软件工程最重要的一个方面是设计程序，以便能够方便地改变它。正如1.5.4小节所述，软件系统在其生命周期内会发生重大改变。如果预先知道改变不可避免，并且尽力支持这种改革过程，那么就可以充分降低进行实行改变的成本，实行改变的成本通常比一开始就产生软件的成本要高得多。当然，事先您不会确切知道改变是什么，但是您如果遵守软件设计的一些简单原则，就可以简化未来维护人员的工作。

3.6.1 可读性的重要性

要遵守的最重要的原则是程序必须用一种方便人们阅读的方法来编写。让Java编译器接受您的程序，并且不会产生各类错误消息——作为努力学习陌生语言语法细节的新的编程人员，必须将大部分精力放在这个问题上。然而，在某种意义上说，这个问题比较简单。如果将来要让其他人能够维护您所编写的代码，就必须将它设计得让人类读得懂，而不仅仅是让编译器读懂。

接下来几章将介绍许多简单策略，以充分提高程序的可读性。例如，您已经注意到形成每种方法主体的语句相对于方法头的缩进级别缩进了几个空格。有了这种简单约定——如果已经使用第4章里的复合语句形式，这就显得非常重要——很可能粗略一看就知道每个句法单元的主体包含哪些行。本章强调的另一种策略是使用变量、方法和类富有含义的名称。甚至是像在每个运算符两边加一个空格这样简单的方法都可以极大地增强程序的可读性。

让人类读者能够理解程序是最终目的，这些策略都是实现这一目标的简单方法。重要的是，编写程序时记住读者，想一想您所写的程序，能够让第一次看到它的人明白。设计程序时，要

知道自己要做什么。关键问题是代码能否将您的意图传达给读者。特别是，如果严格遵守了一些简单的句法规则(正确缩排，选择合适名称)，代码本身就足以表达您的想法。如果做的事情很复杂或不同寻常，为了方便将来的读者阅读，最好在注释里说明这些方面的设计。

3.6.2 使用命名常量支持程序维护

本章的几个程序使用了命名常量让结果程序易于阅读。例如，图 3-4 中的 `CentimetersToFeetAndInches` 就包括下面的常量定义：

```
private static final int INCHES_PER_FOOT = 12;
```

在程序中使用名称 `INCHES_PER_FOOT` 增强了可读性，因为它告诉读者常量有什么意义。如果刚好在程序里看到数值“12”，您可能不知道它表示什么。它可能刚好是一打的数目、时钟里的小时数或者一年的月份数。给常量一个描述性名称就可以减少这种模糊性。

除了增强可读性，命名常量在支持软件发展方面有另外一个重要作用。很可能，1 英尺包含的英寸数不可能随时间改变。这个值在字面上是个真正的常量，在编程中也一样。相反，程序中许多常量指定的内容可能随着程序的发展而改变，即使它们对于程序的特定版本而言是常量。

曾经的示例最能够说明这种原则的重要性。假设您是 20 世纪 60 年代末的一位编程人员，从事 ARPANET 的最初设计。ARPANET 是第一个大规模计算机网络，是今天 Internet 的前身。因为当时资源约束很紧，所以也许需要打破可以连接的计算机(在 ARPANET 时代，它们称为主机)数量的限制——就像 1969 年 ARPANET 的实际设计者做的那样。ARPANET 时代早期，该限制是 127 台主机。如果那时有 Java，可以这样声明常量：

```
private static final int MAXIMUM_NUMBER_OF_HOSTS = 127;
```

然而，后来，网络的爆炸性增长强制要求提高这一限制。如果在程序中使用命名常量，这个过程就相对简单。要将限制的主机数量提高到 1023 台，如下所示改变声明即可。

```
private static final int MAXIMUM_NUMBER_OF_HOSTS = 1023;
```

如果采用这种方式，在程序中任何需要引用最大值的地方使用 `MAXIMUM_NUMBER_OF_HOSTS`，那么这种简单改变将自动扩散到使用该名称的程序的各个部分。

注意，如果使用数字常量 127，情况就会完全不同。在那种情况下，就需要搜索整个程序，将所有使用 127 的实例改变为更大的值。有些情况下 127 可能指其他数量，而不是主机数量的限制，不改变这些值同样很重要。在最可能出错的事件中，发现故障非常困难。

3.6.3 使用命名常量支持程序开发

使用命名常量简化维护过程的最大好处是不受限于改变的种类(在程序的长期演变中会发生这些改变)。给常量命名在开发过程中——特别是当需要进行一些试验来计算特定常量应该有什么值时——也非常重要。这种情况在图形程序中很常见，在那些程序中通常需要调整特定对象的大小和位置，以达到最佳审美效果。

要更好理解使用命名常量如何支持图形程序开发，再看看第 2 章中的 `GRectPlusGOval` 程序会有所帮助。这个程序在红色矩形上添加一个绿色的椭圆，以此说明 `GOval` 类的图形，如图

3-10 所示。



图 3-10 GRectPlusGOval 程序产生的结果

产生该图形的 run 方法如下所示。

```
public void run() {
    GRect rect = new GRect(100, 50, 125, 60);
    rect.setFilled(true);
    rect.setColor(Color.RED);
    add(rect);
    GOval oval = new GOval(100, 50, 125, 60);
    oval.setFilled(true);
    oval.setFillColor(Color.GREEN);
    add(oval);
}
```

随着通过实践对编程美学理解的不断深入，很快就会怀疑用这种方法编写的程序。出错说明对 GRect 和 GOval 构造函数的调用使用了明确的数值，这通常是编程风格不好的标志。问题与程序的可读性有关。数字 100、50、125 和 60 没有提供有关这些值指定什么的任何暗示。如果给这些值起一些像 RECT_X、RECT_Y、RECT_WIDTH 和 RECT_HEIGHT 这样的描述性名称，读者在理解这些值的意义时就会少很多麻烦。

然而，GRectPlusGOval 示例中还有一个更重要的句法问题。要让 GOval 正好与 GRect 内部相切，参数值是什么无关紧要，重要的是在每种情况下参数都必须相同。因此，真正要做的就是用名称定义常量，如 FIGURE_X、FIGURE_Y、FIGURE_WIDTH 和 FIGURE_HEIGHT，然后在 GRect 和 GOval 构造函数里使用这些常量名称。按照这种方法，要改变图形的尺寸或位置，所有要做的就是编辑相关命名常量的定义。在一个地方进行改变，但是该改变的影响会扩散到这些命名常量出现的程序的各个部分。

查看 GRectPlusGOval 程序产生的输出，我的第一个审美反应是图形应该再高点。如果按照前面段落里建议的那样重新写程序，可以反复试验 FIGURE_HEIGHT 值直到找到似乎满意的值为止，这个值好像是 75。我的第二个审美反应是图形实际上应该在窗口中间，而不是稍稍偏左。虽然通过试验不同的 FIGURE_X 和 FIGURE_Y 值可以修正位置，但是不可能精确地找到这个值。要实现所需的效果（让矩形和椭圆都在窗口中间），更好的策略是计算出这些值。

要实现这个目标，需要知道窗口的大小。凑巧的是，要得到这些信息很简单，因为 GraphicsProgram 类包含两种方法——getWidth 和 getHeight——可以返回图形窗口的尺寸。知道了这些尺寸，就可以方便地计算出图形左上角的 x 坐标和 y 坐标。例如，上角的 x 坐标是窗口宽度的一半减去图形宽度的一半，可以用相同的方法计算 y 坐标。这样，使用下面的行就

可以计算出 GRect 和 GOval 左上角的坐标:

```
double x = (getWidth() - FIGURE_WIDTH) / 2;
double y = (getHeight() - FIGURE_HEIGHT) / 2;
```

注意, 即使改变 FIGURE_WIDTH 和 FIGURE_HEIGHT 常量的值或改变窗口的大小, 变量 *x* 和 *y* 也会自动获得正确的值。图 3-11 显示的是合并这些改变的整个程序。

```
/*
 * File: GRectPlusGOval.java
 *
 * This program creates a red GRect and a green GOval using
 * the same parameters. The example illustrates that the
 * GOval fills the boundary set by the enclosing rectangle.
 * This version takes the width and height of the figure from
 * named constants and computes the location of the figure so
 * that it is centered in the window. Changing the value of the
 * constants changes the dimension of the figure on the screen.
 */

import acm.graphics.*;
import acm.program.*;
import java.awt.*;

public class GRectPlusGOval extends GraphicsProgram {
    public void run() {
        double x = (getWidth() - FIGURE_WIDTH) / 2;
        double y = (getHeight() - FIGURE_HEIGHT) / 2;
        GRect rect = new GRect(x, y, FIGURE_WIDTH, FIGURE_HEIGHT);
        rect.setFilled(true);
        rect.setFillColor(Color.RED);
        add(rect);
        GOval oval = new GOval(x, y, FIGURE_WIDTH, FIGURE_HEIGHT);
        oval.setFilled(true);
        oval.setFillColor(Color.GREEN);
        add(oval);
    }
    /* Private constants */
    private static final double FIGURE_WIDTH = 125;
    private static final double FIGURE_HEIGHT = 75;
}
```

图 3-11 GRectPlusGOval 程序改进后的执行

3.7 小结

本章讨论 Java 表达式及在程序中使用它们的方法。介绍的重点是:

- 数据值有许多不同的类型, 每种类型都由域和一组操作定义。
- 常量用于指定程序中不变的值。
- 变量有 3 个属性: 名称、类型和值。Java 程序中使用的所有变量必须使用下面的形式声明。

```
type identifier = expression;
```

它确定了变量的名称、类型和初始值。

- 可以使用类变量创建命名常量。命名常量的声明出现在所有方法之外，包括关键字 static 和 final。
- 表达式由运算符连接的单个项组成。运算符应用的子表达式称为操作数。
- 如果将算术运算符应用于两个 int 类型的操作数，其结果也是 int 类型。如果有一个或者两个操作数都是 double 类型，其结果也是 double 类型。
- 如果将 / 运算符应用于两个整数，结果是整数。第一个操作数除以第二个操作数并舍去余数，就得到结果。使用 % 运算符可以得到余数。
- 当不同类型的值出现在算术表达式中或给更通用类型的变量赋值时，就会发生数字类型间的自动转换。
- 使用类型强制转换可以指定数值类型间的明确转换。
- 表达式中的运算顺序由优先级规则确定。目前为止介绍的运算符都属于表 3-4 所示的优先级类，表中优先级的顺序从上到下依次降低，即最上方运算符的优先级最高。

表 3-4 运算符的优先级

一元运算符 - + + - (类型强制转换)			
*	/	%	
+	-		
<	<=	>	>=
==	!=		
&&			
赋值运算符			

除了赋值以外，当相同优先级类的两个运算符竞争同一个操作数时，Java 按从左至右的顺序应用二进制运算符(赋值运算符和二进制运算符按从右至左赋值，但是本书没有使用理解该规则所需的示例)。

- 使用赋值语句可以改变变量的值。每个变量每次只能保存一个值：给变量赋新值后，以前的值就丢失了。
- Java 包括赋值语句的缩写形式，其中，

```
variable op= expression;
```

是下面这个较长表达式的简写：

```
variable = variable op expression;
```

- Java 包括特殊运算符++和--，它们分别指定从变量里加 1 和减 1。
- Java 定义称为 boolean 的数据类型，它用于表示 Boolean 数据。Boolean 类型只有两个值：true 和 false。
- 使用关系运算符 (<、<=、>、>=、== 和 !=) 可以生成 Boolean 值；使用逻辑运算符(&&、|| 和 !)可以合并 Boolean 值。

- 逻辑运算符`&&`和`||`按从左至右的顺序赋值，按这种方法，只要程序确定了结果，赋值即停止。这种行为称为短路赋值。
- 命名常量在写易于改变的程序时特别有用。

3.8 复习题

1. 定义数据类型的两个属性是什么？
 2. 确定下面哪些在 Java 中是合法常量。对于合法常量，指出它们是整数还是浮点常量。

- | | |
|----------|---------------|
| (1) 42 | (7) 1,000,000 |
| (2) -17 | (8) 3.1415926 |
| (3) 2+3 | (9) 123456789 |
| (4) -2.3 | (10) 0.000001 |
| (5) 20 | (11) 1.1E+11 |
| (6) 2.0 | (12) 1.1X+11 |

3. 以 Java 的形式，用科学计数法重写下面的浮点常量。

- | |
|---------------------------------|
| (1) 6.02 252 × 10 ²³ |
| (2) 2 997 925 000 0.0 |
| (3) 0.000 000 005 291 67 |
| (4) 3.141 592 653 5 |

(顺便说一下，这里每个常量分别表示化学、物理或数学中重要值的近似值：①阿伏加德罗常数值；②以厘米每秒为单位的光速；③以厘米为单位的波尔半径；④数学常量 π。在这种情况下，使用科学符号形式不是很好，但是仍然可以这样写，您应该知道怎么做。)

4. 确定下面哪些在 Java 中是合法的变量名称。

- | | |
|----------------------|---------------------------------|
| (1) x | (7) aReasonablyLongVariableName |
| (2) formula | (8) total output |
| (3) average_rainfall | (9) 12MonthTotal |
| (4) %correct | (10) marginal-cost |
| (5) short | (11) b4hand |
| (6) tiny | (12) _stk_depth |

5. 指出下面表达式的值和类型。

- | | |
|------------|-----------|
| (1) 2+3 | (4) 3*6.0 |
| (2) 19/5 | (5) 19%5 |
| (3) 19.0/5 | (6) 2%7 |

一元负数运算符和二元减法运算符之间有什么不同？

运用合适的优先级规则，计算下面每个表达式的结果：

- | |
|------------------------------|
| (1) 6+5/4-3 |
| (2) 2+2*(2*2-2)%2/2 |
| (3) 10+9*((8+7)%6)+5*4%3*2+1 |

(4) $1+2+(3+4) * ((5*6\%7*8)-9)-10$

8. 如果将变量 k 声明为 int 类型, 程序执行下面的赋值语句后, k 的值是什么?

```
k = (int) 3.14159;
```

执行下面的赋值语句之后, k 的值又是什么?

```
k = (int) 2.71828;
```

9. Java 中, 如何指定数字类型间的转换?

10. 变量 cellCount 的值乘以 2, 可以使用哪个习语?

11. 在 Java 中要写一条与下面的语句有相同效果的语句, 最常见的方法是什么?

```
x = x + 1;
```

12. boolean 数据类型的两个值是什么?

13. 编程人员在条件表达式中使用了数学中的等于符号(=), 会出现什么情况?

14. 如何写 Boolean 表达式来测试整数变量 n 的值是在 0~9 之内还是在它之外?

15. 用英语描述下面条件表达式的意思:

```
(x != 4) || (x != 17)
```

如果该条件是 true, 那么 x 取什么值?

16. “短路赋值”是什么意思?

17. 用自己的语言描述命名常量的使用如何让程序随着时间的推移而更易于改变。

3.9 编程练习

1. 扩充图 3-2 中的 InchesToCentimeters 程序, 让它输入两个值: 英尺数和紧接着单独一行的英寸数。图 3-12 是程序的运行示例。

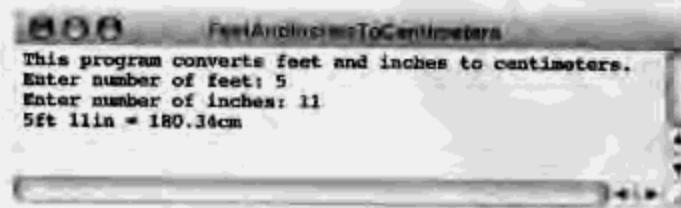


图 3-12 InchesToCentimeters 程序的样本运行

2. 编写一段程序, 输入两个数: 一个账户余额, 一个以百分比表示的年利率。程序要显示一年后的余额。没有存款或取款——只算支付利息。程序应该能够复制图 3-13 所示的运行示例。

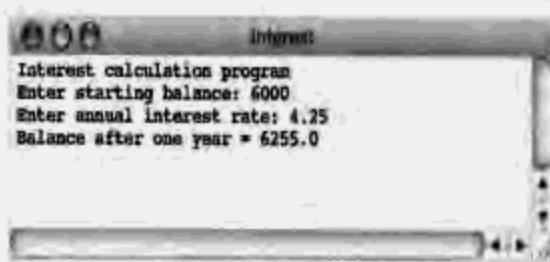


图 3-13 支付利息计算程序的样本运行

3. 扩充练习 2 中编写的程序，让它能够显示两年后的余额，如图 3-14 所示。

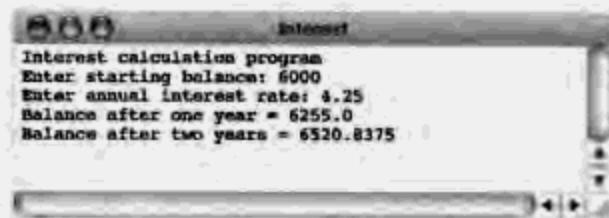


图 3-14 支付利息计算程序的样本运行

注意，该示例中每年以复利计算利息，即第一年的利息添加到银行余额，因此第二年它也有利息。第一年，6 000 美元收取 4.25% 的利息，也就是 255 美元。第二年，账户对全部 6 255 美元收取 4.25% 的利息。

4. 写一段程序，让用户告诉圆的半径，然后用公式

$$A = \pi r^2$$

计算圆(A)的面积。

注意，Java 中没有“幂”运算符。使用 Java 中已经学过的算术运算符，如何写出能够实现所需结果的表达式？

5. 编写一段程序，输入华氏温度，返回相应的摄氏温度。转换公式是

$$C = \frac{5}{9}(F - 32)$$

图 3-15 所示程序运行示例。

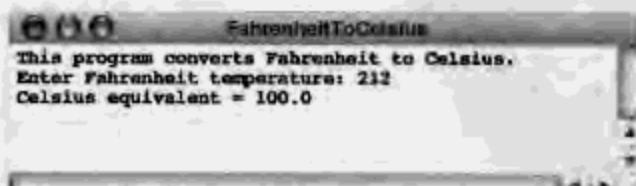


图 3-15 华氏温度转换成摄氏温度的样本运行

如果编写程序时不小心，答案就总是 0。什么故障导致了这种行为？

6. 在 Norton Juster 的儿童读物 *The Phantom Tollbooth* 中，Mathemagician 要 Milo 解决下面的问题：

```
4+9-2*16 +1/3*6=67 +8*2-3*26 -1/34 +3/7+2-5
```

根据 Milo 的计算，并得到了 Mathemagician 的确认，该表达式“结果总是等于零”。然而，如果是您进行计算，只有从头开始并严格按照从左至右的顺序应用运算符，表达式才等于零。如果使用 Java 的优先级规则给 Mathemagician 的表达式赋值，答案又是什么呢？写一段程序验证您的计算。

7. 编写一段程序，将以千克为单位的公制重量转换为相应的以磅和盎司为单位的英语重量。所需的转换公式是：

$$1 \text{ 千克} = 2.2 \text{ 磅}$$

$$1 \text{ 磅} = 16 \text{ 盎司}$$

8. 编写一段程序计算 4 个整数的平均值。

9. 编写一个古老的童谣这样唱：

我去圣路易斯的路上，遇到了一个人，他有 7 位妻子，每位妻子背着 7 个袋子，每只袋子装着 7 只猫，每只猫有 7 只小猫——小猫、大猫、袋子和妻子，一共多少人要去圣路易斯？

最后一句是一个有欺骗性的问题：只有说话的那个人要去圣路易斯；其他人大概在朝相反的方向走。然而，假设要知道这个组合群体——小猫、大猫、袋子和妻子——中有多少来自圣路易斯。写一段 Java 程序计算并显示这个结果。要让程序符合问题的结构，以便所有阅读程序的人能够理解在计算什么值。

10. 图 2-19 所示的 TicTacToeBoard 程序生成的输出如图 3-16 所示。



图 3-16 TicTacToeBoard 程序的输出

然而，这个程序的编程风格不太好，因为用代码明确指定了每一行的坐标。重写该程序，让棋盘在窗口居中，使用名为 BOARD_SIZE 的单个常量来定义图形的高度和宽度。

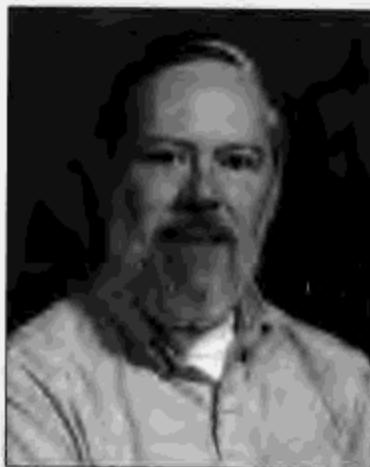
11. 第 2 章的练习 4 要求写图形程序，生成一个结构简单的房子图形。如果要让 DrawHouse 程序易于改变，应该定义哪个命名常量来提供必要的灵活性？使用这些命名常量而不是明确的坐标值重写程序。

第 4 章

语句形式

The statements was interesting but tough.

—Mark Twain, Adventures of Huckleberry Finn, 1884



Dennis Ritchie

作为贝尔实验室一名年轻的研究科学家，Dennis Ritchie 在 1970 年创建了 C 编程语言来简化 UNIX 操作系统的开发。他还和 Ken Thompson 联合开发了 UNIX 操作系统。UNIX 系统和 C 语言都在计算领域产生了重大影响。特别是 C 语言，它为包括 Java 在内的当今许多最重要的编程语言奠定了基础。Ritchie 设计的 C 语言语句结构，几乎按原样延伸到了 Java 里。1983 年，Ritchie 和 Thompson 共同获得了以英国计算机科学家先驱 Alan Turing 的名字命名的“ACM 图灵奖”，这是计算专业的最高荣誉。

第 2 章已经介绍了一些扩充 Program 层次结构的 Java 程序的示例。执行包含在名为 run 方法主体内的语句可以运行这些程序。本章将讨论 Java 中几种不同的语句类型，同时扩充用于解决问题的工具集。

4.1 Java 的语句类型

Java 的语句类型主要有 3 类：

- 简单语句，它执行某个行为。
- 复合语句，它将其他语句结合到一系列操作之中。
- 控制语句，它影响其他语句执行的方式。

到目前为止已经学习过 Java 的各种简单示例，如赋值和对 `println` 方法的调用。即使不知道其中的参数项，但实际上从第一个程序起就已经使用了复合语句，因为每个 `run` 方法的主体就是复合语句。然而，控制语句是全新的内容，本章主要将讨论它。详细介绍这些控制语句之前，先抽象分析语句的类别。

4.1.1 简单语句

在第 2 章和第 3 章的程序中，可以找到用于完成各种任务的简单语句。例如，本文开始包含 `run` 方法的程序就由下面的简单语句组成：

```
add(new GLabel("Hello, world", 100, 75));
```

同样，还介绍了赋值语句，例如简写形式：

```
balance += deposit;
```

显示信息的语句，例如：

```
println("The total is " + total + ".");
```

简单语句的语法

简单语句由表达式及后面的分号组成。

简单地说，可以将这些语句类型当成单独的工具，也可以将它们当作项使用。如果要读取整数，第 2 章有个术语可以实现这一功能，记住这一点极有帮助。只要将它写下来就行了。然而，如果仔细考察语句，就会发现它们共享统一的结构，这个结构让 Java 编译器很容易识别程序中的合法语句。Java 中，所有简单语句——不管它们有什么功能——都有语法形式，如上边的语法说明所示。所以简单语句的模板很简单：

```
expression;
```

在表达式后面添加一个分号，可以让表达式成为合法语句。

在 Java 中，即使表达式后面加上分号成为了合法语句，也不是每个这样组合的语句都是有用的语句。语句必须有某些明确的效果才有用。语句

```
n1 + n2;
```

由表达式 `n1 + n2` 及后面的分号组成，因此是合法语句。然而，它没有用，因为没有结果；语句相加变量 `n1` 和 `n2`，舍弃了结果。Java 中的简单语句通常是赋值(包括简写赋值和递增/递减运算符)或执行某些有用操作的方法调用，如 `println`。下面的程序行

```
println("The total is " + total + ".") ;
```

是合法语句，这很容易理解，因为它们符合简单语句的模板。依据第2章给出的表达式的定义，方法调用是合法的表达式，因此上面行中的方法调用——除了分号以外的部分——也是合法表达式。在行的结尾加上分号就将表达式变成了简单语句。

但是赋值又怎么样呢？如果像下面的行

```
total = 0;
```

要符合简单语句的模板，一定要

```
total = 0
```

本身是表达式。

在Java中，用于赋值的等号只是一个二进制运算符，和+或/一样。`=`运算符有两个操作数，一个在左边，一个在右边。左边的操作数是变量名称。Java执行赋值运算符时，先给右边的表达式赋值，然后将结果值存储在左边的变量里。因为用于赋值的等号是运算符，所以

```
total = 0
```

确实是表达式，因此下面的行

```
total = 0;
```

是简单语句。

4.1.2 复合语句

简单语句允许编程人员指定行为。然而，除了第2章的HelloProgram示例以外，到目前为止看到的所有程序都需要多条简单语句来完成任务。对大多数程序而言，解决方案策略由几个连续步骤组成的协同行为组成。例如，第2章的Add2Integers程序，必须先读取第一个数，然后读取第二个数，再将二者相加，最后显示结果。将这一系列行为转换为实际程序步骤，就要使用几条单独语句，这些语句都是主程序主体的一部分。

要指定系列语句是连贯单元的一部分这个事实，可以将这些语句组合成复合语句(也称代码块)，它是波形括号内语句和声明的集合，如下所示：

```
{  
    statement;  
    statement;  
    statement;  
    ...  
    statement;  
}
```

这些语句都可以由声明取代，最简单地理解，就是将声明作为语句的另一种形式看待。而且，符号“*statement*”出现在语法模板或术语模式时，都可以用单个语句或复合语句取代它。

代码块内部的语句通常相对于封装的上下文缩排。编译器会忽略缩排，但是对于人类读者而言，这种视觉效果很有用，因为它让程序结构从页面格式上看很令人喜欢。经验研究说明，在每个新层次使用3~4个空格可以让人容易理解程序结构；本书的程序在每个新层次使用4

个空格。缩排对于好的编程而言至关重要，因此，应该保证在程序中坚持使用一贯的缩排风格。

4.1.3 控制语句

如果没有任何指令，代码块里的单个语句会按它们出现的顺序依次执行。然而，对于大多数应用程序而言，只有这种严格的由上到下的排序还不够。现实世界中问题解决策略注重于这样的操作，例如重复一组步骤或选择二选一的行为。影响其他语句执行方式的语句称为控制语句。

Java 中的控制语句主要有两类。

1. 条件语句

条件语句在解决问题的过程中，根据某种条件测试，通常需要在程序中两个或更多单个路径中进行选择。例如，要求写一段程序，如果某个值是负数，以一种方式行动；如果不是，则按另一种方式行动。需要做决定的一类控制语句称为条件语句。Java 中，条件语句有两种形式：if 语句和 switch 语句。

2. 迭代语句

迭代语句特别是处理涉及多个数据项的问题时，只要满足某个条件的操作，程序通常要按指定次数重复操作。在编程中，这种重复称为迭代，重复的代码部分称为循环。Java 用作迭代基础的控制语句是 while 语句和 for 语句。

Java 中的每条控制语句都由两部分组成：控制行和主体。控制行指定重复或条件的类型，主体由控制行所影响的语句组成。条件语句中，主体可以分成单独的部分，在某些条件下执行一组语句，在另一些条件下执行另一组语句。

每条控制语句的主体都由语句组成，通常都是代码块控制行——不管它是否指定重复或条件来执行——应用于整个主体。而且，主体可以包括控制语句，这些控制语句依次包含其他语句。在其他控制语句主体内使用控制语句，称为嵌套。嵌套控制语句(一个在另一个里面)是现代编程语言最重要的特征。

4.2 控制语句和问题解决

讨论 Java 控制语句的细节之前，整体思考一下控制语句在编程过程中的作用大有裨益。学习介绍性课程的学生通常认为，一定有某个规则来决定何时使用 Java 提供的不同控制语句。这不是编程的方法。控制语句是用于问题解决的工具。在确定什么样的控制语句在特殊语境中有意义之前，必须认真思考要解决的问题以及解决该问题所选择的策略。确定如何解决根本问题之后，再写程序代码。编程过程中没有什么是自动的。

没有什么魔法规则可以将问题语句转换为可以运行的程序，虽然知道这点会后很失望，但正是这一点让编程成为了一种颇有价值的技能。如果可以依据某些定义好的运算法则执行程序过程，那么就可以直接自动执行整个过程，完全不需要人类编程人员。编程需要很强的设计能力和创造力。编程的本质是解决问题，许多需要解决的问题非常复杂，极具挑战性。解决这样的问题会让计算机编程变得困难；当然也让编程变得非常有趣。

4.2.1 一般化 Add2Integers 程序

到目前为止在本书中看到的程序，都为了说明如何写简单程序，而不是介绍解决问题的经验。现在，可以体验通过编程解决问题过程中的这种兴奋。是解决一些有趣问题的时候了。虽然第2章的 Add2Integers 程序本身不值得详述，但是可以将它作为更有趣示例的基础。如果要让程序在实际中有用，要做的第一件事就是一般化程序，让它可以相加多个数。大多数人完全能够以合理的速度和精确度相加两个整数。对于这样简单的任务，可以不需要计算机。但是如果要相加 10 个整数，或者 1000 个整数，甚至是 100 万个整数呢？在这种情况下，计算机的优势就突现出来了。

但怎样改变 Add2Integers 程序让它能够在更大范围内相加整数呢？如果要加 4 个整数，只要在 run 方法里再简单加上几行就可以，如下所示。

```
public void run() {
    println("This program adds four integers.");
    int n1 = readInt("Enter n1: ");
    int n2 = readInt("Enter n2: ");
    int n3 = readInt("Enter n3: ");
    int n4 = readInt("Enter n4: ");
    int total = n1 + n2 + n3 + n4;
    println("The total is " + total + ".");
}
```

这种方法对于相加 4 个整数合适，但是用相同的策略相加 10 个整数，就很单调乏味，更别说相加 1000 或 100 万个整数了。

假设在不需要 10 个变量声明的情况下，写一段程序相加 10 个整数。让我们想一想这个问题。假设不用计算机相加 10 个整数，有人将这些数据给您：7, 4, 6, 等等。怎么做呢？可以将这些数字写下来，最后将它们相加。这与 Add2Integers 程序中使用的策略类似。这种方法有效，但在速度和灵敏性方面却并不具优势。另一种选择是可以这样相加这些数：7 加 4 等于 11，11 加 6 等于 17，依此类推。如果采用这种策略，就不需要记住单个数字，只记住当前的总数就可以了。听到最后一个数，就可以宣布答案了。

如何相加 10 个整数而不必声明 10 个变量？不必记住每个单独的数，这种思想有助于回答这个问题。运用这种新策略，应该能够写出新的只使用两个变量的 Add10Integers 程序。一个表示听到的每个数(value)，一个表示目前这些数的和(total)。每次给 value 输入新的数时，就将它与 total 相加，这样就可以记住连续的总数。完成之后，可以重新使用 value 来保持下一个数，用相同的方法进行处理。这种理解应该能够让您着手编码使用新策略的程序。每个输入值都必须执行如下步骤：

- (1) 从用户获得整数值，并将它存储在变量 value 里。
- (2) 将 value 与存储在变量 total 里的连续和相加。

已经知道了如何编码第一步；它是第2章“从用户读取整数”模式的完美示例，通用形式如下：

```
int variable = readInt(prompt)
```

您也知道如何编写第二步的代码。相加 value 和 total 是第3章介绍的简写赋值模式的实例。要相加 value 和 total，形式如下：

```
total += value;
```

这两种模式——一种用于整数读取，一种用于将整数与连续和相加——提供了编码操作（Add10Integers 程序中每个输入值必须产生这些操作）所需的所有工具。对于 10 个输入值而言，程序都要执行下面的语句：

```
int value = readInt(" ? "); total += value;
```

这里，要做的是找到让程序执行这组语句 10 次的方法。

4.2.2 重复 N 次模式

本章稍后将讨论 for 语句，以及在各种应用程序中使用它的方法。在 Java 中，for 语句比在大多数语言中更加灵活也更加强大。对于有经验的编程人员而言，这种灵活性和强大能力非常有用；我的经验表明，大多数 for 循环适用于很简单的模式，这种模式的唯一功能是以预定的次数重复代码块。即使没有理解 for 语句本身的细节，也可以将它作为编程习语与第 2 章里的行一起使用。

“重复 N 次模式”形式如下：

```
for (int i = 0; i < N; i++) {
    statements to be repeated
}
```

这种模式中，值 N 表示要重复的次数。例如，如果用 5 替代 N，大括号内的语句将执行 5 次。要在 Add10Integers 程序里使用这种模式，需要用 10 替代 N。大括号内的语句是：①将整数输入 value；②将该值与 total 相加。如果在范例中进行这些替代，会得到如下代码：

```
for (int i = 0; i < 10; i++) {
    int value = readInt(" ? ");
    total += value;
}
```

此时，几乎可以写出完整的 Add10Integers 程序了，但是还要解决一个小问题。变量 value 被声明为整数作为模式的一部分，用来从用户读取整数，然而，没有声明变量 total。要让该循环正确运行，必须在循环外声明 total，并定义初始值为 0。这样，在 for 循环之前，需要包含声明：

```
int total = 0;
```

来确保该变量的功能是作为连续总数。给变量设置一个合适的初始值称为初始化。许多语言中，没有初始化变量是一个常见的故障。然而，Java 很容易检查出未初始化的变量，并提示需要进行初始化。

最后一道难题是要完成 Add10Integers 程序。所需的 run 方法如下：

```
public void run() {
    println("This program adds ten integers.");
    int total = 0;
    for (int i = 0; i < 10; i++) {
        int value = readInt(" ? ");
        total += value;
    }
    println("The total is " + total + ".");
}
```

1

然而，该程序还是没有设计成想要的样子。不能总是刚好相加 10 个整数，应该很容易改变代码，以便对不同数目的整数进行相加。在目前情况下，必须深入代码内部，改变初始消息和 for 循环的极限。更好的方法是引入第 3 章介绍的命名常量。如果选择调用常量 N_VALUES，就可以得到类似 AddNIntegers 的程序，如图 4-1 所示。

```
/*
 * File: AddNIntegers.java
 *
 * This program adds a predefined number of integers and
 * then prints the sum at the end. To change the number
 * of integers, change the definition of N_VALUES.
 */

import acm.program.*;

public class AddNIntegers extends ConsoleProgram {

    public void run() {
        println("This program adds " + N_VALUES + " integers.");
        int total = 0;
        for (int i = 0; i < N_VALUES; i++) {
            int value = readInt(" ? ");
            total += value;
        }
        println("The total is " + total + ".");
    }

    /* Specifies the number of values */
    private static final int N_VALUES = 10;
}
```

图 4-1 添加预定义整数值的程序

4.2.3 “读取到指定条件为止”模式

即使已经指定数值作为命名常量，以 AddNIntegers 程序当前的形式，还是不可能满足用户的所有需要。当前版程序最重要的问题是更新程序让它相加不同数量的整数，这需要对程序进行改变和重新编译。真正需要的是更一般的程序，不必预先指定输入值的数量，就可以相加任何输入的值。从用户的角度而言，要预先计数让程序看起来很麻烦。如果使用这样的程序，要做的就是输入数字，直到全部完成为止。在那种情况下，需要告诉程序已经输完了所有数字。

解决这种问题的常见方法是定义特定的输入值，用户输入该值来表示输入列表的结尾。用于终止循环的特定值称为哨兵。选择合适值作为哨兵，这种选择取决于输入数据的类型。选择作为哨兵的值不能是合法数据值；也就是说，它不能是用户需要作为正常数据输入的值。例如，相加一列整数时，值 0 就是合适的终止条件。选择 0 作为终止条件说明，不能输入 0 作为数据值，但是所有使用程序相加一列值的人都可以完全忽略数据中所有的 0 值，因为它们不影响结果。如果写程序求考试分数的平均值，情况就不一样。在求平均值时，0 分却会改变结果，我们知道，有的同学有时会得 0 分。在这种情况下，0 是一个合法数据值。为了允许程序的用户输入 0 作为分数，需要选择不代表实际分数的其他终止条件值。因为考试通常不可能得负分，所以选择像 -1 这样的值作为该应用程序的终止条件就很合适。

要将 AddNIntegers 扩充到新的 AddIntegerList 程序，唯一要做的改变是循环结构。for 循环——最常见用于按预定次数执行一组操作——不再合适。需要一种新模式，它可以读取数据，直到用户输入表示输入结束的哨兵为止。这种模式称为“读取到哨兵为止”模式，其形式如下：

```
while (true) {
    prompt user and read in a value
    if (value == sentinel) break;
    rest of body
}
```

这种基于终止条件循环的新模式让您能够完成 AddIntegerList 程序，如图 4-2 所示。

```
/*
 * File: AddIntegerList.java
 *
 * This program reads integers one per line until the
 * user enters a special sentinel value to signal the
 * end of the input. At that point, the program
 * prints the sum of the numbers entered so far.
 */

import acm.program.*;

public class AddIntegerList extends ConsoleProgram {

    public void run() {
        println("This program adds a list of integers.");
        println("Enter values, one per line, using " + SENTINEL);
        println("to signal the end of the list.");
        int total = 0;
        while (true) {
            int value = readInt(" ? ");
            if (value == SENTINEL) break;
            total += value;
        }
        println("The total is " + total + ".");
    }

    /* Specifies the value of the sentinel */
    private static final int SENTINEL = 0;
}
```

图 4-2 相加一列整数(这些整数以指定终止条件标记结尾)的程序

本章稍后将详细介绍控制语句，“读取到指定终止条件为止”模式形成于这些控制语句之外。甚至在理解细节之前，就会发现这种模式很有用。然而，随着对编程学习的不断深入，您会发现，即使是专业编程人员也通常使用没有彻底理解的代码。实际上，专业编程人员的标志之一，就是在没有完全理解内在细节的情况下，能够使用库或一些代码。随着程序越来越复杂，能够使用大体理解的工具是一项非常重要的技能。

4.3 if 语句

Java 中表示条件执行最简单的方法是使用 if 语句，它有两种形式：

```
if (condition) statement
if (condition) statement1 else statement2
```

该模板中的 condition 组件是第 3 章定义的 Boolean 表达式。if 语句的简单形式中，只有当指定的条件值为 true 时，Java 才执行语句。如果条件值为 false，Java 会完全跳过 if 语句的主体。在包含关键字 else 的形式中，如果条件是 true，Java 执行 *statement₁*；如果条件是 false，则 Java 执行 *statement₂*。在 if 语句的两种形式中，条件表达式是 true 时，Java 执行的代码称为 *then* 子句。在 if-else 形式中，当条件是 false 时执行的代码称为 *else* 子句。

决定是使用简单的 if 语句还是 if-else 形式，完全取决于解决方案策略如何操作。如果适合特殊条件，那么解决方案策略就要求执行一组语句，在这种情况下，可以使用简单的 if 语句。程序必须在基于测试结果的两组独立行为之间做出选择，这种情况下，使用 if-else 的形式。通常可以依据用英语描述问题的方法来作决定。如果描述包含“否则”(otherwise)或类似词语，可能就需要使用 if-else 形式。如果英语描述中没有这类词语，很可能使用简单的 if 语句就足够了。

图 4-3 中的 LeapYear 程序说明了 if 语句的用法，该程序使用第 3 章介绍的 Boolean 表达式来确定给定的年份是否为闰年。图中所示的程序需要使用 if-else 形式，因为如果指定的年份是闰年，程序需要打印一条消息；如果不是，则打印另一条消息。相反，如果问题的结构是只有闰年才打印消息，那么使用简单的 if 形式就可以了，如下所示：

```
if (isLeapYear) {
    println(year + " is a leap year.");
}
```

```
/*
 * File: LeapYear.java
 *
 * This program reads in a year and determines whether it is a
 * leap year. A year is a leap year if it is divisible by four,
 * unless it is divisible by 100. Years divisible by 100 are
 * leap years only if divisible by 400.
 */
import acm.program.*;

public class LeapYear extends ConsoleProgram {
    public void run() {
        println("This program checks for leap years.");
        int year = readInt("Enter year: ");
        boolean isLeapYear = ((year % 4 == 0) && (year % 100 != 0))
            || (year % 400 == 0);
        if (isLeapYear) {
            println(year + " is a leap year.");
        } else {
            println(year + " is not a leap year.");
        }
    }
}
```

图 4-3 确定某一年是不是闰年的程序

else 子句在 if 语句中是可选的，这种情况有时会造成含糊不清，称为空悬 *else(dangling-else)* 问题。如果写几条 if 语句，一个嵌套在另一个里面，其中有些使用 *else* 子句，有些则没有，这

种情况下很难说清哪个 `else` 与哪个 `if` 匹配。面对这种情况，每条 `else` 子句与最近且没有 `else` 子句的 `if` 语句相匹配，Java 编译器遵循这一简单规则。虽然这条规则对于编译器而言很简单，但对于人类读者而言很难快速识别每条 `else` 子句属于哪里。采用比 Java 要求更严格的编程风格，则可能消除空悬 `else` 模糊。

if/else 代码块规则

`type identifier = expression;`

其中：

`type` 是变量的类型；

`identifier` 是变量的名称；

`expression` 指定初始值。

本书采用上面所示的代码块规则来消除空悬 `else` 问题。所以，`if` 语句只以下面 4 种形式之一出现：

- 用于非常简短条件的单行 `if` 语句。
- 多行 `if` 语句，这种情况下语句被封装在代码块里。
- `if-else` 语句，它总是使用代码块来封装由 `if` 语句控制的语句，即使它们由单个语句组成。
- 级联 `if` 语句，用于表示一系列条件测试。

下面几节将详细讨论这些形式。

4.3.1 单行 if 语句

简单的单行形式如下边的语法框所示，它只用于这样的 `if` 语句：这些语句中没有 `else` 子句，这些语句的主体是非常简短的单个语句，简短到可以应用与 `if` 相同的行。在这种情况下，使用大括号并将 `if` 语句从一行扩充到 3 行，让程序变得更长、更难阅读。

单行 if 语句的语法

`if (condition) statement;`

其中：

`condition` 是 Boolean 表达式；

`statement` 是 `condition` 为 `true` 时执行的单个语句。

4.3.2 多行 if 语句

如果 `if` 语句的主体由复合语句或对于单行来说太长的简单语句组成，那么该代码应该封装在代码块里，如下边的语法框所示。这种形式中，如果条件是 `true`，就执行代码块里的代码。如果条件是 `false`，程序根本不会执行代码块，而是继续执行 `if` 之后的语句。

多行 if 语句的语法

```
if (condition) {
    statements;
}
```

其中：

condition 是 Boolean 表达式；

statements 是条件为 true 时执行的语句代码块。

4.3.3 if-else 语句

要避免空悬 else 问题，包含 else 子句的 if 语句的主体总是封装在代码块里，如下边的语法框所示。从技术上说，当有多条条件控制语句时才需要使用包含代码块的花括号。然而，系统使用这些花括号，可以大大降低混淆的可能性，让程序更易于维护。

if-else 语句的语法

```
if (condition) {  
    statementsT  
} else {  
    statementsF  
}
```

其中：

condition 是 Boolean 表达式；

statements_T 是 *condition* 为 true 时执行的语句代码块；

statements_F 是 *condition* 为 false 时执行的语句代码块。

4.3.4 级联 if 语句

if 语句的最后一个语法框介绍了一种重要的特殊情况。这种情况对于应用程序很有用，其中可能的情况不止两个。典型形式是条件的 else 部分由另一种测试组成，它用来检验一种二选一的条件。这种语句称为级联 if 语句，可以包含任何数量的 else if 行。例如，图 4-4 中的 SignTest 程序使用了级联 if 语句来报告某个数是正数、零还是负数。注意，没有必要明确检验 $n < 0$ 的条件。如果程序执行到最后一条 else 子句，就没有其他可能性，因为前面的测试已经排除了正数和零的情况。

级联 if 语句的语法

```
if (condition1) {  
    statements1  
} else if (condition2) {  
    statements2  
} else if (condition3) {  
    statements3  
}  
...  
} else {  
    statementsnone  
}
```

其中：

每个 *condition_i* 都是 Boolean 表达式；

每个 *statements_i* 都是 *condition_i* 为 true 时执行的语句代码块；

statements_{none} 是没有 *condition* 为 true 时执行的语句代码块。

```
/*
 * File: SignTest.java
 *
 * This program reads in an integer and classifies it as negative,
 * zero, or positive depending on its sign.
 */

import acm.program.*;

public class SignTest extends ConsoleProgram {
    public void run() {
        println("This program classifies an integer by its sign.");
        int n = readInt("Enter n: ");
        if (n > 0) {
            println("That number is positive.");
        } else if (n == 0) {
            println("That number is zero.");
        } else {
            println("That number is negative.");
        }
    }
}
```

图 4-4 根据符号分类整数的程序

许多情况下, 使用 switch 语句选择单独情况比使用级联 if 的形式要好, 4.4 节将介绍 switch 语句。

4.3.5 ?:运算符

Java 编程语言提供另一种更简洁的方法来表示条件执行, 这种方法在某些情况下非常有用。这种方法就是?:运算符(该运算符称为问题标记冒号, 即使实际上两个字符没有紧挨着出现)。在 Java 中, ?:运算符写成两个部分, 需要 3 个操作数, 这一点与其他运算符不同。该操作的通常形式是:

condition ? *expression₁*: *expression₂*

当 Java 遇到?:运算符时, 它首先给条件赋值。如果条件是 true, 就给 *expression₁* 赋值, 并将它用作整个表达式的值; 如果条件是 false, 值就是赋值 *expression₂* 的结果。因此?:运算符是 if 语句的简写形式。

```
if (condition) {
    value = expression1;
} else {
    value = expression2;
}
```

其中, ?:整个表达式的值应该存储在扩充的 if 语句的 *value* 变量里。

例如, 可以使用?:运算符来设置 *max* 是 *x* 或 *y* 中值较大的那个, 如下所示。

max = (*x* > *y*) ? *x* : *y*;

从技术上说不需要条件外面的圆括号, 但是许多 Java 编程人员倾向于在上下文中包含它

们，以增强代码的可读性。

?:运算符最有用最常见的情况之一是调用 `println`，其中的输出依据某些条件而稍有不同。例如，假设编写程序计算一些项的数量，计算完后，将项的数量存储在变量 `nItems` 里。如何将这个值告诉用户？很明显的方法是使用下面的语句调用 `println`。

```
println(nItems + " items found.");
```

但如果是语法纯粹主义者，当 `nItems` 刚好是值 1 时，阅读这样的输出(如图 4-5 所示)可能有点别扭。



图 4-5 ItemCount 程序的输出

然而，可以在下面的 if 语句中包含 `println` 行来更正英语：

```
if (nItems == 1) {
    println(nItems + " item found.");
} else {
    println(nItems + " items found.");
}
```

唯一问题是这种解决方案策略需要 5 行语句来表达相对简单的想法。作为另一种方法，可以使用?:运算符，如下所示。

```
println(nItems + " item" + (nItems == 1 ? "" : "s") + " found.");
```

如果 `nItems` 等于 1 而字符串 “s” 不等于 1，那么输出中的“item”字符串后面紧接着空字符串。注意，下面这个表达式中的圆括号必不可少。

```
(nItems == 1 ? "" : "s")
```

?:运算符与+运算符相比，优先级相对较低，说明 Java 会先进行串联。虽然?:在某些上下文中非常有用，但是也容易过度使用它。如果将决策结构的基本部分嵌入?:运算符，就会产生问题。因为很难找到决策的代码。另一方面，如果使用?:就可能处理小的细节而不必写复杂的 if 语句，这个运算符可以极大简化程序结构。

4.4 switch 语句

if 语句对于程序逻辑要求双向决策点的应用程序非常完美：条件要么是 `true` 要么是 `false`，程序据此采取行动。然而，有些应用程序要求更复杂的包括不止两个选择的决策结构，这些选择可以分成互相排斥的一组情况：一种情况下，程序应该执行 `x`；另一种情况下，应该执行 `y`；第三种情况，它应该执行 `z`，等等。许多应用程序中，这种情况下最合适的是 `switch` 语句。

其语法如下边的语法框所示。

switch 语句的语法

```
switch (e) {
    case c1:
        statements1;
        break;
    case c2:
        statements2;
        break;
    . . . more case clauses . . .
    default:
        statementsdef;
        break;
}
```

其中：

e 是控制表达式，它用来选择执行什么语句；

c_i 都是常量；

statements_i 是 *c_i* 等于 *e* 时执行的一系列语句；

statements_{def} 是没有 *c_i* 与表达式 *e* 匹配时执行的一系列语句。

switch 语句的标题行是：

```
switch (e)
```

其中 *e* 是给整数赋值的表达式(或者，像整数这样的任何值，如字符。这将在第 8 章讨论)。在 switch 语句的上下文中，表达式称为控制表达式。switch 语句的主体分为下面两个关键字之一引入的单个语句组：case 和 default。case 行以及从它开始一直到关键字的下一个实例之间的语句称为 case 语句；default 行及其相关语句称为 default 语句。例如，左边语法框所示的模板中，语句

```
case c1:
    statements1;
    break;
```

构成第一个 case 子句。

程序执行 switch 语句时，控制表达式 *e* 被赋值，并与值 *c₁*、*c₂* 等进行比较，每个都必须是常量。如果其中某个常量与控制表达式的值匹配，就会执行相关 case 子句中的语句。程序一直到子句结尾的 break 语句，该子句指定的操作才完成，程序继续执行下面整个 switch 语句。如果没有 case 常量匹配控制表达式的值，就会执行 default 子句中的语句。

常见错误

在 switch 语句内每个 case 子句结尾加上 break 语句是一个好的编程习惯。这样做有利于避免出现很难发现的编程错误。包含 default 子句也是一个好的习惯，除非包含了所有情况。

上边语法框里所示的模板说明，break 语句是语法不可或缺的一部分。鼓励这样看待 switch

语法。Java 定义，如果没有 `break` 语句，程序在执行完选择的语句之后，开始执行下一个子句的语句。这种设计有些情况下有用，但是它导致的问题比解决的问题还多。记住要包含 `break` 语句，这很重要。为了强化这种重要性，本文中的每条 `case` 子句都以明确的 `break` 语句结尾(有时以 `return` 语句结尾，如第 5 章所述)。

该规则的一个例外是，在相同语句组之前，指定不同常量的多个 `case` 行可以依次出现。例如，`switch` 语句可以包括如下代码：

```
case 1; case 2:  
    statements  
    break;
```

它表示，如果 `select` 表达式是 1 或者 2，就应该执行指定的语句。Java 编译器将该结构作为两个 `case` 子句看待，第一个子句是空的。因为空的子句没有 `break` 语句，选择该路径的程序就继续执行第二个子句。然而，从概念上来看，将这种结构当作代表两种可能性的单个 `case` 子句可能更好。

在 `switch` 语句中，`default` 子句是可选的。如果没有情况与条件匹配，也没有 `default` 子句，程序就会继续执行 `switch` 语句后的下一条语句。要避免程序可能忽略特殊情况的可能性，在每条 `switch` 语句中都包含 `default` 子句是一种好的编程实践，除非可以肯定已经列举了所有可能性。

因为 `switch` 语句可能很长，如果 `case` 子句本身很简短，程序就更容易阅读。如果没有空间这么做，那么将 `case` 标识符、组成子句主体的语句及 `break` 语句放在同一行，也很有帮助。图 4-6 中的 `CardRank` 程序就说明了这种风格。图中是 `switch` 语句的示例，它可能在编写纸牌游戏程序时有用。在此游戏中，每一副纸牌都用数字 1~13 表示。对于 2~10 的纸牌，显示上面的数字即可。但是这种风格的输出对于值 1、11、12 和 13 就不太令人满意，因为它们分别用 Ace、Jack、Queen 和 King 表示。`CardRank` 程序使用 `switch` 语句来显示每张纸牌正确的标记。

```
/*  
 * File: CardRank.java  
 *  
 * This program reads in an integer between 1 and 13 and  
 * prints the appropriate symbol for a playing card  
 * of that rank.  
 */  
  
import acm.program.*;  
  
public class CardRank extends ConsoleProgram {  
  
    public void run() {  
        println("This program converts integers to card ranks.");  
        int n = readInt("Enter an integer between 1 and 13: ");  
        switch (n) {  
            case 1: println("Ace"); break;  
            case 11: println("Jack"); break;  
            case 12: println("Queen"); break;  
            case 13: println("King"); break;  
            default: println(n); break;  
        }  
    }  
}
```

图 4-6 将纸牌中的整数转换为相应级别的程序

switch 语句只可以用来选择整数(或类似整数)常量确定的情形，这实际上限制了它的使用。您会遇到这样的情况：就是要使用字符串值来选择各种情况，或者用作 case 标识符的值不是常量。由于在这些情况下不能使用 switch 语句，所以要采用级联 if 语句。另一方面，在可能的情况下尽量使用 switch 语句能够让程序更具可读性，更高效。

4.5 while 语句

最简单的迭代结构是 while 语句，它重复执行简单语句或代码块，直到条件表达式的值变成 false 为止。while 语句的模板如下边语法框所示。使用 if 语句，如果主体由单个语句组成，Java 编译器允许省略主体外面的花括号。然而，为了提高可读性，一般会将主体封装在大括号里。

while 语句的语法

```
while (condition) {
    statements
}
```

其中：

condition 是条件测试，用来决定循环是不是应该继续到下一周期；

statements 是要重复的语句。

整个语句，包括 while 控制行本身和主体内的语句，组成了 while 循环。程序执行 while 语句时，先对条件表达式求值，看看它是 true 还是 false。如果值为 false，循环终止；程序继续整个循环后的下一条语句。如果条件是 true，程序执行整个主体；之后，程序会回到最上面，再次核对条件。经过主体内的语句一次构成一个循环周期。

运行 while 循环要遵守两条重要原则：

- 每个循环周期(包括第一个周期)之前执行条件测试如果最初测试是 false，就不会执行循环主体。
- 只是在循环周期开始时执行条件测试。如果在循环过程中某个时候条件值刚好是 false，程序在执行整个周期之前不会注意到这一点。在那种情况下，程序会再次对测试条件求值。如果还是 false，程序终止。

4.5.1 使用 while 循环

学习有效使用 while 循环通常需要考察几个示例，在这些示例中 while 循环很自然地出现在解决方案策略中。例如，要求编写一段调用 DigitSum 的程序来相加正整数。假设输入“1729”，程序的运行如图 4-7 所示。

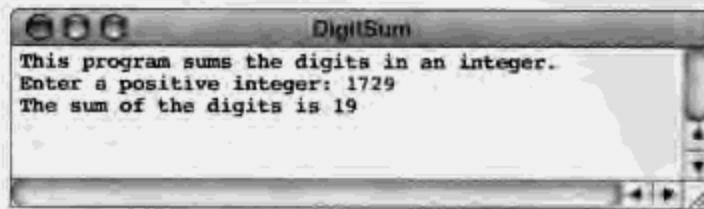


图 4-7 DigitSum 程序的运行结果

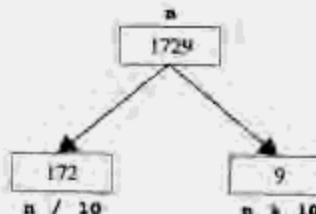
值 19 是对输入数字相加的结果: $1 + 7 + 2 + 9 = 19$ 。怎样编写这样的程序呢?

本章前面讨论图 4-1 的 AddNIntegers 程序时, 已经介绍了如何进行连续的加法运算。要解决 DigitSum 问题, 需要为和声明一个变量, 并将它初始化为 0, 再相加每个数字, 最后显示和。其结构以及用英语写的问题的余下部分如下所示。

```
public void run() {
    println("This program sums the digits in an integer.");
    int n = readInt("Enter a positive integer");
    int dsum = 0;
    For each digit in the number, add that digit to dsum.
    println("The sum of the digits is " + dsum);
}
```

语句 “*For each digit in the number, add that digit to dsum*” 明确指定了某种类型的循环结构, 因为数中的每个数字都需要重复一种运算。如果确定一个数里包含有多少个数字很容易, 可以选择使用 for 循环计算数字的个数。很遗憾, 找出在整数里有多少数字和首先将它们相加一样困难。编写这个程序的最好方法是不断相加数字, 一直加到最后一个为止。在某个条件出现之前运行的循环通常使用 while 语句编码。

这个问题的本质是决定如何将数分解为组成它的数字。算术运算符 / 和 % 足以完成任务, 想到这一点至关重要。整数 n 的最后一位数字是 n 除以 10 之后的余数, 这是表达式 $n \% 10$ 的结果。数剩下的部分——由所有数字(最后一位数除外)组成的整数——通过 $n / 10$ 确定。例如, 如果 n 的值是 1729, 可以使用 / 和 % 运算符将这个数分解为两部分, 172 和 9, 如下所示。



因此, 为了相加数中的数字, 要做的是每个循环周期都将值 $n \% 10$ 和变量 $dsum$ 相加, 然后将数 n 除以 10。下一个周期将相加原始数的第二位到最后一位数字, 依此类推, 直到按这种方法处理完整个数为止。

但是, 怎么知道何时停止呢? 最后, 因为每个周期中都将 n 除以 10, 所以总会有 n 变成 0 的时候。那个时候所有数中的数字就处理完, 可以退出循环了。换句话说, 只要 n 的值大于 0, 就要继续。因此, 这个问题所需的 while 循环如下所示。

```
while (n > 0) {
    dsum +=
    n % 10;
    n /= 10;
}
```

整个 DigitSum 程序如图 4-8 所示。

```

/*
 * File: DigitSum.java
 *
 * This program sums the digits in a positive integer.
 * The program depends on the fact that the last digit of
 * an integer n is given by n % 10 and the number consisting
 * of all but the last digit is given by the expression n / 10.
 */

import acm.program.*;

public class DigitSum extends ConsoleProgram {
    public void run() {
        println("This program sums the digits in an integer.");
        int n = readInt("Enter a positive integer: ");
        int dsum = 0;
        while (n > 0) {
            dsum += n % 10;
            n /= 10;
        }
        println("The sum of the digits is " + dsum);
    }
}

```

图 4-8 相加整数中数字的程序

4.5.2 无限循环

在程序中使用 while 循环时，用于控制循环的条件最终应该变成 false，以便退出循环，确保这一点非常重要。如果 while 控制行的条件的值总是为 true，计算机会一个周期接一个周期地不停执行。这种情况称为无限循环。

例如，假设写 DigitSum 程序中的 while 循环，无意间将控制行里的 >运算符写成了 \geq 运算符，如下所示。

```

while (n >= 0) {
    dsum +=
    n % 10;
    n /= 10; }

```



常见错误

仔细思考 while 循环中使用的条件表达式，确保循环最终可以退出。永远不能完成的循环称为无限循环。

当 n 减少为 0 时循环也不会像正确编码的示例里那样停止。相反，每次 n 等于 0 时，计算机就会一遍又一遍地执行主体。

要停止无限循环，必须输入在键盘上输入特殊命令序列来打断程序，强制使其退出。计算机与计算机间这种命令序列有所不同，应该确保学会在自己的计算机上使用什么命令。

4.5.3 解决“循环到中途”问题

执行循环主体内的所有语句之前，在重复操作开始时能够应用某个测试条件，while 循环适用于这样的情形。如果要解决的问题满足这种结构，while 循环就是一个完美的工具。很遗

憾，许多编程问题不完全符合标准的 while 循环模板。有些问题不是在操作开始时就有方便的测试，它们的结构是，用来决定循环是否完成的测试就在循环中间。

例如，考察读取输入数据直到终止条件出现为止这个问题(在本章前面“读取到终止条件为止”模式中讨论过)。用英语表达时，基于终止条件循环的结构就是重复下面的步骤：

- (1) 读入值。
- (2) 如果值等于终止条件，退出循环。
- (3) 执行该值要求的所有操作。

很遗憾，在循环一开始时不能执行测试来确定循环是否完成。当输入值等于终止条件时就是循环的终止条件；为了核对这个条件，程序必须首先读入某个值。如果程序仍没有读入值，终止条件就没有意义。在程序进行有意义的测试之前，它必须执行部分读入输入值的循环。当循环包含一些必须在完成测试之前执行的操作时，它就是编程人员所说的循环到中途(loop-and-a-half)问题实例。

Java 中解决这个问题的方法之一是使用 break 语句，它除了在 switch 语句中使用之外，还可以立即终止最里面的循环。使用 break 语句，就可以依据问题本质结构为哨兵问题编码循环结构。

```
while (true) {
    prompt user and read in a value
    if (value == sentinel) break;
    process the data value
}
```

需要解释一下首行：

```
while (true)
```

定义 while 循环，以便它可以继续执行直到圆括号内的条件变成 false 为止。符号 true 是常量，因此它绝对不可能变成 false。那么，说到 while 语句本身，循环就从不会终止。程序可以退出循环的唯一方法是在内部执行 break 语句。

编码这种类型的循环可以不使用 while (true) 控制行或 break 语句。然而，要这么做，必须改变循环内操作的顺序，并要求在两个地方输入数据：一次在循环开始之前，第二次是在循环主体内部。以这种方法构建，基于终止条件循环的模板如下：

```
prompt user and read in the first value
while (value != sentinel) {
    process the data value
    prompt user and read in a new value
}
```

图 4-9 所示为，不使用 break 语句，如何将该模板用于执行图 4-2 中的 AddIntegerList 程序。

遗憾的是，使用这种策略有两个缺点。第一个问题是循环中的操作顺序不是像大多数期望的那样。许多对解决方案策略的英语解释中，第一步是读取一个数，第二步是将它与总数相加。图 4-9 中使用的 while 循环模板颠倒了循环内语句的顺序，使程序变得难以理解。第二个问题是这种模板需要用来读入数的语句的两个副本。代码复制产生了严重的维护问题，因为对一组语句的后续编辑可能在另一组语句内就实施不了。经验研究表明，学习使用 break 语句解决

循环到中途(loop-and-a-half)问题的同学比那些没有学习使用该语句的同学，更可能写出正确的程序。

```
/*
 * File: AddIntegerList.java
 *
 * This program reads integers one per line until the
 * user enters a special sentinel value to signal the
 * end of the input. At that point, the program
 * prints the sum of the numbers entered so far.
 */

import acm.program.*;

public class AddIntegerList extends ConsoleProgram {
    public void run() {
        println("This program adds a list of integers.");
        println("Enter values, one per line, using " + SENTINEL);
        println("to signal the end of the list.");
        int total = 0;
        int value = readInt(" ? ");
        while (value != SENTINEL) {
            total += value;
            value = readInt(" ? ");
        }
        println("The total is " + total + ".");
    }
    /* Specifies the value of the sentinel */
    private static final int SENTINEL = 0;
}
```

图 4-9 不使用 break 语句重写 AddIntegerList 程序

4.6 for 语句

Java 中最重要的控制语句之一是 for 语句，它通常用于以特定次数重复某个操作。for 语句的通用形式如下边的语法框所示。

for 语句的语法

```
for (init; test; step) {
    statements
}
```

其中：

init 是初始化循环索引变量的声明；

test 是用来确定循环是否应该继续的条件测试，就像在 while 语句中一样；

step 是用于准备下一个循环周期的表达式；

statements 是需要重复的语句。

for 循环操作由 for 控制行里的 3 个斜体表达式确定: *init*、*test* 和 *step*。*init* 表达式通常声明索引变量并设置其初始值, 这样来表示 for 循环应该如何初始化。例如, 如果编写

```
for (int i = 0; . . .
```

循环通过声明索引变量 i 并将它的值设置为 0 开始。如果循环开始是

```
for (int i = -7; . . .
```

变量 i 将从 -7 开始, 等等。

test 表达式是条件测试, 就像 while 语句中的测试一样。只要表达式值是 true, 循环就会继续。因此, 到目前为止在作为规范示例的循环中,

```
for (int i = 0; i < n; i++)
```

循环从 i 等于 0 开始, i 只要小于 n(它表示 n 周期的总数), 循环就会继续; i 从取值 0、1、2, 依次类推, 直到最后一个值 n-1 为止。循环

```
for (int i = 1; i <= n; i++)
```

从 i 等于 1 开始, 只要 i 小于或等于 n 就继续。这个循环也运行 n 个周期, 从 i 取值 1、2, 依次类推, 直到 n 为止。

step 表达式表示索引变量的值如何一个周期一个周期地变化。步骤规范最常见的形式是使用++运算符递增索引变量, 但这不是唯一可能的形式。例如, 可以使用--运算符进行倒数, 或使用+=2 代替++两个地计数。

作为反方向计数的说明, 图 4-10 中的 Countdown 程序从 10 到 0 向下计数。执行 Countdown 程序时, 生成如图 4-11 所示结果。

```
/*
 * File: Countdown.java
 *
 * This program counts backwards from the value START
 * to zero, as in the countdown preceding a rocket
 * launch.
 */

import acm.program.*;

public class Countdown extends ConsoleProgram {
    public void run() {
        for (int t = START; t >= 0; t--) {
            println(t);
        }
        println("Liftoff!");
    }

    /* Specifies the value from which to start counting down */
    private static final int START = 10;
}
```

图 4-10 火箭发射数列中倒计时到 0 的程序



图 4-11 Countdown 程序的运行结果

Countdown 程序说明任何变量都可用作索引变量。在这种情况下，变量称为 *t*，可能因为它是火箭倒数计秒传统变量，就像在短语“T 减 10 秒，计数”。在任何情况下，必须在程序开始时声明索引变量，就像其他变量一样。

表达式 *init*、*test* 和 *step* 都是可选的，但是分号是必需的。如果没有 *init* 表达式，就不会执行初始化。如果没有 *test*，就假定为 true。如果没有 *step* 表达式，循环周期之间就不会发生任何行动因此，控制行

```
for (;;) {
```

在效果上与下面的语句一样。

```
while (true) {
```

4.6.1 for 和 while 之间的关系

刚好，*for* 语句

```
for (init; test; step) {
    statements;
}
```

的效果与 *while* 语句是一样的。

```
init;
while (test) {
    statements;
    step;
}
```

即使使用 *while* 语句可以很容易重写 *for* 语句，也要在可能使用 *for* 语句的情况下尽量使用它，这样做很有好处。使用 *for* 语句，语句的标题行包含了正确理解将执行哪个周期所需的全部信息。例如，在程序中看到语句：

```
for (int i = 0; i < 10; i++) {
    . . . body . . .
```

就会知道，循环主体中的语句会执行 10 次，*i* 从 0 到 9 的每个值都执行一次。在与它对等的 while 循环形式中：

```
int i = 0;
while (i < 10) {
    . . .
    body . . .
    i++;
}
```

如果主体较大，就很容易丢失循环最下面的递增操作。

4.6.2 在浮点数据中使用 for 语句

因为 for 循环的 init、test 和 step 部分可以是任意表达式，所以没有明显的理由说明 for 循环的循环索引必须是整数。可以使用 for 循环两个两个地从 0 计数到 10，这说明

```
for (int i = 0; i <= 10; i += 2) . . .
```

通过声明循环索引变量为 double，也可以按递增 0.1 从 1.0 计数到 2.0，例如，要显示这个范围的值，可以按如下形式编写：

```
for (double x = 1.0; x <= 2.0; x += 0.1) {
    println(x);
}
```

然而，如果在 Java 中运行程序，不会看到从 1.0、1.1、1.2 依次类推到 2.0 的列表，而是会看到如图 4-12 所示的输出。

```
1.0
1.1
1.2000000000000002
1.3000000000000003
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
```

图 4-12 ForLoopWithDouble 的输出结果

有许多无关的数字，没有值接近 2.0。

问题是浮点数不是太精确。值 0.1 近似于数学中的分数 $\frac{1}{10}$ ，但不是完全等于它。由于 0.1 不断地加入索引变量 *x* 中，不精确度会积累到一定程度，当用 2.0 测试 *x* 来确定循环是否完成时，它的值可能是类似 2.000000001 这样的值，大于 2.0。因此 for 循环的条件没有满足，循环在多运行几个周期后终止。解决这一问题的最好方法是对使用整数作为 for 循环的索引变量加以限制。因为整数是精确的，所以就不会产生问题。

常见错误

测试浮点数是否相等时要小心。因为浮点数只是近似值，它们与数学里的实数不同。一般来说，最好不要使用浮点变量作为 for 循环索引。

有关比较浮点数是否相等的相同警告除了适用于 for 循环之外，还适用于其他许多上下文，考虑到浮点数精确度的限制，看起来似乎刚好相等的数也未必相等。

4.6.3 嵌套 for 语句

许多应用程序中，会发现需要在一个 for 循环中再写另一个 for 循环，这样在最里面循环里的语句，for 循环索引值每个可能的组合它都要执行。这种情况通常在图形应用程序中出现，这种程序中需要在 x 和 y 这两个方向上重复某些操作。例如，如何编写一段程序生成如图 4-13 所示的西洋跳棋盘图案。

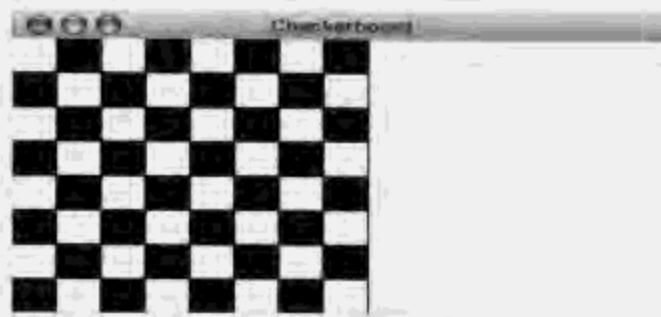


图 4-13 西洋跳棋盘图案

西洋跳棋盘由 8 条水平行和 8 条垂直列组成。要创建单个方块，就需要一对嵌套的 for 循环：生成每行的外部循环和生成每列的内部循环。内部 for 循环里的代码每一行每一列都执行一次，总共 64 个单独方块。

画这种西洋跳棋盘的程序如图 4-14 所示。值得花点时间仔细考察一下代码，特别要注意下面的细节：

- 改变命名常量 N_ROWS 和 N_COLUMNS 的定义，就可以很方便地改变棋盘的大小，程序应该这样设计。
- 应该这样安排棋盘：让它的左边与图形窗口的左边对齐。计算的单个方块大小，让棋盘填充所有的垂直空间，意思就是说，每个方块的大小必须是窗口的高度除以行数。
- 内部 for 循环里的前几条语句计算当前方块与计算方块之间的坐标以及 for 循环索引 i 和 j。
- 检验行数和列数的和是偶数还是奇数，可以决定是否填充方块。和是偶数，方块就是白色；和是奇数，方块就是黑色。但是注意，不需要在代码里包含 if 语句来测试条件。要做的就是用合适的 Boolean 值调用 setFilled 方法。

```
/*
 * File: Checkerboard.java
 *
 * This program draws a checkerboard. The dimensions of the
 * checkerboard are specified by the constants N_ROWS and
 * N_COLUMNS, and the size of the squares is chosen so
 * that the checkerboard fills the available vertical space.
 */

import acm.graphics.*;
import acm.program.*;

public class Checkerboard extends GraphicsProgram {
    public void run() {
        double sqSize = (double) getHeight() / N_ROWS;
        for (int i = 0; i < N_ROWS; i++) {
            for (int j = 0; j < N_COLUMNS; j++) {
                double x = j * sqSize;
                double y = i * sqSize;
                GRect sq = new GRect(x, y, sqSize, sqSize);
                sq.setFilled((i + j) % 2 != 0);
                add(sq);
            }
        }
    }

    /* Private constants */
    private static final int N_ROWS = 8;
    private static final int N_COLUMNS = 8;
}
```

图 4-14 画西洋跳棋盘的程序

4.6.4 简单的图形动画

4.6.3 小节里的 Checkerboard 程序很清楚地说明, for 循环在图形程序里和在基于控制台的程序里一样有用。在 Checkerboard 示例中, for 循环使构造图表变得非常容易,

它通过为 8 行 8 列重复相同的代码使之成为可能。到目前为止学习其他所有 GraphicsProgram 示例中, 程序生成的图像都是固定的。画出棋盘之后, 它们被动地出现在屏幕上。如果要写一些更有意思的程序, 让图形对象在屏幕上来回移动、或者改变其大小和颜色就显得很重要。在计算机图形图像中, 更新显示图像使其随着时间的推移而改变的过程称为动画。

使图形程序变成动画最简单的方法是在里面包含一个循环, 这个循环可以在屏幕上对图形对象稍做改变, 并将程序暂缓一小会儿。这种图形风格的简单示例如图 4-15 所示, 它将方块沿屏幕对角线从初始位置左上角移动到最终位置右下角。图 4-16 说明了程序的操作, 用灰色表示方块的初始位置, 用箭头表示大概轨迹。

```

/*
 * File: AnimatedSquare.java
 *
 * This program animates a square so that it moves from the
 * upper left corner of the window to the lower right corner.
 */

import acm.graphics.*;
import acm.program.*;

public class AnimatedSquare extends GraphicsProgram {
    public void run() {
        GRect square = new GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
        square.setFilled(true);
        add(square);
        double dx = (getWidth() - SQUARE_SIZE) / N_STEPS;
        double dy = (getHeight() - SQUARE_SIZE) / N_STEPS;
        for (int i = 0; i < N_STEPS; i++) {
            square.move(dx, dy);
            pause(PAUSE_TIME);
        }
    }

    /* Private constants */
    private static final int N_STEPS = 1000;
    private static final int PAUSE_TIME = 20;
    private static final double SQUARE_SIZE = 50;
}

```

图 4-15 沿屏幕对角线移动方块的程序。

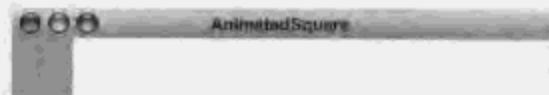


图 4-16 移动方块的简单示例

图 4-15 中 run 方法的前 3 行创建方块，设置为填充，并将它显示在屏幕的左上角。run 方法余下的行负责将方块以一序列时间步长移动到最终位置，这些时间步长中方块一点点地移动。执行中，命名常量 N_STEPS 指定时间步长的数量。要确定在每个时间步长里方块移动多远，需要用移动的总距离除以步长数。下面的行

```

double dx = (getWidth() - SQUARE_SIZE) / N_STEPS;
double dy = (getHeight() - SQUARE_SIZE) / N_STEPS;

```

在每个方向上执行这种计算。改变每个周期内水平和垂直方向上的 dx 和 dy 来移动方块的位置，程序可以确保方块最后会到达右下角这个期望的位置。

动画本身由 run 法末尾的 for 循环执行，每个时间步长它都要循环，如下所示：

```
for (int i = 0; i < N_STEPS; i++) {  
    square.move(dx, dy);  
    pause(PAUSE_TIME);  
}
```

每个时间步长中，程序都调用

```
square.move(dx, dy);
```

它用各方向上适合的距离来调整方块的位置。for 循环的第二行是

```
pause(PAUSE_TIME);
```

它让程序暂停指定毫秒的时间。如果暂停时间为 20ms，说明显示更新频率为 50 次/s，这远远低于人眼感知运动连续的极限。调用 AnimatedSquare 程序的 pause 对于实现动画效果必不可少。今天计算机运行得如此迅速，如果不减缓移动速度以便让人能够看到该操作，方块就会马上跳到右下角。

可以采用两种方法改变动画速度。第一种方法，可以改变将对象移动特定距离所需的时间步长的数量，例如，如果将时间步长数从 1000 改变为 250，方块的移动速度就会快两倍，因为 dx 和 dy 的值比原来大 4 倍。然而，有些情况下，dx 和 dy 的值可能很大，以至于动作很急促。第二种方法是改变延迟时间。从直观上看，如果将每个周期的延迟时间减半，好像就可以让显示的速度加倍，尽管这种策略在实践中也有局限。不能保证程序调用 pause 时出现的延迟很精确。而且，关于暂停程序总有某个极限，不可能以小于该极限的周期时间运行动画。因此，总会在某种情况下，减少示例中的 PAUSE_TIME 值不会再产生任何明显效果。

4.7 小结

第 2 章我们从整体上讲解了编程的过程。同时也介绍了几种非正式的控制语句。本章详细研究了这些语句使用方法。本章介绍的重点是：

- 简单语句由后面加分号的表达式组成。
- 用于指定赋值的=在 Java 中是运算符。因此，赋值是合法的表达式，这就可能写嵌套赋值和多重赋值。
- 可以将单个语句集合成复合语句(通常称为代码块)。
- 控制语句属于两类：条件语句和迭代语句。
- 当仅在某些情况下执行部分代码或程序需要在两种可选路径中做出选择时，if 语句才指定条件执行。
- 当问题的结构如下时，switch 语句指定条件执行：情况 1，这样做；情况 2，那样做。以此类推。
- while 语句指定重复，这种重复只要满足某个条件就会发生。
- for 语句指定重复，在这种重复中，为了更新索引变量的值，每个周期都要求完成某种动作。

- 在循环的每个周期稍稍更新图形图像，然后调用 `pause` 来将程序延时一小会，可以使图形图像变成动画。

4.8 复习题

1. 下面为种结构在 Java 中是合法语句吗？它有用吗？

17:

2. 什么是代码块？“复合语句”一词传达了代码块的哪些重要信息？这个概念的另一个名字是什么？

3. 两类控制语句是什么？

4. 嵌套两个控制语句是什么意思？

5. 本文中使用的 if 语句的 4 种不同格式是什么？

6. 用英语描述 switch 语句的一般操作。

7. 假设 while 循环的主体包含语句，当执行时，会导致该 while 循环的条件值变成 false。循环此时会立即终止吗？它会执行完当前周期吗？

8. 对于图 4-8 中的 DigitSum 程序而言，指定整数是正数为什么很重要？

9. 什么是“循环到中途”问题？本文为解决这个问题提供了哪两种策略？

10. for 语句控制行中有 3 个表达式，每个表达式的目的分别是什么？

11. 在下面这些情况下，应该使用什么样的 for 循环控制行？

(1) 从 1 到 100 计数。

(2) 从 0 开始，7 个 7 个地计数，直到数超过两位数为止。

(3) 从 100 到 0，两个两个地倒数计数。

12. 最好不要使用浮点变量作为 for 循环里的索引变量，为什么？

13. 用自己的话描述本章用来执行简单动画的策略。

14. 在图 4-15 所示的 AnimatedSquare 程序中，调用 `pause` 的目的是什么？如果删除对 `pause` 的调用，会出现什么情况？

4.9 编程练习

1. 作为在长途汽车上打发时间的一种方法，现在的美国年轻人都知道唱下面这首反复性的歌曲：

墙上有 99 瓶啤酒。

99 瓶啤酒。

拿下来一瓶，传出去。墙上有 98 瓶啤酒。

墙上有 98 瓶啤酒……

因此，问题就是，编写一段 Java 程序表达这首歌的意思。在测试程序的过程中，使用其他常量(不是 99)作为瓶子的初始数，这样会更好。

2. 上面还在讨论打发时间的歌曲，而下面是另一首老歌 *This Old Man*，第一段是：

```

This old man, he played 1.  

He played knick-knack on my thumb.  

With a knick-knack, paddy-whack,  

Give your dog a bone.  

This old man came rolling home.

```

除了数字和第二行押韵的单词不同之外(用下面的取代)，后面的每段乐章都相同。

2—shoe	5—hive	8—pate
3—knee	6—sticks	9—spine
4—door	7—heaven	10—shin

写程序显示这首歌全部 10 段。

3. 编写一段程序，读入正数 N，然后计算并显示前 N 个奇数整数的和。例如，如果 N 等于 4，程序应该显示 16，即 $1+3+5+7$ 。

4. Why is everything either at sixes or at sevens?

—Gilbert and Sullivan, *H.M.S. Pinafore*, 1878

编写一段程序，显示 1~100 之间可被 6 或 7 除尽，但不能同时被 6 和 7 除尽的整数。

5. 使用图 4-2 中的 AddIntegerList 程序作为模型，写一段名为 AverageList 的程序，它读入一列表示考试分数的整数，然后打印出平均值。因为有些没有准备好的学生可能会得 0 分，所以程序应该使用 -1 作为终止条件来标记输入结束。

6. 重写图 4-8 中的 DigitSum 程序，让它不是相加数中的数字，而是生成数字以相反顺序排列的数，运行结果如图 4-17 所示。

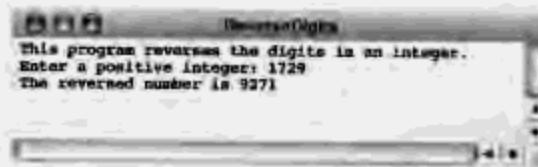


图 4-17 ReverseDigits 程序的样本运行

该练习的思想不是将整数按字符分解(这将在第 8 章讨论)，而是需要使用算术来计算顺序相反的整数。本示例中，循环第 1 个周期后新的整数是 9，第 2 个周期后是 92，第 3 个周期后是 927，第 4 个周期后是 9271。

7. 整数 n 的数字根定义为重复求数字和直到只有一位数字时的结果。例如，1729 的数字根可以使用下面的步骤进行计算：

$$\begin{array}{rcl}
 \text{第1步: } 1 + 7 + 2 + 9 & = & 19 \\
 \text{第2步: } 1 + 9 & = & 10 \\
 \text{第3步: } 1 + 0 & = & 1
 \end{array}$$

因为第 3 步结束后的总数是单个数字 1，这个值就是数字根。

重写图 4-8 中的 DigitSum 程序，让它计算输入值的数字根。

8. 重写图 4-10 中的 Countdown 程序，让它使用 while 循环，而不是 for 循环。

9. 在数学中,有一个著名的数列称为 Fibonacci 数列,是以 13 世纪意大利数学家 Leonardo Fibonacci 的名字命名。数列中的前两项是 0 和 1,每个数列项都是前两个数的和。因此, Fibonacci 数列的前几项如下所示:

$F_0 = 0$
$F_1 = 1$
$F_2 = 1 (0 + 1)$
$F_3 = 2 (1 + 1)$
$F_4 = 3 (1 + 2)$
$F_5 = 5 (2 + 3)$
$F_6 = 8 (3 + 5)$

编写一段程序,显示数列从 F_0 到 F_{15} 的值。

10. 修改前面练习中的程序,让它不是指定末项的索引,而是显示 Fibonacci 数列中小于 10 000 的项。

11. 写画金字塔的 GraphicsProgram 子类。该金字塔由水平放置的砖块组成,每一行砖块的数量向上递减 1 块,运行结果如图 4-18 所示。

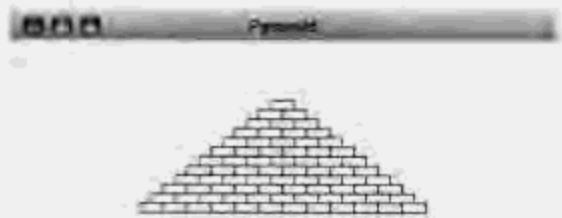


图 4-18 金字塔图案

金字塔应该在窗口居中,应该为下面参数使用命名常量:

- BRICK_WIDTH 每块砖的宽度
- BRICK_HEIGHT 每块砖的高度
- BRICKS_IN_BASE 基座砖块的数量。

12. 编写一段 ConsoleProgram 程序,让它读入一列整数,一个数一行,直到用户输入哨兵值 0(也可以轻易地改变为其他值)为止。读取终止条件时,程序应该在列表上显示最大值,如图 4-19 所示。

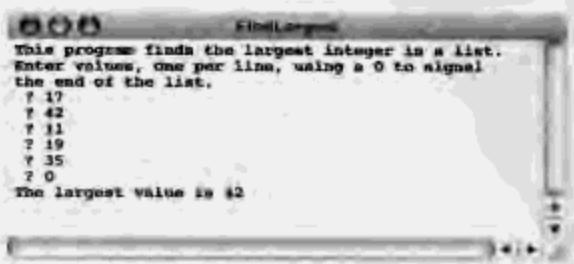


图 4-19 FindLargest 程序的样本运行

13. 扩充前面练习中的程序，让它找出列表中最大和第二大的值，如图 4-20 所示。

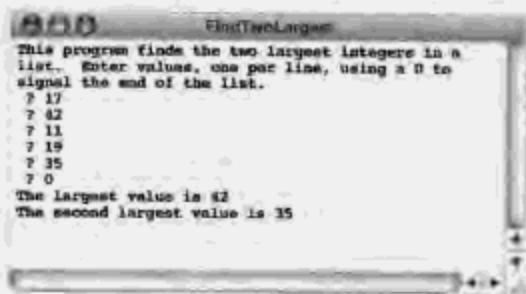


图 4-20 FindTwoLargest 程序的样本运行

14. 改进 Checkerboard 程序，让它水平居中棋盘，画出游戏开始状态相应的红色和黑色棋子，如图 4-21 所示。

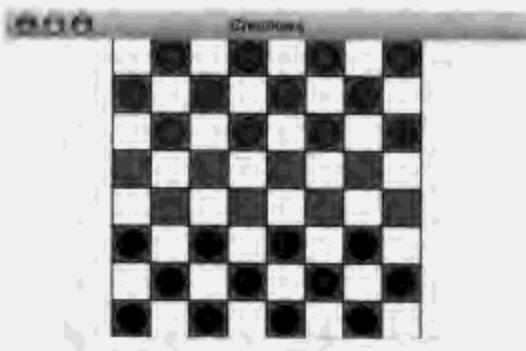


图 4-21 棋盘图案

程序的另一个改变是黑色方块的颜色从黑色变成了灰色，这样就让黑色棋子不会与背景混淆。

15. 使用 AnimatedSquare 程序作为模型，编写一段动画 BouncingBall 程序，在图形窗口的边界内弹起一个球。程序开始应该在窗口中心放置一个 GOval 表示球。每个时间步长中，程序应该将球的位置移动 dx 和 dy 个像素，其中 dx 和 dy 的初始值为 1。球的前沿接触到窗口的一个边界时，程序应该通过求相应的 dx 或 dy 的反值让球弹起来。例如，如果球碰到窗口的底边，程序应该通过求 dy 的反值让球垂直弹起来。因此，程序开始应该描绘出球的轨迹，如图 4-22 所示，其中灰色的圆圈表示球的原始位置，箭头表示球运动的轨迹。



图 4-22 BouncingBall 程序示意图

不必为如何停止程序而担心。在这个练习中，可以使用

```
while (true)
```

循环，这样在退出程序之前，它会不停地将球弹起来。

16. 在纽约时代广场，通过查看大型显示屏(它们只显示文本的一行)上的标题行就可以了解新闻。标题行最初出现在屏幕右边，接着快速从右边移动到左边。请编写一段 GraphicsProgram 程序，通过在屏幕上移动 GLabel 来模拟这种类型的标题行显示。

例如，假设要用程序显示著名的 *Chicago Tribune* 标题，当时报纸错误地声称 1948 年总统竞选结果。

程序应该创建一个包含标题行的 GLabel，然后定位它，以便剪切掉超出屏幕右边的整个文本。接着程序应该执行动画循环，按每个时间步长将 GLabel 向左移动几个像素。几个时间步长之后，会显示标题行的第一个字母，如图 4-23 所示：

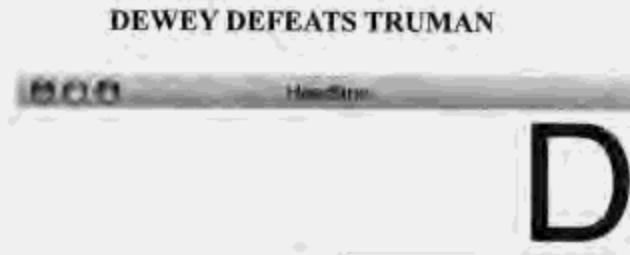


图 4-23 标题(1)

标题行继续在屏幕上滚动，几秒钟之后，第一个单词完全显示出来了，如图 4-24 所示。



图 4-24 标题(2)

标签继续滚动，随着新字母不断出现在左边，这些字母会从屏幕左边消失。这样，在某个时候，屏幕上可见的文本将包含标题行的最后一个字符，如图 4-25 所示。



图 4-25 标题(3)

文本会向左继续滚动，直到整个 GLabel 从视线中消失为止。

第 5 章

方 法

The statements was interesting but tough.

—Agatha Christie, *Poirot Investigates*, 1924



David Parnas

David Parnas 是爱尔兰利默里克大学(Limerick University)计算机科学教授，管理着那里的软件质量研究实验室(Software Quality Research Laboratory)，同时他还在德国、加拿大和美国的一些大学任教。他对软件工程最具影响力的贡献是 1972 年写的一篇开创性论文 *On the criteria to be used in decomposing systems into modules*，这篇论文奠定了结构化编程技术的基础。美国国防部专门小组负责研究假定战略防御倡议(通常称为“星球大战”)的软件，因为不可能实现系统要求，Parnas 教授 1985 年从该小组辞职，这引起了大家的广泛注意。为了表彰他揭露这些问题的勇气，1987 年 Parnas 获得了美国“为了社会责任的计算机职业者协会(CPSR)”授予的诺伯特·维纳奖(Norbert Wiener Award)。

本章详细介绍了方法的概念(第 2 章曾提及)。方法是一组集合在一起并命名的语句。因为方法允许编程人员通过使用单个名称调用整个操作，因此程序变得更简短、更简单。没有方法，程序随着规模和复杂性的增加会变得不可控制。

为了理解方法如何减少程序复杂性，需要从两个方面理解这个概念。从简化论的角度来看，需要理解方法在操作时的工作原理，以便预测它们的行为。同时，必须退一步从整体看待这些方法，以便理解它们的重要性以及有效使用它们的方法。

5.1 方法概述

从第 2 章开始就在使用方法。在详细讨论方法的工作原理之前，回顾前面介绍的有关方法

的基本术语很有意义。首先，方法由一组集合在一起并命名的语句组成。执行与方法相关的一组语句，称为调用方法。在 Java 中要表示方法调用，可以写方法的名称，后面紧跟着包含在圆括号里的一列表达式。这些表达式（称为参数）允许调用者将信息传递给方法。

例如，考察图 2-2 中 Add2Integers 程序的第一行，如下所示：

```
println("This program adds two integers.");
```

这条语句表示对 `println` 方法的调用，它会在控制台上打印一行信息。然而，要理解这种意思，`println` 方法必须知道要显示的信息是什么。这里，信息通过参数提供，这个参数由下面的字符串组成：

```
"This program adds two integers."
```

调用之后，方法将提供的数据当作参数，执行操作，然后返回到程序中产生调用的位置。记住调用程序在做什么并能够准确回到产生调用的位置，是方法调用机制的定义特征之一。回到调用程序的操作称为从方法返回。作为返回操作的一部分，方法也可以将结果发送回调用程序，现在很熟悉的声明

```
int n1 = readInt("Enter n1: ");
```

中的 `readInt` 方法可以说明这一点。`readInt` 方法执行从用户读入整数的任务之后，将该整数传回调用程序作为调用的值。这种操作称为返回值。

5.1.1 作为降低复杂性机制的方法

`Add2Integers` 程序第一次出现在第 2 章时，介绍了每项中 `println` 方法的功能。它将提供的字符串作为参数，并让它显示在作为每个 `ConsoleProgram` 一部分的控制台上。然而，本章不讨论 `println` 如何完成该操作。基本实现方式的细节仍是一个巨大的秘密，在某种程度上它让 `println` 的操作看起来像魔法。

J. K. Rowling 所著的 *Harry Potter and the Chamber of Secrets* 中，Arthur Weasley 警告说“决不要相信任何事，如果不知道它要干什么的话。”在神界这可能是一条明智的建议，但编程人员经常必须单凭信仰接受东西。在学习编程的过程中，如果不编写 `println`，确实无法打印。但要理解它的运行原理，却非常困难。即使这样，也没有什么能够阻止你有效使用它。

实际上，方法最大的好处在于，可以使用它们但无需理解其内在复杂性。方法提供的方法可以隐藏更低级别的实现方式细节，因此调用者也不会受这些细节的困扰。计算机科学中，这种技术称为信息隐藏。David Parnas 在 20 世纪 70 年代早期支持的基本思想是，编程系统的复杂性最好通过确保细节只在程序相关级别可见来管理。例如，只有实现 `println` 并确保其运行的编程人员需要知道该操作的具体细节。而仅仅使用 `println` 的编程人员对其根本机制可以毫不知情。

5.1.2 作为编程人员工具而不是用户工具的方法

刚开始学习编程的同学有时难以理解方法和程序之间的区别。在某种程度上，程序和方法都有收集一系列语句并给它命名的功能。例如考察 `Add2Integers` 程序，它的行为就像是包含在 `run` 方法内一组步骤的简写。

然而，区别这两个概念十分重要。方法和程序之间的主要区别在于谁或者什么在使用它。当用户坐在计算机前启动并运行应用程序时，该程序代替用户执行某些动作。因此，通过调用程序，可以满足外部用户的需要。另一方面，方法提供了一种机制，通过该机制，程序可以调用以前定义的操作。这样，方法的操作就在程序域的内部。

常见错误

仔细区别程序域中输入和输出的概念以及方法域中参数和结果的相关概念。输入输出指程序与其用户之间的通信；参数和结果表示方法与其调用者之间的通信。

参数和程序输入之间、返回值与程序输出之间也存在类似混淆。很容易将使用 `readInt` 输入的输入数据当成类似作为参数传递的值。毕竟，两者都表示将数据传递到某些程序的方法。尽管概念类似，但是明确区分输入操作(例如 `readInt`)和方法域中参数的使用，非常重要。与 `readInt` 类似的方法为用户输入数据提供了机制。当 `readInt` 需要输入值时，坐在终端前的人必须在键盘上手动输入。另一方面，方法的参数为方法接收调用者的输入提供了方法，它只是程序的另一部分。以参数形式传递的数据可能由用户在程序先前的部分输入，但是只能算作程序运行的一部分。要仔细区分输出操作(例如 `println`)使用与返回结果的方法。使用 `println`，输出会显示在控制台。方法返回结果时，信息返回给其调用者，只要程序需要，可以任意使用该结果。当逻辑上要求使用参数和结果时，新的编程人员倾向于使用方法里的输入/输出操作。

5.1.3 作为表达式的方法调用

第3章将方法调用列为Java的一种表达式。方法调用只是一种表达式，它可以用于可以使用表达式的所有语境，在试图理解方法如何适应整个Java架构时，记住这一点十分有用。而且，方法的参数通常就是表达式，它们本身可以包含方法调用或其他表达式中的合法操作。

为了说明方法及其参数都是表达式，这里介绍 `Math` 类中的几个标准方法，如图5-1所示。从现有方法的列表中可以看出，`Math`类包含许多在高中代数学和三角学中学习的标准数学函数。例如，`Math`类中，方法 `sqrt` 用于求参数的平方根，`sin` 和 `cos` 用于分别求三角正弦和余弦。这些方法都使用 `double`类型的数据作为参数，返回的结果也是 `double`类型。可以在简单语句中使用这些方法，例如：

```
double root3 = Math.sqrt(3.0);
```

或者在较为复杂的语句中使用它们。例如，使用标准的平面上两点间距离公式，可以计算从原点到点(x, y)的距离。

$$\text{distance} = \sqrt{x^2 + y^2}$$

Java中，此公式与下面的语句相对应。

```
double distance = Math.sqrt(x * x + y * y);
```

图 5-1 所示的方法定义为 Math 类的一部分(作为标准程序包 java.lang 的一部分，所有 Java 程序都有 Math 类)。然而，不像 Java 中其他大多数方法，Math 类中的方法与对象没有关联，它们全部都是类的一部分。这种方法称为静态方法。调用静态方法时，Java 要求指定类的名称及方法名称。因此，调用 Math 类中的静态 sqrt 方法时，必须写成 Math.sqrt。

Math.abs(x)	返回 x 的绝对值，可以是任何数字类型
Math.min(x, y)	返回 x 和 y 中较小的那个数
Math.max(x, y)	返回 x 和 y 中较大的那个数
Math.sqrt(x)	返回值 x 的平方根
Math.log(x)	返回 x 的自然对数，使用数学常量作为其基数
Math.exp(x)	返回 x 的反对数，即 e^x
Math.pow(x, y)	返回 x 的 y 次幂
Math.sin(theta)	返回角 theta 的三角正弦，以弧度为单位
Math.cos(theta)	返回角 theta 的三角余弦
Math.tan(theta)	返回角 theta 的三角正切
Math.asin(x)	返回正弦为 x 的角
Math.acos(x)	返回余弦为 x 的角
Math.atan(x)	返回正切为 x 的角
Math.toRadians(degrees)	将角从度转换成弧度
Math.toDegrees(radians)	将角从弧度转换成度

图 5-1 从 Math 类中选择的方法

5.1.4 作为消息的方法调用

在类似 Java 这样的语言中，Math 类提供的这种静态方法有些落伍，它是面向对象编程出现之前的编程风格。面向对象语言使用方法的方式与传统语言不同，主要因为面向对象范例依赖于基本过程中不同的概念模型。理解了概念模型之后，对 Java 中方法调用的结构以及用来描述它的术语就有了很好的直观认识。

在面向对象世界里，对象通过相互发送信息和请求进行交流。总体而言，这些对象间的传输称为消息。发送消息相当于让一个对象调用属于另一个对象的方法。为了与发送消息的概念模型相一致，发起方法的对象称为发送方，作为传输目标的对象称为接收方。Java 中，发送方通过产生下面这种形式的调用来标识接收方：

```
receiver.name(arguments)
```

前面已经学习过这种风格的方法调用，在第 2 章就是利用它将消息发送到图形对象的。

除了对静态方法(总是要包括明确的类名)的一些调用之外，本书所有的方法调用都有接收方，哪怕没有明确指定它。如果调用没有指定接收方的方法，则 Java 假定发送对象和接收对象相同。这种解释使下面的调用

```
println(value)
```

仍然可能运用发送和接收消息的概念模型。如果这种形式的行出现在程序中，可以将它解释为发送消息，该消息需要与方法名称 `println` 相关的动作——特别是，在输出设备上显示值。由于调用中没有明确的接收方，消息的目标就是程序本身。刚好，程序能够对消息作出适当反应，因为方法 `println` 被定义为 `Program` 类的一部分。`Program` 类的每个子类都继承了这种方法，这意味着这些子类可以使用简化的方法调用来自行操作。如果要明确接收方——有些教师这么做——可以使用 Java 关键字 `this`，它代表当前对象，如下所示。

```
this.println(value)
```

5.2 编写自己的方法

到目前为止，我们所接触到的大多数方法都被定义为某个库程序包的一部分。库方法当然有用，但是要解决复杂编程问题，仅有这些方法当然不够。面对复杂问题时，需要开发自己的方法来控制代码的复杂性。将程序分解为方法，以便能通过更小的单元考察程序的不同部分，这些较小的单元更容易被理解。编写自己的方法也是给自己提供工具的一种方式，这些工具不是现有库程序包的一部分。

例如，大多数国家使用摄氏温度，而美国使用华氏温度。假设您的任务是编写一段程序，生成从摄氏温度到华氏温度的换算表。您可能想定义一种简单的换算方法，以便在程序的其他部分使用它。这一计算相对简单，因为将一种温度转换成另一种温度就是简单应用下面的公式。

$$F = \frac{9}{5}C + 32$$

5.2.1 方法定义的格式

Java 中要定义新的方法，要从编写方法头开始，如下面的语法框所示。除非开始定义自己的类，否则公有方法和私有方法之间的区别就完全不重要。即使这样，如果可能，也最好让方法保持私有。因此应该将只在单个类里使用的方法声明为 `private`。模式里的 `type` 元素表示方法返回值的类型；如果方法没有返回值，应该指定 `type` 元素为 `void`。`name` 元素表示方法名称，它可以是根据第 2 章给出的规则形成的任何标识符。最重要的是，名称应该让阅读程序的人很容易确定方法要做什么。圆括号里的代码(在模式中标记为 `parameters`)指定方法需要哪些参数。除了没有指定初始值以外，参数表与变量声明的形式一样，拥有多个参数的方法用逗号将这些声明区分开。

方法头的语法

```
visibility type name (parameters)
```

其中：

visibility 通常是 `public` 或 `private`；

type 是返回值的类型；如果方法没有返回值，它就是 `void`；

name 是方法名；

parameters 是参数声明列表。

本示例中，编写一个方法将摄氏度转换为华氏度。方法头行如下所示。

```
private double celsiusToFahrenheit(double c)
```

方法头行之后是方法体，它通常是代码块，由封装在花括号里的语句组成。代码块里的语句可以包括变量声明，如许多程序示例中出现的那些声明。

5.2.2 return 语句

如果方法返回结果，方法体内的语句必须至少包含一条 `return` 语句，用于指定返回的值。`return` 语句的一般形式如下面的语法框所示。

return 语句的语法

```
return expression;
```

其中：

expression 是返回的值。

如果方法没有结果，语法则只是

```
return;
```

大多数情况下，`return` 语句包含表示结果的值的表达式，尽管如果没有返回值会省略该表达式。当它以下面的形式使用时，

```
return expression;
```

`return` 语句让方法立即返回指定的值。这样，`return` 语句包括了下面两个想法：“已完成”和“这是答案”。在有些编程语言中(例如 Pascal 和 Fortran)表示方法执行完成和指定其结果是两个单独的操作。如果有使用那些语言的经验，要适应 Java 中的 `return` 语句可能需要一点时间。

`return` 语句完成了一系列工具，写 `celsiusToFahrenheit` 方法的实现需要这些工具：

```
private double celsiusToFahrenheit(double c) {
    return 9.0 / 5.0 * c + 32;
}
```

要创建温度换算表，需要创建一个 `ConsoleProgram` 子类，该子类的 `run` 方法为表中的每个条目调用 `celsiusToFahrenheit`。整个程序如图 5-2 所示。

```

/*
 * File: TemperatureConversionTable.java
 *
 * This program creates a table of Celsius to Fahrenheit
 * equivalents using a function to perform the conversion.
 */

import acm.program.*;

public class TemperatureConversionTable extends ConsoleProgram {

    public void run() {
        println("Celsius to Fahrenheit table.");
        for (int c = LOWER_LIMIT; c <= UPPER_LIMIT; c += STEP_SIZE) {
            int f = (int) celsiusToFahrenheit(c);
            println(c + "C = " + f + "F");
        }
    }

    /* Returns the Fahrenheit equivalent of the Celsius temperature c. */
    private double celsiusToFahrenheit(double c) {
        return 9.0 / 5.0 * c + 32;
    }

    /* Private constants */
    private static final int LOWER_LIMIT = 0;
    private static final int UPPER_LIMIT = 100;
    private static final int STEP_SIZE = 5;
}

```

图 5-2 生成温度转换表的程序

5.2.3 包含内部控制结构的方法

方法通常不像 `celsiusToFahrenheit` 那样简单。许多情况下，设计方法需要进行测试或编写循环。这种细节增加了实现方式的复杂性，但没有改变其基本形式。例如，`Math` 类中的 `abs` 方法计算其参数的绝对值，目前可以假设它为整数。但是假设方法不存在，如何编写它的实现工具呢？绝对值的定义说明，如果参数是负数，方法返回它的相反数(正数)；如果参数是正数或零，方法原封不动地返回参数的值。因此，可以用如下语句实现 `abs` 方法。

```

private int abs(int n) {
    if (n < 0) {
        return -n;
    } else {
        return n;
    }
}

```

这种实现方式表明，`return` 语句可以在方法体内部任何地方出现，并且可以多次出现。同样，可以定义方法 `min` 来返回两个浮点参数中较小的一个：

```

private double min(double x, double y) {
    if (x < y) {

```

```

        return x;
    } else {
        return y;
    }
}

```

方法内使用的控制结构比前面的示例更复杂。假设要定义名为 factorial 的方法，它读入整数 n ，返回介于 1 和 n 之间的整数乘积。前面几级阶乘如下所示。

```

factorial(0) = 1 (定义)
factorial(1) = 1 * 1
factorial(2) = 2 = 1 * 2
factorial(3) = 6 = 1 * 2 * 3
factorial(4) = 24 = 1 * 2 * 3 * 4
factorial(5) = 120 = 1 * 2 * 3 * 4 * 5
factorial(6) = 720 = 1 * 2 * 3 * 4 * 5 * 6
factorial(7) = 5050 = 1 * 2 * 3 * 4 * 5 * 6 * 7

```

阶乘在数学中通常用感叹号表示(例如 $n!$)，在统计学、组合数学和计算机科学中有广泛应用。在这些领域中，计算阶乘的方法是解决问题的有用工具。factorial 方法读入整数，返回整数，因此它的头行如下：

```
private int factorial(int n)
```

然而，实现 factorial 方法还需要做一些工作。作为编程问题，计算阶乘在许多方面与相加一列数类似。在第 4 章的 AddList 程序中，声明名为 total 的变量来记录连续的总数。程序开始时，total 初始化为 0。随着不断输入新值，这些新值都与 total 相加，因此，total 连续表示到目前为止输入的数的和。除了必须记录乘积而不是和之外，当前问题的情况很类似。因此，可以使用以下步骤：

- (1) 声明名为 result 的变量。
- (2) 将它初始化为 1。
- (3) 乘以介于 1 和 n 之间的每个整数。
- (4) 返回最终的 result 值作为方法的结果。

要循环步骤(3)中要求的每个整数，需要使用 for 循环。它从 1 开始，直到 n 结束。for 循环需要索引变量，传统选择 i 就很合适。变量 result 保存连续的乘积， i 保存索引。

factorial 的实现非常简短，足以一次全部显示，而不需要逐步解释细节。

```

private int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

```

5.2.4 返回非数字值的方法

到目前为止，本章看到的方法都返回数字值，但决不是必需这样。Java 中的方法可以返回任

何数据类型的值。例如，如果编写的程序与日期有关，最好有一种方法能够将1~12间的数字月份转换为String数据类型，分别对应1月到12月的月份名称。在内部使用数字值通常很方便，但是用传统英语名称输出显示通常更具有可读性。要解决这个问题，可以定义方法monthName，如下所示。

```
private String monthName(int month) {
    switch (month) {
        case 1: return ("January");
        case 2: return ("February");
        case 3: return ("March");
        case 4: return ("April");
        case 5: return ("May");
        case 6: return ("June");
        case 7: return ("July");
        case 8: return ("August");
        case 9: return ("September");
        case 10: return ("October");
        case 11: return ("November");
        case 12: return ("December");
        default: return ("Illegal month");
    }
}
```

要使用该方法，可以从程序其他部分调用monthName方法，然后使用println显示结果。例如，如果整数变量month、day和year分别是值7、20和1969(阿波罗11登月的日期)，语句

```
println(monthName(month) + " " + day + ", " + year);
```

生成的输出如图5-3所示。

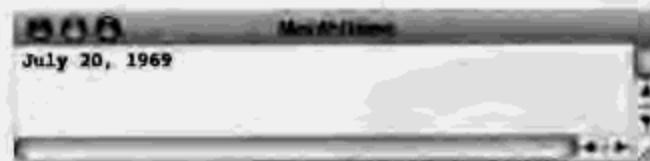


图5-3 monthName方法生成的输出

在monthName方法中的switch语句里，每个case子句里的return语句自动退出整个方法，而不需要明确的break语句。正如第4章关于switch语句的介绍，如果确保每个case从句用一条break或return语句结束，那么就会省去调试过程中的许多麻烦。

假设这些字符串对于Java来说非常完整，以至于它们比对象看起来更像原始类型，那么方法返回其他类型的对象也很有用，如第2章介绍的图形对象。例如，考察下面的方法，它创建圆形的GOval，其圆心是点(x,y)，半径为r个像素，填充颜色由color指定：

```
private GOval createFilledCircle(double x, double y,
                                 double r, Color color) {
    GOval circle = new GOval(x - r, y - r, 2 * r, 2 * r);
    circle.setColor(color);
    circle.setFilled(true);
```

```

    return circle;
}

```

图 5-4 中的 DrawStoplight 程序显示了用这种方法画传统的红绿灯(红灯、黄灯、绿灯, 如图 5-5 所示)的方法。使用 createFilledCircle 方法可以简化代码, 因为只需要调用有合适参数的方法, 程序就可以创建每盏灯。定义包含这种通用代码的方法, 就不用为每盏红灯、黄灯和绿灯重新写相同的步骤。

```

/*
 * File: DrawStoplight.java
 *
 * This program draws a traditional stoplight with a red, yellow,
 * and green light in a gray rectangular frame.
 */

import acm.graphics.*;
import acm.program.*;
import java.awt.*;

public class DrawStoplight extends GraphicsProgram {

    public void run() {
        double cx = getWidth() / 2;
        double cy = getHeight() / 2;
        double fx = cx - FRAME_WIDTH / 2;
        double fy = cy - FRAME_HEIGHT / 2;
        double dy = FRAME_HEIGHT / 4 + LAMP_RADIUS / 2;
        GRect frame = new GRect(fx, fy, FRAME_WIDTH, FRAME_HEIGHT);
        frame.setFilled(true);
        frame.setColor(Color.GRAY);
        add(frame);
        add(createFilledCircle(cx, cy - dy, LAMP_RADIUS, Color.RED));
        add(createFilledCircle(cx, cy, LAMP_RADIUS, Color.YELLOW));
        add(createFilledCircle(cx, cy + dy, LAMP_RADIUS, Color.GREEN));
    }

    /*
     * Creates a circular GOval object centered at (x, y) with radius r.
     * The GOval is set to be filled and colored in the specified color.
     */
    private GOval createFilledCircle(double x, double y,
                                      double r, Color color) {
        GOval circle = new GOval(x - r, y - r, 2 * r, 2 * r);
        circle.setColor(color);
        circle.setFilled(true);
        return circle;
    }

    /* Private constants */
    private static final double FRAME_WIDTH = 50;
    private static final double FRAME_HEIGHT = 100;
    private static final double LAMP_RADIUS = 10;
}

```

图 5-4 使用 createFilledCircle 方法画交通灯的程序



图 5-5 DrawStoplight 程序的输出

5.2.5 断言方法

5.2.4 小节的示例说明方法可以返回不同数据类型的值。例如，方法 factorial 返回 int 类型的值，方法 monthName 返回 string 类型的值。虽然 Java 中的方法可以返回任何类型的值，但是有一种类型的结果值得注意——第 4 章介绍的 boolean 数据类型。返回 boolean 类型值的方法称为断言方法，它在现代编程中具有重要作用。

boolean 类型只有两个值：true 和 false。因此，断言方法——不管它有多少参数或者其内部处理多么复杂——最终必须返回这两个值之一。因此，调用断言方法的过程类似于问“是/否”这样的问题并得到答案。

例如，假设调用者提供特定的整数 n 作为参数，下面的方法定义用来回答“n 是不是偶数？”这个问题：

```
private boolean isEven(int n) {
    return (n % 2 == 0);
}
```

如果这个数除以 2 没有余数，那么它就是偶数。如果 n 是偶数，那么表达式

$n \% 2 == 0$

的值是 true，它会作为 isEven 方法的结果返回。如果 n 是奇数，方法返回 false。因为 isEven 返回 Boolean 结果，所以在条件语境中可以直接使用它。例如，下面的 for 循环使用 isEven 来列出 1~100 之间的所有偶数。

```
for (int i = 1; i <= 100; i++) {
    if (isEven(i)) println(i);
}
```

for 循环运行每个数，if 语句始终问“这个数是不是偶数？”这个问题。如果是，println 在屏幕上显示它；如果不是，就不显示。

Java 中标准的类定义通常包含断言方法。一个特别重要的示例是 equals 方法，它在通用类 Object 中定义，所以可以应用于任何对象。equals 方法询问一个对象的值与另一个对象的值是否相同。相反，== 运算符询问其操作数是不是完全相同的对象，这通常没有用。

用字符串最容易说明 equals 和 == 之间的区别。例如，假设程序包含下面的语句来向用户询问一个简单的“是/否”问题。

```
String answer = readLine("Would you like instructions? ");
```

用户输入响应后，程序需要确定答案是“是”还是“否”或者完全可能是其他某个字符串。

下面的方法无效：



```
if (answer == "yes") . . .
```

用户输入的值可能是由字符“y”、“e”和“s”组成的字符串对象，它与程序中出现的常量字符串“yes”不是同一个对象。相反，要问的应该是两个字符串是否包含相同顺序的字符。而这刚好是 equals 方法的功能。因此，要写的语句应该是：

```
if (answer.equals("yes")) . . .
```

注意，这条语句使用了接收方语法，因此可以解释为：发送 equals 消息到 answer 对象，然后发回比较的结果。

作为断言方法的另一个示例，可以编写程序来测试给出的年份是不是闰年，代码如下：

```
private boolean isLeapYear(int year) {
    return ((year % 4 == 0) && (year % 100 != 0))
        || (year % 400 == 0);
}
```

第 4 章已经介绍过如何使用 Boolean 表达式来确定 year 是不是闰年。在程序的不同位置，将表达式置于方法之中就可以简化核对是否为闰年的过程。定义方法之后，程序其他部分完全可以使用这种形式的语句：

```
if (isLeapYear(year)) . . .
```

5.3 方法调用过程的技巧

到目前为止，本章主要从整体上介绍了方法。用这种方法考察方法有助于理解如何使用它们以及作为编程资源它们能够提供什么。然而，要相信自己编写的方法能够正常运行，就要理解方法内部如何运行。

理解方法技巧的第一步是考察图 5-6 所示的程序，它包括本章前面提到的 factorial 方法和显示阶乘表的主程序。如果将 FactorialTable 程序看成两个单独部分，那么就很容易理解它。主程序用来从 LOWER_LIMIT 到 UPPER_LIMIT 计数。每个周期中，它都调用索引 i 上的 factorial，然后显示结果。通过这种方法，通常可以使用名称 i 来表示其他普通索引变量(该索引变量用于在 for 循环中计算周期)，这没有什么奇怪的。factorial 方法也很直接。和在主程序中相比，有一点接近，但不是太多。理解方法在做什么很容易。特别是，应该知道 n 是正在计算其阶乘的数，result 保存了每个周期累积的乘积，i 是用于记录 for 循环过程的普通索引变量。这样，程序的每个部分都有它自身的意义。

对于许多同学而言，只有整体考察程序时才会产生混淆。如果事先没有学习如何考察方法就要整体理解程序，很可能会遇到各种概念问题。第一，有两个称为 i 的变量，一个在主程序中，一个在 factorial 方法中。每个变量都是循环索引，但是两个变量的值通常不同。第二，程序有的部分用两个不同名称来指代相同的值。在 run 方法中，要计算阶乘的两个数存储在循环索引 i 里，在 factorial 方法中，同样的值称为 n。因此，在整个程序中，有的变量名称相同却

保存着不同的值，有些值相同却存储在名称不同的变量里。

```
/*
 * File: FactorialTable.java
 *
 * This file generates a table of factorials.
 */

import acm.program.*;

public class FactorialTable extends ConsoleProgram {
    public void run() {
        for (int i = LOWER_LIMIT; i <= UPPER_LIMIT; i++) {
            println(i + "!" + factorial(i));
        }
    }

    /*
     * Returns the factorial of n, which is defined as the
     * product of all integers from 1 up to n.
     */
    private int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }

    /* Private constants */
    private static final int LOWER_LIMIT = 0;
    private static final int UPPER_LIMIT = 10;
}
```

图 5-6 生成阶乘表的程序

这种名称和值之间联系的明显不稳定性似乎确实会造成混淆。实际上，效果刚好相反。将编程人员从维护名称和值之间的固定联系中解脱出来，很可能开发出独立程序。随着程序的发展，根本无法从整体上理解它们。理解大型程序的唯一希望就是将它分解为独立的部分，每个部分都很简单，可以单独理解。FactorialTable 程序中，阶乘产生的问题分成两部分——run 方法和 factorial 方法——每个部分都可以单独理解。

5.3.1 参数传递

要理解 FactorialTable 程序的两个部分如何协调运行，先理解 Java 如何记录不同的名称和值会很有帮助。努力这样做，对于编程人员来说至关重要，这让编程人员能够分别考察单个方法。相同的名称可能用于不同的值，不同名称可能表示同一个概念值。Java 是如何处理这种情况呢？要回答这个问题，很有必要对调用者的参数值和方法语境中用于保存这些值的变量进行语义上的区分。方法调用中的参数可以是任意表达式。方法头中对应位置出现的变量是这些值的占位符，称为形参。

调用方法时，在 Java 虚拟机上执行程序的运行系统采取如下步骤：

- (1) 计算调用方法语境中每个参数的值。因为参数是表达式，这种运算可能包括运算符和

其他方法调用；调用新方法之前，必须为它们赋值。

(2) 将每个参数表达式的值复制到对应的形参变量里。如果不止一个参数，运行系统依次将参数复制到形参里，第一个参数复制到第一个形参里，依次类推。如有必要，运行系统可以在参数值和形参之间执行自动类型转换，就像在赋值语句里那样。例如，如果将 int 类型的值传递给方法，而该方法的形参声明为 double 类型，那么运行系统在将值复制到形参变量时，将整数转换为相应的浮点值。

(3) 在 return 语句出现之前，它计算方法体内的语句。

(4) 给 return 语句相关的表达式赋值，如有必要，将它转换为方法指定的结果类型。

(5) 从调用方法停止的位置继续执行其内部的代码，用返回值取代调用。

每次对方法的调用都会创建一组单独的变量。要更好地理解该过程，可以画一些方框图表示每个变量的值，也可采用第 2 章中各种示例中的线，这样可能有帮助。对于细分为单独方法的图形，每当一种方法调用另一种方法时，都需要画一个新的变量方框。对各种方法而言，方法声明的每个变量(包括形参)都需要用单个方框表示。这些变量只有在声明它们的方法里才有意义，因此称为局部变量。

例如，跟踪 FactorialTable 示例的执行时，首先需要为方法 run 中的变量创建空间。方法 run 只声明一个变量——循环索引 i——run 中的变量表示如图 5-7 所示。

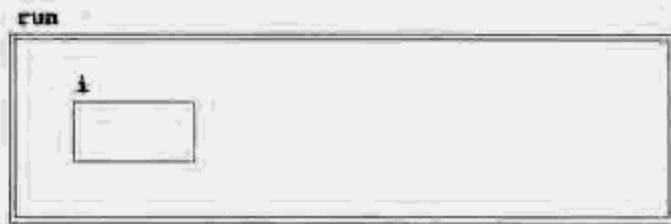


图 5-7 run 方法中的变量

变量方框外面的双线用于包含与特殊方法调用相关的所有变量。这种变量的集合称为该方法的框架或者称为堆栈框架。

自己跟踪程序执行时，表示索引卡上的每个堆栈框架有助于同时跟踪多个框架。程序调用新的方法时，需要创建一个新的索引卡，并将它放在现有索引卡的最上面。方法返回时，可以丢弃索引卡(在堆栈的最上面)。在第 7 章会知道，堆栈框架中的变量存储在内存的物理邻近部分，这说明框架的概念不仅仅是个抽象表示法。

假设将 LOWER_LIMIT 定义为 0，就像在程序清单里一样。在这种情况下，for 循环的第一个周期内，i 的值是 0，和前面一样，在表示变量的方框里标记值，可以在框架里表示这种条件，如图 5-8 所示。

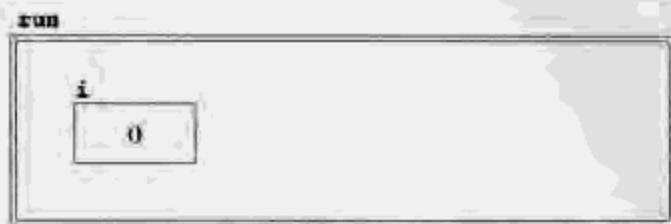


图 5-8 在框架中表示条件

接着，主程序调用 `println`，作为给 `println` 参数赋值的一部分，计算表达式的结果。

```
factorial(i)
```

要用框图表示计算行为，先看看当前框架里 `i` 的值，它的值是 0。必须为 `factorial` 创建一个新的框架，值 0 是它的第一个(也是唯一)参数。`factorial` 方法有 3 个变量：形参 `n`、局部变量 `result` 和 `i`。因此 `factorial` 的框架需要 3 个变量单元，如图 5-9 所示。

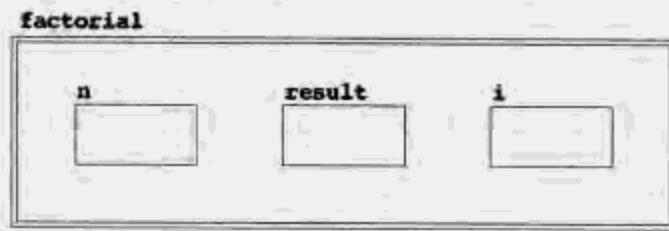


图 5-9 factorial 方法的变量

在这种图形执行中，Java 运行系统将形参 `n` 初始化为相对应的实参值(0)。框架的内容如图 5-10 所示。

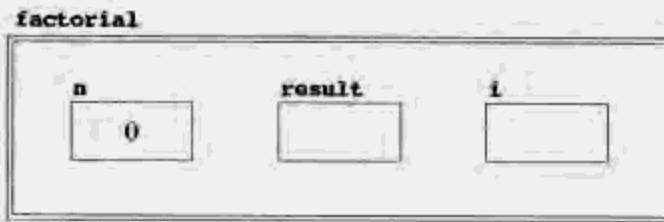


图 5-10 初始化变量 n

为 `factorial` 创建框架时，`run` 方法的框架并没有完全消失。框架会保持悬浮，直到 `factorial` 操作完成为止。要在概念模型中表示这种情况，只需要将每个新框架的索引卡放在前面一个索引卡的上面即可，也就是掩盖它。例如，调用 `factorial` 方法时，表示 `factorial` 框架的索引卡会移动到 `run` 框架的上面。

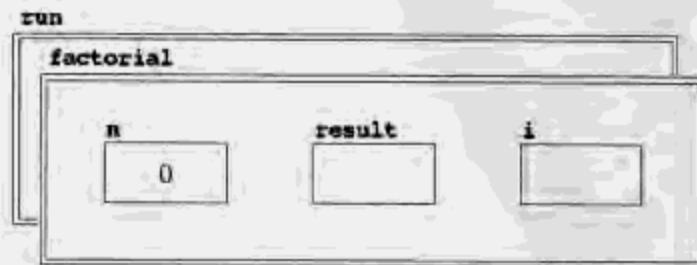


图 5-11 新框架的索引卡在最上方

如图 5-11 所示，整个框架形成了堆栈，最新的框架在堆栈的最上面。`run` 框架仍然还在，只要激活方法 `factorial`，就看不到它。特别是，名称 `i` 不再指在 `run` 方法里声明的变量，而是指 `factorial` 方法里名为 `i` 的变量。

Java 运行时系统初始化形参之后，通过运行当前框架里的每个步骤来执行 `factorial` 方法。

的主体。变量 `result` 初始化为 1，程序运行到 `for` 循环。在 `for` 循环中，变量 `i` 初始化为 1，但是由于值总是比 `n` 大，所以根本不会执行 `for` 循环的主体。因此，程序运行到 `return` 语句时，框架如图 5-12 所示。

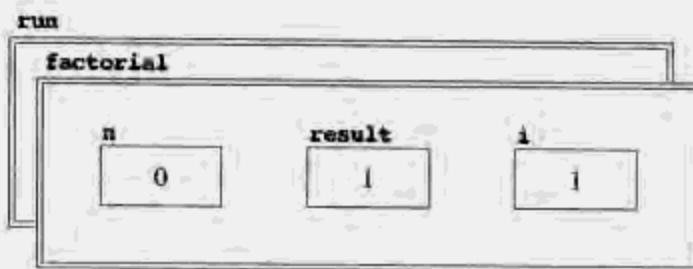


图 5-12 程序运行到 `return` 语句的框架

`factorial` 返回时，作为方法的结果，`result` 的值会传回给调用者。从方法返回也会丢弃它的堆栈框架，再一次显示 `run` 中的变量，如图 5-13 所示。



图 5-13 重新显示 `run` 方法

因此结果 1 传递到 `println` 方法，`println` 方法经过相同的过程；然而，用户却看不见这种操作的细节，因为用户不知道 `println` 在内部怎样运行。最后，`println` 在屏幕上显示值 1，然后返回，之后方法 `run` 继续执行下一个 `for` 循环周期。

5.3.2 从其他方法内调用方法

定义方法后，不仅可以在主程序语境中使用它，而且可以将它作为实现其他更复杂方法的工具，方法的强大功能即来源于此。可以从其他方法调用这些方法，创建非常复杂的层次结构。

为了说明多层次方法的思想，假设在您面前的桌子上有一组 `n` 个不同的对象。要从这一组对象中选择 `k` 个对象。从大小为 `n` 的原始集合中选择 `k` 个对象，怎样写一种方法，让它计算出有多少种不同的方法？

要让问题更具体，假设桌上的对象是 5 枚美国硬币：一分硬币、五分硬币、一角硬币、二角五分硬币和五角硬币。例如，要确定从桌子上取两枚硬币共有多少种方法，可以取一分硬币和五分硬币、一分硬币和一角硬币、五分硬币和二角五分硬币，或者其他几种组合。如果列出所有可能性，就会发现有 10 种不同的组合：

一分硬币+五分硬币	五分硬币+二角五分硬币
一分硬币+一角硬币	五分硬币+五角硬币
一分硬币+二角五分硬币	一角硬币+二角五分硬币
一分硬币+五角硬币	一角硬币+五角硬币

五分硬币+一角硬币

二角五分硬币+五角硬币

本示例中, n 等于 5, k 等于 2。问题的本质是找一种方法计算从一组 n 个对象中选择 k 个对象有多少种方法, 作为值 n 和 k 的函数。这个函数在概率和统计学中经常出现, 称为组合函数, 在数学中通常写作:

$$\binom{n}{k}$$

函数符号如下:

$$C(n, k)$$

事实上, 阶乘与组合函数之间有一个简单定义:

$$C(n, k) = \frac{n!}{k!(n-k)!}$$

例如, 通过逐步计算下面的数学表达式, 可以验证 $C(5,2)$ 确实是 10:

$$\begin{aligned} C(5,2) &= \frac{5!}{2!3!} \\ &= \frac{120}{2 \times 6} \\ &= 10 \end{aligned}$$

如果要在 Java 中实现这种方法, 最好使用比数学中使用的单个 C 更长的名称。作为通用规则, 本文使用的方法名比局部变量名更长、更有表现力。与局部变量名称不同, 方法调用经常出现在程序的各个部分, 这些位置与定义这些方法的位置距离很远。由于在大型程序中很难定位定义, 所以最好选择包含有关方法足够信息的方法名称, 以便读者不需要查看定义就可以理解方法。另一方面, 局部变量只用于单个方法的主体内, 这样就很容易明白它们的意思。为了让用户看到组合方法的名称就能理解该方法, 使用名称 combinations 而不是 C 作为方法名。

要实现 combinations 方法(按照阶乘定义), 可以利用已经有 factorial 实现方式。假设在实现方式中可以调用 factorial, combinations 方法的主体是其数学定义的直接转换:

```
private int combinations(int n, int k) {
    return factorial(n) / (factorial(k) * factorial(n - k));
}
```

这样就可以编写一个简单的主程序来测试 combinations 方法, 代码如下:

```
public void run() {
    int n = readInt("Enter number of objects in the set (n): ");
    int k = readInt("Enter number to be chosen (k): ");
    println("C(" + n + ", " + k + ") = " + combinations(n, k));
}
```

完整的程序——Combinations.java——如图 5-14 所示。图 5-15 所示为该程序的一次运行

结果。

```
/*
 * File: Combinations.java
 *
 * This program computes the mathematical combinations function
 * C(n, k), which is the number of ways of selecting k objects
 * from a set of n distinct objects.
 */

import acm.program.*;

public class Combinations extends ConsoleProgram {
    public void run() {
        int n = readInt("Enter number of objects in the set (n): ");
        int k = readInt("Enter number to be chosen (k): ");
        println("C(" + n + ", " + k + ") = " + combinations(n, k));
    }

    /*
     * Returns the mathematical combinations function C(n, k),
     * which is the number of ways of selecting k objects
     * from a set of n distinct objects.
     */
    private int combinations(int n, int k) {
        return factorial(n) / (factorial(k) * factorial(n - k));
    }

    /*
     * Returns the factorial of n, which is defined as the
     * product of all integers from 1 up to n.
     */
    private int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

图 5-14 计算组合函数 $C(n,k)$ 的程序

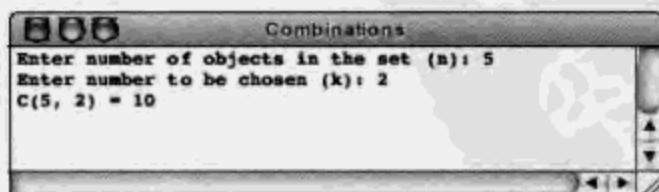


图 5-15 Combinations 的一次运行结果

程序运行时计算机内部发生了什么呢？正如在阶乘示例中一样，运行系统给方法 run 创建一个框架(方法 run 现在声明两个变量 n 和 k)。用户输入两个值后，程序执行到 println 语句，框架内的变量取值如图 5-16 所示。

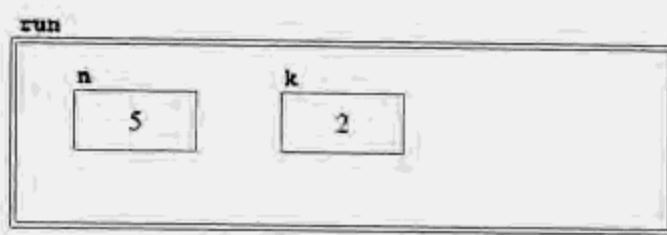


图 5-16 变量取值

要执行 `println` 语句，计算机必须对 `combinations` 方法的调用求值，这会创建一个新框架覆盖以前的框架，如图 5-17 所示。

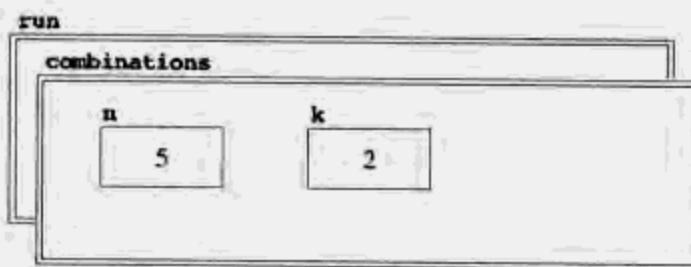


图 5-17 新框架覆盖原框架

这个示例的方法调用比阶乘示例中要多。其中，进行调用之前，每个新框架必须准确记录程序在做什么。例如，对 `combinations` 的调用来自于 `run` 中最近的行，如下面程序文本中的标记 `R1` 所示：

```
public void run() {
    int n=readInt("Enter number of objects in the set (n): ");
    int k=readInt("Enter number to be chosen (k): ");
    println("C(" + n + "," + k + ")=" + combinations(n, k));
}
```



计算机执行新的方法调用时，它知道调用完成后在调用程序中应该从哪里继续。执行应该返回的位置(称为返回地址)在这些代码中用圆圈标记表示。在程序执行中要知道在哪里，就需要在框架图中记录调用的位置，如图 5-18 所示。

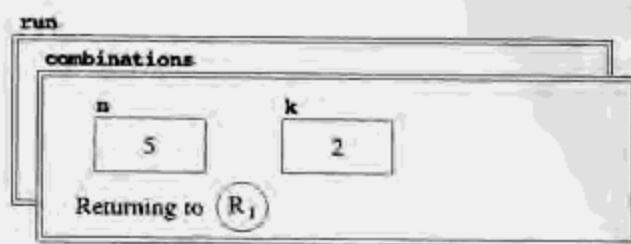


图 5-18 标记返回位置

创建新的框架以后，程序就开始执行 combinations 方法的主体，与 factorial 调用(用标记标注)类似，如下所示：

```
private int combinations(int n, int k) {
    return factorial(n) / (factorial(k) * factorial(n - k));
}
```

要执行语句，计算机必须产生对方法 factorial 的 3 个指定调用。依据 Java 指定的规则，编译器必须按从左至右的顺序产生这些调用。第一个调用需要计算机计算 factorial(n)，结果会创建如图 5-19 所示框架。

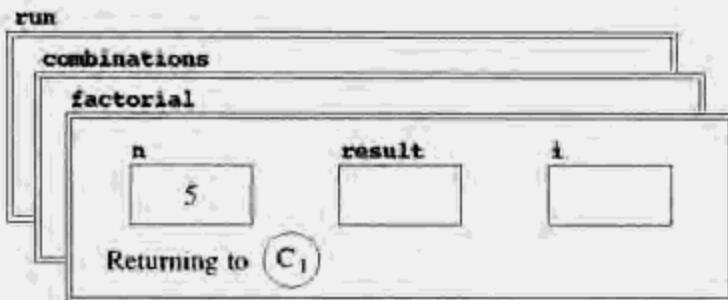


图 5-19 第一个调用的框架

factorial 方法和以前一样运行，将值 120 返回到调用者里标记 C₁ 表示的位置。factorial 方法的框架消失，只剩下 combinations 框架，准备继续到计算的下一个阶段。通过用返回值代替原始调用，可以说明当前状态，如下所示。

```
return 120 / (factorial(k) * factorial(n - k));
```

上面语句中，“120”外面的方框表示包含的值不是程序的一部分，而是前面计算的结果。从这里开始，计算机继续对 factorial 的第二个调用求值，其中参数是 k。由于 k 的值是 2，因此会创建如图 5-20 所示框架。

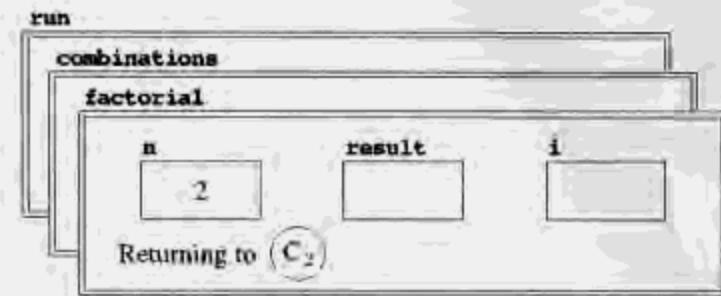


图 5-20 创建第二个调用的框架

`factorial` 方法会再次完成其操作而不会产生进一步调用，方法返回到 C₂，值为 2!（结果是 2）。将该值插入记录当前结果的表达式中，表示下面这种状态的计算：

```
return 120 / ( [2] * factorial(n - k) );
          ↑
          C3
```

现在只能产生一个调用，它开始于对参数表达式(n-k)求值。给定 `combinations` 框架里 n 和 k 的值，参数表达式的值等于 3，这样就会创建如图 5-21 所示的框架。

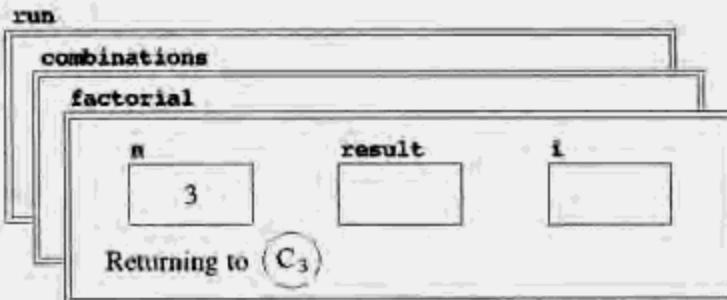


图 5-21 第三个框架

从这里开始，`factorial` 计算 3! 的值，并将值 6 返回到调用者中的 C₃ 位置，会出现如下情况：

```
return 120 / ( [2] * [6] );
```

`combinations` 方法现在具有计算结果(10)必需的所有值。要理解结果与什么有关，需要考察 `combinations` 框架，它再次回到堆栈的最上面，如图 5-22 所示。

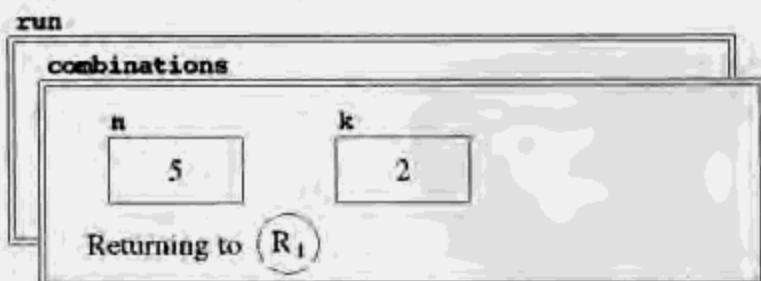


图 5-22 combinations 框架

框架表示，程序应该读取返回值，将用它代替方法 run 中 R₁ 处的调用。如果用 10 替代 `println` 语句中对 `combinations` 的调用，会出现如下情况：

```
println("c(" + n + ", " + k + ")=" + [10]);
```

有了这个结果，`println` 就可以生成所需结果。

分析全部内部细节的练习有助于理解 Java 中的方法调用机制。在这种细节层面上偶尔

研究一下自己的程序会很有益，但是不能养成这种习惯。相反，应该更加随意地考察方法，尽量对其运行原理形成直观感受。程序调用方法时，方法执行其操作，程序接着从产生调用的位置继续运行。如果方法返回结果，调用程序在后面的计算中就可以使用该结果。作为编程人员，要知道可以充分考虑过程而无需担心细节的位置。毕竟，计算机会替您完成这些细节。

5.4 分解

编程人员会面临的最重要挑战是想办法减少程序概念上的复杂性。大型程序通常难以从总体上理解。能够让人理解这些程序的唯一方法是将它们分解为更简单、更易处理的部分。这个过程称为分解。

分解是一条基本策略，应用于编程过程的各个层次。在 Java 中，大型系统首先分解成程序包，程序包再分解成类，类本身就包含一套方法。第 6 章将会学习在这些抽象层次如何应用分解。在方法层次，分解是将大型任务分解为较简单的子任务，总体上这些子任务可以共同完成任务。这些子任务本身也可能需要进一步分解，这类子任务的层次结构如图 5-23 所示。该图表示了典型解决方案的大体结构，没有关于问题本身的细节。图中的整个任务被分解为 3 个主要子任务。第二级子任务进一步细分为更低层次的两个子任务。根据实际问题的复杂性，细分可能需要更多子任务和更多分解层次。

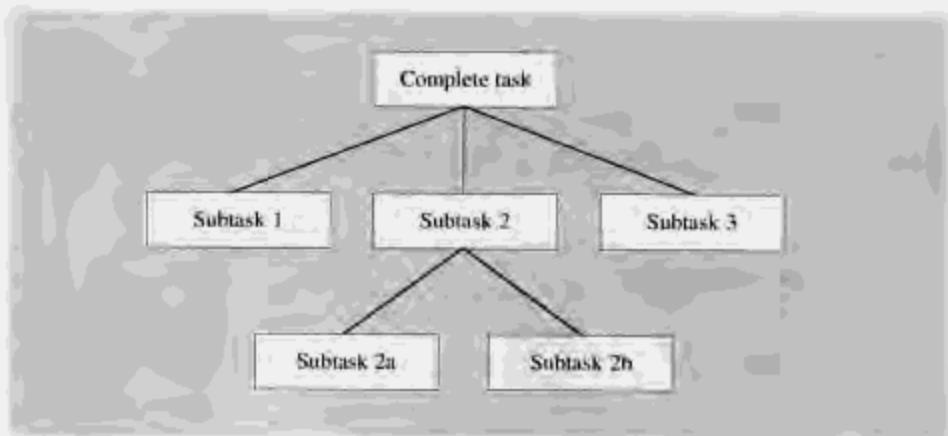


图 5-23 将问题分解为子任务

学习如何找出最有用的分解需要大量实践。如果合适定义单个子任务，每个子任务作为单元有概念上的完整性，会让程序从整体上更容易理解。如果选择的子任务不合适，分解就会遇到麻烦。虽然本章提供了一些有用的指导，但是选择特定分解没有必须遵守的规则；通过实践会知道如何应用该方法。

5.4.1 逐步细化

要找到有效的分解策略，最好的方法是从抽象化的最高级别开始，向下分解到细节。可以从整体考察程序开始。假设程序很大，需要分解，下一步就是将整个问题分解为主要组件。明白主要子任务是什么后，就可以重复分解所有需要分解的子任务（如果子任务本身太大，用简

单几行无法解决)。该过程结束后，就只剩下一组单个任务，每个单独的任务都非常简单，可以作为单个方法执行。这个过程称为从上到下设计，或者称为逐步细化。

学习简单示例是理解逐步细化过程的最好方法。假设要编写 `GraphicsProgram` 程序，用来创建有 3 节车厢的火车，这 3 节车厢由黑色火车头、绿色货车车厢和红色守车组成，如图 5-24 所示。

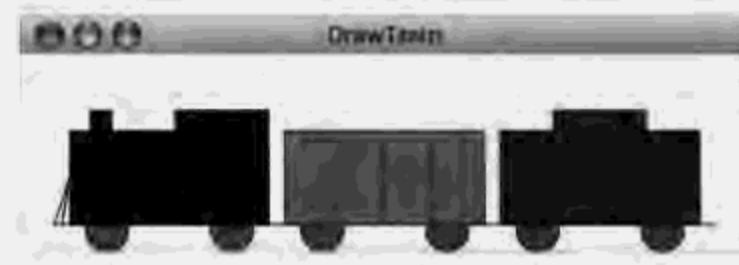


图 5-24 具有 3 节车厢的火车示意图

这样的程序该如何设计呢？

图 5-24 中有两个重要问题需要注意。首先，它完全由已经知道画法的形状组成。每节车厢都由填充的 `GRect` 组成，火车头上的烟囱、火车头上的驾驶室、守车上的炮塔都是这样。货车车厢上的门没有填充 `GRect`。车轮填充为灰色。火车头前面的排障器及火车车厢间的连接器是 `GLine`。由于已经知道如何创建图片的组件，可以开始写 `run` 方法，将每个图形对象添加到位置、大小、尺寸都合适的画布上。然而，这样做之前，应该静下来想一想有没有简化问题的方法。

对于火车图片，第二个需要注意的重要问题是，即使图形比目前为止本书中其他图形示例要复杂，但它很容易分解为更小的部分。画这个特定火车的任务很明显可以分解成 3 个子任务：画火车头，画货车车厢，画守车。因此，如果要采用分解的方法来编写 `run` 方法，应该将它分解为 3 个方法：`drawEngine`、`drawBoxcar` 和 `drawCaboose`。每个方法分别负责画特定类型的车厢。责任划分如图 5-25 所示。



图 5-25 分解车厢示意图(1)

将程序分解为方法有几个好处。第一，这种类型的细分每次着重分析问题的一个部分，所以让程序更容易编写。例如，编写 `drawEngine` 时，可以忽略 `drawBoxcar` 和 `drawCaboose` 的细节。也许更重要的是，这样的分解让程序更易阅读，因此对将来需要修改代码的编程人员而言，充分减少了概念上的复杂性。首先，方法名让人很容易理解代码的每个部分在做什么。如果将所需的单个步骤结合起来使图片创建到单个方法中，读者将很难知道哪些代码行属于图片的特

定部分。最后，分解通常提供了可以重复使用的工具。在这个简单的火车图片中，只有一个货车车厢。在现实世界中，只有一节货车车厢的货车肯定只能亏本运行。要节约成本，一节火车头应该牵引多节这样的货车车厢。需要多少节货车车厢，`drawBoxcar` 方法就可以画多少节这样的车厢。

5.4.2 指定参数

现在，我们已经知道将画火车的问题分解为画各种类型车厢的问题，这样做很有意义。下一步就是决定这些方法如何获得正确绘制图片所需的信息。为了绘制图 5-25 中所示的图片，每个方法都需要不同的信息。例如，`drawBoxcar` 方法需要知道货车车厢的大小、颜色及在画布上的位置，以便它能与其他车厢正确地链接在一起。

可以使用两种策略给方法提供这样信息。一种是以参数的形式从调用者那里传递信息。另一种方法是通过定义有合适值的命名变量来包含作为程序一部分的信息。添加更多参数提供了更大的灵活性，但也让理解方法的运行原理变得更难。调用方法时，必须知道每个参数的意义。如果绘制货车车厢需要指定颜色、它在画布上的位置及车厢本身的大、车门的大小、车轮的半径，可能很难保证这些细节正确无误。另一方面，如果选择将每个参数定义为命名常量，那么就只能绘制一种颜色、一种大小和一个位置的货车车厢。

要避免各种极端问题，需要在这两种策略间寻求平衡。作为通用规则，决策时应用下面的指导原则会有所帮助：

- 单个调用者可能给形参提供不同的值时，使用实参。
- 调用者如果对选择的值满意，使用命名常量。

在火车图片示例中，调用者可能对控制车轮半径不感兴趣。因此，它的值可以指定为命名常量。另一方面，调用者确实需要指定特定车厢应该放置在画布的什么位置。因此应该通过实参传达这种信息，通常在画布上以 x 和 y 的形式体现。其他值，如车厢的大小和颜色，取极大值和极小值的中间值即可。允许调用者指定这些值提供了额外的灵活性，但是也增加了调用概念上的复杂性。本示例中，可以进行如下假设：

- 所有车厢大小相同。
- 火车头都是黑色的。
- 守车都是红色的。
- 货车车厢有多种颜色。

这些假设表示，只有 `drawEngine` 和 `drawCaboos` 参数是放置车厢的位置。`drawBoxcar` 方法需要与车厢颜色相同的信息。因此这 3 种方法的方法头如下：

```
private void drawEngine(double x, double y)
private void drawBoxcar(double x, double y, Color color)
private void drawCaboos(double x, double y)
```

创建图片所需的其余信息可以使用命名常量指定。对于图 5-24 中所示的火车图片而言，这些常量的值如图 5-26 所示。

```
/* Dimensions of the frame of a train car */
private static final double CAR_WIDTH = 75;
private static final double CAR_HEIGHT = 36;

/* Distance from the bottom of a train car to the track below it */
private static final double CAR_BASELINE = 10;

/* Width of the connector, which overlaps between successive cars */
private static final double CONNECTOR = 6;

/* Radius of the wheels on each car */
private static final double WHEEL_RADIUS = 8;

/* Distance from the edge of the frame to the center of the wheel */
private static final double WHEEL_INSET = 16;

/* Dimensions of the cab on the engine */
private static final double CAB_WIDTH = 35;
private static final double CAB_HEIGHT = 8;

/* Dimensions of the smokestack and its distance from the front */
private static final double SMOKESTACK_WIDTH = 8;
private static final double SMOKESTACK_HEIGHT = 8;
private static final double SMOKESTACK_INSET = 8;

/* Dimensions of the door panels on the boxcar */
private static final double DOOR_WIDTH = 18;
private static final double DOOR_HEIGHT = 32;

/* Dimensions of the cupola on the caboose */
private static final double CUPOLA_WIDTH = 35;
private static final double CUPOLA_HEIGHT = 8;
```

图 5-26 控制火车图片参数的命名常量

如标题行所示, drawEngine、drawBoxcar 和 drawCaboose 方法使用 x 和 y 参数来表示画布上车厢的位置。使用这些方法的调用者需要知道这些坐标的意义, 根据相对于点(x, y)的位置来决定将车厢画在哪里。这有许多种可能和解释, 编程人员的工作就是选择合适的设计。一种可能是采用 acm.graphics 程序包使用的约定, 将点(x, y) 定义为车厢左上角。与此相反, 在没有其他具有说服力的理由的情况下, 可以选择一种与现有库程序包相匹配的模型。然而, 在这里, 也许确实有一个具有说服力的理由。火车车厢左上角的位置取决于车厢的高度。本示例中, 火车头和守车在各自顶部有图形, 这样它们就比货车车厢高。因此, 要计算车厢顶部的 y 坐标, 就要知道车厢的类型。另一方面, 计算车厢底部的 y 坐标很容易, 因为所有车厢都在铁轨上。因为所有车厢的基线相同, 所以让 x 参数表示从车箱伸出来的连接器的左边, 用 y 参数表示铁轨的水平面。因此这种解释与 acm.graphics 程序包用于 GLabel 类是相同的。因为每个字母都在水平基线上, 所以 GLabel 类中的 y 坐标就表示基线的位置。

5.4.3 自顶向下设计

既然已经决定了 rawEngine、drawBoxcar 和 drawCaboose 方法的方法头, 从逻辑上说, 下一步似乎是继续完成它们。但是通常在开始下一步之前, 最好在每个分解层面上完成代码。这样做, 就可以确信自己选择了正确的分解方法。如果遗漏了子任务或没有在方法的参数中包括足够的灵活性, 当编码程序的最高级时就会发现这些问题。自顶向下设计的原则说明, 应该从 run 方法开始, 然后具体分析细节。

将问题分解为辅助方法使 run 方法变得相当简单。run 方法的主体包括对 drawEngine、drawBoxcar 和 drawCaboose 方法的调用。唯一剩下的问题是弄明白给每个调用中的坐标取什么值。如果查看一下图 5-24 中的图片，就会发现火车位于画布的底部。因此铁轨的 y 坐标等于画布底部的 y 坐标；因为向下移动时 Java 坐标值会增加，所以画布底部的 y 坐标就是画布的高度，该高度可以通过调用 getHeight 方法获得。找到每节车厢的 x 坐标需要更多的计算。本示例中，整列火车位于窗口中央。因此，计算火车头的 x 坐标需要知道整列火车的长度，然后用画布中心的坐标减去该长度的一半。每节随后的车厢开始于某个点，该点的 x 坐标是向右移动一节车厢的宽度加上其连接器的宽度的距离。Java 中表达这些计算使用下面的 run 方法：

```
public void run() {
    double trainWidth = 3 * CAR_WIDTH + 4 * CONNECTOR;
    double x = (getWidth() - trainWidth) / 2;
    double y = getHeight();
    double dx = CAR_WIDTH + CONNECTOR;
    drawEngine(x, y);
    drawBoxcar(x + dx, y, Color.GREEN);
    drawCaboose(x + 2 * dx, y);
}
```

5.4.4 寻找共同特征

既然 run 方法没有问题，就可以将注意力转移到画单个车厢的方法上。一种方法就是急不可待地拿出键盘，开始输入代码，但是花点时间仔细考虑一下这个问题也会大有好处。选择一种分解时，期待的一件事就是出现的子任务不仅仅是问题的一部分。要理解这种策略如何应用于当前的问题，有必要考察一下这 3 个不同类型的车厢，如图 5-27 所示。



图 5-27 分解车厢示意图(2)

如果仔细考察这 3 节车厢的图形，就会发现它们具有一些共同特征：车轮相同，链接车厢的连接器也相同。实际上，车厢本身除了颜色不同以外，其主体也相同。因此，每种类型的车厢都具有共同架构，如图 5-28 所示。



图 5-28 车厢的共同架构示意图

如果将该图片的灰色部分填充上适合的颜色，就可以将它作为这 3 种类型车厢的基础。对于火车头而言，需要添加烟囱、驾驶室和排障器。对于货车车厢而言，需要添加两扇门。对于守车而言，需要添加炮塔。在这种情况下，定义画车厢基本架构的方法可以取代画车厢的大量

工作。这种方法——本身分解为画每个车轮——如下：

```
private void drawCarFrame(double x, double y, Color color) {
    double x0 = x + CONNECTOR;
    double y0 = y - CAR_BASELINE;
    double top = y0 - CAR_HEIGHT;
    add(new GLine(x, y0, x + CAR_WIDTH + 2 * CONNECTOR, y0));
    drawWheel(x0 + WHEEL_INSET, y - WHEEL_RADIUS);
    drawWheel(x0 + CAR_WIDTH - WHEEL_INSET, y - WHEEL_RADIUS);
    GRect r = new GRect(x0, top, CAR_WIDTH, CAR_HEIGHT);
    r.setFilled(true);
    r.setFillColor(color);
    add(r);
}

private void drawWheel(double x, double y) {
    double r = WHEEL_RADIUS;
    GOval wheel = new GOval(x - r, y - r, 2 * r, 2 * r);
    wheel.setFilled(true);
    wheel.setFillColor(Color.GRAY);
    add(wheel);
}
```

5.4.5 完成分解

拥有创建单个车厢架构的工具之后，只要填充 drawEngine、drawBoxcar 和 drawCaboose 方法的定义就可以完成 drawTrain 程序。其中有些方法很简单，无需进一步分解就可进行编码。例如，drawBoxcar 的实现是画背景画面，然后添加两个门板。

```
private void drawBoxcar(double x, double y, Color color) {
    drawCarFrame(x, y, color);
    double xRight = x + CONNECTOR + CAR_WIDTH / 2;
    double xLeft = xRight - DOOR_WIDTH;
    double yDoor = y - CAR_BASELINE - DOOR_HEIGHT;
    add(new GRect(xLeft, yDoor, DOOR_WIDTH, DOOR_HEIGHT));
    add(new GRect(xRight, yDoor, DOOR_WIDTH, DOOR_HEIGHT));
}
```

然而，可以选择进一步分解某些方法。例如，决定将火车图形分解为其组成部分，然后如下编码 drawEngine：

```
private void drawEngine(double x, double y) {
    drawCarFrame(x, y, Color.BLACK);
    drawSmokestack(x, y);
    drawCab(x, y);
    drawCowcatcher(x, y);
}
```

处理问题较小部分的私有方法通常称为辅助方法，如本示例中的 drawSmokestack、drawCab 和 drawCowcatcher。

如果采用这种分解策略，就必须为这些辅助方法以及仍未执行的 drawCaboose 方法编写实现工具。在 5.8 节的练习 8 中有机会完成这种分解。

逐步细化是一种相当重要的技能，与此同时，面向对象语言还提供了另一种简化问题的方法。这种方法不是将程序分解为连续执行更简单的子任务，而是通过定义类的层次结构(反映了相同的分解策略)实现相同的目标。用 Java 编程时，这种策略通常有明显的优势。在第 9 章，会定义类层次结构，这些层次结构中每个特殊的车厢类型——火车头、货车车厢和守车——都是包含所有单个类型的常见类的子类。

5.5 算法方法

方法除了是管理复杂程序的一种工具之外，对于编程也至关重要，因为它们是算法实现(第 1 章简要介绍过)的基础。算法是一种抽象策略，编写方法是用编程语言表达该算法的一种常规方法。因此，将算法作为程序的一部分执行时，通常编写一种方法(这种方法可能依次调用其他方法来处理部分工作)来实现该算法。

虽然已经见过示例程序中执行的几种算法，但没有机会理解算法过程本身的实质。目前为止遇到的许多编程问题都非常简单，脑海中会立即出现适合的解决方案方法。然而随着问题越来越复杂，其解决方案需要更深的思考，在编写最终程序之前，需要考虑多种策略。

作为算法策略形成方法的示例，本节将考察经典数学问题的两种解决方案，该数学问题就是找出两个整数的最大公约数。假设两个整数 x 和 y ，最大公约数(简写为 gcd)是能被这两个整数整除的最大整数。例如，49 和 35 的 gcd 是 7，6 和 18 的 gcd 是 6，32 和 33 的 gcd 是 1。

假设要求编写一种方法，输入整数 x 和 y ，返回最大公约数。从调用者的观点来看，需要的是方法 $\text{gcd}(x, y)$ ，该方法将这两个整数作为参数，返回另一个是最大公约数的整数。因此该方法的标题行如下：

```
public int gcd(int x, int y)
```

如何设计算法来执行该计算呢？

5.5.1 “强力”方法

在许多情况下，计算 gcd 最明显的方法是尝试各种可能性。可以“猜测” $\text{gcd}(x, y)$ 是 x 和 y 中较小的那个数，因为所有较大的值不可能被较小的数除尽。然后继续用 x 和 y 除以猜测的数，看看它能否被这两个数整除。如果可以，答案就出来了；如果不能，从猜测的数中减去 1，然后再试。尝试每种可能性的策略通常称为“强力”方法。Java 中计算 gcd 函数的“强力”方法如下：

```
public int gcd(int x, int y) {
    int guess = Math.min(x, y);
    while (x % guess != 0 || y % guess != 0) {
        guess--;
    }
    return guess;
}
```

在确定这种实现实际上是计算 gcd 函数的合法算法之前，需要问问自己有关代码的几个问题：gcd 的强力实现方式总是能得出正确答案吗？它总是会终止吗？方法会永远继续吗？

要知道程序能否得出正确答案，需要考察 while 循环中的条件。

```
x % guess != 0 || y % guess != 0
```

通常，while 条件表示在什么情况下循环继续。要找出导致循环终止的条件，必须对 while 条件求反。涉及`&&`或`||`的求反条件要非常小心，除非记得如何应用 De Morgan 法则。De Morgan 法则指出，当 while 循环存在时，必须满足下面的条件：

```
x % guess == 0 & y % guess == 0
```

从这个条件出发，就会发现 guess 的最终值就是公约数。要确认它就是最大公约数，必须考察嵌入 while 循环的策略。要注意的重要因素是，程序向后计数所有可能性。最大公约数决不会比 x (或 y) 大，因此强力搜索从 x 和 y 中的较小值开始。如果程序超出了 while 循环，那么它必须尝试过 x 和 guess 当前值之间的每个值。因此，如果有一个更大的值被 x 和 y 除尽，程序在 while 循环的早期迭代中总是会找到这个值。

要确认方法终止，关键是 guess 的值最终必须取到 1，即使没有找到最大公约数也是如此。此时，while 循环确实会终止，因为 1 能被 x 和 y 整除，不管这些变量取什么值。

5.5.2 欧几里得算法

然而，“强力”算法并不是唯一有效策略。虽然“强力”算法在其他语境中有一席之地，但如果考虑效率，对于 gcd 函数而言，“强力”算法不是一个好的选择。例如，如果调用方法计算整数 1 000 005 和 1 000 000 的最大公约数，在得出 5 之前，“强力”算法会执行 while 循环主体几乎 100 万次——而只要考察这两个数就很容易得出 5 这个值。要找出一种算法，它可以确保找到正确答案就终止，同时所需的步骤比强力算法少。这是问题的高明解答和明确理解的体现。很幸运，古希腊数学家欧几里得(Euclid)已经提供了所需的创造性理解，他的 *elements(book 7, proposition II)* 讨论了这种问题的完美解决方案。欧几里得的算法可以描述如下：

- (1) 用 x 除以 y，计算余数，将余数称为 r。
- (2) 如果 r 是 0，过程结束，答案就是 y。
- (3) 如果 r 不等于 0，就将 x 设置为 y 以前的值，y 设置等于 r，重复整个过程。

很容易将这种算法描述为如下的 Java 代码：

```
int gcd(int x, int y) { int r = x % y; while (r != 0) { x = y; y = r; r = x % y; } return y; }
```

gcd 方法的这种实现方式也正确找到了两个整数的最大公约数。它有两点与“强力”实现方式不同：一方面，它能更快速地计算出结果；另一方面，证明它正确更难。

5.5.3 讨论欧几里得算法的正确性

虽然欧几里得算法正确性的正式证明超出了本书的范围，但是采用希腊人使用的数学智力模型，可以对算法的运行原理形成直观认识。在希腊数学中，几何学处于中心地位，它将数字作为距离看待。例如，欧几里得要找出两个整数(如 55 和 15)的最大公约数，会将这个问题转换为寻找最长的量尺，该量尺可以用来划分这两段距离。这样，可以用两根长条将特殊问题形

象化，这两根分别为 55 个单位长和 15 个单位长，如图 5-29 所示。

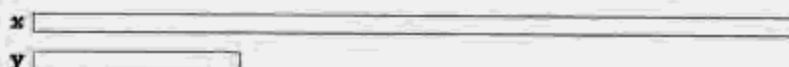


图 5-29 划分为两段距离

问题是找出新的量尺，该量尺可以首尾相连地放置在这两根长条上，以便刚好覆盖 x 和 y 的距离。

欧几里得算法从用较短的长条划分较长的长条开始，如图 5-30 所示。



图 5-30 用较短的长条划分较长的长条

除非较小的数刚好是较大数的约数，否则就有余数，如图 5-30 上面那根长条的阴影部分所示。本示例中，55 除以 15 等于 3，余数为 10，说明阴影部分有 10 个单位长度。欧几里得的基本观点是两段原始距离的最大公约数必须是较短长条长度和图中阴影部分所表示的距离的最大公约数。

通过将原始问题简化为包含更小数的更简单的问题，可以解决原始问题。本例中，新的数字是 15 和 10，重复欧几里得算法可以得出它们的最大公约数。用适合长度的量尺表示新的值 x' 和 y' 开始。然后用较小单位的长条划分较长的长条，如图 5-31 所示。



图 5-31 再次划分

该过程结束后又有一段剩余部分，这一次它的长度为 5。如果再次重复该过程，就会发现，长度为 5 的阴影部分本身是 x' 和 y' 的公约数，因此，按照欧几里得的观点，它也是原始数字的公约数。这个新值确实是原始数的公约数，如下图 5-32 所示。



图 5-32 确定最大公约数

欧几里得在《Elements》中提供了其命题的完整证据。如果您对 2500 年前的数学家如何思考这样的问题感兴趣，请查看原始资料。

5.5.4 两种算法的效率比较

为了说明用来计算最大公约数的这两个算法策略在效率方面的不同，再考察一下 1 000 005 和 1 000 000 这两个整数。要找出这两个整数的最大公约数，“强力”算法需要 100 万步；欧几里得算法只需要两步。欧几里得算法开始时，在第一个循环周期内， x 是 1000005， y 是 1 000 000， r 设置为 5。由于 r 不等于 0，程序将 x 设置为 1 000 000，将 y 设置为 5，重新开始。在第二个循环周期内， r 的值为 0，因此程序退出 while 循环，报告答案为 5。

本章介绍的计算最大公约数的两个策略清晰说明，算法的选择对解决方案的效率有重大影响。如果继续学习计算机科学，就会知道如何量化性能方面的不同，以及提高算法效率的一些常见方法。

5.6 小结

本章学习了方法，让您使用单个名称就可以引用完整的运算集。更重要的是，允许编程人员忽略内部细节，着重考察方法的整体效果。方法为简化程序概念上的复杂性提供了重要工具。本章介绍的重点如下：

- 方法由一组集合在一起并命名了的程序语句组成。程序的其他部分可以调用方法，并可能以参数的形式传递信息，接收方法返回的结果。
- 返回值的方法必须包含指定结果的 `return` 语句。方法可以返回任何类型的值。
- 在像 Java 这样的面向对象语言中，方法调用用来表示从一个对象向另一个对象发送消息的过程。使用方法调用发送消息时，需要指定接收方作为方法调用的一部分，例如：
`receiver.name(arguments)`
- 返回 `Boolean` 值的方法称为断言方法，因为使用该方法的结果来指定 `if` 或 `while` 语句中的条件，所以断言方法在编程中举足轻重。
- 在方法主体内，作为参数值占位符的变量称为形参。
- 方法声明的变量对于方法而言是局部的，不能在方法之外使用。在内部，所有在方法内声明的变量(包括参数)都存储在堆栈框架里。
- 方法返回时，它从进行调用的地方继续。计算机将该点称为返回地址。
- 方法是简化程序复杂性的强大工具，因为它允许将大型任务分解为更小、更易于管理的子任务。
- 很多情况下，从将问题看作整体开始分解程序，然后深入到细节，这种方法很有效。这个过程称为自顶向下设计，或者称为逐步细化。
- 因为方法与语句集合联系在一起以便产生特殊效果，所以 Java 中方法为表述算法提供了标准架构。
- 解决特殊问题通常有许多不同的算法。选择最适合应用程序的算法是编程人员的重要任务之一。

5.7 复习题

1. 用自己的语言说明方法和程序之间的不同。
2. 给出下列术语应用到方法时的定义：调用、参数、返回。
3. 使用参数给方法传递信息与使用方法(如 `readInt`)读取输入数据之间有何不同？分别在何时适合？
4. Java 中如何指定方法的结果？

5. 在方法主体内可以有多个 return 语句吗？
6. 如果想将方法应用于另一个对象，该如何表示？
7. 前几章提到的程序中，为什么没有必要指定接收方？
8. 在本章讨论的 monthName 方法中，每个 case 子句末尾为什么都无需包含 break 语句？
9. 什么是断言方法？
10. 如何判断两个字符串是否包含相同的字符？
11. 实参与形参之间是什么关系？
12. 方法内声明的变量称为局部变量。在此语句中，单词“局部”有何重要性？
13. 术语“返回地址”是什么意思？
14. 用自己的语言解释逐步细化过程。
15. 什么是强力算法？
16. 使用欧几里得算法计算 7735 和 4185 的最大公约数。计算过程中局部变量 r 取什么值？
17. 本章运用欧几里得算法计算 gcd(x, y) 的示例中，x 总是大于 y。如果 x 比 y 小，情况又如何？

5.8 编程练习

1. 在高中代数中，我们知道标准的一元二次方程

$$ax^2 + bx + c = 0$$

有两个解，其公式是：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

用+取代 \pm 可以得出第一个解；用-取代 \pm 可以得出第二个解。

编写 Java 程序，给定 a、b 和 c 的值，然后求这两个解。如果平方根符号内的数是负数，方程就无解，程序应该对这种结果显示一条消息。可以假设 a 的值非零。程序的运行结果可如图 5-33 所示。

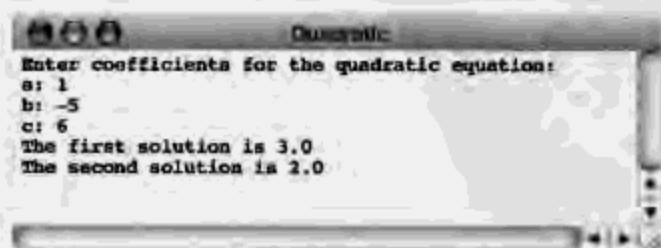


图 5-33 Quadratic 的运行结果

2. 第 4 章练习 9 介绍了斐波纳契(Fibonacci)数列，该数列中新的项是前面两项之和。重新编写该程序，改变实现方式，让程序调用方法 fibonacci(n) 来计算第 n 个斐波纳契数。从所需的数学计算的次数来看，新的实现方式有没有第 4 章使用的实现方式有效？

3. 编写方法 `raiseIntToPower`, 让它读入两个整数 n 和 k , 返回 n^k . k 从 0 取到 10, 利用您的方法显示 2^k 计算值的列表。

4. 编写方法 `countDigits(n)`, 返回整数 n 中数字的个数, 可以假设 n 为正数。设计主程序测试您的方法。提示: 可以回头查看图 4-8 中的 `DigitSum` 程序。

5. 使用图 5-4 中的 `createFilledCircle` 方法重写第 2 章练习 5 中的 `Target` 程序。另外, 改变程序, 让靶子总是位于窗口中央, 让圆的数量和尺寸由下面的命名变量控制:

```
private static final int N_CIRCLES = 5;
private static final double OUTER_RADIUS = 75;
private static final double INNER_RADIUS = 10;
```

给定这些值, 程序应该生成如图 5-34 所示的结果。



图 5-34 箭靶

6. 重写图 5-4 中的 `DrawStoplight` 程序, 让它也包含私有方法 `createFilledRect`。该方法创建填充的 `GRect` 对象的方法与 `createFilledCircle` 创建填充的 `GOval` 的方法相同。问题是计算出 `createFilledRect` 方法需要采用什么参数。

7. 编写断言方法 `askYesNoQuestion(prompt)`, 打印字符串 `prompt` 作为用户的问题, 然后等待响应。如果输入 “yes”, `askYesNoQuestion` 方法应该返回 `true`; 如果用户输入 “no”, 该方法返回 `false`; 如果用户输入其他字符串, `askYesNoQuestion` 方法应该提醒用户这是一个寻求“是—否”回答的问题, 然后重复问题。例如, 如果程序包括语句

```
if (askYesNoQuestion("Would you like instructions? "))
```

则与用户的互动如图 5-35 所示。

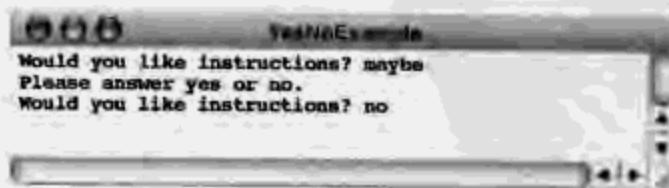


图 5-35 YesNoExample 的样本运行示意图

8. 利用方法 `drawSmokestack` `drawCab` `drawCowcatcher` 和 `drawCaboose` 完成 `DrawTrain` 程序的实现。

9. 编写 `GraphicsProgram` 程序, 画出如图 5-36 所示的图形。



图 5-36 GraphicsProgram 生成的图形

确保使用逐步细化的方式将图形分解为有用的部分。

10. 如果您认为前面练习中的房子有点普通，那么就画一个 House of Usher, Edgar Allan Poe 描述如下：

第一眼看到这栋建筑，一种深深的忧郁便笼罩了我的灵魂。……我的眼前——只有房子和简单的风景——暗淡的墙壁——像眼睛一样神情茫然的窗口。……几根白色的腐朽的树干——绝对沮丧的灵魂……

从 Poe 的描述中，可以想象一座房子和周围的场景，如图 5-37 所示。

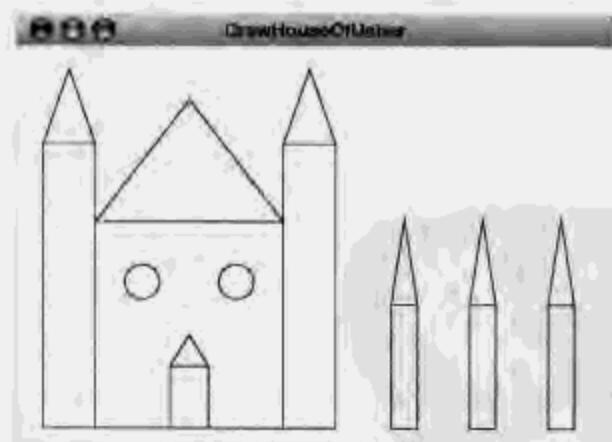


图 5-37 DrawHouseOfUsher 的样本运行

左边的图形是有“像眼睛一样神情茫然的窗口”的房子，右边的 3 个图形是“几根白色的腐朽的树干”的程式化表述。

编写 GraphicsProgram 程序画出如图 5-33 所示的图形。确保使用逐步细化的方式对程序进行分解，在图形中寻找重复元素，使用命名常量指定组成画面中不同图形的尺寸。

11. 大于 1 的整数，如果它除了自身和 1 之外没有其他约数，那么该整数称为素数。例如，17 是素数，因为除了 17 和 1 之外，它没有其他约数。然而，91 就不是素数，因为它可以被 7 和 13 除尽。编写断言方法 isPrime(n)，如果整数 n 是素数，返回 true；否则，返回 false。作为

开始策略，使用测试每个可能约数的强力算法执行 `isPrime`。运行这种版本之后，努力改进算法，在确保其正确性的情况下提高运行效率。

12. 希腊数学家对等于其完全约数之和的数很感兴趣(n 的完全约数是小于 n 本身的所有约数)。他们将这种数称为完全数。例如，6 就是完全数，因为它是 1、2、3 的和，它们都是小于 6 的整数，6 可以整除它们。同样，28 也是完全数，因为它是 1、2、4、7、14 的和。

编写断言方法 `isPerfect(n)`，如果整数 n 是完全数，返回 `true`；否则，返回 `false`。写主程序来测试您的实现方式，该程序可以使用 `isPerfect` 方法依次测试 1~9999 之间的每个数来核对完全数。确定一个完全数后，程序应该在屏幕上显示该数。输出的前两行应该是 6 和 28。程序应该还可以在此范围内找出其他两个完全数。

13. 虽然前面练习中欧几里得算法和寻找完全数的问题都来自数学领域，希腊人对其他类型的算法也很有兴趣。例如，在希腊神话中，雅典的特修斯神拿着一个线球，边走边放线，然后按照线的路径回到了出口，这样他从人身牛头怪物的迷宫中逃脱了出来。特修斯的策略代表了一种从迷宫逃脱的算法，但是这不是解决这一问题的唯一算法。例如，如果迷宫没有内部循环，那么将右手一直放在墙壁上就可以走出迷宫，这种算法称为右手规则。

例如，假设特修斯在图 5-28 所示的迷宫中迷路了。图中特修斯的位置用希腊字母 Θ 标记。

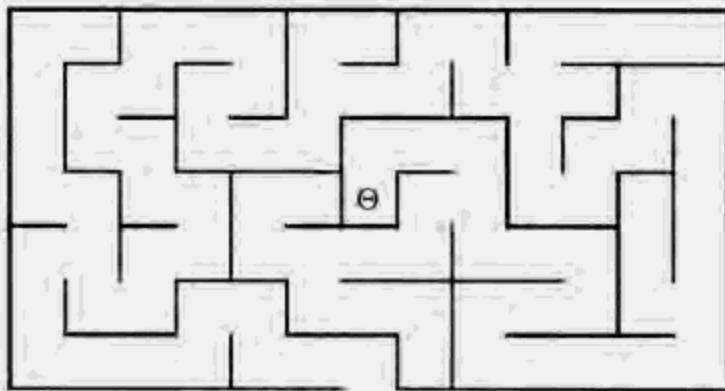


图 5-38 迷宫示意图

要逃脱出来，特修斯沿着图 5-39 中点线所示的路径行走，此过程中他的右手不能离开墙壁。

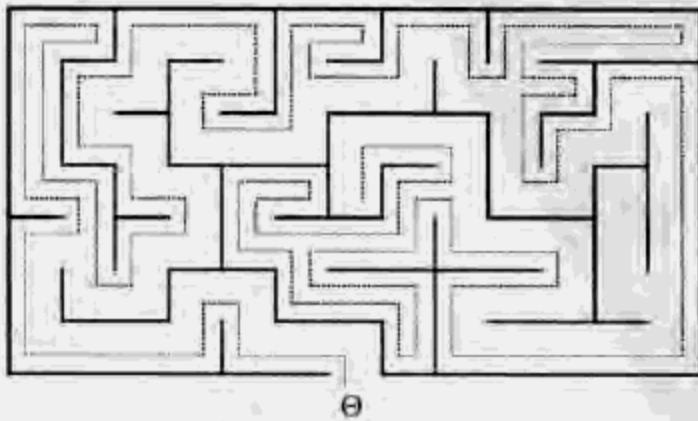


图 5-39 右手规则示意图

假设要编写一段程序让机器人从迷宫中走出来。假设已经访问了名为 `MazeRunningRobot` 的类，该类执行下面的方法：

```
void moveForward()      /*Move forward one square*/  
void turnRight()        /*Turn right without moving*/  
void turnLeft()         /*Turn left without moving*/  
boolean isBlocked()    /*True if facing a wall*/  
boolean isOutside()    /*True if outside the maze*/
```

请利用这些方法编写算法方法

```
private void solveMaze(MazeRunningRobot robot)
```

让机器人利用用右手规则走出迷宫。



第6章

对 象 和 类

To beautify life is to give it an object.

—Jose Mart, On Wilde, 1882



Kristen Nygaard(1926—2002) Ole-Johan Dahl(1931—2002)

40多年前，挪威计算机科学家 Kristen Nygaard 和 Ole-Johan Dahl 在开发编程语言 SIMULA 的过程中，形成了面向对象编程的中心思想。SIMULA 的早期版本出现在 20世纪60年代早期，但稳定版发行于1967年，该版本中的这些观念引起了全世界的广泛注意。SIMULA最初的工作在挪威计算中心(Norwegian Computing Center)展开，该中心是挪威国家研究实验室，专门开发更好的软件工程技术。后来，Kristen Nygaard 和 Ole-Johan Dahl 都成了挪威奥斯陆大学的教师。虽然业内花了几十年时间来接受他们的成果，但对面向对象技术的兴趣却在过去20年内迅速增长，特别是在现代面向对象语言(如C++和Java)出现之后。鉴于他们所做的贡献，2001年Nygaard 和 Dahl 同时获得了计算机协会授予的图灵奖，同年获得了电气和电子工程师协会授予的 John von Neumann 奖章。

第2章介绍将程序作为整体对待，以此为开端，第3~5章采用简化的方法，从头开始介绍了编程基本组成要素的使用方法——表达式、语句和方法。在某种程度上，本章通过考察对象和类，扩充了级别，从 Java 语言层次结构的下一级开始。另一方面，对象和类是用高级、抽象方法考察程序的主要工具。从高级的观点来看，本章又重新回到了第2章的整体观点。本章以一个示例开始，该示例用来说明如何使用本身非常重要的现有类——`acm.util` 程序包里的 `RandomGenerator` 类。本章其余部分讨论在 Java 应用程序中如何实现自己的类。

虽然对类和对象采用整体观点对于有效使用它们至关重要，但是这种观点通常不足以理解对象在实践中的表现。对绝大多数同学而言，开发对象的智力模型也很重要，这种智力模型能够形象化地说明在机器内部如何表示对象。为了避免高级设想与太多细节之间产生混淆，本书将这些内部表示的问题放在第7章讨论。

6.1 使用 RandomGenerator 类

到目前为止，本书中的所有程序都可确定地运行，意思是说，由给定的输入值可以预测其动作。这种程序行为是可重复的。如果今天运行程序，它产生了特定结果，明天它也会产生同样的结果。

然而有些编程应用程序中(如游戏或模拟)，程序行为不可预测就显得非常重要。如果计算机游戏总是出现相同结果，这个游戏就很乏味。为了创建可以随机产生行为的程序，需要某个表示随机过程的机制，如抛硬币或掷骰子。模拟这种随机事件的程序称为非确定程序。

因为早期计算机主要用于数字应用程序，所以使用计算机生成随机事件的思想通常表达为能够生成特定范围内的随机数。从理论上讲，在一组相等可能的结果中，如果不能预先确定这个数取什么值，那么这个数就是随机的。例如，掷骰子生成的随机数介于 1~6 之间。如果骰子是正常的，就无法确定会出现哪个数。6 个可能值出现可能性相等。

虽然随机数的思想能形成直观感觉，但是在计算机内部表示它却很困难。计算机通过执行内存里的一系列指令运行，所以其运行模式是确定的。如果确定的过程中生成某个数，所有用户应该都能够执行那一组相同的规则，预见计算机的响应。这种情况似乎很荒谬，如何在遵守确定规则的同时，让计算机生成不可预测的结果？

6.1.1 伪随机数

几乎所有情况下，今天的计算机编程人员都通过放弃真实随机的思想来回避这种悖论。使用随机数的 Java 程序实际上使用确定的程序计算这些数。虽然从理论上讲，这样做意味着用户遵循相同的规则、预见计算机的响应，但实际上这种策略有用，因为实际没有人执行这些计算。在大多数实际应用程序中，数是不是随机的并不重要；重要的是数要看起来似乎是随机的。要数看起来随机，应该满足两个条件：从统计的观点来看它们随机；用户事先很难预知。计算机内部算法过程生成的“随机”数正式称呼为伪随机数，用来强调不包含真正的随机行为。然而，在非正式场合通常称它们为随机数，尽管不完全正确。

按照随机数的计算要求生成随机数的方法很简单，但是按照其数学基础生成随机数则非常复杂。斯坦福大学教授 Don Knuth 是里程碑式系列图书《计算机程序设计艺术》的作者，他用了 100 多页的篇幅来讨论如何编码随机数生成器，以便让输出尽可能随机。这种数学分析的细节远远超出了介绍性文本的范围。很幸运，几乎没有编程人员通过编写代码来生成随机数。有的入学完了必需的数学知识，辛苦地解决了必需的复杂的理论和实践问题，开发出了这样的代码。大多数编程人员只对使用随机数感兴趣，对如何计算等细节不感兴趣。对于这些编程人员而言，可以将随机数生成器当成是一个黑盒子。向它请求随机值，输出另一个随机数。

将黑盒子当作程序的一部分，这是面向对象编程概念的基本思想。要生成随机数，需要随机数生成器，如下面的盒子所示。



在 Java 域内，黑盒子构成一个对象。它的类是 RandomGenerator，在 `aem.util` 程序包里定义，如标签所示，盒子是 RandomGenerator 类的特定实例。要生成随机值时，给该对象发送一条消息就可以了，如下所示。



然后随机数生成器对象会输出下一个随机数，这个数由某个算法过程计算出来，计算过程的细节隐藏在黑盒子内部。例如，请求一个随机数，随机数生成器对象可以回答如下：



和面向对象编程中的常见情况一样，消息采用方法调用的形式。回答就是方法返回的值。随机数生成器对象依据应用程序所需的随机值类型，对几种不同方法作出响应，这将在下一节介绍。

6.1.2 使用 RandomGenerator 类

RandomGenerator 类中最重要的公有方法如图 6-1 所示。这些方法大部分很容易理解。唯一需要进一步解释的方法是静态 getInstance 方法，它返回 RandomGenerator 类的新实例。要记住的是，RandomGenerator 类本身不是随机值，而是生成随机值的对象。虽然在程序中通常要生成许多不同的随机值，但只需要一个生成器就够了。而且，因为可能要在程序内的几个方法中使用该生成器，因此最好将它声明为实例变量，如下所示。

```
private RandomGenerator rgen = RandomGenerator.getInstance();
```

此行既声明了变量 rgen，又将它初始化为整个程序随机值的源。

static RandomGenerator getInstance()	返回 RandomGenerator 类的实例，整个程序都共享它
int nextInt(int n)	返回随机整数。它从范围 0~n-1(包括 n-1)的 n 个值中选出
int nextInt(int low, int high)	返回范围 low~high(包括)的随机整数
double nextDouble()	返回范围在 0 ≤ d < 1 里的随机双精度 d；不包括一边的范围称为半开
double nextDouble(double low, double high)	返回在半开范围 low ≤ d < high 里的随机双精度 d
boolean nextBoolean()	返回大约 50% 的时间是 true 的 boolean 值
boolean nextBoolean(double p)	返回概率为 p(介于 0 和 1 之间)时为 true 的 boolean 值
Color nextColor()	返回随机 Java 颜色
void setSeed(long seed)	设置“种子”来表示伪随机序列的起始点

图 6-1 RandomGenerator 类中有用的方法

声明和初始化 rgen 变量之后，就可以使用 RandomGenerator 类中的其他方法生成相应类型的下一个随机值。选择什么方法取决于应用程序。例如，如果需要特定范围内的随机整数，可以使用 nextInt 的双参数形式。因此，要模拟掷标准的 6 面骰子，可以编写如下代码：

```
int die = rgen.nextInt(1, 6);
```

如果要生成介于 -1.0 和 +1.0 之间的随机实数，可以调用

```
double x = rgen.nextDouble(-1.0, +1.0);
```

如果要模拟抛硬币，可以使用 nextBoolean：

```
String coinFlip = rgen.nextBoolean() ? "Heads" : "Tails";
```

图 6-2 中的程序说明了 RandomGenerator 类在整个应用程序语境中的使用方法。程序模拟 Craps 娱乐游戏，其玩法如下：掷两个 6 面骰子，计算总数，依据第一掷的情况，游戏分解为下面几种情形：

```
/*
 * File: Craps.java
 *
 * This program plays the casino game of Craps. At the beginning of the game,
 * the player rolls a pair of dice and computes the total. If the total is 2,
 * 3, or 12 (called "craps"), the player loses. If the total is 7 or 11
 * (called a "natural"), the player wins. If the total is any other number,
 * that number becomes the "point." From here, the player keeps rolling the
 * dice until (a) the point comes up again, in which case the player wins or
 * (b) a 7 appears, in which case the player loses. The numbers 2, 3, 11,
 * and 12 no longer have special significance after the first roll.
 */

import acm.program.*;
import acm.util.*;

public class Craps extends ConsoleProgram {
    public void run() {
        int total = rollTwoDice();
        if (total == 7 || total == 11) {
            println("That's a natural. You win.");
        } else if (total == 2 || total == 3 || total == 12) {
            println("That's craps. You lose.");
        } else {
            int point = total;
            println("Your point is " + point + ".");
            while (true) {
                total = rollTwoDice();
                if (total == point) {
                    println("You made your point. You win.");
                    break;
                } else if (total == 7) {
                    println("That's a 7. You lose.");
                    break;
                }
            }
        }
    }

    /* Rolls two dice and returns their sum. */
    private int rollTwoDice() {
        int d1 = rgen.nextInt(1, 6);
        int d2 = rgen.nextInt(1, 6);
        int total = d1 + d2;
        println("Rolling dice: " + d1 + " + " + d2 + " = " + total);
        return total;
    }

    /* Create an instance variable for the random number generator */
    private RandomGenerator rgen = RandomGenerator.getInstance();
}
```

图 6-2 玩 Craps 游戏的程序

第一次掷出的是 2、3 或 12。第一次掷时掷出这些数称为 crap，说明输了。

第一次掷出的是 7 或 11，如果其中的某个数在第一掷中出现，称为 natural，获胜。

第一次掷出的是其他数(4、5、6、8、9 或 10)。这种情况下，掷出的数称为 Point(点数)，继续掷骰子，直到第二次掷出 Point，获胜；或者掷出 7，输了。如果掷出其他数——包括 2、3、11 和 12，就继续掷，直到掷出 point 或 7 为止。

图 6-2 中的程序直接将英语规则转换成了 Java 代码。图 6-3 显示了各种可能的运行结果。

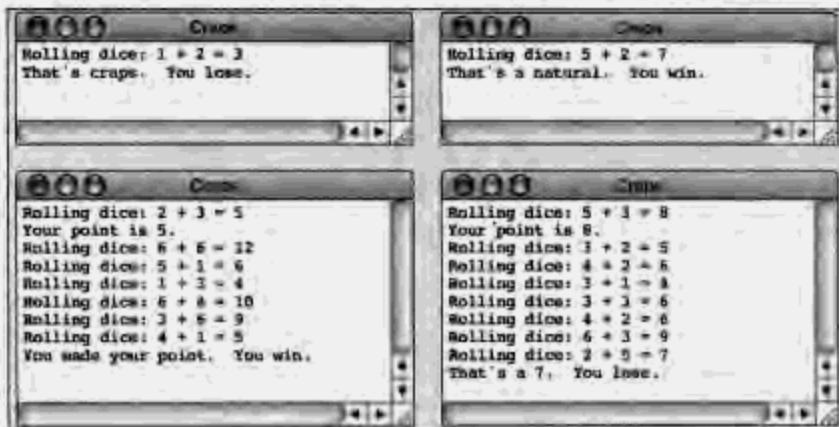


图 6-3 说明 Craps 游戏可能结果的样本运行

Craps.java 程序为使用随机数提供了常见编程模式。通常，在程序结尾使用下面的行声明和初始化随机数生成器：

```
private RandomGenerator rgen = RandomGenerator.getInstance();
```

利用这样的生成器，就可以在 rollDice 方法中使用它来创建一系列随机值，如下所示：

```
int d1 = rgen.nextInt(1, 6);
int d2 = rgen.nextInt(1, 6);
```

每个声明都使用随机数生成器 rgen，获得介于 1~6 之间(包括 1 和 6)新的随机整数。

6.1.3 随机数种子的作用

图 6-1 中所示的 RandomGenerator 类中大多数方法返回特定类型的随机值，可能是整数、双精度数，特定概率选择的 Boolean 值或图形程序中使用的随机颜色。除了 getInstance 之外，另一个不适合该架构的方法是列表末尾的 setSeed 方法。要理解此方法可以做什么，有必要稍微具体地考察一下代表随机数生成器的黑盒子内部发生的情况。

假设第 2 章中将对象定义为概念上集中的实体，该实体封装行为和状态，理解随机数生成器对象的工作原理必须考虑这两个方面：第一，对象的行为由它响应的消息所定义；第二，随机数生成器对象必须维护内部状态，以便每次调用一种方法时可以生成新的随机值。要实现典型的伪随机数生成器，以便它从给定开始点产生一系列看起来似乎随机的值。开始点称为种子。每次调用一种方法都会产生随机值，生成器读取当前的种子，应用数学函数生成随机序列中的下一个值，将结果作为种子存储到内存。接着，方法将新种子转换为适合方法调用的特定类型和特定范围的值。例如，如果调用

```
rgen.nextInt(1, 6);
```

方法将当前随机种子从其作为 long 的内部值转换成介于 1~6 之间(包括 1 和 6)的 int 值。

由于每次对随机数生成器的调用都会更新种子，所以对生成器内方法的一系列调用会产生一系列不同的随机值。如果不知道序列的起始点，也就不可能预见随机数模式。Java 将初始种子默认设置为内部系统时钟上注册的时间。因此，每次运行程序值的序列都会改变。然而，可以用 `setSeed` 方法给随机序列指定具体的起始点。如果在程序开始时这样做，则所谓的随机值每次都一样。

从其字面上看，对程序而言，每次生成相同随机序列的想法有时可能有用，但是似乎有点疯狂。如果玩计算机游戏，最不愿看到的事情就是每次的行为都一样。这样的游戏完全可以预见结果，很快就会变得乏味。即使这样，仍然有人希望——至少是暂时——在计算机游戏环境中设置专用随机种子，一个很重要的理由是确定的程序通常更容易调试。

为了说明这一原则，假设编写了一段程序，用来玩一个复杂游戏，如 Monopoly。通常和新写的程序一样，该程序有一些故障。在复杂程序中，故障相对比较模糊，因为它们只在很少的地方出现。假设您正在玩游戏，发现程序行为有些古怪，但没有对所有相关征兆引起足够警觉。您想再次运行程序，进行仔细观察。

然而，如果程序以一种非确定的方式运行，程序第二次运行时产生的行为可能与第一次运行时不一样。第一次出现的故障第二次可能不会出现。一般来说，如果程序以随机方式运行，则很难复制导致程序故障的条件。另一方面，如果程序确定地运行，它每次运行时都会做相同的事情。这种行为让您能够重新找到问题，这正是在调试阶段所需要的。

在调试过程中可以强加确定的行为，这是 `RandomGenerator` 类包括 `setSeed` 方法的主要原因。例如，在程序开发的调试阶段，程序开始时可以包括调用

```
rgen.setSeed(1);
```

程序每次运行时都会重复相同的一组值。如果将种子设置为其他值，程序将产生不同序列的随机值，但仍然会确定地运行。在发布 Monopoly 程序之前，当然可以删除该语句，但是要让一切正常，它就很有帮助。

6.2 javadoc 文档系统

图 6-1 中 `RandomGenerator` 类的方法概述对于许多应用程序而言已经足够了，但是它并不是该类的完整描述。首先，`RandomGenerator` 类也包括一些专门方法，初学者可能不会使用这些方法，但是对于熟练的用户而言，它们非常重要。其次，图 6-1 中的描述仅限于方法头和一句话小结。如果需要了解其中某个方法的更多细节，就必须到其他地方寻找。在现代面向对象语言开发之前的早期计算中，这种附加信息的最终来源是实现 `RandomGenerator` 类的代码。然而，查看代码不是最佳选择。如果只对使用类感兴趣，就不太可能关注该类的实际实现方式。代码将实现方式带到前台，在那里，过多的细节很容易妨碍用户获得真正要知道的东西。因而必须有一个更好的方法。

Java 采用一种方法将普通文档从基本实现的细节中分离出来，在讨论这种方法之前，有必要介绍两个术语，这两个术语正确表达了实现类的编程人员和使用类的编程人员之间的不同。很显然，实现类的编程人员称为实现者。因为词语“用户”指运行程序的人而不是编写程序的人，所以计算机科学家用术语“客户”表示利用实现者提供的工具的人员。作为客户，不需要知道有关类如何运行的细节，而是要知道它包含什么公有方法。作为实现者，任务是尽可能地对客户隐藏实现的细节，只发布客户最需要的信息。

在 Java 中，客户不希望看到代码，理解如何使用特定的类。相反，客户通常做的是参考类基于 Web 的文档，它由称为 javadoc 的工具自动产生。javadoc 系统是 Sun 公司标准 Java 开发工具箱的一部分，读取程序包代码，为它包含的每个类创建文档。

图 6-4 显示了 RandomGenerator 类的 javadoc 页面。javadoc 描述中的许多词语是超链接，它链接到其他页面或本页面上的特定条目。这些超链接由 javadoc 系统自动产生，它让浏览文档变得更加容易。

The screenshot shows the javadoc page for the `RandomGenerator` class. At the top, there's a navigation bar with links for Overview, Package, Student, Complete Tree, Index, and Help. Below the navigation bar, there are tabs for CLASS, FIELD, CONSTRUCTOR, and METHOD. The main content area starts with the package declaration `java.util`, followed by the class name `RandomGenerator`. It includes the class hierarchy: `Object` → `Random` → `RandomGenerator`. The class definition is shown as:

```
public class RandomGenerator extends Random
```

A note below the class definition says: "This class implements a simple random number generator that allows clients to generate pseudorandom integers, doubles, booleans, and colors. To use it, the first step is to declare an instance variable to hold the random generator as follows:"

```
private RandomGenerator rgen = RandomGenerator.getInstance();
```

Below this, there's a note about the default seed: "By default, this `RandomGenerator` object is initialized to begin at an unpredictable point in a pseudorandom sequence. During debugging, it is often useful to set the internal seed for the random generator explicitly so that it always returns the same sequence. To do so, you need to invoke the `setSeed` method."

Another note explains shared instances: "The `RandomGenerator` object returned by `getInstance` is shared across all classes in an application. Using this shared instance of the generator is preferable to allocating new instances of `RandomGenerator`. If you create several random generators in succession, they will typically generate the same sequence of values."

Constructor Summary

<code>RandomGenerator()</code>	Creates a new random generator.
--------------------------------	---------------------------------

Method Summary

<code>RandomGenerator getInstance()</code>	This method returns a <code>RandomGenerator</code> instance that can be shared among several classes.
<code>boolean nextBoolean(double p)</code>	Returns a random boolean value with the specified probability.
<code>Color nextColor()</code>	Returns a random opaque <code>Color</code> whose components are chosen uniformly in the 0-255 range.
<code>double nextDouble(double low, double high)</code>	Returns the next random real number in the specified range.
<code>int nextInt(int low, int high)</code>	Returns the next random integer in the specified range.

Inherited Method Summary

<code>boolean nextBoolean()</code>	Returns a random boolean that is true 50 percent of the time.
<code>double nextDouble()</code>	Returns a random double d in the range $0 \leq d < 1$.
<code>int nextInt(int n)</code>	Returns a random integer k in the range $0 \leq k < n$.
<code>void setSeed(long seed)</code>	Sets a new starting point for the random generator sequence.

图 6-4 RandomGenerator 类的 javadoc 页面

Constructor Detail**public RandomGenerator()**

Creates a new random generator. Most clients will not use the constructor directly but will instead call `getInstance` to obtain a `RandomGenerator` object that is shared by all classes in the application.

Usage: `RandomGenerator rgen = new RandomGenerator();`

Method Detail**public static RandomGenerator getInstance()**

This method returns a `RandomGenerator` instance that can be shared among several classes.

Usage: `RandomGenerator rgen = RandomGenerator.getInstance();`

Returns: A shared `RandomGenerator` object

public boolean nextBoolean(double p)

Returns a random boolean value with the specified probability. You can use this method to simulate an event that occurs with a particular probability. For example, you could simulate the result of tossing a coin like this:

```
String coinflip = rgen.nextBoolean(0.5) ? "HEADS" : "TAILS";
```

Usage: `if (rgen.nextBoolean(p)) ...`

Parameters: `p` A Value between 0 (impossible) and 1 (certain) indicating the probability

Returns: The value true with probability `p`

public Color nextColor()

Returns a random opaque `Color` whose components are chosen uniformly in the 0-255 range.

Usage: `Color color = rgen.nextColor();`

Returns: A random opaque `color`

public double nextDouble(double low, double high)

Returns the next random real number in the specified range. The resulting value is always at least `low` but always strictly less than `high`. You can use this method to generate continuous random values. For example, you can set the variables `x` and `y` to specify a random point inside the unit square as follows:

```
double x = rgen.nextDouble(0.0, 1.0);
double y = rgen.nextDouble(0.0, 1.0);
```

Usage: `double d = rgen.nextDouble(low, high);`

Parameters: `low` The low end of the range

`high` The high end of the range

Returns: A random double valued in the range `low ≤ d < high`

public int nextInt(int low, int high)

Returns the next random integer in the specified range. For example, you can generate the roll of a six-sided die by calling

```
rgen.nextInt(1, 6);
```

or a random decimal digit by calling

```
rgen.nextInt(0, 9);
```

Usage: `int k = rgen.nextInt(low, high);`

Parameters: `low` The low end of the range

`high` The high end of the range

Returns: The next random `int` between `low` and `high`, inclusive

图 6-4 (续)

如果看看图 6-4 中 javadoc 描述的开头, 就会发现 `acm.util` 程序包里的 `RandomGenerator` 类是 `java.util` 程序包中 `Random` 类的子类。和所有子类一样, `RandomGenerator` 继承了其超类的方法。图 6-4 中的方法小结让这种继承更加明显。虽然有些方法(如 `nextColor`)由 `RandomGenerator` 类本身定义, 但是其他方法——例如简单的 50% 时间返回 `true` 的 `nextBoolean` 方法——却继承了

Random。在很多情况下，这两类哪个真正实现特定方法根本不重要。作为 RandomGenerator 类的客户，重要的是已经准备好访问全部方法了。

Random 和 RandomGenerator 类是面向对象编程中的一种常见情况。这种情况下，一个类提供了一些但不是所有您想要的功能。定义子类能够扩充这些性能，因为新的类继承了原来类的方法。可以重复此过程，以便子类继承整个超类的行为。因为每个新的类都建立在其前身提供的架构上，这种层次结构称为分层抽象化。

6.3 定义自己的类

至少在某种程度上，在 Java 中定义自己的类这种想法应该不是特别难实现。毕竟，从第 2 章起用户就已经定义自己的类了。每个程序都是扩充某个 Program 子类的类定义，这说明已经有足够的示例来理解类定义的基本结构。本节以及接下来几节中示例的目的都是说明 Java 中类定义运行的细节。

6.3.1 类定义的结构

类定义的语法结构如下面的语法框所示。在 Java 中，客户可以使用的每个类——与在程序包语境中定义相反——必须标记关键字 public。而且，每个公有类定义都应该在名为 *name.java* 的单独源文件里，其中 *name* 是新类的名称。

extends 规范指明超类，类从它开始扩充。这种规范是随意的；如果没有它，新类就成为了嵌入 Object 类的直接子类，Object 类是 Java 类层次结构的根。如果从某个类开始，顺着以超类开始的链，继续通过超类，这些链通常结束于 Object 类。因此，每个 Java 类——直接或间接——都是 Object 的子类。而且，Java 中的所有对象都是 Object 类的实例。

公有类定义的语法

```
public class name extends super {  
    class body  
}
```

其中：

name 是新类的名称；

super 是超类的名称；

class body 是一系列定义，通常包括构造函数、方法、命名常量和实例变量。

类的主体由一系列不同类型的定义组成，统称为条目。在目前为止出现的所有类中，这些定义仅限于方法和命名常量。另外，类通常包含构造函数，用于指定如何创建类的新实例；还包含以及实例变量，用于维护对象的状态。构造函数的定义方法与方法类似，但有两点不同：其一，构造函数的名称总是与类的名称相同；其二，构造函数不指定结果类型。实例变量声明除了它们出现在类的最上一级而不是方法定义内之外，看起来与其他变量声明一样。实例变量与方法内的变量定义十分类似，除了它们整体出现在类的某一级之外。

6.3.2 控制条目的可见性

类定义中的条目使用关键字标记，表示什么类可以访问这些条目。在 Java 中，可以使用下面 3 个关键字之一来控制对类中单个条目的访问。

- **public:** 程序里的每个类都可见。
- **private:** 只有定义它的类里可见。
- **protected:** 只有在相同程序包里的超类和类可见。

也有第 4 种选择，就是完全省略关键字。没有关键字的条目称为私有程序包，只有相同程序包里的类可见，其他程序包里的超类不可见。

虽然受保护的和私有程序包有其用途，但是本书示例只使用 **public** 和 **private** 选项。指定为 **public** 的条目称为导出。作为通用规则，应该指定条目为 **private**，除非有强制理由要求导出它们。

6.3.3 封装

类最重要的功能之一是将相关数据片结合为一个单元。在现实世界处理数据时，单个数据值通常结合在一起形成更大的概念上的结构。例如，在 `acm.graphics` 程序包中用来表示坐标位置的 `x` 和 `y` 坐标值很少单独出现。几乎所有情况下，都是同时提供 `x` 和 `y` 的值。从概念上说，两个值一起表示图形空间上的点。在第 9 章会发现，`acm.graphics` 程序包包括名为 `GPoint` 的类，它将 `x` 和 `y` 坐标结合为一种结构，这种结构可以作为单个单元使用。例如，通过调用 `getLocation(它返回 GPoint 对象)`，可以确定任何图形对象的位置。使用 `GPoint` 通常比使用分别表示坐标的两个单独变量要方便。

将几个数据值结合为单个对象是过程的一部分。计算机科学家称之为封装。打个比喻说，用这种方法集合数据的过程与将对象放入密封舱里类似。对象仍然在密封舱里，但是现在可以将它当成一个整体使用。例如，现实中，可以拿着密封舱，可以将它运送给另外一个人，他可以打开密封舱，恢复内容。在编程领域，情况类似，因为可以将单个数据值结合为新的封装对象，然后将此对象作为单个值传递给方法。如有必要，方法可以打开对象恢复原始数据值，但也可以作为整体使用封装对象。

然而，词语“封装”在计算机科学中有其他含义。虽然将数据值结合为集合单元是过程的重要部分，但是封装的概念也说明内容在某些方面受到保护。正如太空舱保护宇航员，使其与外部空间的危险隔开一样，设计良好的封装对象通常也会保护其元素不被程序其他部分任意访问。本章剩余部分的示例强调了封装过程的这个方面，以及 Java 中如何写类定义的更多常见问题。

6.4 表示学生信息

要理解类定义如何运行，最好的方法是考察示例。假设大学里的管理计算部门请您实现一个新的基于 Java 的系统，记录哪些学生可以毕业。这种实现方式的中心元素是称为 `Student` 的类，它记录作出决定所需的信息。虽然 `Student` 类的真正实现肯定包括更多细节，但这里只要求包括如下数据：

- 学生姓名。
- 学生的 6 位 ID 号。
- 学生取得的学分(该大学可能需要包括小数点后面的数字,以记录 0.5 和 0.25 学分的课程)。
- 用来表示学生是否付清了所有学费的标记。

只有取得了必需的学分而且没有欠费的同学才符合毕业条件。

6.4.1 声明实例变量

开始设计 Student 类的结构时,首先要回答的一个问题是需要用什么实例变量来存储每个对象的状态。依据本节介绍的规则,每个 Student 对象必须包含 4 个信息片段。而且,可以使用 Java 的预定义类型表示每个数据值。可以使用 String 表示学生姓名, int 表示 ID 号, double 表示学分数, boolean 表示是否付清所有学费。这样,使用 4 个实例变量就可以记录 Student 对象的状态:

```
private String studentName;
private int studentID;
private double creditsEarned;
private boolean paidUp;
```

除了两点不同之外,这些声明与局部变量的声明类似。首先,它们在文件中出现的位置不同。局部变量在方法体内定义,只能在该方法内使用。实例变量在方法之外定义,类里定义的所有方法都可使用它。其次,实例变量声明通常包括关键字来表示范围,该变量对此范围内其他类里的方法是可见的。虽然 Java 在 6.3.2 小节提供了 4 种可见性选项,但是使用关键字 private 声明所有实例变量很有好处。这样做的主要原因是封装既是保护数据的策略,也是集合数据的策略。通过禁止访问内部实例变量,类设计师可以保护类的完整性。例如,允许 Student 类的客户独立改变学生姓名和 ID 号就很危险。许多大学里,学生在校整个期间都使用相同的 ID 号。如果总记录维护系统的其他部分能够给现有同学赋予一个新 ID,那么就很难维护存储信息的完整性。将这些实例变量指定为私有,就可以让系统内的其他类不能改变这些值,而不必依赖 Student 类设计师提供的方法。

实际上,私有实例变量的好处非常有说服力,本书中所有变量——甚至是 ACM 库的实现工具中使用的实例变量——都使用关键字 private 来标记。虽然偶尔有使用公有实例变量,但如果要维护只使用私有实例变量的约定,就要养成更好的编程习惯。

常见错误

声明类里的实例变量时,通常最好将这些变量声明为 private,将实例变量限制到它们出现的类的方法里,这样可以提高信息的安全性。

6.4.2 完成类定义

然而,实例变量只能表示类定义的一部分。正如 6.3 节所述,类定义通常包括构造函数、方法、命名常量定义及实例变量声明。Student 类至少包含了各个条目类型的示例,其完整定义如图 6-5 所示。

```
/*
 * The Student class keeps track of the following pieces of data
 * about a student: the student's name, ID number, the number of
 * credits the student has earned toward graduation, and whether
 * the student is paid up with respect to university bills.
 * All of this information is entirely private to the class.
 * Clients can obtain this information only by using the various
 * methods defined by the class.
 */

public class Student {

    /**
     * Creates a new Student object with the specified name and ID.
     * @param name The student's name as a String
     * @param id The student's ID number as an int
     */
    public Student(String name, int id) {
        studentName = name;
        studentID = id;
    }

    /**
     * Gets the name of this student.
     * @return The name of this student
     */
    public String getName() {
        return studentName;
    }

    /**
     * Gets the ID number of this student.
     * @return The ID number of this student
     */
    public int getID() {
        return studentID;
    }

    /**
     * Sets the number of credits earned.
     * @param credits The new number of credits earned
     */
    public void setCredits(double credits) {
        creditsEarned = credits;
    }
}
```

图 6-5 学生信息的类定义

```

/**
 * Gets the number of credits earned.
 * @return The number of credits this student has earned
 */
public double getCredits() {
    return creditsEarned;
}

/**
 * Sets whether the student is paid up.
 * @param flag The value true or false indicating paid-up status
 */
public void setPaidUp(boolean flag) {
    paidUp = flag;
}

/**
 * Returns whether the student is paid up.
 * @return Whether the student is paid up
 */
public boolean isPaidUp() {
    return paidUp;
}

/**
 * Creates a string identifying this student.
 * @return The string used to display this student
 */
public String toString() {
    return studentName + " (" + studentID + ")";
}

/* Public constants */

/** The number of credits required for graduation */
public static final double CREDITS_TO_GRADUATE = 32.0;

/* Private instance variables */

private String studentName; /* The student's name */
private int studentID; /* The student's ID number */
private double creditsEarned; /* The number of credits earned */
private boolean paidUp; /* Whether student is paid up */
}

```

图 6-5 (续)

如果只考察图 6-5 的长度，很容易理解 Student 类——毕竟，它是第一个占用一个页面以上的编程示例——在某种程度上比前面几章里的示例要复杂。但事实胜于雄辩，如果仔细阅读代码，就会发现构造函数只有两行，其他每个方法的主体只有一行。大多数空间被注释占据。更严重的是，这些注释文本看起来大量重复。例如，方法 isPaidUp 的注释如下所示：

```

/**
 * Returns whether the student is paid up.
 * @return Whether the student is paid up
 */

```

很显然，有人会想，要理解其意思有第一行就足够了。

6.4.3 编写 javadoc 注释

实际上，图 6-5 中代码例证了一种编码风格，这种风格在 Java 开发中非常有用。虽然改变不十分明显，但是仍然可以发现 Student 类里的公有方法和常量的注释以字符/**开始，而不是用来标记注释的更传统的/*。这样的注释对于人类读者以及对于本章前面描述的 javadoc 文档而言，都是特意安排的。Java 编译器和以前一样会忽略注释文本。然而，如果通过有标准 Java 工具的 javadoc 应用程序运行该程序，它会自动创建描述类的 Web 页面及图 6-4 中所示的行。javadoc 应用程序同时读取注释和代码来创建类的标准化描述。

方法 javadoc 条目的概述部分显示了方法的标题行及注释的第一句。javadoc 条目的细节部分也显示了标题行和注释的全部文本。另外，细节部分包括方法接受的每个参数及其所产生的结果的描述。这些描述减少了 javadoc 注释中出现的@param 和@result 标记。养成写代码时包括这些标记的习惯很有必要，因为这样做更容易生成自动文档，同时这些标记为阅读代码的读者提供了有用信息。

6.4.4 写构造函数

Student 类的构造函数如下：

```
public Student(String name, int id) {
    studentName = name;
    studentID = id;
}
```

正如 6.3 节所述，构造函数除了没有结果类型外，其他方面与方法定义的形式相同。客户通过关键字 new、类的名称及与形参列表匹配的一列实参，可以调用构造函数。使用如下所示的声明，程序的另一部分可以创建学生条目：

```
Student topStudent = new Student("Hermione Granger", 314159);
```

该语句声明名为 topStudent 的变量，给它指定调用有指定实参的 Student 构造函数产生的结果。构造函数创建新的 Student 对象，将形参值分别给实例变量 studentName 和 studentID 赋值。这样，studentName 实例变量被分配字符串“Hermione Granger”，studentID 变量的值为 314159。剩余的实例变量取各自适合类型的默认值。creditsEarned 变量初始化为 0，paidUp 变量初始化为 false。

在 Student 构造函数的实现方式中，参数使用的标识符名称与实例变量名称不同。有些 Java 编程人员喜欢使用相同的标识符名称，因为参数名称与实例变量名称取相同的值。然而，不可能像下面这样编码构造函数：

```
public Student(String studentName, int studentID) {
    studentName = studentName;
    studentID = studentID;
}
```

Java 编译器转换该代码时，会将每条赋值语句的两端解释为参数名称。这样，构造函数的这种不正确形式只能取 studentName 变量的值，并将其赋值给该变量本身。如果要让参数和实



例变量使用相同的名称，需要用某种方式区分两者。在 Java 中，进行区分的最简单方法是使用关键字 `this`，它表示当前对象。所以，只要将构造函数改为如下所示，就可以更正它的故障实现：

```
public Student(String studentName, int studentID){  
    this.studentName = studentName;  
    this.studentID = studentID;  
}
```

修改之后，Java 编译器就会知道赋值语句左边的变量是属于该对象的实例变量。

让参数和实例变量使用相同名称，这一策略存在的唯一问题是当参数名称匹配实例变量名称时，有些 Java 编译器会发出报警消息。要避免发出这种报警，文本和 ACM 库中的代码总是使用不同的名称。

6.4.5 getters and setters

Student 类里的大多数方法用来检索或更改实例变量的值。例如，可以调用

```
String name = topStudent.getName();
```

来获得存储在变量 `topStudent` 里的学生姓名。或者调用

```
topStudent.setCredits(97);
```

将 `topStudent` 的学分设置为 97。

能够检索实例变量值或为其赋某个新值的方法分别称为 `getter` 和 `setter`(Java 的正式写法中，这样的方法称为存取器和增变器方法，但是如果使用这些充满想象的术语，几乎没有人能够理解)。按惯例，`getter` 以前缀 `get` 开始，`setter` 以前缀 `set` 开始，如 `Student` 类的实现方式示例所示。这种命名约定的唯一例外(示例中也有)是返回 `boolean` 类型值的 `getter`，第 5 章将这种类型定义为断言方法。对于检索 `boolean` 值的方法而言，`getter` 通常以前缀 `is` 开始。因此，即使调用

```
topStudent.setPaidUp(true);
```

设置显示 `topStudent` 已付清所有费用的标记，测试条件的代码也会调用

```
if (topStudent.isPaidUp()) ~ ~ .
```

毕竟，`boolean` 值用来表示回答“是—否”问题，命名约定让程序非常易读。

`Student` 类的实现方式不包括学生姓名或学生 ID 的 `setter`，注意这一点很重要。只有构造函数可以设置这些值，此后不能改变。这种设计不可能打破学生姓名和学生 ID 号之间的联系，也不希望改变这种联系。

6.4.6 `toString` 方法

`Student` 类的实现方式中唯一不是 `getter` 或 `setter` 的方法是 `toString` 方法，它负责将对象的值转换为人类读者可以理解的字符串。`toString` 方法在 Java 中有特殊作用。Java 要将对象转换为字符串——通常在使用`+`运算符指定串联的表达式中出现，调用对象的 `toString` 方法可以实现。例如(假设变量 `topStudent` 仍然初始化为表示 `Hermione Granger`，如本章前面所述)，执行行

```
println("Top student = " + topStudent);
```

将产生如图 6-6 所示的输出。

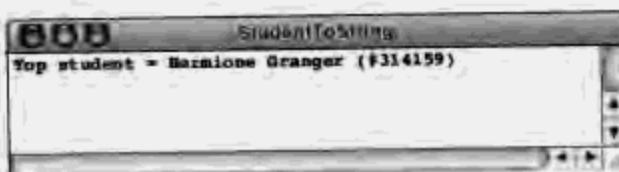


图 6-6 StudentToString 的输出

如果不考虑 `toString` 方法的实现方式, Java 会使用 `Object` 类中 `toString` 方法的默认定义。这种情况下结果如图 6-7 所示。

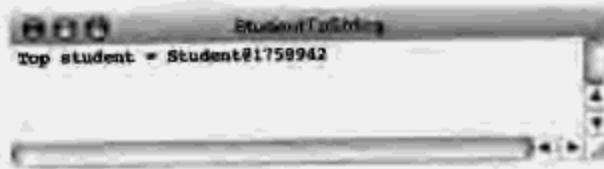


图 6-7 StudentToString 的输出

这样提供的信息就很少。输出中的数字字符串——不同计算机结果可能不同——与 `Hermione` 学生 ID 号无关, 而是一个内部代码(通过此代码 Java 可以确定对象)。

给 `Student` 类中的 `toString` 提供定义, 意思就是这种新定义优先于 `Object` 类中的原始定义。在面向对象编程中, 该过程称为重写。6.6.3 小节将详细讨论重写方法机制。

6.4.7 定义类中的命名常量

许多类导出命名常量定义及其他条目类型。因为客户不能改变命名常量, 所以让它们变成公用就几乎没有危险。这样做意味着导出类的客户可以在自己的代码里使用该常量。为此, 客户可以写类名称、点(.)和常量名称。

导出常量最容易理解的一种示例是 `PI`, 它在 `java.lang` 程序包的 `Math` 类里定义如下:

```
public static final double PI = 3.14159265358979323846;
```

因为其定义标记为 `public`, 通过 `Math.PI`, 每个类都可以访问数学值 π 。其他包括在 `Color` 类里的常量, 如 `Color.GREEN` 和 `Color.MAGENTA`, 将对象作为它们的值而不是原始类型。`Student` 类的定义导出命名常量, 用来指定需要毕业的学生的学分:

```
public static final double CREDITS_TO_GRADUATE = 32.0;
```

使用命名常量(包括此定义)有两个好处。第一, 使用命名常量的代码容易阅读, 假设选择常量名称让其意思显而易见。第二, 代码易于维护, 因为可以在程序内的单个地方改变毕业所需的学分。

6.4.8 使用 `Student` 类

虽然很难编写包括 `Student` 对象的令人感兴趣的应用程序(除非知道如何表示整个学生列表), 但是仍然可以知道客户端编程人员如何编写利用 `Student` 类的方法。下面的方法使用

Student 对象，确定学生是否满足毕业条件：

```
private boolean isEligibleToGraduate(Student student) {
    return student.getCredits() >= Student.CREDITS_TO_GRADUATE &&
        student.isPaidUp();
}
```

6.5 有理数

虽然前面定义的 Student 类说明了类的基本机制，但深入理解这一问题还需要通过更复杂的示例。要理解特定示例的重要性，假设您回到小学，要做下面的分数加法。

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 1$$

运用基本算术——甚至凭直觉——就可以知道从数学上讲正确答案是 1，但在计算机上使用双精度算法就很难得出该答案。例如，考察下面的 run 方法：

```
public void run() {
    double sum = 1.0/2.0 + 1.0/3.0 + 1.0/6.0;
    println("1.0/2.0 + 1.0/3.0 + 1.0/6.0 = " + sum);
}
```

如果执行包含此 run 方法的 Java 程序，得到的输出结果如图 6-8 所示。

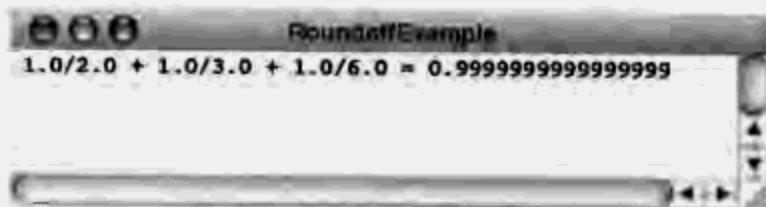


图 6-8 RoundoffExample 的输出

对于实际应用而言，虽然结果 0.999999999999999 很精确，但是如果在课堂上给出这个答案，您的小学老师肯定会怀疑您使用了计算设备。

问题是计算机内用来存储数字的内存单元存储能力有限，因此限制了可以提供的精度。在双精度算法的限制内， $\frac{1}{2} + \frac{1}{3} + \frac{1}{6}$ 等于 0.9999999999999999，虽然在数学上这种结果不太令人满意。要得到准确答案，需要跳出双精度数的范围，进入有理数领域，这些数的值可以表示为两个整数的商。有理数作为数学概念很容易理解，它们也有自己的运算规则，如图 6-9 所示。遗憾的是，Java 的预定义类型不包括有理数。要在 Java 中使用有理数算法，需要定义新的类来表示它们。

加 $\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$	乘 $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$
减 $\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$	除 $\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

图 6-9 有理数运算规则

如果能够定义 Rational 类，就可以在 Java 程序中使用它实现数学上的精确计算。例如，下面的 run 方法可以实现前面所示的分数计算。

```
public void run() {
    Rational a = new Rational(1, 2);
    Rational b = new Rational(1, 3);
    Rational c = new Rational(1, 6);
    Rational sum = a.add(b).add(c);
    println(a + " + " + b + " + " + c + " = " + sum);
}
```

该代码声明了 3 个变量——a、b 和 c，并将它们初始化为与计算中分数相关的有理值。下一行声明变量 sum，并将它初始化为 a、b、c 的和。此行中唯一可能产生的混淆是 Rational 类的加法必须使用 add 方法表示，因为只为原始类型定义+运算符。最后一行打印 3 个值及它们的和。程序输出结果如图 6-10 所示(假设 Rational 类可以正常运行)：



图 6-10 RationalTest 的输出

图 6-11 显示 Rational 类的简单实现方式，它执行 4 种基本算术运算：加、减、乘、除。许多类定义很直接，但代码确实解释了一些新特征，如下所示。

- 乘法构造函数。图 6-11 中的 Rational 类定义了 3 个不同的构造函数。调用 new Rational() 创建新的有理数，其值为 0，内部表示为分数 0/1。调用 new Rational(n) 创建新的等于整数 n 的有理数，表示为分数 n/1。最后，调用 new Rational(x, y) 创建等于分数 x/y 的有理数。通过查看客户提供的参数，Java 就知道调用哪个构造函数。在相同名称下有多种参数形式的构造函数和方法称为重载构造函数或方法。
- 关键字 this 的新用法。从本章前面 Student 类的讨论中可知，Java 定义了关键字 this 来表示当前对象。Rational 类使用关键字 this 的方法有两种。第一种是在构造函数中，它使用 this 将对象的创建委托给含有不同参数的其他构造函数。第二种出现在实现算法的方法中，其中 this 用来强调 num 和 den 实例变量都在当前有理数里。例如，考察 multiply 方法的定义：

```
public Rational multiply(Rational r) {
    return new Rational(this.num * r.num, this.den * r.den);
}

/**
 * The Rational class is used to represent rational numbers, which
 * are defined to be the quotient of two integers.
 */
public class Rational {

    /** Creates a new Rational initialized to zero. */
    public Rational() {
        this(0);
    }

    /**
     * Creates a new Rational from the integer argument.
     * @param n The initial value
     */
    public Rational(int n) {
        this(n, 1);
    }

    /**
     * Creates a new Rational with the value x / y.
     * @param x The numerator of the rational number
     * @param y The denominator of the rational number
     */
    public Rational(int x, int y) {
        int g = gcd(Math.abs(x), Math.abs(y));
        num = x / g;
        den = Math.abs(y) / g;
        if (y < 0) num = -num;
    }

    /**
     * Adds the rational number r to this one and returns the sum.
     * @param r The rational number to be added
     * @return The sum of the current number and r
     */
    public Rational add(Rational r) {
        return new Rational(this.num * r.den + r.num * this.den,
                           this.den * r.den);
    }

    /**
     * Subtracts the rational number r from this one.
     * @param r The rational number to be subtracted
     * @return The result of subtracting r from the current number
     */
    public Rational subtract(Rational r) {
        return new Rational(this.num * r.den - r.num * this.den,
                           this.den * r.den);
    }
}
```

图 6-11 表示有理数的类定义

```

/**
 * Multiplies this number by the rational number r.
 * @param r The rational number used as a multiplier
 * @return The result of multiplying the current number by r
 */
public Rational multiply(Rational r) {
    return new Rational(this.num * r.num, this.den * r.den);
}

/**
 * Divides this number by the rational number r.
 * @param r The rational number used as a divisor
 * @return The result of dividing the current number by r
 */
public Rational divide(Rational r) {
    return new Rational(this.num * r.den, this.den * r.num);
}

/**
 * Creates a string representation of this rational number.
 * @return The string representation of this rational number
 */
public String toString() {
    if (den == 1) {
        return "" + num;
    } else {
        return num + "/" + den;
    }
}

/**
 * Calculates the greatest common divisor using Euclid's algorithm.
 * @param x First integer
 * @param y Second integer
 * @return The greatest common divisor of x and y
 */
private int gcd(int x, int y) {
    int r = x % y;
    while (r != 0) {
        x = y;
        y = r;
        r = x % y;
    }
    return y;
}

/* Private instance variables */
private int num;      /* The numerator of this Rational */
private int den;      /* The denominator of this Rational */
}

```

图 6-11 (续)

方法生成新的 Rational 值，其分子是对象的分子和 r 的分子的乘积，其分母同样是这两个对象分母的乘积。

- 公有和私有方法同在一个类中。Rational 类里的公有方法是那些类的客户需要调用的方法。它包括构造函数，实现算术运算的方法，以及前面描述的 toString 方法。然而，这

也包括自己实现方式中使用的私有方法，该实现方式不需要导出到客户。这种方法是第5章的gcd函数，它实现欧几里得算法，这里用来将分数化成最简形式。通用规则是，应该尽可能将方法标记为private。

- 将整数转换为字符串。toString方法的实现工具以两种方式将Rational数的值转换为字符串。大多数情况下，通过给表达式赋值，用分子分母间的斜线将分子和分母连接起来创建字符串：

```
num + "/" + den
```

在分母是1的特殊情况下，Rational数实际上是整数，toString方法会省去分子和斜线。然而，不可能只返回num的值，因为它是int类型而不是string类型。虽然还有其他方法可以将整数转换为字符串，大多数Java编程人员采用简写形式：

```
" " + num
```

它用不包括字符的字符串连接num的值。

Rational类明显支持实例变量为私有。对于有些客户而言，通过选择对象内的r.num和r.den变量，能够获得有理数r的分子和分母，这样很有帮助。将这些变量设置为公有会影响类的完整性。Rational类的实现确保下面的属性。

- 有理数的分母总是正数。如果分母是负数，构造函数将符号转换给分子，让分子值变成负数。
- 用分子和分母分别除以其最大公约数，将用分子和分母表示的分数化为最简式。这种设计确保有理数总是显示为最简式。

在整个代码主体内都确保正确的属性称为不变量。如果类可以防止客户有意或无意地破坏这些不变量，那么就有可能维护它们。如果实例变量是公有的，任何客户都可以将r.den设置为0，就会打乱有理数r的第一个不变量，或扰乱所有数的第二个不变量。

图6-11中Rational类的实现方式实际上进一步保护类的不变量。将实例变量声明为私有，可确保客户在赋值语句中不能给它们赋新的值。如果仔细考察Rational类里的方法——与构造函数相反，就会发现没有方法可以将值赋给实例变量。构造函数给实例变量赋值以后，在对象的整个生命周期内，这些值都不能改变。客户，甚至是方法调用，都不能改变其内部状态的类称为不可变类。不可变类的类型有几个很有用的属性，在后面几章会经常出现。

6.6 扩展现有类

如果再查看一下本章前面给出的Student和Rational类的定义，就会发现标题行都没有包括extends从句来指定超类。因此，这些类都变成了Java的内置类Object类的直接子类。这种情况有点不同寻常。大多数情况下，定义的新类会扩展现有类，让新类继承超类的行为。新类通常通过定义新方法来扩展行为，也可以通过重新定义现有方法来更改行为。因为扩展类的过程对于Java的面向对象结构至关重要，所以有必要理解新类扩展现有类时会发生什么。

6.6.1 创建类表示实心三角形

第5章已经介绍了如何使用方法来简化创建共享特定属性的图形对象。第5章的主要示

例是方法 `createFilledCircle`, 它创建圆形的 `GOval` 对象, 该对象自动用调用者指定的颜色进行初始化填充。新的类可以扩展现有某个 `GObject` 子类的行为, 通过定义此新类可以实现相同目标。

假设要编写一个图形应用程序, 并且几乎所有要显示的三角形都是实心的而不是只有轮廓。对于这个应用程序, `GRect` 类的行为(默认使用轮廓形式)不是特别有用。当然, 可以用第 5 章 `createFilledCircle` 方法和 `createFilledRect` 方法实现, 但应该选择以一种更面向对象的方式解决问题。`acm.graphics` 程序包提供了一个名为 `GRect` 的类, 它最初显示为三角形轮廓。如果程序包的设计师也定义了名为 `FilledRect` 的类就最好了, 除了自动填充三角形而不是只画轮廓之外, 它与 `GRect` 基本相同。

即使 `acm.graphics` 的设计师没有包括 `FilledRect` 类, 也不会妨碍您定义一个。凭直觉, 新类与现有的 `GRect` 类非常类似, 唯一不同的是初始化。不像 `GRect` 构造函数那样将填充指针设置为 `false`, `FilledRect` 类的构造函数将它设置为 `true`。`GRect` 类里的其他方法保持不变。在 Java 中, 让其他所有方法保持不变这个问题很简单, 只要让 `FilledRect` 成为 `GRect` 的子类即可。`FilledRect` 类会继承 `GRect` 类中所有方法的定义。因此, 类的标题行如下:

```
public class FilledRect extends GRect
```

剩下的唯一一个问题是如何改变构造函数, 让它将必需的 `setFilled` 操作添加到初始化代码中。

过程的第一步是确定要 `FilledRect` 构造函数根据其参数结构看起来是什么样子。`FilledRect` 只是 `GRect` 的一种特殊类型, 问题的答案即来源于此。要创建 `GRect` 对象, 需要提供其左上角的 `x` 坐标和 `y` 坐标以及宽度和高度。`FilledRect` 构造函数也需要相同的信息, 因此其标题行是:

```
public FilledRect(double x, double y, double width, double height)
```

这里, `FilledRect` 构造函数需要调用 `GRect` 构造函数实现的操作来初始化其内部数据结构。这些数据结构表示对象和状态, 存储在实例变量里。在保持良好编程习惯的实践中, 这些实例变量对于 `GRect` 类而言是私有的, 即 `FilledRect` 类不能直接使用它们。`FilledRect` 类能做的唯一一件事就是请求 `GRect` 类实现其标准初始化。因此, `FilledRect` 构造函数必须将 `x`、`y`、`width` 和 `height` 信息传递给 `GRect` 构造函数。

在 Java 中, 通过在首行指定关键字 `super`, 类可以调用其超类的构造函数。在此语境中, `super` 好像是方法调用, 将超类构造函数的参数包含在圆括号内。由于 `FilledRect` 构造函数要使用它接收到的参数调用 `GRect` 构造函数, 因此, 构造函数的第一行如下:

```
super(x, y, width, height);
```

这一行能够影响 `GRect` 数据结构的初始化。`FilledRect` 构造函数可以继续进行自身操作所需的初始化。在这里, 其他所需要的操作就是将三角形的填充状态设置为 `true`。因此, 整个 `FilledRect` 构造函数如下:

```
public FilledRect(double x, double y, double width,
                  double height) {
    super(x, y, width, height);
    setFilled(true);
```

)

假设 FilledRect 类继承了所有 GRect 方法，此时就可以停止了。在新的构造函数之外的类定义的主体内是空的。但也可以添加一些新功能。如果解决第 5 章使用 createFilledCircle 方法的练习，很可能会发现在创建对象的同时设置其颜色会很方便。要同样方便地使用 FilledRect 类，可以添加其他构造函数，用其他参数指定颜色。正如图 6-11 中的 Rational 类所示，可以定义多个构造函数，只要 Java 编译器通过查看参数能够知道需要哪个构造函数即可。如果添加新的构造函数，它用第 5 个参数来表示颜色，当调用者提供参数时，Java 会调用构造函数。

定义新构造函数的方法之一是从原始构造函数里复制代码，然后添加 setColor 行，代码如下：

```
public FilledRect(double x, double y, double width,
                  double height, Color color) {
    super(x, y, width, height);
    setFilled(true);
    setColor(color);
}
```



虽然代码产生了所需的结果，但它不是最佳设计。问题是两个构造函数包含了相同的复制代码。虽然这里复制的数量的确很小，但是单独定义这两个构造函数很容易导致维护出现问题。将来，有人可能想给 FilledRect 类添加其他功能。很可能有些扩充需要添加到构造函数中出现的初始化代码里。在目前情况下，想进行扩充的人必须记住要将这些变化合并到两个构造函数里。复制代码强迫维护人员在多个位置更新代码这种情况，这常常是程序的生命周期内出现问题的根源。

常见错误

要避免在程序的多个部分编写相同的代码。复制代码以通常会产生严重的维护问题，因为这很容易让粗心的维护人员改变代码的某个副本而没有同时更新其他副本。

如果再仔细考察包含颜色参数的新构造函数，会发现它分为两个概念部分，如下面的定义所示，它用 Java 的组合表示，英语中通常称为伪代码。

```
public FilledRect(double x, double y, double width,
                  double height, Color color) {
    Create a filled rectangle object using the first four arguments.
    Set the color of this object as indicated by the last argument.
```

可以重写构造函数，以便它更接近伪代码。首先要做的是调用 FilledRect 类的原始构造函数，然后将对象设置为 color 参数指定的颜色。该方法产生的代码如下所示。

```
public FilledRect(double x, double y, double width,
                  double height, Color color) {
    this(x, y, width, height);
    setColor(color);
```

FilledRect 类的完整代码如图 6-12 所示。使用该类可以更简洁地编写一些图形程序，如下面的 run 方法所示。

```

public void run() {
    double width = getWidth();
    double height = getHeight();
    double stripe = width / 3;
    add(new FilledRect(0, 0, stripe, height, Color.BLUE));
    add(new FilledRect(stripe, 0, stripe, height, Color.WHITE));
    add(new FilledRect(2 * stripe, 0, stripe, height, Color.RED));
}

```

```

/*
 * File: FilledRect.java
 *
 * This file defines a graphical object that appears as
 * a filled rectangle.
 */

import acm.graphics.*;
import java.awt.*;

/**
 * This class is a GObject subclass that is almost identical
 * to GRect except that it starts out filled instead of
 * outlined.
 */
public class FilledRect extends GRect {

    /** Creates a new FilledRect with the specified bounds. */
    public FilledRect(double x, double y, double width, double height) {
        super(x, y, width, height);
        setFilled(true);
    }

    /** Creates a new FilledRect with the specified bounds and color. */
    public FilledRect(double x, double y, double width, double height, Color color) {
        super(x, y, width, height);
        setColor(color);
    }
}

```

图 6-12 FilledRect 的类定义

该方法创建了 3 个垂直条纹，每个条纹的宽度都等于窗口的 1/3，扩充到整个垂直空间。条纹的颜色是蓝色、白色和红色，组成了一幅法国国旗的图片，如图 6-13 所示。

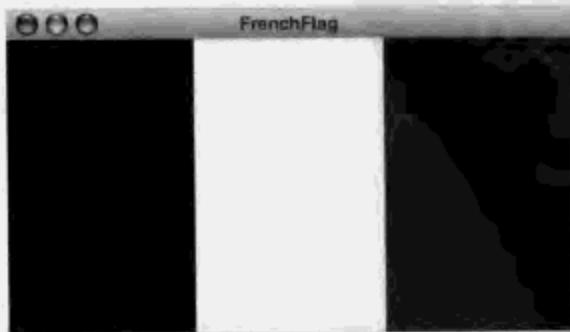


图 6-13 法国国旗图片

6.6.2 继承构造函数的规则

虽然构造函数与方法有很多共同点,但 Java 中的构造函数有时更微妙。这种微妙来自于类形成层次结构。6.6.1 小节定义的 FilledRect 类的实例也是 GRect、GObject 和 Object 的实例,因为这些类是列表中下一个类的子类。要确保新的 FilledRect 对象正确初始化,Java 必须确保在构造过程中调用这些类的构造函数。第一步是通过调用 Object 构造函数初始化与每个 Object 相关的数据。从此,Java 必须调用每个子类级的构造函数,沿着继承链向下,一直到 GObject、GRect 和 FilledRect。FilledRect 构造函数明确调用其超类的构造函数,但本书中大多数编程示例没有这样做。即使遗漏了调用,对 Java 来说,调用超类构造函数来确保维持超类的所有非变量也很重要。

要确保调用超类构造函数,Java 中没有明确包括将 super 和 this 调用作为其首行的构造函数,自动执行,就像构造函数以语句

```
super();
```

开始一样。它调用没有参数的超类构造函数。没有参数的构造函数称为默认构造函数。这样,Java 类里的每个构造函数都可以用下面 3 种方式之一调用超类构造函数:

- 以明确调用 this 开始的类调用该类其他构造函数中的某一个,将确保调用超类构造函数这一责任委托给该构造函数。
- 以明确调用 super 开始的类调用超类里的构造函数,该超类匹配提供的参数列表。
- 不以调用 super 或 this 开始的类调用没有参数的默认超类构造函数。

剩下的唯一问题是理解如何定义超类构造函数。就像前面几章里的类定义没有包括对超类构造函数的明确调用,这些定义也没有包括无参数的构造函数的定义。如果没有合适的构造函数定义能够调用,Java 如何调用超类构造函数呢?问题的答案是,如果类没有定义任何自己的构造函数,Java 自动定义默认构造函数。自动生成的默认构造函数主体就是空的,因此实际上它不进行任何初始化。然而,定义这样的构造函数确实可以确保继承层次结构里的每个类都有可以调用的构造函数。但是要注意,如果类定义包括构造函数,就不再创建默认构造函数。在这种情况下,所有子类的构造函数必须明确调用其超类里的某个构造函数。

6.6.3 继承方法的规则

第一次遇到继承和重写的概念时,很容易搞不清特定对象应该调用方法的那个版本。在 Java 中,规则是所执行的在层次结构中与对象的实际类最接近的方法。因此,假设有一个方法调用了 FilledRect 类的对象,Java 编译器首先在 FilledRect 类中寻找有合适名称和结构的方法。如果定义了方法,编译器使用那个版本。如果没有,编译器会在 GRect 类里寻找,GRect 类是链中的下一个类。编译器以这种方式继续寻找,直到找到合适的定义或发现不存在这样的方法为止。

容易混淆的是,Java 总是依据对象实际上是什么来作决定,而不是依据在程序的特殊位置声明它是什么来作决定。再考察一下本章前面对 Student 类的讨论。在 6.4.6 小节,介绍了调用

```
println("Top student = " + topStudent);
```

会自动调用 Student 类里的 toString 方法,产生容易被人理解的输出,如图 6-14 所示。

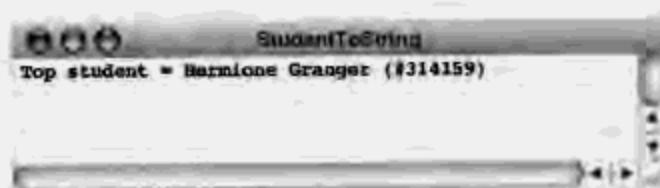


图 6-14 StudenttoString 的输出

但如果执行下面的行，情况又如何？

```
Object obj = topStudent; println("Top student = " + obj);
```

将 `topStudent` 变量赋给 `obj` 变量是完全合法的，因为每个 Java 类都是 `Object` 的子类。现在的问题是，当 Java 将参数赋值给 `println` 时，要调用 `toString` 方法的哪个版本？从前面的示例中可知，`Object` 有自己的 `toString` 定义。假设代码现在将该值声明为 `Object` 而不是 `Student`，Java 还会使用 `toString` 的 `Object` 版本(它产生的结果难以理解)吗？答案是否定的。存储在变量 `obj` 里的值仍然是 `Student`，Java 运行系统会正确调用方法的 `Student` 版本。

虽然重写方法提供了极大的灵活性和功能，但它也会带来一些风险。例如，假设设计一个类，其中一个公有方法调用类里的另一个公有方法。编写代码时，假设调用的方法有您设计的特殊效果。遗憾的是，如果其他人声明您的类的子类，他们通过重写您的代码所依赖的方法，可以使假设无效。这种情况很快会给维护带来极大麻烦。更糟糕的是，如果问题出现在优先类(它们执行客户不能执行的操作)里，其他编程人员用一段代码取代另一段代码就很容易产生安全漏洞。

常见错误

在现有类里重写方法时要小心，因为新的定义可能不符合原始类所作的假设。作为一般规则，只有当这些方法的文档特别要求这样做时才重写方法。

如上边提示框所示，最好保守使用重写，而且限制在原始类的设计师明确允许可以重写的情况下使用。类设计师总是发出这样的邀请。例如，`Object` 类里 `toString` 方法的注释开始如下：

```
/*
 * Returns a string representation of the object. In
 * general, the toString method returns a string that
 * "textually represents" this object. The result should
 * be a concise but informative representation that is
 * easy for a person to read. It is recommended that all
 * subclasses override this method.
 */
```

`Object` 类的设计师不仅邀请您重写 `toString` 方法，而且主动建议这样做。

有时，能够从重写方法本身的代码里调用方法的原始定义很有用。例如，当类定义执行初始化操作的 `init` 方法(这些操作与构造函数定义的操作分离)时，情况就出现了。子类可能想添加其他初始化代码，但必须确保超类里的所有初始化代码有效。Java 中，可以使用关键字 `super` 调用超类的行为，就好像它是方法的接收方一样。这样，如果要编写超类提供的 `init` 方法的扩充，可以这样写：

```
public void init() {
    super.init();
    Code to perform any further initialization goes here.
}
```

首先确保超类所需的所有初始化都是完整的，之后的方法可以执行此级别所需的其他任何初始化操作。

6.7 小结

本章的目的是从整体上和概念上介绍类和对象的思想。在 Java 中，类表示有共同结构和行为的一组值的模板，对象是类的特殊示例。因此，考察单个类定义，肯定有很多该类的实例。

除了给出了对象和类的完整定义之外，本章介绍的重点还包括：

- 计算机在很大程度上是确定地运行，意思就是产生的输出由输入确定，相同的输入总是产生相同的输出。然而，对于计算机游戏和模拟等应用程序而言，通过将随机引入其行为从而让程序非确定运行是很重要的。因为计算硬件通常不支持真正的随机，所以大多数编程环境通过提供产生伪随机数的方法来模拟随机行为。
- `acm.util` 程序包包括 `RandomGenerator` 类，该类提供了几种模拟随机事件的方法。`RandomGenerator` 类提供的方法如图 6-1 所示。
- `acm.util` 中定义的 `RandomGenerator` 类实际上是 `java.util` 中 `Random` 类的子类，所以它提供全部标准库类提供的能力及一些扩充。因为 `RandomGenerator` 的新扩充添加到 `Random` 基本性能的最上方，这些类表示分层抽象化。
- 对编程人员而言，在 Java 中定义新类可以给其他需要利用这些功能的编程人员提供有用的服务。为类编写代码的编程人员称为实现者。在程序中将类作为资源使用的编程人员称为客户。
- 客户和实现者对类有不同的看法。客户需要知道如何使用类提供的性能，但通常不需要知道这些特征运行的细节。相反，实现者必须理解这些细节。而且，要确保客户能够在合适的抽象高度上工作，类的实现者应该尽可能隐藏细节。
- 给客户提供他们所需的 Java 类信息的通常策略是使用 javadoc 文档系统。javadoc 应用程序阅读类的代码，自动生成类中方法的文档 Web 页面。
- 将类的定义封装入关键字 `class` 标识的代码块里，这样可以定义自己的类。许多类定义指定新类扩充的特定超类的名称；在没有特定超类时，新类扩展 `java.lang` 里的 `Object` 类，它是 Java 层次结构的根。
- 类定义通常包括构造函数、方法、命名常量和实例变量。构造函数指定如何创建类的新对象；方法定义类的行为；命名常量提供机制（第 3 章讨论过），给常量值赋以更容易理解的名称；实例变量允许对象保持其内部状态。总的来说，这些定义称为条目。
- 类里的每个条目都可以用关键字 `public` 或 `private` 标记，用来表示谁可以访问该条目。只有客户需要使用构造函数、方法和常量时它们才是公有的。本书中，所有实例变量都是私有的，以确保内部状态细节不被客户看到。

- 类可以包含有相同名称的几个构造函数和方法，只要它们的参数结构不同就行，编译器利用这些结构可以确定需要哪个版本。如果某个特定名称存在多个方法定义，该方法名称称为被重载。
- 类可以提供在其超类里已定义的方法的新版本。此过程称为重写。Java 总是选择适合于对象实际上是什么的方法版本，而不是在语境中声明它是什么的方法版本。
- 鼓励 Java 编程人员重写每个类里 `toString` 方法的定义，以便新的定义产生人类可以理解的对象描述。
- Java 定义关键字 `this` 来表示当前对象。如果可能产生混淆，就可以在表达式中使用该关键字来指明当前对象里的实例变量。可以在构造函数中使用关键字 `this` 来调用类里其他某个构造函数。
- Java 定义关键字 `super` 表示当前对象的超类。这个关键字用来调用超类里的构造函数和方法。
- 对象创建之后就不允许客户改变其属性的类称为不变类，如 6.3 节的 `Rational` 类，后面几章将介绍几种不变类。

6.8 复习题

- 对于程序而言，非确定地运行为什么很有用？
- 术语“伪随机数”是什么意思？
- 本章鼓励定义名为 `rgen` 的实例变量来保持生成每个随机值的 `RandomGenerator` 对象。这一行的正确语法是什么？
- 如何使用 `RandomGenerator` 类里的方法生成下列每个值？
 - 介于 0~9 之间的随机数字。
 - 介于 -1.0 ~ +1.0 之间的 `double` 值。
 - 1/3 时间是 `true` 的 `boolean` 值。
 - 用于设置图形对象颜色的随机颜色值。
- 假设 `d1` 和 `d2` 声明为 `int` 类型的变量，可以使用多赋值语句
`d1 = d2 = rgen.nextInt(1, 6);`
 来模拟掷双骰子的过程吗？
- 在什么环境下调用 `setSeed` 方法有用？
- 正确描述客户与实现者之间的不同。
- 客户和实现者在类设计的哪些方面应该一致？
- `javadoc` 文档系统有什么作用？
- 什么是分层抽象化？
- 许多类定义用关键字 `extends` 来指定超类。如果省略这个关键字，情况如何？
- 在类定义中，条目有哪些常见类型？
- 判断题：本书及 ACM 库程序包中的所有实例变量都使用关键字 `private` 定义。
- 重载和重写有什么不同？

15. Java 中, 关键字 this 和 super 表示什么意思? 本章中在什么语境中使用这些关键字?
16. 什么属性让类保持不变?
17. 在每个新 Java 类的定义里包括 `toString` 方法的定义很有用, 为什么?

6.9 编程练习

1. 从一副完整的 52 张扑克牌中随机选择一张, 编写程序显示该纸牌的名称。每张牌由等级(Ace、2、3、4、5、6、7、8、9、10、Jack、Queen、King)和花色(梅花、方块、红桃、黑桃)组成。程序应该显示纸牌的完整名称, 如图 6-15 所示。

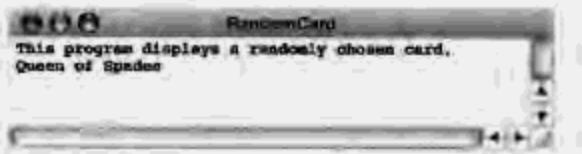


图 6-15 RandomCard 的输出

2. Heads...
Heads...
Heads...
A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.

——Tom Stoppard, *Rosencrantz and Guildenstern are Dead*, 1967

- 编写程序模拟抛硬币, 一直抛到出现 3 个连续的正面为止。这里, 程序应该显示硬币被抛的总次数, 如图 6-16 所示。



图 6-16 ConsecutiveHeads 的可能输出

3. 虽然在机会游戏时很容易想到随机数, 但是它们在计算机科学和数学中有其他更实际的用途。例如, 可以通过编写简单程序模拟飞镖板, 使用随机数生成常量 π 的大致近似值。假设墙上挂着一块飞镖板。它由正方形的背景幕上画的一个圆组成, 如图 6-17 所示。

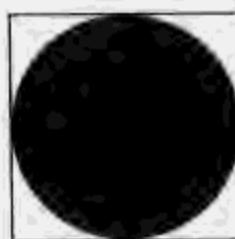


图 6-17 飞镖板

如果随机朝这块板扔飞镖，有些会落在圆圈以内。如果真的是随机投掷，那么落在圆圈内的飞镖数与落在正方形里的飞镖总数之比，应该大致等于这两个面积之比。面积之比与飞镖板的实际大小无关，如下公式所示：

$$\frac{\text{落在圆圈内部的飞镖}}{\text{落在正方形内的飞镖}} = \frac{\text{圆圈面积}}{\text{正方形面积}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

要在程序中模拟此过程，假设飞镖板画在标准的笛卡尔坐标平面里。生成两个随机数 x 和 y （介于 -1 和 1 之间），可以模拟朝正方形投飞镖的随机过程。点 (x,y) 总是在正方形内部。如果

$$\sqrt{x^2 + y^2} < 1$$

那么点就在圆圈内部。

如果对不等式两边进行平方就会极大简化条件，从而产生更有效的测试，如下所示：

$$x^2 + y^2 < 1$$

如果多次执行该模拟，计算落在圆圈内飞镖的分数，结果接近 $\pi/4$ 。

编写程序模拟投 10 000 次飞镖，然后使用本练习中描述的模拟技术生成和显示 π 的近似值。如果只有前面一些数字测试答案是正确的，请别担心。这个问题中使用的策略不是特别精确，即使在计算大致近似值时它偶尔有用。数学中这种技术称为蒙特卡洛综合，以赌博中心摩纳哥首都命名。

4. *shall never believe that God plays dice with the world.*

—Albert Einstein, 1947

爱因斯坦的相对论、物理学上的电流模型及量子理论，都是建立在自然确实包括随机过程这一思想之上。例如，放射性原子不会因为人类理解的特定理由而停止衰减，而是原子在一定时期内有衰减的随机概率。有时会，有时不会，无法确定。

因为物理学家认为放射性衰减是一个随机过程，所以可以用随机数模拟该过程。假设从原子集合开始，每个时间单位内每个原子都有衰减的可能。依次取每个原子，随机确定它是否衰减，考察概率，这样就可以近似衰减过程。

编写程序模拟包括 10 000 个放射性原子样本的衰减，其中每个原子每年有 50% 的衰减机会。程序输出应该显示每年结束时剩余原子的数量，如图 6-18 所示。

```
There are 10000 atoms initially
There are 4969 atoms at the end of year 1
There are 2464 atoms at the end of year 2
There are 1207 atoms at the end of year 3
There are 627 atoms at the end of year 4
There are 311 atoms at the end of year 5
There are 165 atoms at the end of year 6
There are 89 atoms at the end of year 7
There are 40 atoms at the end of year 8
There are 21 atoms at the end of year 9
There are 9 atoms at the end of year 10
There are 5 atoms at the end of year 11
There are 2 atoms at the end of year 12
There are 0 atoms at the end of year 13
```

图 6-18 RadioactiveDecay 的输出

输出的数字显示，样本中每年几乎有一半原子衰减。物理学中，这种观测结果习惯性称为样本的半衰期为一年。

5. 从蒙特卡洛到拉斯维加斯的赌场中，最常见的赌博设备之一是“赌博机”。典型的赌博机有 3 个轮子，它们在一个狭窄的窗口后面旋转。每个轮子都标记为 CHERRY、LEMON、ORANGE、PLUM、BELL 和 BAR。然而，窗口每次只允许看到一个轮子的一个符号。例如，窗口可能显示的配置如下：



如果赌博机中投入 1 美元硬币，推拉旁边的手柄，轮子就会旋转，最终显示为某个新的配置。如果显示的配置与赌博机前输出的某个获胜模式匹配，就可以取回一些钱。如果不匹配，就输了 1 美元。下面显示的是典型的获胜模式及其相关回报。

BAR	BAR	BAR	支付	\$ 250
BELL	BELL	BELL/BAR	支付	\$ 20
PLUM	PLUM	PLUM/BAR	支付	\$ 14
ORANGE	ORANGE	ORANGE/BAR	支付	\$ 10
CHERRY	CHERRY	CHERRY	支付	\$ 7
CHERRY	CHERRY	—	支付	\$ 5
CHERRY	—	—	支付	\$ 2

符号 BELL/BAR 说明 BELL 或 BAR 都可以在那个位置出现，破折号说明所有符号都可能出现。这样，在第一个位置出现 CHERRY 就自动获得 2 美元，不管其他轮子出现什么。LEMON 符号没有回报，即使出现 3 个也是这样。

编写程序模拟赌博机。程序应该给用户提供 50 美元的原始赌金，然后让用户开始玩，直到钱输光或用户决定退出为止。每一轮，程序应该取走 1 美元，模拟轮子旋转，查看结果，给用户合适的奖金。例如，用户可能很幸运，如图 6-19 所示。

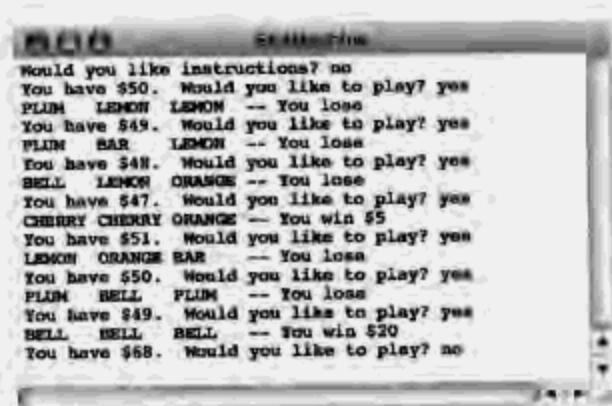


图 6-19 赌博机运行示例

即使这样也不现实(这样赌博机让赌场赚不到钱), 应该假设每个轮子上的 6 个符号出现的机会均等。

6. 随着计算机在学校不断普及, 很有必要在教学过程中使用计算机帮助工作。这种需要促进了教育软件产业的发展, 已经产生了许多帮助教学的程序。

作为教育应用程序的示例, 写程序出一些简单的算术题让学生回答, 如图 6-20 所示。

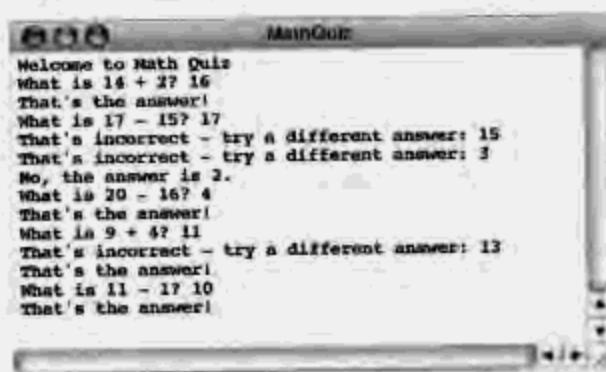


图 6-20 算术问答的运行示例

程序应该满足如下要求:

- 每组应该问 5 个问题, 因为这种限制, 问题数量应该编码为命名常量, 以方便改变。
- 每个问题应该都是两个数的加法或减法问题, 如“ $2+3$ 等于几?” “ $11-7$ 等于几?”
- 每个问题的类型——加法或减法——应该随机选择。
- 要确保问题适合于 1、2 年级的学生, 所有数包括答案都必须大于 0 且小于 20。这种限制意味着程序不能问像“ $11+13$ 等于多少?” 或“ $4-7$ 等于多少?” 这样的问题, 因为答案超出了合理范围。在这种限制内, 程序应该随机选择数。
- 每个问题程序都应该给学生 3 次回答机会。如果学生回答正确, 程序应该以适当的祝贺方式表示结果, 并继续下一个问题。如果学生 3 次都没有回答正确, 程序应该给出答案, 然后继续下一个问题。

7. 即使练习 6 中的程序在学生回答正确时提供鼓励, 但是过一会就单调地重复像“答案正确!”这样的句子, 则会产生负面效果。要在互动中添加变化, 就要修改练习 6 的解决方案。

学生回答正确时，让它随机选择 4~5 个不同的消息，如图 6-21 所示。

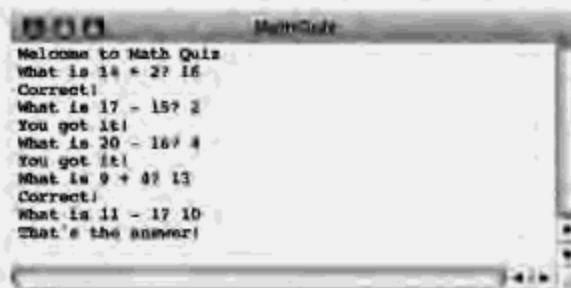


图 6-21 算术问答的输出示例

8. 实现名为 Card 的新类，它必须包括如下条目：

- 4 种花色(CLUBS、DIAMONDS、HEARTS、SPADES)的命名常量和 4 个等级(一般表示为 ACE、JACK、QUEEN、KING)。等级常量的值应该分别是 1、11、12 和 13。
- 构造函数，它读取一个等级和一种花色，使用这些值返回 Card。
- 命名为 getRank 和 getSuit 的 Getter 方法，检索纸牌的等级和花色组件。
- `toString` 方法的实现方式，该方法返回纸牌的完整名称，如练习 1 中所示。记住可以使用+运算符来连接字符串的不同部分，如图 6-11 中 Rational 类的 `toString` 实现方式中所示。

9. 使用图 6-5 中的 Student 类作为模型，实现新的名为 LibraryRecord 的类，记录图书的如下信息：

- 书名。
- 作者。
- 美国国会图书馆目录号。
- 出版者。
- 出版年份。
- 本书是否正在流通。

该类应该导出如下条目：

- 读取这 6 个值并利用它们创建新的 LibraryRecord 对象的构造函数。
- 构造函数的第二版本，它只读取前 5 个值，并将书初始化为流通。
- 6 种信息合适命名的 getter 方法。
- 流通/非流通标记的 setter 方法。
- `toString` 方法的相应实现方式。

10. 本章 Rational 类的实现方式在实践中不是特别有用，因为它不允许分子和分母不是整数，即使有理算术中经常出现较大的值也如此。解决该问题的方法之一是使用 `java.math` 程序包中的 `BigInteger` 类，它定义整数的扩充类型，该整数可以取任意大的值。重写 Rational 类的实现方式，在不改变参数和所有公有方法结果类型情况下，将私有实例变量 `num` 和 `den` 声明为 `BigInteger` 而不是 `int`。要理解 `BigInteger` 如何运行，可以参考 javadoc 程序包。

第7章

对象和内存

Yea, from the table of my memory.

I'll wipe away all trivial fond records.

—William Shakespeare, Hamlet, 1600



Jay Forrester

Jay Forrester 在中西部一个没有电的养牛场长大，之后他在内布拉斯加州大学和 MIT(美国麻省理工学院)学习电机工程，在那里他成为 1944 年“海军旋风计划(Navy's Project Whirlwind)”的主任。Whirlwind 系统与费城摩尔学校的 ENIAC 系统及哈佛的 MARK I 系统一起，从早期模拟设计到数字系统(已成为今天的行业标准)不断发展，在计算机的早期历史中发挥了重要作用。Forrester 对计算机硬件设计作出的最重要贡献是开发核心内存，里面小的铁素体磁盘被磁化为两个方向，表示二进制的 0 和 1。磁化核心内存使硬件设计发生了革命性的变化，到 20 世纪 70 年代它被集成电路内存取代以前，基本上所有计算机都在使用它。1956 年，Forrester 加入了斯隆管理学校(Sloan School of Management)，在那里他建立了新的系统动力学原理，该原理着重从整体上关注大型系统及其交互作用，而不是只看它们的单个部分。

Java 等高级语言最大的优势是，让用户不再担心计算机硬件内部低层次表示的细节。管理内部表示法的细节通常很耗费时间，也很单调乏味，让语言理解这些细节则可以提高编程人员的生产率。但另一方面，不知道内部表示法如何运行就很难理解编程过程。编程人员从整体上使用的结构——继续形成像 Java 这样的语言的基础——反映了运行程序的硬件的性能。了解这些低层次机器性能为更高更抽象地理解编程提供了概念上的架构。开始使用对象而不是原始数据时，对计算机内部数据表示法如何进行形成直观感觉特别重要。在 Java 中，将对象作为参数传递给方法，似乎与使用原始类型作为参数大不相同。然而，如果理解计算机内部表示对象的方法，就会很清楚这种明显不同的原因，实际上它们最初出现时完全一样。

7.1 内存结构

第1章已经从抽象层面上简单介绍了计算机的内部结构。图1-1说明了典型计算机系统的主要硬件组件，包括CPU、内存、辅助存储器和I/O设备。对象如何存储在计算机内？要开发这个智力模型，有必要详细讨论一下内存系统的结构。

7.1.1 位、字节和字

在最原始级别，所有数据值都是以基本信息单元(称为位)的形式进行存储。位记录了最简单的可能值，可能是两种可能状态中的一种。如果将机器内的电路看成微型灯开关，那么这两个状态就是off(关)和on(开)。如果将位看成 Boolean 值，那么可以使用 false 和 true 标注。然而，词语“位”最初是二进制数字(binary digit)的缩写，所以最常见的是使用值 0 和 1 来标注其状态。

由于单个位保存的信息太少，所以单个位不能提供存储数据的便利机制。为了更方便地存储数和字符等传统类型的数据，必须将位集合起来形成更大的单元，并将该单元当成存储的完整单元对待。最小的结合单元称为字节，由 8 个单独位组成。在绝大多数计算机中，字节组合成更大的结构，称为字——在 Java 中定义为 4 个字节——它大得足以保存一个整数值。具体到每种计算机可用内存的大小差别很大。早期支持内存的计算机，其大小用千字节(KB)计算；今天的计算机内存大小用百万字节(MB)甚至十亿字节(GB)计算。在许多科学中，前缀 kilo, mega 和 giga 分别表示千、百万和十亿。然而，在计算机世界里，这种基于 10 的值不能很好适应计算机的内部结构。所以，按照传统，用这些前缀来表示最接近它们传统解释的 2 的幂。因此，在编程中，前缀 kilo, mega 和 giga 的意思如下所示：

$$\text{kilo (K)} = 2^{10} = 1\,024$$

$$\text{mega (M)} = 2^{20} = 1\,048\,576$$

$$\text{giga (G)} = 2^{30} = 1\,037\,741\,824$$

一台 20 世纪 70 年代早期的 64KB 计算机，其内存为 64×1024 即 65 536 字节。同样，512MB 的计算机内存为 $12 \times 1\,048\,576$ 字节即 536 870 912 字节。

7.1.2 二进制和十六进制表示法

计算机内的每个字节都保存数据，这些数据的含义取决于系统如何解释单个位。依据用来操作它的硬件指令，特定系列的位可以表示整数、字符或浮点值，其中每一个都需要某种编码方案。最简单的编码方案是整数的编码方案，其中位用来表示用二进制表示法表示的整数。在二进制表示法中，只有 0 和 1 是合法的值。二进制表示法在结构上与人们熟悉的十进制表示法类似，但是它使用 2 而不是 10 作为基数。二进制数字对整个数的贡献取决于它在整个数内的位置。最右边的数字表示单位字段，其他每个位置的数字比它右边的数字大 2 倍。

例如，考察包含下面二进制数字的 8 位字节：

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

这种位序列表示数 42，可以计算单个位的贡献来验证这个值，如图 7-1 所示。

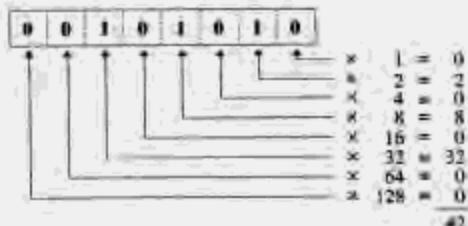


图 7-1 二进制与十进制的关系

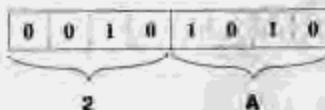
图 7-1 说明如何将整数映射为使用二进制表示法的位，但这种表示法不是特别方便。二进制数很麻烦，主要因为它们太长。十进制表示法很直观，我们也很熟悉，但很难理解如何将数转换为位。为了有助于理解数如何转换为二进制表示法而不必使用延伸到整个页面的二进制数，计算机科学家倾向于使用十六进制(基数为 16)表示法。

十六进制表示法有 16 个数字，表示 0~15 的值。十进制数字 0~9 完全可用作前 10 个数字，但是传统算术没有定义表示剩余 6 个数字的其他符号。计算机科学家通常使用字母 A~F 来表示，如表 7-1 所示。

表 7-1 十六进制中的 10~15 用 A~F 表示

十六进制	十进制值
A	10
B	11
C	12
D	13
E	14
F	15

十六进制的优点是可以在十六进制和基本的二进制表示法之间快速转换。所有要做的就是将位 4 个 4 个地分组。例如，数 42 可以像下面这样从二进制转换为十六进制。



前 4 个数字表示数 2，后 4 个表示数字 10。这两个数转换为相对应的十六进制数字就是 2A，相加单个数字值，同样可以验证这个数的值是 42，如图 7-2 所示。

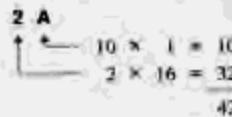


图 7-2 十六进制与十进制的关系

大多数情况下，本书中出现的数字表示法使用十进制表示法以方便阅读。如果上下文中的基数不太清楚，文本通常使用下标来表示基数。这样，数 42 的 3 种最常见表示法——十进制、二进制和十六进制——如下所示：

$$42_{10} = 101010_2 = 2A_{16}$$

关键是数本身总是相同的：数字基数只影响表示法。42 的真实解释与基数无关。真实解释

可能最容易在小学生使用的表示法中看到，毕竟它是另一种写数的方法，如图 7-3 所示。



图 7-3 另一种表示法

此表示法中线段数是 42。用二进制、十进制或其他基数写数是表示法的属性，而不是数本身的属性。

7.1.3 内存地址

在典型的计算机内存系统中，每个字节都由数字地址标识。计算机中第一个字节编号为 0，第二个字节编号为 1，依次类推，直到计算机里的字节数。例如，可以图解 64KB 计算机的内存字节，如图 7-4 所示。然而，开始时可能不太熟悉编号方案，因为写地址使用的是十六进制数而不是十进制数。7.1.2 小节介绍了十六进制表示法，以及将十六进制值转换为其二进制或十进制值的方法。要记住的重要一点是地址只是数，基数只与写这些数的方式有关。图中最后一个地址是 FFFF，它对应的十进制值是 65 535。用十进制表示法写地址很容易，也能更容易理解这些数。然而，本文使用十六进制表示法，理由如下：

- 地址数通常是十六进制，Java 调试器和运行环境以这种形式显示地址。
- 以十六进制形式写地址数和使用 sans-serif 字体，更容易看出特定数表示地址而不是表示某个未标识的数。
- 使用十六进制值更容易理解选择特定限制的原因。写为十进制值时，数 65 535 更像随机值。如果用十六进制值 FFFF 表示相同的数，就很容易看出这个值是用 16 位表示的最大值。
- 数看起来不太熟悉使用户不能从算术上考察它们。在对编程人员隐藏基本地址计算方面，Java 很出色。重要的是明白地址用数表示，至于这些数是什么——通常也不可能确定——根本不重要。

早期，字节值很有用，因为字符可以用单个字节表示。今天，字节太小了，不足以表示现代计算机中的较大字符集。例如，第 8 章讨论的 Unicode 表示法中每个字符需要两个字节。因此，char 类型的值占用内存两个字节单元，如图 7-5 中的阴影字节所示。



图 7-5 char 类型值得表示

需要多个字节的数据值由第一个字节的地址标识，因此阴影区表示的字符地址为 FFF0。第二个示例是，double 类型的值需要内存 8 个字节，因此存储在地址 1000 里的 double 类型的变量将占用地址 1000 和 1007(包括 1000 和 1007)之间的所有字节，如图 7-6 所示。

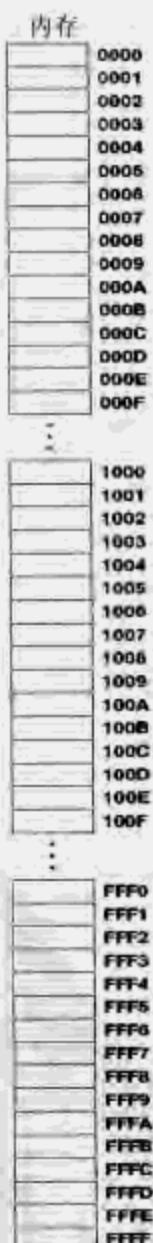


图 7-4 内存字节

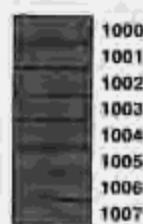


图 7-6 double 类型值得表示

虽然内存的地址通常按字节指定，但是字节太小因而所以不太有用，这导致硬件设计师将连续字节组成较大的单元。今天典型的体系结构包括以下面的大小使用内存单元的指令：单字节(8位)、双字节(16位)、4字节(32位)和8字节(64位)，这些单元对应于内置的 Java 类型 byte、short、int 和 long。为了节省空间，通常用这些较大的单元而不是单个字节来存图表。存储在编号为 1000~1007 之间所有字节里的 double 值可以用图 7-7 所示的形式更简洁地图解，其中内存划分为 4 个字节的字。



图 7-7 4 字节数据的表示法

7.2 将内存分配给变量

在程序里声明变量时，编译器必须保留内存空间来保存变量的值。保留内存空间的过程称为分配。Java 中，依据其声明的方式，从 3 种不同的来源分配内存，如图 7-8 所示。下面分别介绍 3 种分配策略。

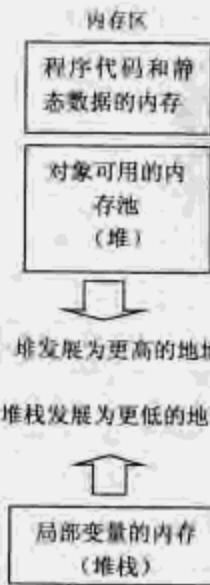


图 7-8 3 种分配变量的来源

1. 静态变量和常量

所有声明中包含关键字 static 的变量，都作为整体而不是单个对象应用于类。通常在内存空间开头分配这些变量，旁边是用于存储程序指令的内存单元。本书示例中，唯一的静态声明是命名常量声明。

2. 动态分配的对象

所有使用关键字 new 创建的对象从称为堆的内存区分配存储空间。虽然没有明确规则加强此约定，但是 Java 虚拟机的大多数实现工具在固定区之后立即开始分配堆内存，这些固定区分配给类里的静态声明。

3. 局部变量

方法内所有声明为局部变量的变量，从称为堆栈的内存区分配，它通常至少部分在硬件里实现。现代体系结构中，堆栈从内存里最高的合法地址开始，随着新方法的调用，它向更低的地址发展。调用方法从数量上增加了堆栈的大小，这些数量是保存方法声明的局部变量所必需的。如第 5 章所述，分配给方法局部变量的内存称为堆栈框架。方法返回时，声明堆栈框架，恢复其调用者的框架。

声明对象变量可能保留堆和堆栈里的内存，认识到这一点很重要。用来存储对象的局部变量出现在堆栈中，好像它是保存原始类型的局部变量一样。然而，特定变量只保存对象的内存地址，该地址存储在堆里。如第 6 章所述，该地址称为引用。

7.2.1 Rational 类的内存图

这里，有必要用一个简单示例来说明堆存储和堆栈存储之间的关系。假设决定使用 6.3 节介绍的 Rational 类，它讨论的是分数的精确表示法。在内部，每个 Rational 对象都声明两个实例变量 num 和 den，分别存储分数的分子和分母。问题是，当执行如下声明时，计算机内部会发生什么情况？

```
Rational half = new Rational(1, 2);
```

变量 half 是局部变量，因此出现在其调用者的堆栈框架里。然而，该变量只包含了足够保存内存地址的存储空间，它通常是 32 位寻址计算机的 4 个字节，虽然 64 位体系结构不断增长的数值说明，随着时间的推移，所需的内存大概要增加到 8 字节。任何时候，堆栈中的内存量总是太小。不管对象本身有多大，其引用可以用单个字进行存储。

Java 语句执行过程中发生了什么要看堆和堆栈里发生了什么。等号右边初始值表达式的执行在堆里创建了一个新的对象。该对象的堆存储包含整数值 num 和 den 以及所有对象的一些常见附加信息的空间。Java 运行系统使用这些信息管理内存里的对象。从编程人员的观点来看，附加信息的细节并不重要。看看实例变量的内存要求，会发现对象占用的空间比希望的要多，附加信息有时有助于理解这一点。本书中，附加管理信息称为对象系统开销，在内存图中用阴影表示。例如，对 new Rational(1, 2) 的调用创建的对象如图 7-9 所示。

在堆里分配了新的 Rational 对象之后，声明过程的下一步是将对象的地址存储在变量 half 里。变量分配在堆栈上，所以在大多数计算机，它出现在内存堆栈框架中高端的某个位置。如

如果没有看到整个方法的代码，就无法知道堆栈框架内的其他值是什么，虽然这样，但是分配给 half 的字可能如图 7-10 所示。

new Rational		1000
num	1	
den	2	

图 7-9 创建的对象

half	1000	FFC0

图 7-10 分配给 half 的字

变量 half 显示的值 1000 不是整数，而是实际对象定位的地址。选择 1000——以及本书中使用的其他每个堆和堆栈变量的地址——是完全随意的。一般来说，Java 编程人员无法知道对象的地址，也没有理由知道。然而，每个对象都有地址，而且 Java 程序通过跟踪对象的地址可以访问对象里的数据，记住这一点非常重要。

即使熟悉对象以后，也会发现它有助于创建图表，该图表更容易让人明白内存里正在发生的情况。要说明此过程，有必要考察在某种程序上更详细的示例。考察下面的 run 方法，它在第 6 章出现过：

```
public void run() {
    Rational a = new Rational(1, 2);
    Rational b = new Rational(1, 3);
    Rational c = new Rational(1, 6);
    Rational sum = a.add(b).add(c);
    println(a + " + " + b + " + " + c + " = " + sum);
}
```

当然，该程序的思想是添加 3 个 Rational 变量并显示它们的和。然而，这里关注的是计算机内部如何表示这些值。

前 3 个声明足够简单。如变量 half 的简单示例所示，每个声明都构造了一个新的 Rational 对象，并将其地址分配到 run 方法的堆栈框架中的变量，产生的内存图如图 7-11 所示。

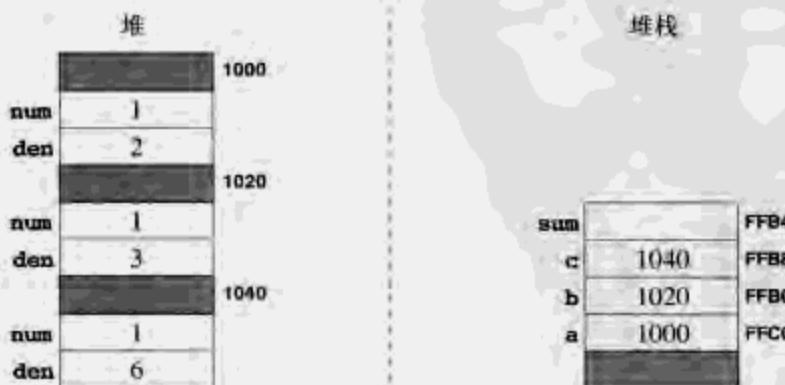


图 7-11 half 的内存图

和在大多数语言中一样，调用方法时 Java 会给方法创建完整堆栈框架，因此全部 4 个局部变量——a、b、c 和 sum——都已经分配到堆栈上了，但只有前 3 个在执行时被初始化。用

来初始化 sum 的值是表达式

```
a.add(b).add(c)
```

的结果。有必要逐步分析该操作。

过程的第一步是调用 Rational 对象 a 上的 add 方法，并将 b 的值作为参数传递给它。因为该操作是方法调用，因此必须为它创建新的堆栈框架，将方法的形参值与调用里的实参值结合起来。如图 6-11 所示，add 方法的代码如下：

```
public Rational add(Rational r) {
    return new Rational(this.num * r.den + r.num * this.den,
                        this.den * r.den);
```

形参是 add 方法内命名的 r，意思就是说必须给 r 赋实参的值，即调用框架里命名的 b。目前变量 b 的值是数 1020，它是堆中相应对象的地址。Java 通过复制地址来初始化新框架里的变量 r，即新框架里的变量 r 也包括 1020。

本示例中，在接收对象上调用的所有方法，其堆栈框架都包括对那个对象的引用，Java 用关键字 this 标识，记住这一点至关重要。从直观上讲，最好将 this 作为表示接收对象的局部变量看待，在这里它就是存储在变量 a 里的值。接收方的特征和参数值一起被复制到新的框架里作为简单地址。这样，创建新的框架之后，内存的状态如图 7-12 所示。

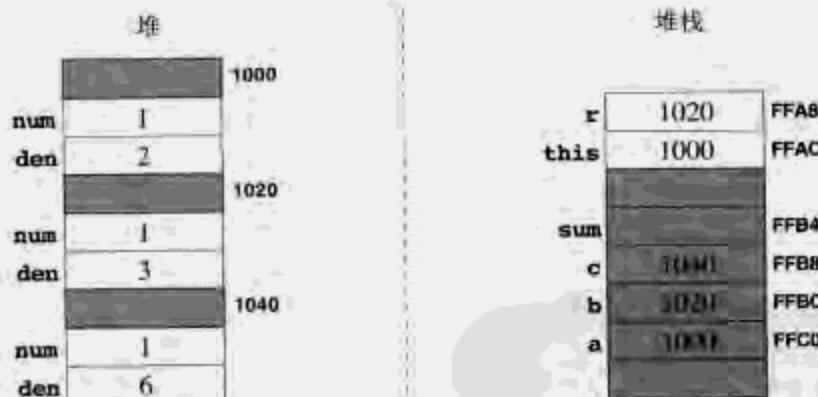


图 7-12 创建新框架后的内存

图 7-12 中，当前框架以下的堆栈区显示为灰色，以强调 add 方法不能访问阴影区。可访问 add 的唯一变量是当前框架里存储在变量 r 和 this 里的引用。这些引用变量正是 add 方法实现其计算所必需的。add 方法返回新的 Rational 值，其分子由下面的表达式确定：

```
this.num * r.den + r.num * this.den
```

其分母的值是：

```
this.den * r.den
```

表达式里的每一项都表示选择操作——从合适的 Rational 值中选择 num 或 den 两个字段之一。例如，r.den 项指定 den 字段值的选择，该值的引用存储在 r 里。由于 r 包含地址 1020，所以 r.den 表示存储在地址 1020 里对象的分母部分，即整数 3。以相同的方法扩充每一项的值：

就可以说明表达式

```
new Rational(1 * 3 + 1 * 2, 2 * 3)
```

确定 add 方法的结果。可以进一步简化为：

```
b = new Rational(5, 6)
```

该表达式构造了一个新的 Rational 对象(它在堆里有指定的组件)，返回对象的地址作为调用 a.add(b) 的值，它返回时清除了 add 的堆栈框架。

和到目前为止看到的许多计算结果不一样，该方法计算的新值不存储在变量里。计算 a.add(b) 只是较长表达式

```
a.add(b).add(c)
```

的子表达式而已。

因此，a.add(b) 的值变成了新的 add 调用(参数为 c)的接收方。为新的方法调用设置框架让内存处于图 7-13 所示状态，让新的 Rational 对象显示在堆中地址 1060 上。

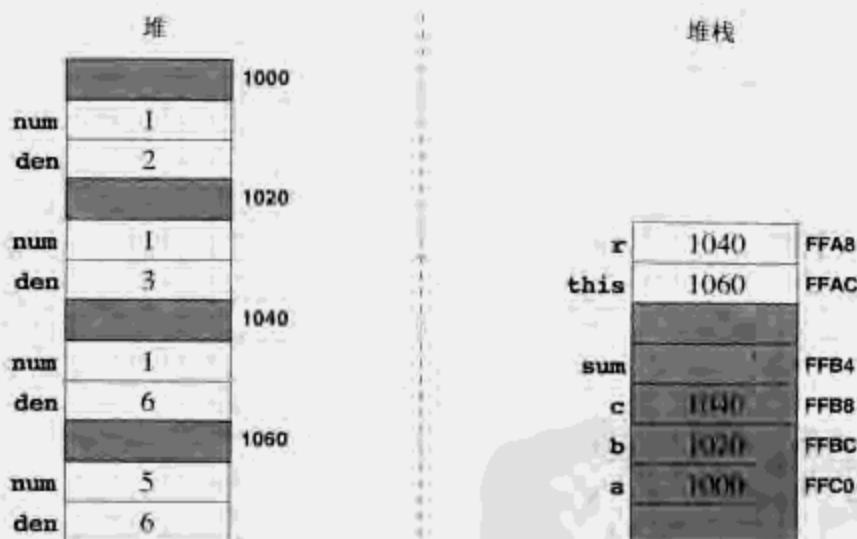


图 7-13 内存状态

add 方法的求值再次要求在当前框架可以访问的两个 Rational 值中确定字段的值。此时，结果表达式的直接扩充表示最后的值与下面的构造函数调用相等：

```
new Rational(5 * 6 + 1 * 6, 6 * 6)
```

结果是

```
new Rational(36, 36)
```

幸运的是，Rational 的构造函数使用欧几里得算法将分数化为最简式，这样对结果进行了额外简化。从分子和分母中约去最大公约数 36，就产生了新的 Rational 对象，其 num 和 den 字段的值都是 1。该值赋给变量 sum 以后，堆和堆栈如图 7-14 所示。

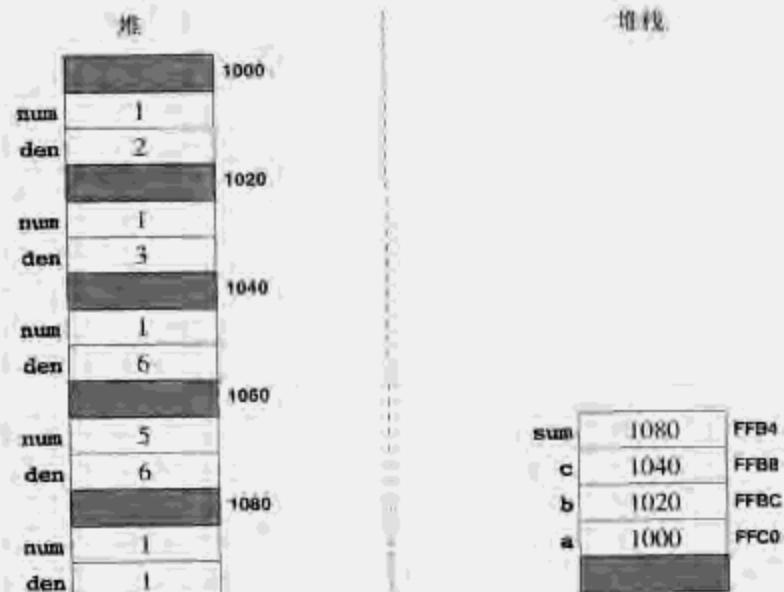


图 7-14 给 sum 变量赋值后的堆与堆栈

7.2.2 无用单元收集

7.2.1 小节的示例说明对象存储在内存时通常会出现有趣的皱纹。如果仔细查看最近的图表，就会发现堆包含 5 个不同的 Rational 对象，即使在堆栈上只有 4 个这样的对象有引用。计算过程中创建了存储在地址 1060 的分数 5/6，将它作为子表达式

```
a.add(b)
```

的中间结果。

虽然这个值是完成计算所必需的，但实现最后的加法以后它就没什么用了。用计算机科学的传统用语来说，该值现在是无用单元。面向对象语言中的计算通常生成一些无用单元，将之作为复杂计算的中间结果。遗憾的是，这些值会占用堆里的空间。如果允许保留该无用单元，即使运行中的数据有足够的空间，最后堆也会被装满。

为了解决堆被不再有用的对象装满这个问题，Java 运行时系统采用无用单元收集策略。当堆里的可用内存快要不足时，为了收集堆里的所有无用单元，并将该内存返回可分配存储池，Java 会推迟正在做的事。为此，Java 以下面的方法使用二阶段策略。为了让基本思想更清楚，这里对该过程进行了简化。

(1) 仔细检查堆栈上每个变量引用和静态存储里的每个变量引用，在对象的系统开销空间设置一个标记，通过这种方法将相应用对象标记为“使用中”。另外，给对象设置“使用中”标记时，确定对象是否包括对其他对象的引用，将那些对象也标记为“使用中”。

(2) 仔细检查堆中的每个对象，删除没有设置“使用中”标记的所有对象，将其使用的存储空间返回到堆，以便继续再分配。删除这些对象必须安全，因为如果在程序变量里仍然存在对这些对象的引用，程序无法访问它们。

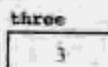
因为无用单元收集算法以二阶段操作：第一阶段标记使用中的对象，第二阶段清理收集不可访问对象的内存，所以这种策略通常称为标记——清理收集器。

7.3 原始类型与对象

从高级概念上的角度来看，考察 Java 对象的方法与考察原始类型(如 int 和 double)的方法相同。如果写

```
int three = 3;
```

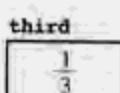
来声明整数变量，可以这样图解该变量：画一个名为 three 的方框，在里面写上 3，如下所示。



在某种程序上，可以用相同的方法处理对象。例如，如果使用

```
Rational third = new Rational(1, 3);
```

创建新的 Rational 变量，在概念上可以考察此变量，就像它有真实的值一样，如下所示。



然而，重要的是理解这种类似有什么不一样。从本章前面内部表示法的讨论中可以知道，变量 third 实际上不包括实际对象，而是包含该对象的地址。这种表示对象的模式有几种含义，作为编程新手，理解这些含义非常重要。

7.3.1 参数传递

当将对象作为参数传递给方法时，就会出现一个重要的结果(Java 为对象使用这种表示法方案)。乍一看，作为参数传递对象的规则似乎与传递原始值的规则不一样。例如，假设写下面的一对方法：

```
public void run() {  
    int x = 17;  
    increment(x);  
    println("x = " + x);  
}  
  
private void increment(int n) {  
    n++;  
    println("n = " + n);  
}
```

运行程序产生如图 7-15 示的输出。



图 7-15 SimpleParameterTest 的输出

从输出可见, increment 方法里的++运算符只影响局部变量 n 的值, 不影响调用框架中 x 的值。这种行为反映了这些值存储在堆栈上的方式。调用 increment 时, 局部变量 n 的堆栈条目的值复制到新的变量 n 的堆栈条目中, 因此, 堆栈看起来如图 7-16 所示。



图 7-16 堆栈的状态

改变新框架内 n 的值对原始值没有影响。

如果这些值是对象而不是简单的整数, 情况又如何? 回答这个问题的方法之一是设计简单的类, 让它包含嵌入的整数值, 如图 7-17 所示。EmbeddedInteger 类的定义包括构造函数, 用来设置内部值的 setValue 方法, 用来取回该值的 getValue 方法, 以及让使用 println 显示对象更容易的 toString 方法。利用这些方法, 可以编写一个简单的测试程序, 代码如下:

```
/*
 * This class allows its clients to treat an integer as an object.
 * The underlying integer value is set using setValue and returned
 * using getValue.
 */
public class EmbeddedInteger {

    /**
     * Creates an embedded integer with the value n.
     */
    public EmbeddedInteger(int n) {
        value = n;
    }

    /**
     * Sets the internal value of this EmbeddedInteger to n.
     */
    public void setValue(int n) {
        value = n;
    }

    /**
     * Returns the internal value of this EmbeddedInteger.
     */
    public int getValue() {
        return value;
    }

    /**
     * Overrides the toString method to make it return the string
     * corresponding to the internal value.
     */
    public String toString() {
        return "" + value;
    }
}

/* Private instance variable */
private int value;      /* The internal value */
}
```

图 7-17 EmbeddedInteger 类的实现方式

```

public void run() {
    EmbeddedInteger x = new EmbeddedInteger(17);
    increment(x);
    println("x = " + x);
}

private void increment(EmbeddedInteger n) {
    n.setValue(n.getValue() + 1);
    println("n = " + n);
}

```

如果运行程序的这个版本，输出就与前面不同，如图 7-18 所示。



图 7-18 ObjectParameterTest 的输出

在基于对象的实现方式中，方法返回后继续反映基本对象中字段值的变化。这种行为上的不同是直接从内存表示对象的方法中产生的。新的实现方式中 `increment` 调用开始时，堆和堆栈处的状态如图 7-19 所示。



图 7-19 increment 调用开始时的堆与堆栈

这里，改变对象内的 `value` 字段会影响 `n` 和 `x` 的概念值，因为这两个变量包含相同的引用。表示法中这种不同的直观结果是对象——和原始值形成对比——在调用方法和被调用的方法之间共享。然而，他们的基本机制完全相同。初始化新的局部变量时，旧的值复制到新变量的堆栈位置。如果值是对象，复制的就是引用，而不是基本值。

7.3.2 包装类

图 7-17 中的 `EmbeddedInteger` 类确实不只简单说明了对象中的参数传递。在后面几章会发现，Java 库程序包包含很多类，这些类可以应用于任何类型的对象。例如，第 10 章介绍的 `ArrayList` 类就可能维护对象的规则列表。对于 `ArrayList` 类，可以添加新值，删除现有值，实现许多有用的操作，这些操作在列表抽象的语境中有意义。使用列表的操作其本质与列表包含的值的类型无关。因此，要使存储任何类型的对象成为可能，`ArrayList` 类使用通用的类 `Object`。Java 类层次结构中的其他所有类都从该类继承。

遗憾的是，能够在 `ArrayList` 类中存储所有类型的对象无法提供用户想要的灵活性。Java 原始类型不是对象，所以不能与这些非常方便的类共同使用。为了解决这个问题，Java 定义了一个类（通常称为包装类）来对应 8 种原始类型，如表 7-2 所示。

表 7-2 原始类型与包装类

原始类型	包装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

每个包装类都有一个构造函数，用来从相对应的原始类型中创建新对象。例如，如果变量 n 是 int 类型，声明

```
Integer nAsInteger = new Integer(n);
```

就会创建一个新的 Integer 对象，其内部值是 n，并将它赋给 nAsInteger。每个包装类都定义方法来检索基本值。方法的名称总是在原始类型的名称加上后缀 Value。因此，初始化变量 nAsInteger 之后，可以回到存储在它里的值：

```
nAsInteger.intValue()
```

因为 nAsInteger 是合法对象，所以可以将值存储在 ArrayList 或其他复合结构里。

与 7.3.1 小节的 EmbeddedInteger 类不同，包装类不提供方法来设置现有对象里基本变量的值。如第 6 章所述，包装类实际上是不变的。不变类有很好的属性，即总是可能考察它们，就像它们表示纯数值一样，这种方法与考察 Java 原始类的方法相同。使用不变类，当将值从一个方法传递到另一个方法时，不用担心该值是否被共享或复制，因为这两种方式都不能改变该值。

包装类也包括几个静态方法，您会发现它们很有用，特别是在 Integer 类里的那些静态方法，它们支持任意基数的数字转换。图 7-20 列出了 Integer 和 Double 类中一些最重要的静态方法：Character 类里最有用的静态方法在第 8 章讨论。因为这些方法是静态的，所以需要包括类名称。因此，要将十进制整数 50 转换为十六进制字符串，代码如下：

```
Integer.toHexString(50, 16)
```

Integer 类里的静态方法

static int parseInt(String str) 或 parseInt(String str, int radix)

使用指定的基数（默认为 10），将数字字符串转换为 int 类型

static String toString(int i) 或 toString(int i, int radix)

使用指定的基数（默认为 10），将 int 类型数据转换为字符串表示法

Double 类中的静态方法

static double parseDouble(String str)

将数的字符串表示法转换为相应的 double

static String toString(double d)

将 double 转换为相应的字符串表示法

图 7-20 Integer 和 Double 类中的数字转换方法

7.3.3 装箱和拆箱

如果使用 Java 的最新版本，通常会忽略原始类型与包装类之间的区别。从 Java 5.0 开始，当语言的语义要求时，编译器可以在原始类型与其相关包装类间自动转换。因此，要求写下面这样的声明：

```
Integer nAsInteger = new Integer(n);
```

Java 5.0 可以将声明简化为：

```
Integer nAsInteger = n;
```

当 Java 5.0 编译器发现用户试图将 int 类型的值赋给声明为 Integer 类型的变量时，在实现赋值之前，它会自动创建一个有合适值的 Integer 对象。相反，如果 Java 希望是 int 类型的值，则使用变量 nAsInteger，Java 5.0 编译器也会自动实现转换。添加或删除所需包装类的过程称为装箱和拆箱。

实现自动装箱和拆箱的好处是可以创建更易于理解的代码。创建包装类的构造函数和从包装对象中选择原始内容的方法完全消失了。当要将原始数据存储为收集的一部分时，这种转换就很有价值，这将在第 13 章讨论。

常见错误

即使使用 Java 5.0，使用自动装箱和拆箱时也要很小心。特别是对象和原始值的运行有时不同，例如当作为参数传递时或用来测试表达式时。记住这一点至关重要。

然而，装箱和拆箱策略也有一些主要不足。一是隐藏原始数据值转换成了对象这一事实，让人很难理解在内存如何表示值。如包括 EmbeddedInteger 类的示例所示，知道作为参数传递的值是对象还是原始数据值有时很重要，因为两者的行为通常大不相同。因为值实际上是不是对象还不够清楚，所以自动装箱和拆箱技术让程序的行为更难理解。

第二点不足是，只有当操作调用的值无法解释时，Java 才应用自动装箱和拆箱。因此，有些 Java 操作调用自动转换，而其他最初似乎相同的操作却没有这样做。这种情况，可以用使用 Java 5.0 的 run 方法说明，代码如下：

```
public void min() {  
    Integer x = 5;  
    Integer y = new Integer(x);  
    println("x == y -> " + (x == y));  
    println("x < y -> " + (x < y));  
    println("x > y -> " + (x > y));  
}
```

方法里的语句创建了两个 Integer 类型的值，它们的值都是 5。赋给 x 的值由自动装箱创建；赋给 y 的值是一个新构造的 Integer 类型数据，它的值来自于 x 的拆箱内容。接下来 3 行分别显示关系表达式 $x == y$ 、 $x < y$ 和 $x > y$ 的值。

如果使用 Java 5.0 编译和运行此程序，输出刚开始看起来很奇怪，如图 7-21 所示。

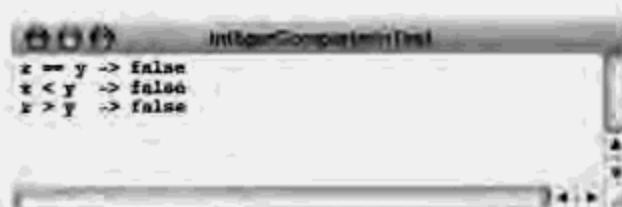


图 7-21 IntegerComparisonTest 的输出

按照数学规则，不管给 x 和 y 赋什么值，只有一个表达式是对的。所有 3 个比较都是 `false` 的原因是，只有使用`<` 和 `>`运算符时才发生对象的拆箱；等于运算符`=`应用于 `Integer` 类型的对象时有自身定义的意义，它表示 Java 不会将它们转换为原始值。应用于对象时，只有当`=`两边的值是相同的对象时，等于运算符才返回 `true`。这里， x 和 y 是具有相同内部值的不同对象，如图 7-22 的堆—堆栈图所示。

图 7-22 x 和 y 具有相同内部值

7.4 链接对象

“对象在内部表示为引用”这句话有多个含义。即使这一话题的充分讨论要等到本书最后，本章也有必要考察这些含义。因为引用很小，所以对象可以方便地包括对其他对象的引用。如果对象以它们出现在堆中的完整形式存储，那么小对象包含大对象就明显不可能。而且，由于每个对象都需要一些系统开销，因此一个堆对象包含另一个相同大小的对象也不可能，因为没有空间存储 Java 需要的记录信息。然而，如果对象存储为引用，就不存在这些限制。从物理上说，小对象不可能包含较大对象这是正确的；但是一个对象包含对其他对象的引用是可以的，不管对象会占用多大的堆存储空间。毕竟，引用只是对象在内存中的地址，因此只用几个字节就可以表示。这样，一个对象包含对不同类的其他对象(甚至是相同类的对象)的引用就不成问题。

创建包含其他对象引用的对象是一种非常强大的编程技术。这种对象称为链接对象。虽然深入讨论链接结构超出了本章的范围，但是考察简单示例会有所帮助，因为示例会强化对内存里表示链接结构方法的理解，同时它有力说明了消息传递的基本概念。Java 使用这个比喻来描述系统内对象之间的通信。当对象需要与其他对象交互时，通过调用接收对象里的方法实现。然而，在面向对象范例中，调用方法的行为通常描述为给接收方“发送消息”。在接收方，该消息可能会依次引起响应，例如状态的改变或生成其他消息，这些消息将此消息传递给其他对象。即使消息传递的实现方式取决于方法调用，但记住基本比喻还是很重要的。

7.4.1 链接结构里的消息传递: Gondor 灯塔

Gandalf 朝他的马儿大声喊到：“Shadowfax！我们必须赶快，时间来不及了。看！Gondor 灯塔都点亮了，他们在求援。战争爆发了。看，Amon Din 上的烽火已经点燃了，Eilenach 上的也点燃了；正在向西传递：Nardol、Erelas、Min-Rimmon、Calenhad 和 Rohan 边界的 Halifirien”

—— J. R. R. Tolkien, *The Return of the King*, 1955

Peter Jackson 采用这一幕布作为它三部曲的结束语，他在电影中创建了历史上最有召唤力最生动的消息传递示例。第一座灯塔位于 Minas Tirith，每个信号塔的看守人看到前面灯塔的烽火点燃后，就点燃自己灯塔的烽火。这样我们看到信号从一个山顶传递到另一个山顶，Gondor 危急的消息向 Rohan 迅速传递，如图 7-23 所示。



图 7-23 消息传递链

使用方法传递范例如何模拟 Gondor 灯塔的点亮过程？假设每个信号塔是一个单独对象，它包含塔名等信息。然而，每座塔必须能够确定链上的下一座塔，以便知道向哪里传递消息。最直接的策略是让每座塔包含对其后继者的引用。这样，表示 Minas Tirith 的对象包含对用于 Amon Din 模型的对象的引用，Amon Din 又包含对 Eilenach 对象的引用，依次类推。如果采用这种方法，每个对象的私有数据必须包括——应用程序作为整体需要的其他信息——下面的实例变量：

```
private String towerName;
private SignalTower nextTower;
```

第一个变量包含塔名，第二个变量是对下一座塔的引用。

该模型对 Minas Tirith 和链中的中间塔有意义。使用这种策略表示 Rohan 就有点问题，因为它是链中最后的一个。和其他塔一样，Rohan 有 nextTower 字段，假设其包含对下一座塔的引用。而实际上 Rohan 没有引用，问题是应该如何表示这种情况。幸运的是，Java 正好为这种情况定义了特定值，称为 null。常量 null 表示对不存在值的引用，可以指派给保存对象引用的任何变量。因此，确保 Rohan 的 nextTower 字段是 null，以说明 Rohan 是链中最后一座塔这一事实。

图 7-24 包含 SignalTower 类的简单定义，它实现了顺着塔链传递消息的思想。构造函数读取塔名和对下一座塔的引用。其他唯一一个公有方法是 signal，它是与塔间传递的“点燃烽火”消息相关的行为。塔接收到 signal 消息时，它点燃自己的烽火，如果需要，它通过将 signal 消息发送给后面一座塔以传递消息。

```

/**
 * This class defines a signal tower object that passes a message
 * to the next tower in a line.
 */

public class SignalTower {

    /**
     * Constructs a new signal tower with the following parameters:
     * @param name The name of the tower
     * @param link A link to the next tower, or null if none exists
     */
    public SignalTower(String name, SignalTower link) {
        towerName = name;
        nextTower = link;
    }

    /**
     * This method represents sending a signal to this tower. The effect
     * is to light the signal fire here and to send an additional signal
     * message to the next tower in the chain, if any.
     */
    public void signal() {
        lightCurrentTower();
        if (nextTower != null) nextTower.signal();
    }

    /**
     * This method lights the signal fire for this tower. This version
     * that simply prints the name of the tower to the standard output
     * channel. If you wanted to extend this class to be part of a
     * graphical application, for example, you could override this
     * method to draw an indication of the signal fire on the display.
     */
    public void lightCurrentTower() {
        System.out.println("Lighting " + towerName);
    }

    /* Private instance variables */
    private String towerName;          /* The name of this tower */
    private SignalTower nextTower;      /* A link to the next tower */
}

```

图 7-24 SignalTower 类的实现

在图 7-24 所示的 SignalTower 类的设计中，点燃当前信号塔的过程是 lightCurrentTower 方法的责任。在这种实现方式中，lightCurrentTower 由下面的代码组成：

```

public void lightCurrentTower() {
    System.out.println("Lighting " + towerName);
}

```

它在 System.out 上打印一条包含塔名的消息。System.out 是 Java 程序中总是可用的标准输出流。将来，可能想重写程序，让烽火在屏幕上依次出现。在那种情况下，可以重写 lightCurrentTower 的实现方式，以便实现相应的图形操作。最好创建新的子类，让它重载 lightCurrentTower 的定义。扩充的类可以取代方法本身，但仍然取决于链接塔的基本类所提供的其他特征。第 9 章可能让示例更生动。

SignalTower 类的构造函数很简单，作用就是将参数复制到相对应的实例变量，代码如下：

```
public SignalTower(String name, SignalTower link) {
    towerName = name;
    nextTower = link;
}
```

因为 `SignalTower` 的构造函数的每个调用都要求指定对链中下一座灯塔的链接，所以，如果以链结尾的 `Rohan` 开始并反向朝链前面的 `Minas Tirith` 运行，就很容易创建表示 `Gondor` 中灯塔链的数据结构。如果将每座塔声明为一个实例变量，下面的方法初始化这些变量来反映信号塔链的结构。

```
private void createSignalTowers() {
    rohan = new SignalTower("Rohan", null);
    halifirien = new SignalTower("Halifirien", rohan);
    calenhad = new SignalTower("Calenhad", halifirien);
    minRimmon = new SignalTower("Min-Rimmon", calenhad);
    erelas = new SignalTower("Erelas", minRimmon);
    nardol = new SignalTower("Nardol", erelas);
    eilenach = new SignalTower("Eilenach", nardol);
    amonDin = new SignalTower("Amon Din", eilenach);
    minasTirith = new SignalTower("Minas Tirith", amonDin);
}
```

7.4.2 链接结构的内部表示法

调用 `createSignalTowers` 初始化灯塔之后，堆内对象的内部表示法如图 7-25 所示。注意信号灯塔出现的顺序由每个结构内最后一个内存单元里引用的链表示，而不是由内存里单元出现的顺序表示。为了强调链里的内部连接，图 7-25 用箭头表示每个单元链接到其后继单元。内存里存储的唯一值是数字地址，它足以确定链里的下一个对象。

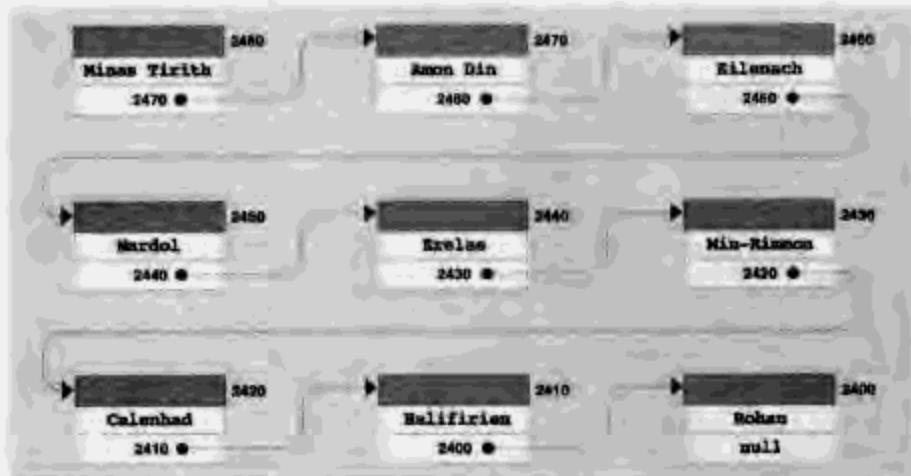


图 7-25 `SignalTower` 链中对象的内存布局

建立图 7-25 所示的数据结构之后，要开始过程，就要调用

```
minasTirith.signal();
```

将 `signal` 消息发送给第一座灯塔。`signal` 方法的实现方式就是模拟点燃 `Minas Tirith` 里的灯

塔, 然后将 signal 消息传递给表示 Amon Din 的对象。对象依次将 signal 消息传递给后继者, 这一过程直到消息到达 Rohan 塔时结束, 此时 nextTower 字段里的 null 表明信号已经到达了链的结尾。

7.5 小结

即使对象模型让数据观点更抽象, 但有效使用对象仍然需要用智力模型来理解内存中如何表示对象。本章介绍了这些对象的存储方法, 讨论了在程序中创建和使用对象时其内部发生的情况。

本章介绍的重点是:

- 现代计算机的基本信息单元是位, 它有两种可能的状态。在内存图中, 位的状态通常表示为二进制数 0 和 1, 也可以依据应用程序将这些值当成 off 和 on, 或者 false 和 true。
- 在硬件内部, 将位序列结合起来形成两种更大的结构: 一种是字节, 它有 8 位; 另一种是字, 它包括 4 个字节或 32 位。
- 计算机内存排列为一系列字节, 其中每个字节由序列中的索引位置(称为地址)确定。
- 计算机科学家倾向于用十六进制(基数为 16)符号编写地址值和内存位置的内容, 因为这样很容易确定单个位。
- Java 中的原始类型需要不同的内存空间。char 类型需要 2 字节, int 类型需要 4 字节, double 类型需要 8 字节。多字节值的地址是它包含的第一个字节的地址。
- 在 Java 程序中创建的数据值分配在 3 个不同的内存区。应用于整个类的静态变量和常量分配在专门用于程序代码和静态数据的内存区。局部变量分配在称为堆栈的区, 堆栈分配在称为框架的结构里, 框架包含方法的所有局部变量。所有对象都分配在称为堆的区, 堆是可用内存池。
- 声明某个对象类的局部变量, 并通过调用其构造函数为其赋初始值时, 堆和堆栈里都要分配内存。堆包括对象中实际数据的存储空间。变量的堆栈条目只包含足够保存对象地址的内存, 称为引用。
- 方法调用的堆栈框架包含由关键字 this 标识的条目, 它将对象标识为方法应用的对象。
- 随着计算继续, 堆上面的对象通常用作临时值, 后续计算不再需要这些值。然而, 这样的值会继续占用堆上的内存, 这些值称为无用单元。Java 运行时系统定期采取无用单元收集程序来搜索堆, 收回这些对象占用的空间。
- 对象从一个方法传递到另一个方法时, 只有对象的引用被复制到新方法的堆栈框架。因为这些引用标识的对象与调用者中的同一引用相同, 所以调用和调用的方法之间共享对象值。
- Java 中的原始类型实际上不是对象, 所以不能在要求是对象的语境中使用。要解决这个问题, Java 定义了一组包装类, 可以将原始类型封装在完全合格的对象里。
- Java 中的对象可以包含对其他对象的引用。这种对象称为链接对象。链接结构在编程中非常有用, 本书稍后将详细讨论它。

7.6 复习题

1. 定义如下术语：位、字节和字。
2. 单词“位”的语源是什么？
3. 384MB 计算机的内存有多少字节？
4. 将下列十进制数转换为相应的十六进制数：
 - (1) 17
 - (2) 256
 - (3) 1729
 - (4) 2766
5. 将下列十六进制数转换为十进制数：
 - (1) 17
 - (2) 64
 - (3) CC
 - (4) FAD
6. 什么是地址？
7. Java 分配给 int 类型的值多少字节？ double 类型需要多少字节？
8. 内存有哪 3 个区，Java 程序中的值可以存储在里面？
9. 利用 7.2 节的示例作为模型，分析执行下面的方法时所出现的堆和堆栈的操作：

```
public void run() {
    Rational x = new Rational(4, 5);
    Rational y = new Rational(5, 2);
    Rational z = x.multiply(y).subtract(z);
    println(x + " * " + y + " - " + y + " = " + z);
}
```

执行到 println 语句时，堆里的哪个对象是元件单元？

10. 判断题：从一个方法到另一个方法传递原始类型值时，Java 总是将该值复制到被调用方法的框架里。
11. 判断题：当从一个方法向另一个方法传递对象时，Java 将对象里的数据复制到新框架。
12. 描述简单的标记——无用单元收集器使用的二阶段。
13. 术语“包装类”是什么意思？包装类的目的是什么？
14. 在整数与其特定基数的字符串表示法之间转换，可以使用什么方法？
15. 什么属性确定链接结构？
16. 假设某类的对象需要内存一些空间，从物理上讲，堆里的对象不可能包含另一个相同类的对象，仍然有空间供其他数据使用。如果要实现相同效果，Java 编程人员可以怎么做呢？
17. 为什么 Java 包含特定值 null 很重要？这个值表示什么意思？

7.7 编程练习

1. 使用静态方法 Integer.parseInt 和 Integer.toString 编写程序，将十六进制值转换为相应的十进制值。程序一直读取十六进制值，直到用户输入 0 为止。程序的运行如图 7-26 所示。

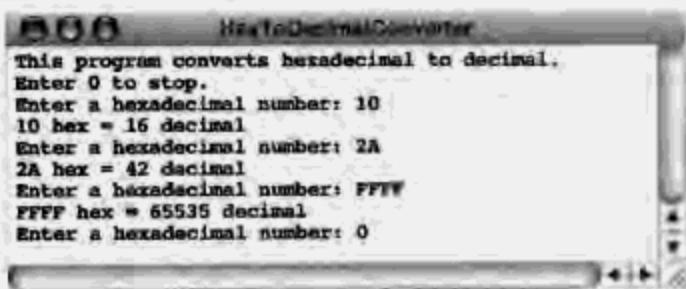


图 7-26 十六进制转换为十进制的样本输出

2. `Integer.parseInt` 方法允许程序将用户输入读取为字符串，然后将它转换为整数。这就有可能编写一个程序，不使用整数——如使用空行——作为表示输入结束的终止条件。重写第 4 章练习 6 中的 `AverageList` 程序，让它使用空行标记输入结束。

3. But don't panic. Base 8 is just like base 10 really—if you're missing two fingers.

—Tom Lehrer. *The New Math*. 1965

重写第 6 章练习 5 中的 `Math Quiz` 程序，提出基数是 8 而不是 10 的问题，如图 7-27 所示。

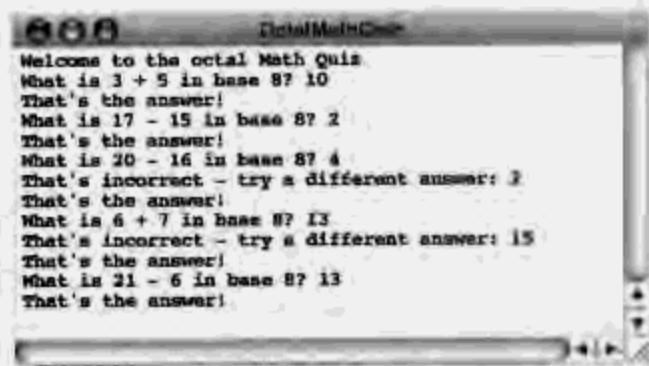


图 7-27 OctalMathQuiz 的输出

4. `java.lang` 程序包里的 `Runtime` 类包括一些简单方法，这些方法有助于更好理解 Java 无用单元收集器的功能。`Runtime` 对象维护有关 Java 虚拟机状态的信息。如果要查看信息，可以调用静态方法 `getRuntime()` 并将结果存储在变量里，进入当前运行环境，代码如下：

```
Runtime myRuntime = Runtime.getRuntime();
```

有了这个变量，调用

```
myRuntime.freeMemory();
```

就可以知道内存还有多少空闲空间可用。因为内存可能很大，`freeMemory` 返回的值是 `long` 类型而不是 `int` 类型，表示可用字节的数量。调用

```
myRuntime.gc();
```

可以明确激活无用单元收集器。编写程序分配 10000 个 `Rational` 对象，不在变量中保存它们，让它们变成无用单元。之后，计算无用单元收集前后的空闲内存大小，使用此差异报告有多少

字节空闲，运行结果如图 7-28 所示。

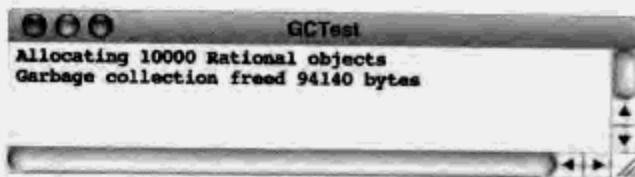


图 7-28 GCTest 的输出

第 8 章

字符串和字符

Surely you don't think numbers are as important as words.

—King Azaz to the Mathemagician
Norton Juster, *The Phantom Tollbooth*, 1961



Herman Hollerith (1860—1929)

以机器可读的形式编码文本这种思想可以追溯到 19 世纪美国发明家 Herman Hollerith 的研究成果。Hollerith 在纽约城市大学和哥伦比亚矿业学校学习工程学，在到美国麻省理工学院任教之前，他在美国人口调查局担任过多年统计员。在人口调查局 Hollerith 相信，机器可以更快更精确地计算调查产生的数据。在 19 世纪 80 年代晚期，他设计建造了用于 1890 年人口普查的编表机。他建立了一家公司使他的发明商业化，这家公司最初称为制表机器公司(Tabulating Machine Company)，1924 年更名为美国国际商用机器公司(IBM)。Hollerith 的打卡编表系统是本章描述的文本编码的前身——FORTRAN 语言的早期版本使用字母 H(表示 Hollerith)来表示文本数据，这一点反映了他的贡献。本书大多数情况下，实际使用的数据——传统上使用数据作为计算的基础——都是用 int 和 double 类型表示的数字数据。Juster 的数学家会说，数固然重要，但是世界上也有许多其他类型的数据。文本数据由出现在键盘和屏幕上的单个字符组成。特别是 20 世纪 80 年代早期个人计算机的发展，计算机使用数字数据比使用文本数据要少。计算机处理文本数据的能力促进了字处理系统、电子邮件、WWW(World Wide Web)、搜索引擎及其他各种有用应用程序的发展。

第 2 章简单介绍了文本数据的概念，它以本书的第一行代码开始，该代码包含字符串：

"hello, world"

此后就偶尔使用字符串值，但每次都是将这些字符串作为整体对待。这种观点非常重要，它确实适用于大多数使用字符串数据的情况。要释放字符串的全部能量，需要知道如何以更成熟的方式操作字符串。

因为字符串由单个字符组成，所以有必要理解字符以及计算机内部表示字符的方法。因此，本章着重讨论 `char` 数据类型及 Java 内置的 `String` 类。在讨论每种类型的细节之前，有必要先全面考察一下计算机内部如何表示非数字数据这个问题。

8.1 枚举的原则

随着计算技术的发展，越来越多的信息以电子形式存储。要在计算机内存储信息，必需以机器可以使用的形式表示数据。特定项的表示法取决于其数据类型。整数在计算机内有一种表示法，这从第 7 章可以知道，它由一序列保存二进制数的位组成。浮点数的表示法又不同，它超出了本文讨论的范围。然而，除了数之外，还有许多类型的有用数据，因此计算机必须也能够表示非数字数据。

要理解非数字数据的本质，可以想一想在一年多的时间内您给相关机构和部门提供的信息。例如，如果您住在美国，要给美国国内税局提供年度纳税申报单数据。许多信息都是数字，如薪水、扣除额、税款和扣交等。有些是文本数据，如姓名、地址和职业。但纳税申报单上的其他项不能以这些形式分类。例如，有个问题是：

- 申报纳税身份(表一)
- 单身
 - 已婚申报共同申报所得稅
 - 已婚申报单独申报所得稅
 - 户主
 - 合格寡妇

表中每个条目的答案都表示数据。然而，回答既不是数字数据也不是文本数据。描述这种数据类型的最好方法是称它为申报纳税身份数据，这是一种全新的数据类型，它的域由 5 个值组成：`single`、`married filing joint return`、`married filing separate return`、`head of household` 和 `qualifying widow(er)`。

可以假设其他许多有相同结构的数据类型。例如，其他表可能会询问性别、种族或学生身份。针对每种情况，可以从可能性列表中选择一个答案，这些可能性构成明显概念类型的域。列举类型域中所有元素的过程称为枚举。通过列举所有元素进行定义的类型称为枚举类型。

因为本章的标题是“字符串和字符”，所以讨论枚举类型似乎不切题。但是，字符在结构上与枚举类型相似，理解枚举类型的运行原理有助于理解字符的运行原理。

然而，枚举类型是个抽象概念。要理解这个概念如何应用于编程，先需要了解计算机内部表示这种值的方法。必须学习在 Java 程序语境中使用枚举类型的方法。下面的两小节将讨论这些问题。

8.1.1 在计算机内部表示枚举类型

美国国内税局核对纳税申报单时，第一步是将纳税申报单的数据输入计算机系统。要存储

这些信息，计算机必须能够表示每种不同的数据项，包括申报纳税身份。如果要开发一种策略记录纳税人的申报纳税身份，该怎么办呢？

解决这个问题需要知道计算机有这种能力。计算机擅长使用整数。作为基本硬件操作，它们可以进行存储、加、减、比较整数以及进行整数的其他各种运算。计算机擅长操作整数，这为解决表示枚举类型问题提供了解决方案。要表示所有类型的有限值，必须给每个值提供一个数。例如，假设有一份允许的申报纳税身份值列表，可以让它们报数，让 single 用 1 表示，married filing joint return 用 2 表示，married filing separate return 用 3 表示，依次类推（实际上，纳税申报表格中直接列出了这些数字代码）。给每个不同的可能性赋一个整数，意思就是可以用整数表示相应的申报纳税身份。

这样，为任何枚举类型定义表示法，就是给其元素编号。给枚举类型的每个元素赋整数的过程称为整数编码——整数表示为原始值的编码。

8.1.2 将枚举类型作为整数表示

在 Java 的较长历史时期中，编程人员都采用整数策略，该整数以可能的最明显方式编码。因为语言缺少对枚举类型的高级支持，可用的唯一机制是为要表示的值定义命名常量。因此，在记录美国国内税局提供的各种申报类别的程序中，可以在 Java 程序中进行如下定义：

```
public static final int SINGLE = 1;
public static final int MARRIED_FILING_JOINT_RETURN = 2;
public static final int MARRIED_FILING_SEPARATELY = 3;
public static final int HEAD_OF_HOUSEHOLD = 4;
public static final int QUALIFYING_WIDOW_OR_WIDOWER = 5;
```

这些定义让用户可以在程序里使用这些名称。例如，如果要实现申报纳税身份是 single 时的一些代码，可以使用下面的 if 语句：

```
if (filingStatus == SINGLE)
```

使用整数常量表示枚举类型时，要注意以下几点：

- 保存枚举值的变量必须声明为 int 类型。很多时候，这种情况令人遗憾，因为值概念上的域决不是 int 类型包含的全部值。无法知道 -1 或 999 的申报纳税身份是什么意思，虽然这些值是合法整数。
- 程序中常量的定义不允许使用输出操作中的常量名称。特别是，如果程序通过如下命令要求申报纳税身份：

```
int filingStatus = readInt("Enter filing status: ");
```

回答中不能输入 MARRIED_FILING_SEPARATELY。

- 显示枚举类型的值没有自动机制，让其名称出现在输出中。如果 filingStatus 的值是 SINGLE，调用

```
println("filing status = " + filingStatus);
```

将只显示一行，此行表示申报纳税身份是 1。

尽管有这些不足，但是枚举类型仍然很有价值，其增加了代码的可读性。

8.1.3 定义新的枚举类型

8.1.2 小节描述的方法在 Java 程序中很常见, 它没有完全利用语言目前提供的性能。Java 5.0 可以定义实际类型名称来表示枚举类型。以最简单的形式, 声明枚举类型的语法如下面语法框所示。

定义枚举类型语法

```
access enum name { elements }
```

其中:

access 是 public 或为空;

name 是新类型的名称;

elements 是用于表示单个值的名称列表, 这些单个值构成了枚举类型。列表中的元素用逗号分开。

例如, 可以定义日期对应的新类型, 如下所示:

```
public enum Weekday {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

或通过写定义

```
public enum Direction {
    NORTH, EAST, SOUTH, WEST
}
```

表示主要罗盘方向的新类型。

在计算机内部, 定义枚举类型的效果与定义命名整数常量的方法相同: 枚举类型的每个元素由整数代码表示。然而, 从编程人员的观点来看, 使用 Java 5.0 策略定义枚举类型有如下优点:

- 编译器能够选择整数代码, 因此免除了编程人员的责任。
- 独立的类型名称通常让程序更易于阅读, 因为声明可以使用富有含义的类型名称而不是使用有多种用途的设计 int 类型。
- 编译器可以检查枚举类型的值是否匹配变量(该值指派给此变量)的声明。
- 枚举值用名称显示。

在程序中使用枚举类型的常量时, 一般需要包含枚举类型名称作为限定符。因此, 如果要声明 Direction 类型的变量 dir, 并将它初始化为值 NORTH, 可以这样写:

```
Direction dir = Direction.NORTH;
```

然而, 有一种情况 Java 不使用枚举类型的名称作为限定符, 如果使用枚举值作为 switch 语句中的表达式, Java 要求在 case 子句中使用 unqualified 元素名称。例如, 如果参数是周末, 下面的方法返回 true:

```
private boolean isWeekendDay(Weekday day)
```

```
    switch (day) {
        case SATURDAY: case SUNDAY: return true;
        default: return false;
    }
```

如果使用 Boolean 逻辑写相同的方法，就需要包含枚举类型名称，代码如下：

```
private boolean isWeekendDay(Weekday day) {
    return day == Weekday.SATURDAY || day == Weekday.SUNDAY;
```

8.2 字符

字符是所有文本数据处理的基础。虽然程序中字符串比单个字符出现得更频繁，但是字符是基本类型它是构造文本数据其他所有形式的“原子”。理解字符如何运行对于理解文本处理的其他问题至关重要。在某种程序上，字符构成嵌入枚举类型，虽然可能字符的完整列表因为太大而不能全部列出。将所有字符的域描述为标量类型更合适，标量类型是可以解释为整数的任何类型。在 Java 中，标量类型特别非常有用，因为在整数可能出现的所有语境中使用它们。例如，标量类型的变量可以用作 switch 语句中的控制表达式。

8.2.1 char 数据类型

Java 使用 char 数据类型表示单个字符，char 类型是预定义数据类型之一。和第 3 章介绍的所有原始类型一样，char 类型也由合法值的域和操纵这些值的操作组成。从概念上说，char 数据类型的域是可以显示在屏幕上或从键盘上输入的符号集。这些符号——字母、数字、标点符号、空格键、Return 键等——都是所有文本数据的积木。

因为 char 是标量类型，所以字符的操作集和整数的一样。然而，理解在字符域里这些操作的意思，需要进一步分析计算机内部表示字符的方法。

8.2.2 ASCII 和 Unicode 编码系统

单个字符在计算机内部的表示法与其他标量类型一样。基本思想是，将这些字符写在列表里，然后逐个地给它们编号，这样可以给每个字符赋一个数。用来表示特定字符的代码称为字符代码。例如，可以用整数 1 表示字母 A，整数 2 表示字母 B，等等。用 26 表示字母 Z 后，可以继续给小写字母、数字、标点符号和其他字符编号为整数 27、28、29 等。

虽然从技术上可以设计一台计算机用数 1 表示字母 A，但这样做肯定是错误。当今世界，信息通常在不同计算机间共享。可以用记忆条将程序从一台计算机复制到另一台计算机，或者让自己的计算机直接与全国或全球网络上的其他计算机通信。为了让这种通信成为可能，计算机必须能够用一种通用语言与其他计算机通信。这种通用语言的基本特征是使用相同的代码表示字符，这样一台计算机上的字母 A 在另一台计算机上不会变成 Z。

在计算早期，不同的计算机实际上使用不同的字符代码。一台计算机上的字母 A 有特定的整数表示法，但在其他制造商生产的计算机上表示法就完全不同，甚至可用字符集也会改变。例如，一台计算机上可能在键盘上有字符 €，而另一台计算机上可能根本无法表示这个字符。

当讲不同语言的人遇到一起，计算机通信就会遇到各种麻烦。

然而，随着时间的推移，计算机有效通信的巨大优势催生了 ASCII 字符编码系统，它是美国信息交换标准码的简称。ASCII 编码系统在 20 世纪 80 年代应用非常广泛，代表了一种标准。然而，与其他语言使用的字符相比，它与英语中使用的字符的联系更紧密。特别是随着 20 世纪 90 年代 WWW 的兴起，它对于将字符域扩充为包含更广泛语言集合必不可少。这种扩充又催生了 Unicode 编码系统，其应用更广泛。Java 是最先采用 Unicode 作为字符表示法的几种语言之一，它更适合于国际架构，今天的计算机都在此架构里操作。

用 Unicode 编码系统阐述枚举原理可能有点困难，因为无法列举所有字符。ASCII 代码只支持 256 个可能的字符，其中只有前面 128 个被完全标准化。这种大小的字符集肯定无法表示所有可用语言，虽然如此，但是形成 ASCII 值表还是很容易。Unicode 有 65 536 个字符空间，虽然不是定义了所有字符。事实证明这种限制还是太小，但对于范围更小的 ASCII 而言，它在灵活性上还是有所改进。

幸运的是，Unicode 的设计师决定合并标准的 ASCII 字符作为 Unicode 集中的前 128 个元素。这些字符及其代码如图 8-1 所示，它显示了 Unicode 表的 ASCII 部分。表中的大多数条目是人们熟悉的键盘上的字符。但也有几个不熟悉的条目，它们的前面用右斜线(\)表示，通常称为反斜线符号，后面紧接着单个字母或数字序列。这些条目称为特殊字符，本章稍后会专门讨论它。

	0	1	2	3	4	5	6	7
00x	\000	\001	\002	\003	\004	\005	\006	\007
010	\b	\t	\n	\013	\f	\r	\016	\017
020	\020	\021	\022	\023	\024	\025	\026	\027
030	\030	\031	\032	\033	\034	\035	\036	\037
040	space	!	"	#	\$	%	&	*
050	()	*	+	.	-	,	/
060	0	1	2	3	4	5	6	7
070	8	9	:	:	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	l	\	l	^	_
140	~	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	}	}	-	\177

图 8-1 Unicode 表的 ASCII 部分

然而，另一方面，此表可能比特殊字符更容易令人混淆。列名为 0~7，而不是在表中用来指定数字代码的 0~9。同样，行名开始以预定比率增长，但接着从 070 直接跳到 100。这是因为 ASCII 和 Unicod 值一般不用十进制(基数为 10)表示，而是用易于转换到数(作为一序列位)

内部表示的基数表示。允许这种方便转换的基数就是 2 的幂，其中最常见的——Java 唯一直接支持的——就是八进制(基数为 8)和十六进制(基数为 16)。图 8-1 使用八进制表示法，其中只有数字 0~7。7.1 节简要介绍了数字基数，现在，对于基于 8 的表示法，每个数位是它右边数位的 8 倍，需要记住这一点。例如，八进制数 177 对应十进制整数：

$$1 \times 64 + 7 \times 8 + 7$$

传统十进制表示法中它是 127。本书中，不是用十进制表示的数都标记有一个表示基数的下标。表示法 177_8 表示八进制值。

在 Java 中，可以用十进制、八进制或十六进制书写整数常量。八进制常量以前置 0 开始；十六进制常量以字符 0x 开始。例如，数 42 可以写为 42、052 或 0x2A。

图 8-1 中每个字符的内部代码是该条目八进制行数和列数之和。例如，字母 A 在表中间，行数为 100，列数为 1。所以字母 A 的 Unicode 值是 $100_8 + 1_8$ ，结果是 101_8 或 65_{10} 。以同样的方法使用此表可以找出所有字符的代码。然而，许多情况下不需要这样做。内部使用数字代码表示字符，虽然知道这一点很重要，但知道什么数字值对应于特定字符通常没有用。输入字母“A”时，嵌入键盘的硬件逻辑自动将此字符转换为 Unicode 值 65，然后发送给计算机。同样，计算机发送 Unicode 值 65 到屏幕时，就出现了字母 A。

8.2.3 字符常量

要在 Java 程序中引用特定字符，标准方法是指定字符常量，字符常量通过在单引号内包含所需的字符表示。例如，字母 A 的 Unicode 表示法，可以写作'A'。Java 编译器知道这种表示法意思就是使用字母 A 的 Unicode 值，即 65。同样，可以写''表示空格，写'9'表示数字 9。注意常量'9'表示字符，不应该与整数值 9 混淆。作为整数，值'9'是 Unicode 表中给出的字符的值，即 71_8 或 57。

常见错误

在程序中要避免使用整数常量表示 Unicode 字符。所有字符常量用单引号包含该字符，例如'A'或''。

由于 Java 在内部将字符表示为它在 Unicode 表中的整数代码，所以大多数情况下可以用整数 65 代替字符常量'A'。程序的运行方式不会变，只是更难让人理解。必须记住，最终其他编程人员要理解您写的程序，除非编程人员记住了 Unicode 代码，在程序中看到整数 65 立即知道它表示字母 A。另一方面，字符常量'A'直接传达这个意思。

图 8-1 可帮助用户更具体理解计算机内部表示字符的方法。只要记住这种思想，就应该忘记特定的字符代码，着重关注字符本身。

8.2.4 Unicode 表示法的重要属性

尽管不用考察特定字符代码这一点很重要，但是也要记住 Unicode 表下面两个结构上的属性：

- 表示数字 0~9 的字符代码是连续的。即使不需要确切知道哪个代码对应于数字代码'0'，但知道数字'1'的代码是次大的整数。同样，如果将 0 与'0'的代码相加，可以得到字符'9'的代码。

- 字母表中的字母分为两个独立范围：一个是大写字母(A~Z)，一个是小写字母(a~z)。然而每个范围内，Unicode 值是连续的，因此可以依次逐个地计算字母。
- 本书后面的程序将会说明每种属性都很有用。

8.2.5 特殊字符

图 8-1 中的大多数字符我们都很熟悉，可以显示在屏幕上。这些字符称为打印字符。然而，Unicode 表也包括一些特殊字符，通常用于控制格式。Java 中，特殊字符使用换码顺序来书写，其组成是反斜线符号后面紧接着字符或一序列数字。图 8-2 列举了预定义换码顺序。

换码顺序	解释
\b	退格
\f	Formfeed(从新的一页开始)
\n	Newline(移到下一行的开始)
\r	Return(不换行返回到行的开始)
\t	Tab(水平移动到下一制表位)
\0	Null 字符(Unicode 值是 0 的字符)
\\\	字符\本身
\\'	字符'(在字符串常量中需要反斜线符号)
\\"	字符"(在字符串常量中需要反斜线符号)
\ddd	Unicode 值是八进制数 ddd 的字符
\uxxxx	Unicode 值是十六进制数 xxxx 的字符

图 8-2 字符和字符串常量中使用的换码顺序

将换码顺序写成常量的一部分，可以在字符串常量里包含特殊字符。虽然一个换码顺序由几个字符组成，但在计算机内部每个顺序都被转换成了单个 Unicode 值。特殊字符的内部代码如图 8-1 所示。

编译器遇到反斜线符号字符时，会将它作为换码顺序中的第一个字符。如果要表示反斜线符号本身，必须在单引号内使用两个连续的反斜线符号，如\"。同样，单引号用作字符串常量时，前面必须有反斜线符号，如\'。

特殊字符也可用于字符串常量。双引号用来表示字符串结束，意思就是说，如果双引号是字符串的一部分，那么必须标记为特殊字符。例如，如果编写的程序包含下面的行：

```
println("\"Bother,\" said Pooh.");
```

输出如图 8-3 所示。

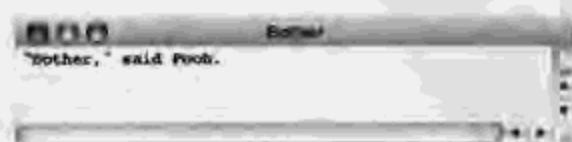


图 8-3 Bother 的输出

Unicode 字符集中的许多特殊字符没有明确的名称，在程序中使用其内部数字代码表示。使用格式\ddd，可以用八进制指定这些字符；使用\uxxxx，可以用十六进制指定这些字符。

8.2.6 字符算法

在 Java 中，可以使用有字符值的算术运算符，就像使用整数一样。不需要使用类型强制转换将字符转换为整数，虽然有时指定这种强制转换让程序更易于理解。在表达式中使用字符时，结果由内部 Unicode 值定义。例如，字符'A'(内部用 Unicode 值 65 表示)，只要它出现在算术语境中，Java 都将它作为整数 65 对待。从整数到字符的转换取决于内部 Unicode 值，并使用相同的策略。在 Java 中，需要使用类型强制转换将整数转换为字符，是因为 char 类型的范围比 int 类型的范围小。Java 的设计师谨慎要求使用明确的强制类型转换，否则在转换过程中可能会丢失信息。

可以使用整数和字符的可转换性，编写 randomLetter 方法，返回随机选择的大写字母。最简单的实现方式是：

```
private char randomLetter() {
    return (char) rgen.nextInt('A', 'Z');
}
```

其中，rgen 是包含 RandomGenerator 的实例变量，如 6.1 节所述。然而，这种实现方式让希望调用 rgen.nextInt 将整数作为参数而不是字符的读者不至于产生混淆。为了这些读者的利益，最好引入类型强制转换明确这些转换，代码如下：

```
private char randomLetter() {
    return (char) rgen.nextInt((int) 'A', (int) 'Z');
}
```

即使对 char 类型的值应用算术运算是合法的，但不是所有操作在该域里都有意义。例如，程序中'A'乘以'B'是合法的。要确定结果，计算机使用内部代码 65 和 66，乘以它们得到 4290。问题是在字符串里该整数没有意义。只有很少一部分算术运算应用到字符时可能有意义。有意义的运算通常有以下几种。

- 整数与字符相加。如果 c 是字符，n 是整数，表达式 c + n 表示编码序列中 c 之后 n 个字符的字符代码。例如，如果 n 介于 0~9 之间，表达式 '0' + n 计算第 n 个数字的字符代码。因此 '0' + 5 计算 '5' 的字符代码。同样，假设 n 介于 1~26 之间，表达式 'A' + n - 1 计算字母表中第 n 个字母的字符代码，从逻辑上说这种运算的结果是 char 类型。
- 从字符中减去整数。表达式 c - n 表示编码序列中 c 之前 n 个字符的字符代码。例如，表达式 'Z' - 2 计算 'X' 的字符代码。从逻辑上说，该运算的结果是 char 类型。
- 从另一个字符中减去一个字符。如果 c1 和 c2 都是字符，表达式 c1 - c2 表示编码序列中这两个字符距离。例如，如果回头再看看图 8-1，并计算每个字符的 Unicode 值，可以确定 'a' - 'A' 的值是 32。更重要的是，小写字母与其对应的大写字母之间的距离是常量，因此 'z' - 'Z' 的值也是 32。从逻辑上说，这种运算的结果是 int 类型。
- 比较两个字符。使用关系运算符比较两个字符是一种常见运算，通常用来决定字母的顺序。例如，如果 Unicode 表中 c1 在 c2 之前，那么表达式 c1 < c2 的值是 true。

要理解这些运算如何应用于实际问题，可以考察计算机如何执行像 readIn 这样的方法。用户输入数(如 102)时，计算机作为字符接收单个按键，因此必须使用输入值 '1'、'0' 和 '2'。因为 readInt 方法必须返回整数，它需要将字符转换为它们表示的整数。为此，readInt 利用了数字在 Unicode 序列中是连续的这一点。

例如，假设 `readInt` 从键盘上读取了一个字符，并将它存储在变量 `ch` 里。通过给表达式

```
ch - '0'
```

赋值，可以将字符转换为其数字形式。假设 `ch` 包含数字字符，其 Unicode 值与数字'0'的 Unicode 值之间的差异必须符合该数字的十进制值。例如，假设变量 `ch` 包含字符'9'。查询 Unicode 表，可以确定字符'9'的内部代码是 57。数字'0'的 Unicode 值是 48， $57 - 48$ 等于 9。关键是方法没有假设'0'的 Unicode 值是 48，意思就是即使有一天 Java 决定使用不同的字符集，相同的方法也可以运行。唯一的假设是数字代码是连续序列。

但是 `readInt` 如何确定字符 `ch` 实际上是不是数字呢？`readInt` 再次利用了数字在 Unicode 表中是连续的这一点。语句

```
if (ch >= '0' && ch <= '9')
```

将数字字符与剩下的 Unicode 集区分开来。同样，语句

```
if (ch >= 'A' && ch <= 'Z')
```

标识大写字母。

```
if (ch >= 'a' && ch <= 'z')
```

标识小写字母。

8.2.7 Character 类中有用的方法

巧的是，在典型的 Java 程序中，前一节结尾通常不会使用 `if` 语句。检查字符是数字还是字母的操作非常普遍，以至于 Java 设计师将它们设计成了 `Character` 类中的方法。`Character` 类在程序包 `java.lang` 中定义，因此在所有没有 `import` 语句的 Java 程序中都可用。

`Character` 类声明了几个用来操作字符值的有用方法，最重要的如图 8-4 所示。和 `Math` 方法一样，这些方法声明为静态方法，意思就是它们不能操作对象，而是更像数学中的传统函数那样运行。例如，可以将小写字母字符 `ch` 转换为其相对应的大写字母，代码如下：

```
ch = Character.toUpperCase(ch);
```

虽然 `toLowerCase` 和 `toUpperCase` 在 `Character` 类里可用，但如果能从 scratch 中实现它们，就能够更好理解它们的操作。可以再次忽略包含的实际 Unicode 值，只依赖于字母是连续的这一事实即可。如果 `ch` 包含大写字母的字符代码，在区分大写字母字符和小写字母字符的值里加上常量差额，就可以将它转换成小写字母形式字符。然而，除了将差额写成明确的常量，如果使用字符算法将它表示为'a'-'A'，程序会更具可读性。因此，可以实现 `toLowerCase` 方法，代码如下：

```
public static char toLowerCase(char ch) {
    if (ch >= 'A' && ch <= 'Z') {
        return ch + 'a' - 'A';
    } else {
        return ch;
    }
}
```

static boolean isDigit(char ch)	确定指定的字符是不是数字
static boolean isJavaIdentifierPart(char ch)	确定指定的字符是不是 Java 标识符的一部分
static boolean isJavaIdentifierStart(char ch)	确定指定的字符能否作为 Java 标识符的第一个字符
static boolean isLetter(char ch)	确定指定的字符是不是字母
static boolean isLetterOrDigit(char ch)	确定指定的字符是字母还是数字
static boolean isLowerCase(char ch)	确定指定的字符是不是小写字母字符
static boolean isUpperCase(char ch)	确定指定的字符是不是大写字母字符
static boolean isWhitespace(char ch)	确定指定字符是不是空白(打印时不可见, 像空格或制表符一样)
static char toLowerCase(char ch)	如果小写存在, 将 ch 转换为小写; 如果不存在, 原样返回 ch
static char toUpperCase(char ch)	如果大写存在, 将 ch 转换为大写; 如果不存在, 原样返回 ch

图 8-4 Character 类中有用的静态方法

方法 `toUpperCase` 的实现方式与此相同。假设示例要匹配 `Character` 类里的实现方式, 方法声明为 `public` 而不是传统的 `private`: `Character` 类中相应的方法也必须是公有的, 否则无法调用它。同样, 必须将方法声明为 `static`, 和在 `Character` 类里一样。

即使 `Character` 类中定义的方法很容易实现, 但是最好使用库方法, 而不是使用自己编写的方法。这样做有 4 个主要理由。

- 由于 `Character` 类里的方法是标准的, 因此其他编程人员可以很方便地阅读您编写的程序。假设这些编程人员都有使用 Java 的经历, 就会识别出类里的方法, 并知道它们的确切含义。
- 依靠库方法而不是自己编写的方法更容易更正程序。因为客户编程人员使用 Java 库, 对于实现者来说正确使用方法会很谨慎。如果自己重写库方法, 很可能在程序中引入故障。
- 自己编写的代码不能利用 Unicode 的国际适用性。前面 `toLowerCase` 的自我实现方式必须假设字符是罗马字母表, 而这不是一个合理假设。其他字母表也有人写字母字符和小写字母字符, `Character` 类里的实现方式也知道如何转换这些字母表中的字符。
- 库程序包里方法的实现方式通常比自己写的实现方式更有效。这些更有效的实现方式如何运行超出了本章讨论的范围, 但重要的是, 使用库形式可以利用附加功效。

8.2.8 包含字符的控制语句

因为 `char` 是标量类型, 所以能够以任何语句形式使用它(整数也以这种语句形式出现)。例如, 如果 `ch` 声明为 `char` 类型, 就可以使用下面的标题行执行 26 次循环, 字母表中的大写字母每个一次。

```
for (char ch = 'A'; ch <= 'Z'; ch++)
```

同样, 可以在 `switch` 语句中使用字符作为控制语句。例如, 如果其参数是英语语言中的元

· 音字母，下面的断言方法返回 true。

```
private boolean isEnglishVowel(char ch) {
    switch (Character.toLowerCase(ch)) {
        case 'a': case 'e': case 'i': case 'o': case 'u':
            return true;
        default:
            return false;
    }
}
```

注意，实现方式使用 `toLowerCase` 方法来识别大写字母形式和小写字母形式的元音字母。

8.3 作为抽象概念的字符串

使用字符的真正动力来自于可以将它们串在一起形成称为字符串的一系列字符。在 Java 中，字符串由名为 `String` 的始终作为导入内容的 `java.lang` 程序包中的类表示。如果每个表示一系列字符的名词以其 Java 形式出现，`String` 类的英语解释就很难理解，所以通常抽象地讨论字符串，称它们为没有大写字母 S，也没有特殊代码字体的字符串。要记住，字符串——作为抽象概念——在 Java 中由名为 `String` 的类表示。正如本章前言所述：从本书的第一段程序开始就在使用字符串显示消息。然而，为了理解字符串给编程带来的巨大力量，需要更深入地学习它。要整体理解字符串，必须从不同层次多角度地考察它们。

8.3.1 从整体和简化论的观点考察字符串

在本书中很多时候都可以从简化和整体的观点分析编程。关心数据表示法的内部细节时，采用的是简化观点。从这个观点出发，任务是理解字符如何存储在计算机内存里，如何存储这些字符让它们形成字符串，以及 200 个字符的字符串如何存入可保存 2 个字符的字符串变量中。这些都是有趣的问题，在适当时候都会知道答案。然而，如果从整体的观点考察字符串，任务是理解如何作为单个逻辑单元来操作字符串。着重考虑字符串的抽象行为，可以知道如何有效地使用它们而不为细节所牵绊。

在某种程度上，Java 要求使用整体的观点对待字符串，因为 `String` 类很少访问基本表示法。即使这样，还是有必要比较一下考虑单个字符的简化观点和不考虑单个字符的整体观点。本书到目前为止，总体上是以整体观点在处理问题。本章其余部分将讨论 `String` 类中的方法，它们既以字符形式操作，又以整体字符串的形式操作。

8.3.2 抽象类型的概念

`String` 类的主要优势——通常在面向对象类定义中——是其定义可能将字符串作为抽象类型，在这种抽象类型中，基本操作只由其行为定义，而不是依据基本表示法定义。Java 中，抽象类型与类定义对应，这些类定义指定可以在该类型对象上实现的操作。特定抽象类型的合法操作称为原始操作，被定义为公有方法。类的实现方式中隐藏了这些操作如何运行，信息在内部如何表示等细节。客户想操作抽象类型的值时，必须使用类提供的方法。

在字符串语境中，想实现哪些原始操作？开始时已经知道如何在程序里指定字符串常量，如何使用串联结合字符串，使用 `println` 和 `readLine` 方法输入和输出字符串。还要做什么呢？例如，使用字符串时，可能还想实现如下操作：

- 找出字符串的长度。
- 选择字符串中的第 i 个字符。
- 摘录一段字符串，形成较短的字符串。
- 确定两个字符串是否包含相同字符。
- 比较两个字符串，看按字母顺序哪个先出现。
- 确定字符串是否包含特殊字符或字符集。

还有其他操作，但这里只提供了一些有趣和有用的操作。这些操作都由 `String` 类里的方法实现，该类提供了使用字符串所需的工具(而不需要理解基本表示法的细节)。不需要理解这些细节是数据抽象的本质。

8.4 使用 `String` 类中的方法

如第 6 章所述，了解类及其方法的最好途径是参考 javadoc 文档。如果这样做，就会发现 `String` 类实现许多方法，其中许多可能从来没有机会使用。最常见的如图 8-5 所示，下面将具体介绍。

<code>int length()</code>	返回字符串长度
<code>char charAt(int index)</code>	在指定索引中返回字符串
<code>String concat(String s2)</code>	将 <code>s2</code> 串联到接收方的结尾，接收方原封不动，返回新字符串
<code>String substring(int p1, int p2)</code>	返回从 <code>p1</code> 开始扩充到 <code>p2</code> (不包括 <code>p2</code>)的子字符串
<code>String substring(int p1)</code>	返回从 <code>p1</code> 开始一直扩充到字符串结尾的子字符串
<code>String trim()</code>	返回删除所有字符串首尾空白的子字符串
<code>boolean equals(String s2)</code>	如果 <code>s2</code> 等于接收方，返回 <code>true</code>
<code>boolean equalsIgnoreCase(String s2)</code>	如果 <code>s2</code> 等于接收方，忽略大小写，返回 <code>true</code>
<code>int compareTo(String s2)</code>	返回一个数值，其符号指出如何按词典顺序比较字符串
<code>int indexOf(char c) or indexOf(String s)</code>	返回字符或字符串的第一个索引；如果不存在，返回 -1
<code>int indexOf(char c, int start) or indexOf(String s, int start)</code>	和有一个参数的 <code>indexOf</code> 一样，但从指定的索引位置开始
<code>boolean startsWith(String prefix)</code>	如果字符串以指定的前缀开始，返回 <code>true</code>
<code>boolean endsWith(String suffix)</code>	如果字符串以指定的后缀结束，返回 <code>true</code>
<code>String toUpperCase()</code>	将字符串转换为大写字母
<code>String toLowerCase()</code>	将字符串转换为小写字母

图 8-5 `String` 类中有用的方法

阅读这些方法的描述，重要的是记住 String 类是不变的，即所有方法都不会改变内部状态。对于许多用户而言，这种行为——实际上非常有用——似乎很反常。发现图 8-5 中包含 toLowerCase 方法，许多同学希望行



```
str.toLowerCase()
```

会将字符串 str 中的字符转换为小写字母。然而，如故障符号所示，这种想法不正确。toLowerCase 方法实际做的是返回新的字符串，新字符串中已经实现了转换。因此，要改变存储在字符串变量 str 里的值，让其中的所有字母以小写形式出现，需要使用赋值语句，例如：

```
str = str.toLowerCase();
```

8.4.1 确定字符串长度

编写程序操作字符串时，通常需要知道特定字符串包含多少个字符。字符串中的字符总数——包括所有字母、数字、空格、标点符号和特殊字符——称为字符串长度。

使用 String 类，通过调用方法 length 可以获得字符串 s 的长度。例如，本书中的第一个字符串是

```
"hello, world"
```

该字符串长度为 12——单词 hello 中的 5 个字符，world 中的 5 个字符，1 个逗号，1 个空格。因此，如果将此字符串赋给变量 message，代码如下：

```
String message = "hello, world";
```

利用

```
message.length()
```

可以计算其长度。如示例所示，String 类使用面向对象的接收方表示法。如果要确定 String 变量的长度，可以给字符串发送一条消息，让它报告长度。该响应作为方法的值返回。

8.4.2 从字符串中选择字符

在 Java 中，字符串内的位置编号从 0 开始。例如，字符串 “hello, world” 中的单个字符编号如图 8-6 所示：

h	e	l	l	o	,		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10	11

图 8-6 “hello world” 中的单个字符编号

字符串中每个字符下面的位置号码称为索引。

为了能够依据字符的索引选择字符串中的特殊字符，String 类提供方法 charAt，它用整数表示索引，返回字符。例如，如果变量 str 包含字符串“hello, world”，调用

```
str.charAt(0)
```

返回字符'h'。同样，调用 str.charAt(5) 返回'.'。要记住，Java 从 0 而不是 1 开始给字符编号。假设 str.charAt(5) 返回字符串中的第 5 个字符，str.charAt(5) 返回在索引位置 5 上的字符，英语中给字符位置编号时，它是第 6 个字符。

8.4.3 串联

字符串最有用的操作之一是串联，就是首尾相连、不带中间字符地将两个字符串结合起来。从图 8-5 可以看出，String 类实现执行此操作的 concat 方法。例如，如果 str 包含"hello, world"，下列语句可以生成在末尾包含感叹号的新字符串。

```
str.concat("!")
```

然而，方法调用只返回新的字符串值而没有改变变量 str 的值，注意到这一点非常重要。要使用 concat 方法，代码如下：

```
str = str.concat("!");
```

Java 中，没有人真正使用 concat 方法，因为其功能以+运算符的形式内置在 Java 中。因此，很可能看到的是

```
str = str + "!";
```

或更简洁的

```
str += "!"
```

不管哪种形式，串联总是中间不带空格地结合两个字符串。如果要在表示为字符串值的两个单词之间放入空格，必须实现其他串联操作。例如，如果变量 word1 包含"hello"，变量 word2 包含"world"，需要编写

```
word1 + " " + word2
```

得到字符串"hello world"。也可以写表达式让它明确调用 concat 方法：

```
word1.concat(" ").concat(word2)
```

这个难看的表达式很快让您相信+运算符的价值。

+运算符也有很多属性，可以将所有不是字符串的操作数转换为字符串表示，因此可以这样写语句：

```
println("The answer is " * answer);
```

不管变量 answer 是什么类型，语句都会运行。Java 将两个值串联起来之前，将 answer 转换为其字符串表示法。

常见错误

使用+运算符指定串联时，要考虑 Java 的优先级规则。不考虑优先级可能会产生一些奇怪的结果。

写包含+运算符的表达式时，需要记住 Java 的优先级规则。语句

```
println("The answer is " + 6 * 7);
```

中，*运算符在+运算符之前赋值，因此输出如图 8-7 所示。



图 8-7 ConcatenationPrecedence 的输出(1)

乍一看，它似乎与语句

```
println("The answer is " + 15 + 25);
```

一样运行，并产生相同的输出。然而，这里依据 Java 的优先级规则，+运算符的两个实例从左到右赋值。结果是字符串 15 首先与字符串串联，然后是字符串 25 与之串联，生成的输出如图 8-8 所示。



图 8-8 ConcatenationPrecedence 的输出(2)

作为使用串联的简单示例，下面的方法返回字符串，该字符串由作为第二参数传递的字符串指定数量的副本组成：

```
private String concatNCopies(int n, String str) {
    String result = "";
    for (int i = 0; i < n; i++) {
        result += str;
    }
    return result;
}
```

如果要在控制台输出中生成分隔符，方法特别方便。例如，语句

```
println(concatNCopies(72, "-"));
```

在一行打印 72 个连字号。

在某种程度上，concatNCopies 中使用的实现策略与第 5 章 factorial 方法中使用的策略相同。在这两个示例中，for 循环的每个周期内，方法使用局部变量来记录中间计算结果。在 concatNCopies 方法中，for 循环的每个周期将 str 的值串联到前面 result 值的后面。因为每个周期都将 str 的副本添加到 result 的后面，n 个周期之后，result 的最终值一定由该字符串的 n 个副本组成。

在 factorial 和 concatNCopies 这两个方法中，用来保存结果的变量的初始化值得注意。在 factorial 方法中，result 变量初始化为 1，因此随着计算的进行，它乘以 i 的每个后续值就是结果。在 concatNCopies 方法中，相应的语句初始化字符串变量 result，因此它通过串联不断发展。循环的第一个周期完成之后，变量 result 必须是字符串 str 的一个副本。因此，第一个周期之前，result 必须包含字符串的 0 个副本，意思就是里面没有字符。根本没有字符的字符串称为空字符串，在 Java 中使用连续双引号("")表示。在现有字符串变量的基础上通过串联连续部分来构造新字符串时，应该将变量初始化为空字符串。

8.4.4 摘录部分字符串

串联让较短的字符串形成较长字符串，而人们常做的是反过来：将字符串划分为它所包含的较短片段。由长字符串的一部分组成的字符串称为子字符串。String 类提供方法 substring，它在字符串中采用两个参数表示索引位置，调用 s.substring(p1, p2) 的结果是摘录 s 中位置 p1 到 p2(不包括 p2)之间的字符。因此，如果 str 包含字符串 "hello world"，方法调用

```
str.substring(1, 4)
```

返回字符串"ell"。由于 Java 中的编号从 0 开始，因此索引位置 1 上的是字符'e'。

substring 方法中的第二个参数是可选的。如果没有它，子字符串返回从第一个参数指定的索引位置一直到字符串结束的字符。

作为使用 substring 的示例，方法 secondHalf(s) 返回子字符串，该子字符串由 s 中的后一半字符组成，如果字符串长度是奇数，还包含中间字符。

```
private String secondHalf(String str) {
    return str.substring(str.length() / 2);
```

8.4.5 比较两个字符串

在编程过程中，许多时候需要核对两个字符串是否有相同的值。此时，几乎肯定会以如下所示不正确的形式进行编码：

```
if (s1 == s2) ...;
```



这里的问题是，Java 中字符串是对象，而==这样的关系运算符只为 int 和 char 等原始 Java 类型定义于传统数学中。更有趣的是，== 运算符在用于字符串时起了作用，但不是最初想要的。对于对象而言，== 运算符测试两边是否为相同的对象。然而，通常要做的是测试两个不同的 String 对象是否有相同的值。

常见错误

在 Java 中比较两个字符串值时，要使用 equals 和 compareTo 方法，不要使用关系运算符。编译器检测不到这种错误，但测试不会产生预期的结果。

要实现想要的结果，String 类定义 equals 方法，可用来测试两个字符串的相等性。和 String 类中的其他方法一样，equals 方法用于接收方对象，意思就是等式测试的语法如下：

```
if (s1.equals(s2)) . . .
```

另一种方法 `equalsIgnoreCase` 可以不考虑字母大小写的区别核对两个字符串量否相等。

有时也会发现有必要确定两个字符串在字母表中的关系。为此，`String` 类提供了方法 `compareTo`，可以用典型的基于接收方形式调用 `compareTo` 方法。

```
s1.compareTo(s2)
```

调用的结果是一个整数，它的符号表示两个字符串间的关系：

- 如果按字母表顺序 `s1` 在 `s2` 之前，`compareTo` 返回负数。
- 如果按字母表顺序 `s1` 在 `s2` 之后，`compareTo` 返回正数。
- 如果两个字符串相同，`compareTo` 返回 0。

如果要确定按字母表顺序 `s1` 是否在 `s2` 之前，代码为

```
if (s1.compareTo(s2) < 0) . . .
```

计算机使用的“字母表顺序”在某些方面不同于字典使用的顺序。`compareTo` 比较两个字符串时，使用基本字符代码确定的数字顺序进行比较。此顺序称为词典顺序，与传统的字母表顺序有些不同。例如，在字母表索引中，会发现 `aardvark` 条目在 `Achilles` 条目之前，因为传统的字母表顺序不区分大写字母和小写字母。如果将 `compareTo` 应用于字符串“`aardvark`”和“`Achilles`”，方法只比较 Unicode 值。在 Unicode 中，小写字母“`a`”在大写字母“`A`”之后，在词典顺序中，先出现字符串“`Achilles`”。方法调用

```
"aardvark".compareTo("Achilles")
```

返回正数。

调用 `compareTo` 时，它从每个字符串中的第一个字符开始比较字符串。如果这些字符不同，`compareTo` 会考察这两个字符值在 Unicode 序列中的相互关系，并返回一个整数来说明结果。如果第一个字符匹配，`compareTo` 继续比较第二个，直到它发现不同为止。如果 `compareTo` 比较完了这两个字符串中的某一个，就自动认为那个字符串在较长的字符串之前，和传统的字母表顺序一样。例如，

```
"abc".compareTo("abcdefg")
```

返回一个负数。只有两个字符串完全匹配并在相同的位置结束，`compareTo` 才返回值 0。

8.4.6 在字符串内搜索

有时需要搜索字符串，看它是否包含特殊字符或子字符串。为此，`String` 类提供一种称为 `indexOf` 的方法，它有几种形式。该调用最简单的形式是如下语句：

```
int pos = str.indexOf(search);
```

其中，`search` 是正在搜索的对象，可以是字符串或字符。调用 `indexOf` 时，方法搜索字符串 `str` 寻找第一个搜索值。如果找到搜索值，`indexOf` 返回匹配开始的索引位置。如果字符直到字符串结尾都没有出现，`indexOf` 返回值 -1。

`indexOf` 方法也使用随意的第二参数表示开始搜索的索引位置。`indexOf` 方法两种风格的

结果如下所示，假设变量 str 包含字符串 "hello, world":

str.indexOf('o')	returns 4
str.indexOf('o', 5)	returns 8
str.indexOf('z')	returns -1

和字符串比较一样，搜索字符串区分大写字母字符和小写字母字符。

可以使用 indexOf 来实现生成只取首字母的缩写词的方法，它是一个新单词，由一系列单词的首字母组成。例如，单词 scuba 就是由 self contained underwater breathing apparatus 的首字母形成的首字母缩写词。方法 acronym 使用由单个单词组成的字符串，返回首字母缩写词。因此，调用方法

```
acronym("self contained underwater breathing apparatus")
```

会返回 "scuba"。

假设单词被单个空格分开，且没有无关字符，acronym 的实现方式可能是选择第一个字母，然后进入搜索每个空格的循环。如果找到一个，它将下一个字符串连接用来保存结果的字符串变量的结尾。如果字符串中没有其他空格，首字母缩写词就完成了。该策略可以转换为 Java 实现方式，代码如下：

```
private String acronym(String str) {  
    String result = str.substring(0, 1);  
    int pos = str.indexOf(' ');  
    while (pos != -1) {  
        result += str.substring(pos + 1, pos + 2);  
        pos = str.indexOf(' ', pos + 1);  
    }  
    return result;  
}
```

Java 中的 String 类也包含两个有用的方法，可以用来核对字符串是否以特殊子字符串开始或结束。如果调用的字符串以作为参数传递的字符串开始，startsWith 方法返回 true。因此，对于下列代码，只有当用户输入以大写或小写字母 Y 开始的答案时，才执行 if 语句的主体。String 类也包含相应的 endsWith 方法，当字符串以特定的子字符串结束时，它返回 true。

```
String answer = readLine("Would you like to play a game?");  
if (answer.startsWith("y") || answer.startsWith("Y")) {  
    . . . code to play the game . . .  
}
```

8.4.7 大小写字母转换

String 类包含两个方法：toUpperCase 和 toLowerCase 这两个方法可以将所有字母表字符转换为指定的大小写。例如，如果 str 包含字符串 "hello, world."，调用方法

```
str.toUpperCase()
```

返回字符串 "HELLO, WORLD."。字符串中的所有非字母表字符——如本示例中的逗号、空格和句点——保持不变。

8.5 字符串处理案例研究

为了更好理解在 Java 中如何实现字符串处理问题，本节讨论 ConsoleProgram 的开发，它逐行读入用户输入的文本，然后将每个单词从 English(英语)转换为 Pig Latin(儿童黑话)。Pig Latin 是一种创造性语言，它根据如下简单规则转换每个英语单词：

- 如果单词以辅音字母开始，将词首的辅音字母字符串(第一个元音字母前的所有字母)从单词的开头移动到末尾，然后加上后缀 ay，这样就形成了它的 Pig Latin。
- 如果单词以元音字母开始，只需要添加后缀 way 即可。

例如，假设单词是 `scram`，因为单词以辅音字母开始，可以将它分成两个部分：一部分由第一个元音字母前的字母组成，一部分由元音和剩下的字母组成：

`s c r` `a m`

然后互换这两个部分，在结尾添加 `ay`，创建 Pig Latin 单词 `amscray`，如下所示。

`a m` `s c r` `a y`

如果单词以元音字母开始，如 `apple`，只要在结尾添加 `way` 即可，这样就变成了 `appleway`。

8.5.1 应用自顶向下设计

由于问题是将英语文本整行转换成 Pig Latin，程序应该能够生成如图 8-9 所示的结果。

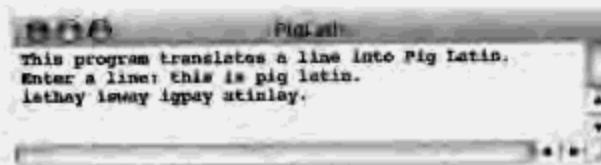


图 8-9 Pig Latin 输出(1)

解决大多数编程问题，最好的方法是应用自顶向下设计，如 5.4.1 小节所述。使用自顶向下设计，可以从 `run` 方法开始，然后使用一系列辅助方法，每个辅助方法解决整个问题的一个简单方面。在最初阶段，定义 `run` 方法作为一系列需要实现的高级步骤。虽然通常可能直接编码步骤作为方法调用，但通常用英语更容易写。例如，只要在纸上设计程序，就可以使用伪代码编写 Pig Latin 程序的大概草稿：

```
public void run() {
    Tell the user what the program does.
    Ask the user for a line of text.
    Translate the line into Pig Latin and print it on the console.
}
```

即使伪代码对于编译器没有意义，但是对编程人员而言它很有用，因为它能够记录逐步细化的过程。写出作为一系列英语步骤的完整程序之后，可以回到伪代码语句，取代实现它们所需的实际 Java 代码。例如，在这种情况下，很容易将前两个伪代码语句转换为 Java 语句，因为它们匹配从第 2 章就开始使用的编程术语。填充这两个语句的细节之后，程序的伪代码如下：

```

public void run() {
    println("This program translates a line into Pig Latin.");
    String line = readLine("Enter a line: ");
    Translate the line into Pig Latin and print it on the console.
}

```

结果仍然是伪代码，但也有所进步。剩下的英语语句很难编码，因此最好应用逐步细化的方法，用实现英语句子的新方法取代伪代码行，这里，需要一种方法“将行转换成 Pig Latin”。可以选择命名 translateLine。使用此名称后，可以完成 run 方法的实现，代码如下：

```

public void run() {
    println("This program translates a line into Pig Latin.");
    String line = readLine("Enter a line: ");
    println(translateLine(line));
}

```

在这种细节层面上，程序非常简单：让用户知道程序在做什么，读入一行文本作为 Java 字符串，调用 translateLine，将字符串转换为 Pig Latin 单词，最后将转换过的行打印到控制台。

虽然没有写 translateLine，但从调用者的观点来看，也可以讨论一下其行为。因为它从调用者读入字符串并返回新的包含转换行的字符串，所以 translateLine 的标题行一定如下：

```
private String translateLine(String line)
```

8.5.2 实现 translateLine

此时可以开始实现 translateLine，它仍然很复杂，需要进一步分解。编程中通常就是这样，有许多策略可以这样做，其中有些更好些。然而，大多数情况下，分解没有特殊策略也非常“正确”。通常需要考察细分问题的几种方法，并努力平衡各个方面。第一个分解策略通常会进入死胡同，让您退回来重新开始。这也是编程过程的一部分。

在实现 translateLine 的过程中，需要解决如何将字符串划分为单词的问题，将每个单词转换为 Pig Latin，然后重新结合单词形成转换字符串。问题的语句建议使用下面概念上的分解：

```

private String translateLine(String line) {
    Divide the line into words.
    Translate each word into Pig Latin.
    Concatenate each of the translated words together.
    Return the concatenated string as a result.
}

```

理论上这种分解是合理的，但也会产生某些现实问题。第一步要求存储单词的整个列表（这在第 12 章讨论，之前不知道怎样做），然而，如果再仔细考察一下问题，就会发现不需要立即记录所有单词。找到一个单词后，可以转换该单词，然后立即将它串联到保存转换行的字符串。这种思想提出了第二种策略：

```

private String translateLine(String line) {
    Initialize a variable to hold the result and set it to the empty string.
    while (there are any words left on the line) {
        Get the next word.
        Translate that word into Pig Latin.
    }
}

```

```

    Append the translated word to the result variable,
}
Return the value of the result variable to the caller
}

```

该策略的伪代码版本中丢失了一些细节，但整体观点还是对的，避免了记录单词整个列表的问题。

8.5.3 考虑空格和标点符号

`translateLine` 的伪代码版本中使用的策略有点小问题。例如，假设程序运行时，用户输入如图 8-10 所示的输入行，

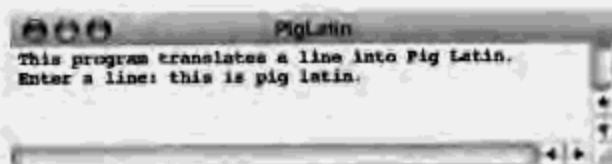


图 8-10 Pig Latin 输入

如果确信输入是 4 个单词“this”、“is”、“pig”和“latin”，假设所有英语步骤像想象的那样正确进行，程序的输出如图 8-11 所示。

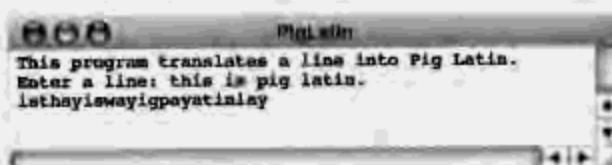


图 8-11 Pig Latin 输出

该输出不是真正想要的结果。所有单词都挤在一起，因为空格和标点符号消失了。`translateLine` 的伪代码版本不考虑空格和标点符号。另一方面，也没有问题的原始英语语句。问题没有完全确定。

编程的实际情况之一是问题的英语描述通常不完全指定。作为编程人员，通常会被某个细节绊倒，问题的制造者没有注意到这些细节或者认为它们太明显了不值得一提。在某些情况下，省略是很严谨的，必须与指定编程任务的人讨论。然而，许多情况下，必须自己选择一种似乎合理的策略。

然而，决定什么是合理的策略有时很棘手。这里，忽略其他标点符号，确定在输出中的每个单词之间打印一个空格。在此语境中，该策略很简单，也很合理。另一方面，它可能不是最好的策略。标点符号让输出更具有可读性。因为标点符号和空格有意义，所以输出中最好在它们出现的地方显示它们。这样，输出可能如图 8-12 所示。

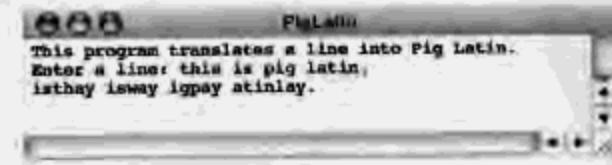


图 8-12 Pig Latin 输出

有许多方法可以重新设计程序，让标点符号正确出现在输出中。例如，改变 run 方法中的循环，让它逐个字符而不是逐个单词地读取字符串。如果使用这种策略，实现的伪代码的结构如下：

```
private String translateLine(String line) {
    Initialize a variable to hold the result and set it to the empty string.
    for (int i = 0; i < line.length(); i++) {
        if (the ith character in the line is some kind of separator) {
            Append that character to the result variable.
        } else if (the ith character is the end of a word) {
            Extract the word as a substring.
            Translate the word to Pig Latin.
            Append the translated word to the result variable.
        }
    }
    Return the value of the result variable to the caller.
}
```

经过努力，可以使用该策略（在本章的练习 17 中有机会这样做）。然而，这样的策略也有一些不足。一是程序结构很复杂，而原始的伪代码设计比较简短，因为它允许在较大单元中使用字符串。

然而，更严重的问题是分解。前一节伪代码的版本包含英语语句

Get the next word.

最近的版本中没有这条语句。作为编程人员，会意识到操作“取下一单词”是有用的通用工具，它远不只应用于简单的 Pig Latin 程序。许多问题要求将文本分解为单词。如果能开发实现此操作的通用方法，那么解决这些问题就有极大优势。另一方面，简单地读取下一个单词不能解决标点符号问题。要在当前应用程序中有用，返回下一个单词的方法必须也能够返回空格和标点符号。

调整 translateLine 伪代码版本的最简单方法是重新定义构成单词的方法。如果看到输入行

this is pig latin.

可能只看到 4 个单词 this、is、pig 和 latin。也可以选择将这一行看成由 8 个单独部分组成，如下所示。

this □ is □ pig □ latin □ .

以这种观点看到这一行，空格和标点符号与单词一样被解释为明确实体。在计算机科学中，作为相关单元的一系列字符称为令牌。因此，上面每个方框内的单元都构成令牌。

将空格和标点符号作为单独令牌，就很容易修改 translateLine 策略，让这些字符包含在返回字符串内。修订后的伪代码策略如下：

```
private String translateLine(String line) {
    Initialize a variable to hold the result and set it to the empty string.
    while (there are any tokens left on the line) {
```

```

Get the next token.
if (the token is a word) {
    Translate the token to Pig Latin.
}
Append the token to the result variable.

Return the value of the result variable to the caller.

```

该策略也很简单。而且，获得令牌和测试令牌是否在行里的单独操作很可能在各种应用程序中都有用。

8.5.4 StringTokenizer 类

实际上，将字符串划分为令牌非常有用，因此 Java 库包含几种不同的类来完成此任务。当然，最简单的是 StringTokenizer 类，它是 java.util 程序包的一部分。该类可以从提供给 StringTokenizer 构造函数的字符串中读取令牌。这样就创建了令牌赋予器(tokenizer)，可以用它从原始字符串逐个读取连续的令牌。从令牌赋予器读取的一系列令牌称为令牌流。

StringTokenizer 类中最重要的方法如图 8-13 所示。 StringTokenizer 的构造函数以 3 种形式出现，最简单的形式如下面的声明所示，它从名为 line 的字符串值变量读取令牌：

```
StringTokenizer tokenizer = new StringTokenizer(line);
```

使用这种形式的构造函数时，令牌赋予器使用空格和制表符——或者让 Character.isWhitespace 方法返回 true 的任何字符——将字符串划分为令牌来标记每个令牌的边界。划分令牌的空格本身不是令牌，不会作为令牌流的一部分返回。

构造函数
new StringTokenizer(String str) 创建新的 StringTokenizer 对象，它从 str 读取单独的空白令牌
new StringTokenizer(String str, String delims) 创建 StringTokenizer，它使用 delims 中的字符作为令牌分隔符
new StringTokenizer(String str, String delims, boolean returnDelims) 创建 StringTokenizer，如果第三个参数是 true，返回定界符令牌
从令牌流读取令牌的方法
boolean hasMoreTokens() 如果要从 StringTokenizer 中读取更多令牌，返回 true
String nextToken() 从令牌流中返回下一个令牌，如果没有令牌，就发送错误

图 8-13 StringTokenizer 类中的有用方法

构造函数的第二种形式允许指定划分单个令牌的字符集，这些字符称为定界符。例如，如果创建使用空格或逗号划分单个令牌的令牌赋予器，可以这样重写声明：

```
StringTokenizer tokenizer = new StringTokenizer(line, " ,");
```

在 StringTokenizer 构造函数的单参数版本中，其声明创建的令牌赋予器使用空格和逗号字符来区分令牌，但不将它们作为令牌流的一部分。按照当前的情况，Pig Latin 示例的伪代码需要可以返回定界符字符的扫描仪。构造函数的第三种形式提供了这种能力，它包含第三个参数，该参数可以确定是否应该将定界符作为令牌返回。如果这个参数是 false，令牌赋予

器会像在双参数形式中一样操作。如果参数是 true，令牌流会作为单个字符串令牌包含每个定界符字符。

在 PigLatin 程序中，需要的令牌赋予器可以将所有标点符号解释为定界符并作为令牌返回这些定界符。要创建这样的令牌赋予器，最简单的方法是定义名为 DELIMITERS 的命名常量(它包含所有合法标点符号)，然后使用下面的代码声明令牌赋予器：

```
 StringTokenizer tokenizer =
    new StringTokenizer(line, DELIMITERS, true);
```

声明令牌赋予器之后，调用 nextToken 方法来依次获得每个令牌，代码就可以从令牌流读取令牌。要确定令牌流何时结束，需要使用 hasMoreTokens 方法，只要还有其他令牌，它就返回 true。这些方法可以用实现方法所需的实际代码取代 translateLine 中的大多数伪代码。如果用对仍未实现的方法 isWord 和 translateWord 的调用取代伪代码的最后剩余位，translateLine 的代码如下：

```
private String translateLine(String line) {
    String result = "";
    StringTokenizer tokenizer =
        new StringTokenizer(line, DELIMITERS, true);
    while (tokenizer.hasMoreTokens()) {
        String token = tokenizer.nextToken();
        if (isWord(token)) {
            token = translateWord(token);
        }
        result += token;
    }
    return result;
}
```

8.5.5 完成实现

此时，只有两种方法未实现：isWord 和 translateWord。这两种方法中最简单的是断言方法 isWord，它确定 nextToken 返回的令牌是否应该转换为 Pig Latin 的单词，或者它是否为标点符号。Pig Latin 的规则只有当单词完全由字母组成时才有意义。所以如果 token 中的每个字符都是字母，isWord(token)返回 true 是合理的。到现在为止，应该很熟悉字符串和字符，可以立即实现该方法，代码如下：

```
private boolean isWord(String token) {
    for (int i = 0; i < token.length(); i++) {
        char ch = token.charAt(i);
        if (!Character.isLetter(ch)) return false;
    }
    return true;
}
```

转换单个单词的过程有点麻烦。在伪代码中，translateWord 的结构镜像 Pig Latin 规则：

```
private String translateWord(String word) {
```

```

Find the position of the first vowel.
if (the vowel appears at the beginning of the word)
    Return the word concatenated with "way".
} else {
    Extract the initial substring up to the vowel and call it the "head."
    Extract the substring from that position onward and call it the "tail."
    Return the tail, concatenated with the head, concatenated with "ay".
}
}

```

伪代码中的第一个英语语句要求实现大多数代码，也是唯一让定义新方法有意义的语句。方法 `findFirstVowel(word)` 返回 `word` 里第一个元音字母的索引位置。因此，`translateWord` 实现方式中的第一个语句可以如下编写：

```
int vp = findFirstVowel(word);
```

如果在设计中过程中很细心，就会注意到方法 `findFirstVowel(word)` 没有完全指定。非正式的描述不包含所有可能的情况。如果 `word` 中没有元音字母，情况又如何？在这种情况下，`findFirstVowel` 仍然必须返回某个信息。假设 `String` 类中的各种搜索方法使用-1 作为终止条件，表示方法搜索的值在字符串中不存在，这里就可以采用相同的策略。

`findFirstVowel` 返回-1，`translateWord` 必须适当响应。由于从未明确定义转换不包含元音字母的单词的结果，因此需要确定 `translateWord` 采取什么行动。对于 `translateWord` 而言，最简单的方法是原封不动地返回原始单词。将这种设计决定结合到 `translateWord` 的伪代码版本，会产生如下更完整的实现方式：

```

private String translateWord(String word) {
    int vp = findFirstVowel(word);
    if (vp == -1) {
        return (word);
    } else if (vp == 0) {
        Return the word concatenated with "way";
    } else {
        Extract the initial substring up to the vowel and call it the "head."
        Extract the substring from that position onward and call it the "tail."
        Return the tail, concatenated with the head, concatenated with "ay".
    }
}

```

`translateWord()` 剩下的步骤是从 `String` 类调用合适的方法，这样就很容易完成编码过程，代码如下：

```

private String translateWord(String word) {
    int vp = findFirstVowel(word);
    if (vp == -1) {
        return word;
    } else if (vp == 0) {
        return word + "way";
    } else {
        String head = word.substring(0, vp);

```

```
    String tail = word.substring(vp);
    return tail + head + "ay";
}

}
```

编码 findFirstVowel 也很简单，特别是当有方法 isEnglishVowel 时。isEnglishVowel 在 8.2.8 小节中有定义，它用来确定字符是不是元音字母。因此 findFirstVowei 的实现方式如下：

```
private int findFirstVowel(String word) {
    for (int i = 0; i < word.length(); i++) {
        if (isEnglishVowel(word.charAt(i))) return i;
    }
    return -1;
}
```

Pig Latin 程序的完整代码如图 8-14 所示。

```
/*
 * File: PigLatin.java
 *
 * -----
 * This file takes a line of text and converts each word into Pig Latin.
 * The rules for forming Pig Latin words are as follows:
 * - If the word begins with a vowel, add "way" to the end of the word.
 * - If the word begins with a consonant, extract the set of consonants up
 *   to the first vowel, move that set of consonants to the end of the word,
 *   and add "ay".
 */

import acm.program.*;
import java.util.*;

public class PigLatin extends ConsoleProgram {

    public void run() {
        println("This program translates a line into Pig Latin.");
        String line = readLine("Enter a line: ");
        println(translateLine(line));
    }

    /* Translates a line to Pig Latin, word by word */
    private String translateLine(String line) {
        String result = "";
        StringTokenizer tokenizer =
            new StringTokenizer(line, DELIMITERS, true);
        while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (isWord(token)) {
                token = translateWord(token);
            }
            result += token;
        }
        return result;
    }
}
```

图 8-14 PigLatin 转换器的完全实现

```

/* Translates a word to Pig Latin and returns the translated word */
private String translateWord(String word) {
    int vp = findFirstVowel(word);
    if (vp == -1) {
        return word;
    } else if (vp == 0) {
        return word + "way";
    } else {
        String head = word.substring(0, vp);
        String tail = word.substring(vp);
        return tail + head + "ay";
    }
}

/* Returns the index of the first vowel in the word (-1 if none) */
private int findFirstVowel(String word) {
    for (int i = 0; i < word.length(); i++) {
        if (isEnglishVowel(word.charAt(i))) return i;
    }
    return -1;
}

/* Returns true if the character is a vowel */
private boolean isEnglishVowel(char ch) {
    switch (Character.toLowerCase(ch)) {
        case 'a': case 'e': case 'i': case 'o': case 'u':
            return true;
        default:
            return false;
    }
}

/* Returns true if token is a "word" (all character are letters) */
private boolean isWord(String token) {
    for (int i = 0; i < token.length(); i++) {
        char ch = token.charAt(i);
        if (!Character.isLetter(ch)) return false;
    }
    return true;
}

/* Defines the characters that delimit a token */
private static final String
    DELIMITERS = "!@#$%^&*()_-+={}:;\\'<,>.?/-`|\\\\ ";
}

```

图 8-14 (续)

8.6 小结

从本章开始介绍如何使用文本数据。在 Java 中，文本数据最常见的形式是字符串，它是单个字符按顺序组成的集合。单个字符用 char 数据类型表示，它是 Java 定义的原始类型之一。

使用 Unicode 编码系统在硬件内部用数字值表示字符。字符串本身使用类 `String` 表示，它正式定义为 `java.lang` 程序包的一部分，但实际上它以多种方式直接结合到语言中。

本章介绍了使用 `String` 类提供的方法操作字符串的方法，这样就可以重点关注字符串的抽象行为，而不必关心其具体细节。

本章介绍的重点如下：

- 在计算机内部，给类型域里的元素编号，然后将这些编号作为原始值的代码。通常这样表示概念值不是数的类型。通过列举元素来定义的类型称为枚举类型。
- Java 可以用两种方法定义新的枚举类型。较早的策略是对这种类型使用 `int`，并为指定值定义命名常量。更现代的策略是使用 Java 5.0 中引入的关键字 `enum`。
- 依据 Unicode 预定义编码方案，在内部可以将字符表示为整数。Unicode 可以表示更大范围内的语言字符。在 Java 中，这些值由原始类型 `char` 表示。
- 使用标准的算术运算可以操作字符值。通常不需要使用类型强制转换将 `char` 转换为 `int`，但 Java 则需要使用类型强制转换。
- `Character` 类包含几种方法，可以用来区分和改变单个字符的大小写。
- `String` 类可以将字符串作为抽象类型使用。
- `String` 类定义了几种操作字符串的方法。这些方法如图 8-5 所示。
- `java.util` 程序包包含 `StringTokenizer` 类，它可以很方便地将字符串划分为作为有意义单元的子字符串。这些方法如图 8-13 所示。

8.7 复习题

1. 用自己的话描述枚举原理。
2. 本章提到的 Java 中表示枚举类型有哪两种方法？目前本书为什么介绍较早的策略？
3. 什么是标量类型？
4. 如何在字符串常量内包含双引号？
5. ASCII 表示什么？
6. ASCII 和 Unicode 之间是什么关系？
7. 参考图 8-1，确定字符 “\$”、“@”、“\t” 和 “x” 的八进制 Unicode 值。将这些值转换为对应的十进制值。
8. Unicode 表中的数字字符是连续的，这一点为什么有用？
9. 本章讨论的哪 4 种算术运算对字符最有意义？
10. 调用 `Character.isDigit(5)` 的结果是什么？如果调用 `Character.isDigit('5')`，结果又是什么？调用 `Character.isDigit("5")` 合法吗？
11. 调用 `Character.toUpperCase('5')` 的结果是什么？
12. 使用 `Character` 类中的方法而不要使用自己写的方法，本章给出了哪 4 个理由？
13. 判断题：在 `switch` 语句内使用字符常量作为 `case` 表达式是合法的。
14. 下面的语句对 `str` 值有何影响？

```
str.trim()
```
15. 前一个问题中，实现表达式预期结果的正确方法是什么？

16. 术语“不变的”是什么意思？

17. 下列表达式的结果是什么？

- | | |
|-----------------------------|-----------------------------|
| (1) "ABCDE".length() | (6) "ABCDE".substring(2) |
| (2) "".length() | (7) "ABCDE".indexOf("C") |
| (3) "t".length() | (8) "ABCDE".indexOf('Z') |
| (4) "ABC".charAt(2) | (9) "XYZZY".indexOf('Z', 3) |
| (5) "ABCDE".substring(0, 3) | (10) "ABCDE".toLowerCase() |

18. 比较字符串时最重要的是要记住什么？

19. 下列表达式的结果是什么？(对 compareTo 的调用，只表示结果的符号)

- | |
|---------------------------------------|
| (1) "ABCDE".equals("abcde") |
| (2) "ABCDE".equalsIgnoreCase("abcde") |
| (3) "ABCDE".compareTo("ABCDE") |
| (4) "ABCDE".compareTo("ABC") |
| (5) "ABCDE".compareTo("abcde") |
| (6) "ABCDE".startsWith("a") |

20. StringTokenizer 构造函数中的第三参数有什么作用？

21. 描述 StringTokenizer 类中 nextToken 和 hasMoreTokens 方法的操作。

8.8 编程练习

1. 实现方法 isEnglishConsonant(ch)，如果 ch 在英语中是辅音字母，它返回 true。辅音字母就是除了 5 个元音字母 “a”、“e”、“i”、“o” 和 “u” 之外的所有字母。和 isEnglishVowel 一样，该方法使用可以识别大写和小写字母的辅音字母。编写 ConsoleProgram 显示所有大写辅音字母。

2. 编写方法 randomWord，返回由随机选择的字母组成的随机构造“单词”。单词的字母数也从命名常量 MIN LETTERS 和 MAX LETTERS 之间随机选择。编写 ConsoleProgram，通过显示 5 个随机单词来测试此方法。

3. 实现方法 capitalize(str)，它返回字符串，该字符串的词首字符是大写字母(如果是字母)，将其他所有字母转换为小写字母形式。字符(而不是字母)应该不受影响，因此，capitalize("BOOLEAN") 和 capitalize("boolean") 应该返回字符串 "Boolean"。

4. 编写方法 createDateString(day, month, year)，它返回字符串，该字符串由月份中的日期、连字号、月份名称中的前 3 个字母、连字号、年的最后两个数字组成。例如，调用方法

```
createDateString(22, 11, 1963)
```

应该返回字符串 "22-Nov-63"。

5. 在许多单词游戏中，单词中的每个字母根据其点值计分，它与英语单词中的频率成反比。在 Scrabble™ 中，点数分配如表 8-1 所示。

表 8-1 游戏点数分配

1	A, E, I, L, N, O, R, S, T, U
2	D, G
3	B, C, M, P
4	F, H, V, W, Y
5	K
8	J, X
10	Q, Z

例如，Scrabble 单词“FARM”值 9 个点。F 值 4 点，A 和 R 各值 1 点，M 值 3 点。写 ConsoleProgram 读入单词，打印出在 Scrabble 中的点数，不考虑游戏中出现的其他奖励。计算分数的时候应该忽略所有不是大写字母的字符。特别注意，假设小写字母表示黑麻将牌，它可以表示任何字母，但分数是 0。

6. 回文是一种顺读和倒读都一样的单词，例如 level 或者 noon。编写断言方法 isPalindrome(str)，如果字符串 str 是回文，它返回 true。另外，设计和编写测试程序，调用 isPalindrome 来说明其运行。在写程序的过程中，应重点关注如何解决问题而不是如何让解决方案更有效。

7. 忽略标点符号和字母的大小写差异之后，练习 6 中回文的概念通常可以扩展开到整句，例如，语句

Madam, I'm Adam.

就是句子回文，因为如果只看字母，忽略大小写字母之间的差异，它顺读和倒读完全一样。写断言方法 isSentencePalindrome(str)，如果字符串 str 符合句子回文的定义，它返回 true。例如，应该能够使用您的方法编写生成如图 8-15 所示结果的主程序。

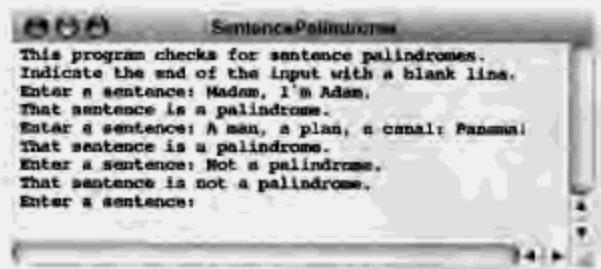


图 8-15 SentencePalindrome 示例

8. 编写方法 createRegularPlural(word)，返回按照下面标准的英语规则形成的 word 的复数。

- (1) 如果单词以 s, x, z, ch 或 sh 结尾，添加 es。
- (2) 如果单词以 y 结尾，并且 y 前面是辅音字母，变 y 为 ies。
- (3) 其他情况，加 s。

编写测试程序，并设计一组测试情况来验证您的程序是否正确。

9. 英语中，现在并持续到将来想法使用现在进行时表示，它需要在动词后加上后缀 ing。例如，句子 I think 的意思是说某个人能够思考，句子“ I am thinking ”说明一个人正在思考。

动词的 ing 形式称为现在分词。

遗憾的是，创建现在分词不是添加 ing 后缀这么简单。常见的例外是以不发音的 e 结尾的单词，如 cogitate。在这种情况下，通常去掉 e，因此现在分词形式是 cogitating。另一个常见例外是以单个辅音字母结尾，现在分词形式通常双写该辅音字母。例如，动词 program 的现在分词是 programming。

虽然有许多例外，但是应用如下规则，可以构造大部分合法的英语现在分词形式：

(1) 如果单词以 e 结尾，并且前面有辅音字母，在添加后缀 ing 之前去掉 e。因此，move 的现在分词是 moving。如果 e 前面不是辅音字母，必须保留 e，因此 see 的现在分词是 seeing。

(2) 如果单词以辅音字母结尾并且前面有元音字母，添加后缀 ing 之前先复制该辅音字母。因此，jam 的现在分词是 jamming。然而，如果单词结尾不止一个辅音字母，就不需要双写，因此 walk 的现在分词是 walking。

(3) 其他情况，直接加 ing 后缀。

编写方法 createPresentParticiple，读入英语动词，可以假设它全部是小写字母并且至少有两个字符长，运用上面的规则形成现在分词。编写 ConsoleProgram 方法测试该方法。

10. 和其他许多语言一样，英语也包含两种类型的数：用于计数的基数(如 one、two、three 和 four)和用于表示序列中位置的序数(如 first、second、third 和 fourth)。在数字形式中，序数通常用数中的数字表示，后面紧跟着表示相应序数的英语单词的最后两个字母。这样，序数 first、second、third 和 fourth 通常表示为 1st、2nd、3rd 和 4th。

确定序数后缀的一般规则可以定义如下：以数字 1、2、3 结尾的数，分别使用后缀 st、nd 和 rd，除了以其中两个数字结合(11、12 或 13)结尾的之外。这些数及其他不以 1、2 或 3 结尾的数，使用后缀“th”。

任务是编写 createOrdinalForm(n)方法，它读入整数 n，返回表示相应序数的字符串。例如，方法应该返回下列值：

createOrdinalForm(1)	返回字符串	"1st"
createOrdinalForm(2)	返回字符串	"2nd"
createOrdinalForm(3)	返回字符串	"3rd"
createOrdinalForm(10)	返回字符串	"10th"
createOrdinalForm(11)	返回字符串	"11th"
createOrdinalForm(12)	返回字符串	"12th"
createOrdinalForm(21)	返回字符串	"21st"
createOrdinalForm(42)	返回字符串	"42nd"
createOrdinalForm(101)	返回字符串	"101st"
createOrdinalForm(111)	返回字符串	"111th"

11. 要让人们不容易阅读消息，最简单的代码类型是字母替代密码，其中原始消息里的每个字母都由该消息编码版本中的不同字母取代。字母替代密码最简单的类型是 Caesar 密码——这样命名是因为罗马历史学家 Suetonius 说 Julius Caesar 曾经使用过此密码——其中每个字符都由字母表中前面固定距离的另一个字母取代。Caesar 密码是循环的，因为转换超过 Z 字母的操作又回到了起始点，重新从 A 开始。

例如，假设要通过将每个字母前移 4 个位置来编码消息。在 Caesar 密码中，每个 A 变成了 E，B 变成了 F，Z 变成了 D(因为又回到了开始)，依此类推。

要解决这个问题，应该首先定义方法。

```
private String encodeCaesarCipher(String str, int shift)
```

它返回新的字符串，将 str 中的每个字符前移 shift 指定的字母个数，如果需要可以回到字母表开始，新字符串就是这样就形成的。实现 encodeString 之后，编写 ConsoleProgram，它可以复制如图 8-16 所示的示例。

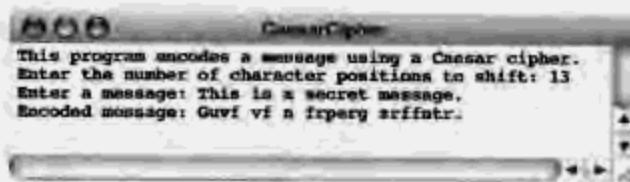


图 8-16 Caesar 密码运行示例(1)

注意编码操作只应用于字母，其他字符在输出中保持不变。而且，字母的大小写不变，小写字母输出是小写字母，大写字母输出是大写字母。

编写程序，让 shift 的负数值说明字母在字母表中向前移动，不是向后移动，如图 8-17 所示。

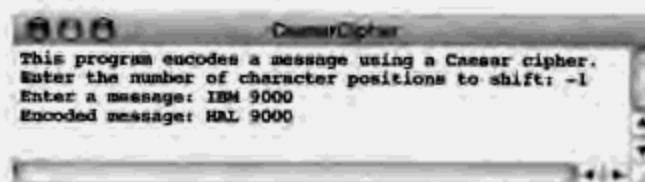


图 8-17 Caesar 密码的运行示例(2)

12. 假设 ConsoleProgram 类不包括从控制台读取整数的 readInt 方法，只支持 readString 方法。编写方法 private int myReadInt(String prompt)，通过读取一行，然后将那一行的字符转换为整数，来模拟 readInt 的操作。myReadInt 的实现方式应该允许输入从 - 字符开始表示负值。然而，除了特殊情况以外，如果它读入任何字符而不是标准的十进制数，实现方式应该显示错误。使用 acm.io 中的 readInt 方法，实现方式应该打印消息并要求用户输入新值，以此记录该错误，如第 2 章的 Add2Integers 程序运行结果，如图 8-18 所示。

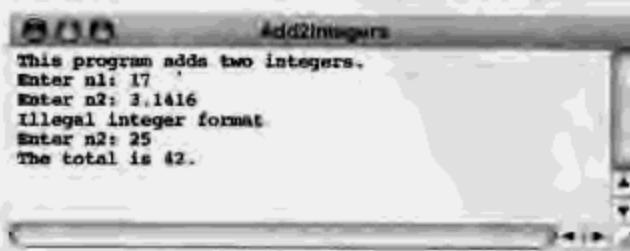


图 8-18 Add2Integers 的运行结果

13. 在纸上写值比较大的数时，传统方式——至少在美国——是用逗号将数分成 3 个一组。例如，100 万通常写成以下形式：

1,000,000

为了让编程人员更方便地以这种方式显示数，请实现方法 private String addCommas(String digits)。它读取表示数的十进制数字字符串，返回字符串，该字符串从右边开始，每隔 3 位插入逗号。例如，如果执行主程序

```
public void run() {
    while (true) {
        String digits = readLine("Enter a number: ");
        if (digits.length() == 0) break;
        println(addCommas(digits));
    }
}
```

addCommas 方法的实现应该能够产生如图 8-19 所示结果。

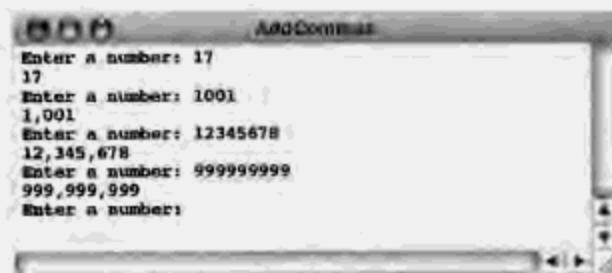


图 8-19 AddCommas 运行结果

14. 填写支票时，通常以数字和口头形式记录支票总数。因此，如果写 \$1,729 美元的支票，口头表示应该这样写：

```
one thousand seven hundred and twenty-nine
```

实现方法 private String numberToWords(int n)，它读入整数——可以假设介于 0~999 999 之间——返回口头上表示的字符串。编写 run 方法测试此方法，该 run 方法重复读入整数，然后将整数转换为单词，输入负数即停止，如图 8-20 所示。

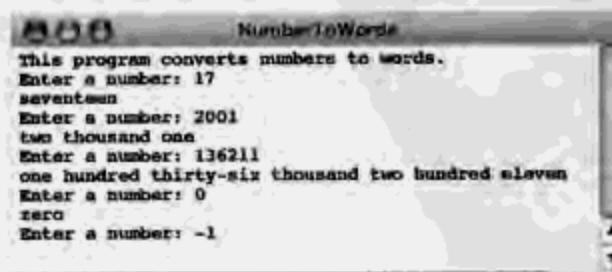


图 8-20 NumberToWords 的运行结果

注意，没有魔法将像 1 这样的数字转换为英语单词 one。要实现这种情况，需要包括显式的代码——很可能以 switch 语句的形式——将每个数字转换为其英语表示。该假设中要考虑的重点是如何分解问题，以便可以重新使用方法转换数的不同部分。

15. 实现方法 `private String longestWord(String line)`, 它返回 `line` 中最长的单词, 其中单词定义为字母的连续字符串, 和 PigLatin 程序中一样。编写 `ConsoleProgram` 测试该实现方式, 该程序可以复制图 8-21 所示的效果。

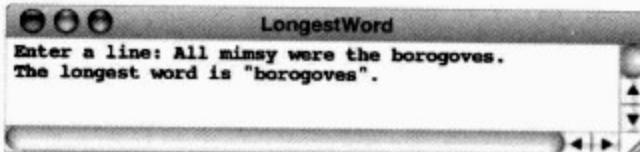


图 8-21 LongestWord 的运行结果

16. 重写 8.4.6 小节描述的 `acronym` 方法, 让它使用 `StringTokenizer` 类来划分参数中的单词。这种改变将允许首字母缩写词方法返回正确的结果, 即使单词没有用空格分开。例如, 实现方式应该可以形成字符串 `self-contained underwater breathing apparatus` 的首字母缩写词 `scuba`, 即使前两个字母被连字号而不是空格分开。

17. 重写本章的 PigLatin 程序, 让它使用逐个字符而不是逐个单词地检查行的算法策略。
8.5.3 小节有该策略的伪代码版本。

18. 如果输入包含以大写字母开头的单词的字符串, 图 8-14 中的 PigLatin 程序行为有些古怪。例如, 如果将句子和 Pig Latin 语言名称中的第一个单词的首字母写成大写, 输出如图 8-22 所示。

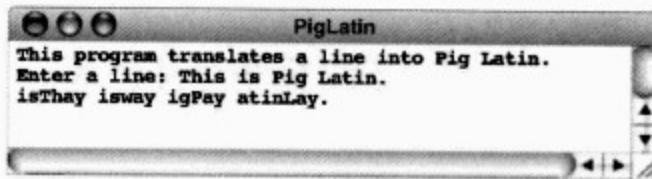


图 8-22 PigLatin 程序的输出

重写 `translateWord` 方法, 让英语行中以大写字母开头的所有单词在 Pig Latin 中仍然以大写字母开头。这样, 改变程序后, 输出应该如图 8-23 所示。

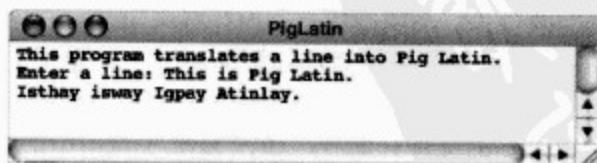


图 8-23 PigLatin 程序的输出

19. 大多数人——至少是说英语的国家里的人——有时会玩 Pig Latin 游戏。然而, 有一种创造性的“语言”, 其中单词使用英语的简单转换创建。其中之一是 Obenglobish 语言, 它的单词通过在英语单词中的元音字母前添加字母 ob 创建。例如, 按照规则, 单词 `english` 的转换通过在两个元音字母前添加 ob 形成 `obenglobish` 完成, 这也是这种语言名称的由来。

在正式的 Obenglobish 中, 只在发音的元音字母前添加字母 ob, 意思就是像“game”会变成“gobame”而不是“gobamobe”, 因为最后一个 e 不发音。虽然不可能完全实现该规则, 但

也可以做得很好，只要在英语中的每个元音字母前添加 ob 就行，除了以下两种情况：

- 元音字母接着其他元音字母。
- 单词结尾是 e。

使用上面的转换规则，编写方法 obenglobish，读入英语单词，返回 Obenglobish。例如，使用 run 方法：

```
public void run() {
    while (true) {
        String word = readLine("Enter a word: ");
        if (word.equals("")) break;
        println(word + " -> " + obenglobish(word));
    }
}
```

应该能够生成如图 8-24 所示结果。

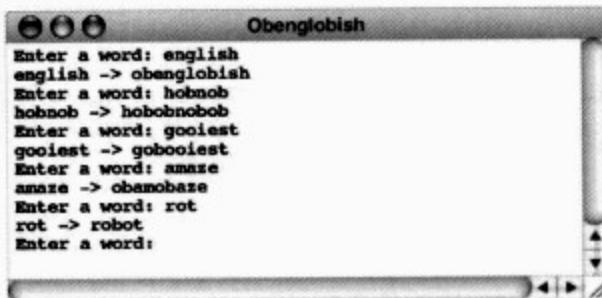


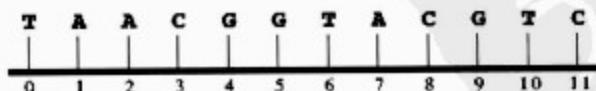
图 8-24 Obenglobish 的运行结果

20. 所有活性生物体的 DNA 都携带其基因代码，DNA 分子具有复制自身结构的强大能力。DNA 分子本身由一长串化学基组成，这些化学基的相同分子互相缠绕在一起形成双螺旋结构。DNA 的复制功能来源于其结构中的 4 个基——腺苷、胞核嘧啶、鸟嘌呤和胸腺嘧啶——以下面的方式互相结合：

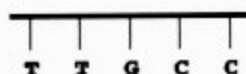
- 腺苷只与胸腺嘧啶链接，胸腺嘧啶也只与腺苷链接。
- 胞核嘧啶只与鸟嘌呤链接，鸟嘌呤也只与胞核嘧啶链接。

生物学家通常将这些基的名称缩写为其初始字母：A、C、G 和 T。

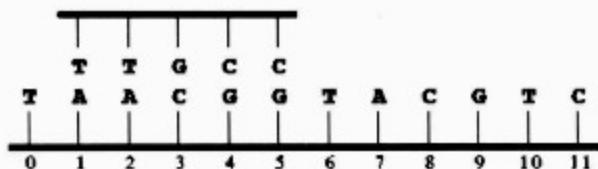
在细胞内，DNA 串像一个模板，其他 DNA 串可以将自己附属到该模板。例如，假设有如下所示的 DNA 串，其中每个基的位置有编号，和在 Java 字符串中的编号一样。



生物学家要解决的问题之一是找出短的 DNA 串附属到较长的 DAN 串的位置。例如，如果要为串



找到匹配, DNA 复制的规则指出这个串可以在位置 1 上绑定到较长的串:



编写方法 `findFirstMatchingPosition(shortDNA, longDNA)`, 让它读取 DNA 串中表示基的两个字母字符串。方法应该返回第一索引位置, 在此位置第一个 DNA 串可以绑定到第二个 DNA 串, 如果没有匹配位置, 返回-1。

21. 即使 `StringTokenizer` 类是 `java.util` 程序包的一部分, 用户自己重新实现 `StringTokenizer` 也可以更好学习字符串操作和类设计。写此类(实现如图 8-13 所示的构造函数和方法)新的实现方式。唯一不知道如何实现的设计是没有令牌可用时调用 `nextToken` 生成错误的特征。要解决这一问题, 编写自己的 `nextToken` 版本, 如果没有其他令牌, 让它返回 `null`。

第 9 章

面向对象图形

*In pictures, if you do it right, the thing happens,
right there on the screen.*

—Director John Huston, as quoted in

—James Agee, Agee on Film, 1964



Ivan Sutherland

Ivan Sutherland 出生于美国内布拉斯加州，在高中时，他就对计算机产生了浓厚兴趣。由于计算机科学当时还不是一门学科，因此 Sutherland 在匹兹堡卡内基技术学院(Carnegie Institute of Technology)，现在的卡内基梅隆大学，Carnegie Mellon University)主修电机工程，后来在美国加利福尼亚理工学院获得了硕士学位，在 MIT(美国麻省理工学院)获得了博士学位。他的博士论文“画板：人机图形通信系统”是计算机图形学的奠基石之一，该论文介绍了图形用户界面的概念，这一概念是现代软件的基本特征。完成学业后，Sutherland 曾在哈佛大学、犹他州大学和加利福尼亚理工学院任教，之后他离开学术界建立了一家计算机图形公司。1988 年 Sutherland 获得了 ACM 图灵奖。

虽然在 acm.graphics 程序包中可以看到几个类和方法，但始终没有将类的集合作为整体考虑。除了继续零散地讨论程序包之外，本章将 acm.graphics 作为工具综合集进行研究。这样做有两个目的。第一，可以介绍程序包提供的许多图形性能。这种理解有助于编写更有趣的图形程序，这应该很令人兴奋。还有一个更微妙的目的，面向对象方法编程与编码风格的关系不大，与设计策略的关系较大。将 acm.graphics 作为连续的程序包重点关注，可以更详细地理解设计类和程序包的方法，这样编写它们就很方便。

9.1 acm.graphics 模型

在讨论 acm.graphics 中可用的类和方法之前，先需要理解假设、约定和隐喻这些程序包建立的基础。一般来说，为程序包定义适当智力图片的思想称为模型。有关在域中应该如何考察运行的各种问题，程序包模型都可以回答。例如，在使用图形程序包之前，需要回答如下问题：

- 在屏幕上使用什么单位指定长度和位置？
- 在程序包设计中有哪些现实的类推和隐喻？

对于第一个问题，在 acm.graphics 模型中，所有坐标位置和长度都用像素表示。它是在屏幕上出现的单个点。通过指定 x 和 y 方向上点的坐标，可以指定屏幕上的点，在画布上向右移动时， x 值会增加，从上向下移动时， y 值会增加。 x 坐标的解释与传统的笛卡尔几何学中一样，但 y 坐标则相反。因为向下移动时 y 值增加，所以 Java 图形模型设计师将点(0, 0)置于窗口的左上角。该点称为原点。

acm.graphics 程序包采用这种坐标模型(与笛卡尔坐标系相反)来匹配其他 Java 程序包中使用的坐标系。在程序包间保持一致的模型有利于从一个程序包转移到另一个程序包。

关于 acm.graphics 程序包，第二个问题(什么样的类推和隐喻适用于程序包)在某种程序上更重要。许多现实类推可用于计算机图形，因为有许多方法可以创建视觉艺术。一种可能的隐喻是绘画。艺术家选择画笔和颜色后，在表示视觉画布的屏幕上移动刷子，绘制图像。如果用铅笔作画，可以形成不同的图形风格，这种风格中线条画占主导地位，并且可以擦去以前画的内容。同样，也可以设计一种基于雕刻隐喻的图形程序包，这样艺术家使用铁笔蚀刻出图画；对于该隐喻而言，擦去大概没有意义。还有许多其他可能的隐喻，就像有许多技术手法一样。

为了支持面向对象设计的概念，acm.graphics 程序包使用拼贴画隐喻。拼贴画艺术家的工作就是选择不同的对象，并在背景画面上组合这些对象。在现实中，这些对象可能是几何形状、从报纸上剪切的单词、字符串组成的行或从杂志上选取的图像。acm.graphics 程序包提供所有这些对象的对应部分。

图像是拼贴画，这一概念对描述创建图案的过程有重要意义。如果在绘画，可能在特定位置使用绘画的技巧，或用涂料填充某个区域。在拼贴画模型中，主要操作是添加和删除对象，以及在背景画布上重新定位它们。

拼贴画也有属性，例如有些对象可以位于其他对象之上，遮蔽它们后面的对象。删除这些对象后才能显示下面使用的对象。拼贴画中从后到前的顺序在本书中称为堆栈顺序，尽管有时更正式的写法是 z 顺序。堆栈顺序总是与由 x 和 y 形成的二维平面上的轴一起出现，这也是名称 z 顺序的由来。在数学中，平面的垂直轴称为 z 轴。

9.2 acm.graphics 程序包的结构

虽然从第 2 章开始已经使用了 acm.graphics 程序包中的一些类，但是还有许多类需要整体介绍。本章的目的是让用户更全面理解 acm.graphics 程序包在图形工具方面提供的性能。图 9-1 显示的是 acm.graphics 内的类结构图。接下来几节描述了各个类及使用这些类的方法。

9.2.1 GCanvas 类

图 9-1 中显示的第一个新类是 GCanvas 类。即使本书中没有提及 GCanvas 类，但是在每个 GraphicsProgram 中都使用过 GCanvas 对象。GCanvas 类表示添加单个 GOBJECTS 的背景。在拼贴画艺术家的世界里，背景是物理画布。在 acm.graphics 程序包中，背景是显示在屏幕上的窗口。

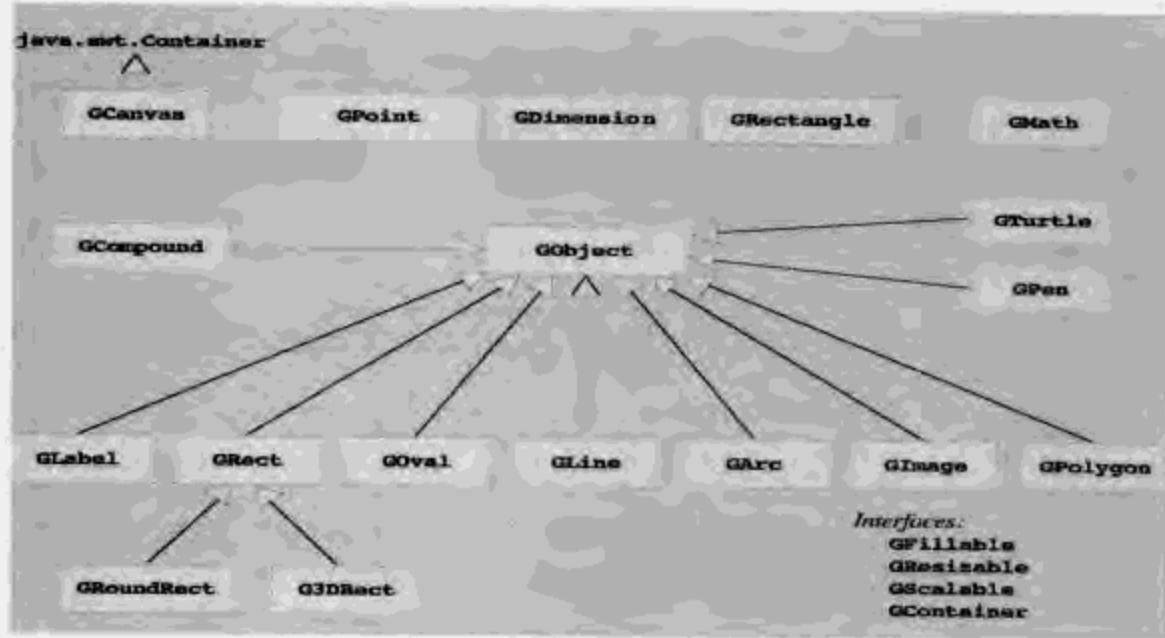


图 9-1 acm.graphics 程序包的结构

虽然通过调用其构造函数可以构造 GCanvas 类的新实例，但是通常不需要明确地这样做。GraphicsProgram 类自动创建 GCanvas 类，并定位它，让它完全填充程序窗口。在某个程序中调用 add 方法时，GraphicsProgram 中 add 的实现将请求传递到基本的 GCanvas。因此，当第 2 章的 HelloProgram 执行

```
add(new GLabel("hello, world", 100, 75));
```

时，它将相同的消息发送到 GCanvas，要求它添加最新生成的标签。GraphicsProgram 类以相同的方法对待与基本画布有关的其他访问调用。它只将请求传递到 GCanvas，由 GCanvas 实现实际操作。该过程称为转接。

GCanvas 类最重要的可用方法(可以通过在 GCanvas 对象上明确调用它们或让 GraphicsProgram 处于它们前面来调用它们)如图 9-2 所示。第一组方法与添加和删除图形对象有关。这些方法很容易理解，尤其是如果记得拼贴画隐喻。将图形对象添加到 GCanvas 相当于将物理对象添加到拼贴画。在此过程中，这些对象可以遮蔽堆栈顺序中新对象后面的对象。删除对象就是将对象从画布上拿掉，显示该对象背后的所有对象。

添加和删除图形对象的方法

```
void add(GObject gobj)
    在图形对象的中心存储位置将它添加到画布
void add(GObject gobj, double x, double y)
    在指定位置将图形对象添加到画布
void remove(GObject gobj)
    从画布上删除指定的图形对象
void removeAll()
    从画布上删除所有图形对象
```

在找出在特定位置的图形对象的方法

```
GObject getElementAt(double x, double y)
    返回包含指定点的最上面的对象。如果这样的对象不存在，返回 null
```

支持图形动画的方法（只有在 GraphicsProgram 中可用）

```
void pause(double milliseconds)
    延缓程序执行一段指定时间。通常用毫秒计算
void waitForClick()
    延缓程序执行，直到用户在画布的任何位置单击为止
```

从超类继承的有用方法

```
int getWidth()
    以像素为单位，返回画布的宽度
int getHeight()
    以像素为单位，返回画布的高度
void setBackground(Color bg)
    改变画布的背景色
```

图 9-2 GCanvas 和 GraphicsProgram 类中有用的方法

add 方法有两种形式。GObject 的构造函数已确定对象的位置时，第一种形式很有用，就像在前面那些示例中一样。当要根据屏幕大小或其他属性在屏幕上放置对象时，使用第二种形式。在那种情况下，通常需要创建没有指定位置的 GObject，弄明白它应该在哪里，然后将它添加到画布上明确的 x 和 y 位置。这种定位风格最常见的示例是在某个位置居中 GLabel；9.3.1 小节将描述这如何实现。

GCanvas 类中的 getElement 方法在包含指定点的画布上返回图形对象。使用鼠标选择对象时，这种方法特别有用。因为该描述很符合其语境，getElement 的细节在第 10 章讨论。

前面已经介绍过 pause 方法，它将程序执行延迟参数所指定的毫秒数。正如第 4 章所述，可以对画布上的图形对象做小的改变，然后调用 pause 来确保程序停止运行，这样可以让程序运动动画的速度比人眼能够跟随的速度慢。使用动画的程序通常有如下形式：

```
public void run() {
    Initialize the graphical objects in the display.
    while (as long as you want to run the animation)
        Update the properties of the objects that you wish to animate.
        pause(delay time);
}
```

当要挂起图形程序直到用户在画布任意位置单击时再启动，这种情况下 waitForClick 方法就很有用。这种策略允许用户确定何时开始继续执行。

和在面向对象类层次结构中一样，GCanvas 类中最有用的方法没有在 GCanvas 中定义，而是从其超类继承。GCanvas 类扩充 java.awt 程序包中称为 Container 的类。因为所有 GCanvas

都是 Container——如果上升到层次结构链中更高位置，它就是 Component。GCanvas 可以显示在窗口中，这样在屏幕上就可以看见它。和其他 Component 一样，GCanvas 对方法 getWidth 和 getHeight 会作出响应，这两个方法确定屏幕上画布的尺寸。调用 setBackground(bg)(其中 bg 是新的背景色)也可以改变 GCanvas 的背景颜色。使用 Color 类的对象可以指定新的颜色，和设置 GObject 的颜色一样。

9.2.2 Color 类的更多细节

只要讨论颜色，就有必要讨论 Color 类的其他一些细节。到目前为止，本文中唯一出现的颜色是 Color 类中定义为命名常量的 13 种颜色，如第 2 章所述。现代图形程序包中，java.awt 程序包中的 Color 类提供了比命名颜色更多的颜色。为了更好地选择颜色，本节介绍如何使用 Color 类中的构造函数来创建数百万种不同的颜色。

虽然 Java 提供了几种定义颜色的模型，但最常见的是 RGB 模型，其颜色由它所包含的红、绿、蓝(光的三原色)的比例定义。颜色的红、绿、蓝组件的值都用介于 0~255 之间的整数表示，其中 0 表示没有颜色，255 表示最大强度的颜色。例如，如果红色是 255，其他颜色值是 0，这样的颜色定义为 RED。同样，如果红、绿、蓝颜色值都是 128，即每种颜色表示为最大强度的一半，这样的颜色定义为 GRAY。要定义颜色，可以调用方法 getRed、getGreen 和 getBlue 来确定其组件。

Color 构造函数最简单的版本采用 3 个整数参数，这 3 个参数分别指定红、绿、蓝的强度。然而，在现有 Java 代码中，通常看到用 6 位十六进制数表示颜色组件，前两位数字指定红色组件，接下来的两位数字指定绿色组件，最后两位指定蓝色组件。定义标准颜色常量的构造函数调用如图 9-3 所示，它同时显示了注释中的十六进制说明。

```
/* Public constants in the Color class from java.awt */
public final static Color WHITE = new Color(255, 255, 255);           /* 0xFFFFFFFF */
public final static Color LIGHT_GRAY = new Color(192, 192, 192);        /* 0xCCCCCC */
public final static Color GRAY = new Color(128, 128, 128);            /* 0x808080 */
public final static Color DARK_GRAY = new Color(64, 64, 64);           /* 0x404040 */
public final static Color BLACK = new Color(0, 0, 0);                  /* 0x000000 */
public final static Color RED = new Color(255, 0, 0);                 /* 0xFF0000 */
public final static Color YELLOW = new Color(255, 255, 0);             /* 0xFFFF00 */
public final static Color GREEN = new Color(0, 255, 0);                /* 0x00FF00 */
public final static Color CYAN = new Color(0, 255, 255);              /* 0x00FFFF */
public final static Color BLUE = new Color(0, 0, 255);                 /* 0x0000FF */
public final static Color MAGENTA = new Color(255, 0, 255);            /* 0xFF00FF */
public final static Color PINK = new Color(255, 175, 175);             /* 0xFFAFAF */
public final static Color ORANGE = new Color(255, 200, 0);             /* 0xFFCC00 */
```

图 9-3 定义 Color 常量

如果要创建其他颜色，可以尝试这 3 种强度的不同值。作为通用规则，减少其主要组件的强度值，可以创建暗色；增加其他组件的比例可以创建亮色。例如，减少红色组件的值可以创建暗红；增加绿色和蓝色组件的值可以创建鲜红色。

9.2.3 GPoint 类、GDimension 类和 GRectangle 类

如 6.3.3 小节所述，将单个 x 和 y 坐标值结合为单个封装单元来表示画布上的点，通常很方便。与此类似，可以结合 width 和 height 值来表示对象的尺寸，或结合这 4 个值来表示矩形的位置和大小。为此，acm.graphics 程序包定义了类 GPoint、GDimension 和 GRectangle。将单个值封

装到合成对象中，这样做的主要优点是可以将对象作为单个实体从一个方法传递到另一个方法。

例如，声明

```
GPoint center = new GPoint(getWidth() / 2, getHeight() / 2);
```

引入了变量 center，并将它初始化为 GPoint 对象，该对象的坐标位于画布中央。同样，要定义 GRectangle 变量 bounds 来表示画布的边界，代码如下：

```
GRectangle bounds =
    new GRectangle(0, 0, getWidth(), getHeight());
```

虽然 GPoint、GDimension 和 GRectangle 类导出了许多有用的方法，可以简化常见的几何操作，但本书还是将它们全部列举出来，以提供更多的细节。最重要的操作是构造函数，它们创建可以检索单个坐标值的复合对象和方法。这些方法如图 9-4 所示。要找出这些类里可用的其他方法，可以参考 javadoc 页面。

GPoint 构造函数和方法	
new GPoint(double x, double y)	创建新的包含坐标值 x 和 y 的 GPoint 对象
double getX()	返回 Gpoint 的 x 组件
double getY()	返回 Gpoint 的 y 组件
GDimension 构造函数和方法	
new GDimension(double width, double height)	创建新的包含 width 和 height 值的 GDimension 对象
double getWidth()	返回 GDimension 的 width 组件
double getHeight()	返回 GDimension 的 height 组件
GRectangle 构造函数和方法	
new GRectangle(double x, double y, double width, double height)	创建新的包含 4 个指定值的 GRectangle 对象
double getX()	返回 GRectangle 的 x 组件
double getY()	返回 GRectangle 的 y 组件
double getWidth()	返回 GRectangle 的 width 组件
double getHeight()	返回 GRectangle 的 height 组件

图 9-4 GPoint、GDimension 和 GRectangle 类中的主要方法

9.2.4 GMath 类

计算图形图像的位置和大小有时需要使用简单的数学函数。虽然 Java 的 Math 类定义了方法来计算三角函数，如 sin 和 cos，但这些方法通常会产生混淆，因为它们采用的坐标模型在某些方面与 Java 用于图形程序包的模型不兼容。在 Java 图形库中，角用度来衡量，而在 Math 类中，角必须用弧度表示。

为了将与这种表示不一致性相关的混淆程度降到最低，acm.graphics 程序包包含一个称为

GMath 的类，它导出的方法如图 9-5 所示。和 Java 的 Math 类中的方法一样，GMath 导出的方法是静态的。因此，调用这些方法需要包含类的名称，在

```
double cos45 = GMath.cosDegrees(45);
```

中，将变量 cos45 设置为 45°余弦。

以度为单位的三角方法
double sinDegrees(double angle) 返回以度为单位的角的三角正弦
double cosDegrees(double angle) 返回以度为单位的角的三角余弦
double tanDegrees(double angle) 返回以度为单位的角的三角正切
double toDegrees(double radians) 将角从弧度转换为度
double toRadians(double degrees) 将角从度转换为弧度
double distance(double x, double y) 返回从原点到点(x, y)的距离
double distance(double x0, double y0, double x1, double y1) 返回点(x0, y0)和(x1, y1)之间的距离
double angle(double x, double y) 返回原点到点(x, y)之间的角，以度为单位
将 double 近似为 int 的方法
int round(double x) 将 double 近似为最接近的 int(不是近似为 Math 类中的 long)

图 9-5 GMath 类中的静态方法

图 9-5 中的前几种方法计算用度表示的角的三角函数。distance 和 angle 方法可以方便地将传统的 x-y 坐标转换为极坐标。在极坐标中，依据点到原点的距离和方向来定义它们。距离通常表示为字母 r ， r 的特定值对应于以它为半径的圆上的点，字母 r 即来源于此。角——数学中通常写作希腊字母 θ ，用变量名 theta 表示——用从 +x 轴逆时针方向上的度来衡量，和古典几何学中一样。

图 9-5 中的最后一个方法是 round，它将 double 四舍五入为最接近的 int 值。虽然 Java 的 Math 类也导出名为 round 的方法，但此版本返回 long 类型值，因此不便使用。

9.2.5 GObject 类及其子类

从图 9-1 中的箭头可以知道，GObject 类在 acm.graphics 程序包中发挥了重要作用。条条大路通罗马，子类到其超类的所有路径也都到达 GObject。GObject 表示所有图形对象。在拼贴模型的架构中，GObject 类表示可以添加到拼贴画上所有对象的一般设计。这些对象可能是三角形或椭圆形等剪切形状；有些可能是从杂志上剪切下来的图像或单词。这些对象共有的属性是可以将它们添加到画布。在 acm.graphics 中，GObject 只是一个能添加到 GCanvas 的对象。

在图 9-1 中，GObject 类的名称以斜体显示。在类图中，斜体用来表示抽象类，这些类不能基于自身构造，而是作为子类集合的模板。因此，没有图形对象将 GObject 作为其主要的类。相反，在屏幕上的对象都是 GLabel、GRect、GOval 或扩充 GObject 的其他子类之一。然而，

Java 类的层次结说明，所有这些类也是 GObject 类，即使从未见过

```
GObject gobj = new GObject();
```



这样的声明，这样的声明是不合法的，因为不能在抽象类的基础上构造该类的实例。然而，构造特定 GObject 子类的实例，然后将它赋给 GObject 变量，却是合法的，例如：

```
GObject gobj = new GLabel("Hello, world");
```

GObject 本身定义了图 9-6 所示的一组方法。其中许多已经介绍过。下面简单介绍一些新方法。

void setLocation(double x, double y)	将对象的位置设置到指定点
void move(double dx, double dy)	使用置换 dx 和 dy 移动对象
void movePolar(double r, double theta)	将对象在 theta(以度为单位)方向上移动 r 个单位
double getX()	返回对象的 x 坐标
double getY()	返回对象的 y 坐标
double getWidth()	返回对象的宽度
double getHeight()	返回对象的高度
boolean contains(double x, double y)	核对某个点是否在对象内部
void setColor(Color c)	设置对象颜色
Color getColor()	返回对象颜色
void setVisible(boolean visible)	设置对象是否可见
boolean isVisible()	如果对象可见，返回 true
void sendToFront()	将对象发送到画布前面，在那里它可以遮蔽更后面的对象
void sendToBack()	将对象发送到画布后面，这样它可能被前面的对象遮蔽
void sendForward()	将对象在堆栈顺序中前移一个位置
void sendBackward()	将对象在堆栈顺序中后移一个位置

图 9-6 所有 GObject 子类都支持的方法

- setLocation、move 和 movePolar 方法允许改变对象在画布上的位置。如果知道新位置的准确坐标，可以使用 setLocation 方法。然而，如果要将对象从现在的位罝移动特定距离，可以使用 move 或 movePolar 方法。例如，如果要将存储在变量 gobj 里的对象向右移动 10 个像素，调用

```
gobj.move(10, 0)
```

可以实现；如果要将对象向东北移动 100 像素，可以调用

```
gobj.movePolar(100, 45);
```

movePolar 的第二个参数是要移动的方向，在极坐标中表示为角。例如，示例中的参数 45 表示从 +x 轴逆时针旋转 45°。

- 断言方法 contains 能够确定一个对象是否包含特定点。
- setVisible 方法能够隐藏屏幕上的对象。如果调用 setVisible(false)，对象会从屏幕上消失，直到调用 setVisible(true)为止。断言方法 isVisible 能够确定对象是否可见。
- 各种 send 方法能够改变堆栈顺序。在画布上添加新对象时，它会处于其他对象上面，因此会遮蔽它背后的对象。如果调用 sendToBack，对象会移动到堆栈后面。相反，sendToFront 会将对象移动到前面。调用 sendForward 和 sendBackward，可以改变对象在堆栈中的顺序，它们分别将对象向前或向后移动一层。

虽然知道这些新方法很有用，但马上就会发现有些使用的方法图 9-6 中根本没有。例如，没有 setFilled 方法，尽管一直在填充 GOval 和 GRect。同样，图中也没有包含 setFont，尽管在 GLabel 语境中使用过这种方法。然而，即使图 9-6 中没有这些方法，也不应该太担心。正如图的标题所述，在 GObject 级别定义的方法都可以应用于 GObject 的所有子类。例如，setFont 应用于 GLabel 类，因此在那个级别定义。

setFilled 方法的情况更有趣。因为它用来填充椭圆、矩形、多边形和弧，所以 setFilled 有 4 个相应的类(加上扩充的 GRoundRect 和 G3DRect 子类)。然而，填充线条、图像或标签表示意思还不太清楚。由于 setFilled 的解释对有些类没有意义，所以在 GObject 级别定义它就不合适，GObject 级别用于所有子类通用的方法。同时，在它有意义的每个子类中单独定义 setFilled 似乎是多余的。对 setFilled 而言，对每个可填充的类采用相同的方法当然很方便。

在 Java 中，定义一组方法(在层次结构中，类的子集共享这些方法)最好的方法是定义 Java 所谓的接口，它是一组类所共享的方法清单。Java 中，在接口中导出所有方法的类称为实现该接口。第 13 章将会更详细地介绍接口结构。可以在任何实现接口的类中调用接口定义的方法，现在理解这一点就足够了。

在 acm.graphics 程序包中，类 GOval、GRect、GPolygon 和 GArc 实现称为 GFillable 的接口，它指定所有可填充对象的行为。除了 GFillable 方法之外，acm.graphics 程序包还包含 GResizable 接口和 GScalable 接口，前者可以重置对象的范围，后者通过缩放系数来缩放对象。这 3 种接口指定的方法如图 9-7 所示。

图 9-7 中剩下的类可划分为如下几组：

- GCompound 类用于将各种 GObject 结合到单个对象之中，然后可以将这些对象作为一个单元操作。在某种程度上，GCompound 类是 GCanvas 和 GObject 模型的结合。使用 GCanvas，可以将图形对象添加到 GCompound。使用其他 GObject，可以将 GCompound 添加到 GCanvas，并让它显示在屏幕上。GCompound 类在许多语境中都很有用，9.4 节将详细讨论它。
- 本书没有介绍 GPen 和 GTurtle 类，尽管它们是 acm.graphics 程序包的一部分。GPen 类可以很方便地画出某类形状，对于这些形状而言，纸笔类推比用于程序包余下部分的拼贴画模型更合适。GTurtle 类在某些方面与 GPen 相似，但它专门用来教孩子们学习图形编程。GTurtle 类的灵感来自于 MIT 的 LOGO 计划(Project LOGO)，Seymour Papert 所著的《Mindstorms》一书中对此有详细介绍。

- 图表底部的类——GLabel、GRect、GOval、GLine、GArc、GImage、GPolygon 以及 GRect 的 G3DRect 和 GRoundRect 子类统称为形状类。这些类是抽象的 GObject 类的具体实现，它们各自都定义了可以添加到画布的某个形状。形状类在图形编程中经常出现，它们非常重要，有必要单独用一节来详细讨论。

GFillable (implemented by GArc, GOval, GPolygon, and GRect)
void setFilled(boolean fill) 设置对象是否填充(true 意味着填充; false 意味着轮廓)
boolean isFilled() 如果填充对象，返回 true
void setFillColor(Color c) 设置用来填充对象的颜色。如果颜色是 null，就用对象的颜色进行填充
Color getFillColor() 返回用来填充对象的颜色
GResizable (implemented by GImage, GOval, and GRect)
void setSize(double width, double height) 将对象的大小改变指定的宽度和高度
void setBounds(double x, double y, double width, double height) 将对象的边界改变为单个参数指定的边界
GScalable (implemented by GArc, GCompound, GImage, GLine, GOval, GPolygon, and GRect)
void scale(double sf) 保持对象位置不变，在各个方向上应用缩放比例系数，调整对象大小
void scale(double sx, double sy) 按指定的缩放系数，在单独在 x 和 y 方向上按比例缩放对象

图 9-7 图形接口指定的方法

9.3 使用形状类

正如 9.2 节结束时定义的，形状类是 GObject 类的具体子类，GObject 类用来在屏幕上表示实际对象。相对于各种图形对象(标签、矩形、椭圆、线条、弧、图像和多边形)有几种不同的形状类可以显示在画布上。每种形状类都继承了 GObject 的方法，但通常单独定义特别适合该子类的其他方法。本节将详细介绍这些形状类。

9.3.1 GLabel 类

GLabel 类是本节的第一个类，即使在某些方面它是这些形状类中最特殊的一个。一方面，GLabel 没有实现图 9-7 中的任何图形接口；另一方面，它使用的几何模型与其他图形对象不同，因为 GLabel 的位置不是定义在左上角。GLabel 类使用的几何模型如图 9-8 所示。

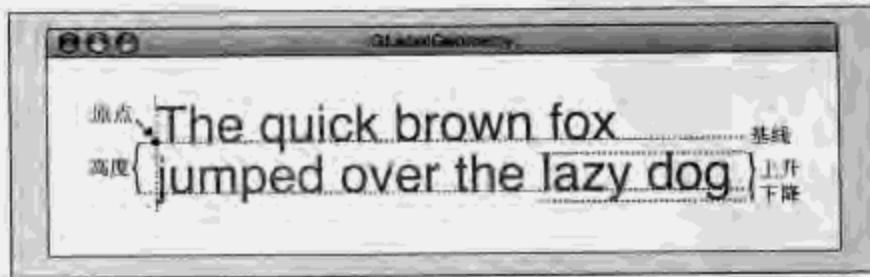


图 9-8 GLabel 类的几何排列

GLabel 类的几何模型与排字机已经使用了几个世纪(从 Gutenberg 发明印刷机至今)的几何模型相同。当然,字体的概念源自打印。打印机必须将不同的尺寸和类型风格加载到印刷中,从而控制字符出现在页面上的方法。Java 用来描述字体和标签的术语也来自于排版。如果了解如下术语,就更容易理解 GLabel 类的行为:

- 基线是放置字符的假想线。
- 原点是标签文本开始的点。在从左到右阅读的语言中,原点是基线上第一个字符左边的点,如图 9-8 所示。在从右向左阅读的语言中,原点是基线右端第一个字符右边的点。
- 字体高度是多行文本中连续基线间的距离。
- 字体上升就是从基线向上扩充的最大距离字符。
- 字体下降就是从基线向下扩充的最大距离字符。

GLabel 类导出的方法如图 9-9 所示。此列表中的第一个条目是构造函数,它有两种形式。构造函数的第一种版本前面已经见过,它读取标签的字符串以及原点的 x 和 y 坐标。第二种版本省略了原点,它将原点设置为默认值(0, 0)。

乍一看,构造函数的第二种版本似乎没有用。毕竟,如果真的在点(0, 0)上显示 GLlabel,基线以上的所有内容将从画布顶部消失。实际上,特别是创建标签时,不知道将它放置在哪里,使用第二种版本通常更方便。

构造函数
new GLLabel(String str, double x, double y) 创建新的包含字符串 str(其原点是点(x, y))的 GLLabel 对象
new GLLabel(String str) 创建新的包含 str(在点(0, 0))的 GLLabel 对象; 实际位置稍后设置
获得和设置标签所显示文本的方法
void setLabel(String str) 将标签显示的字符串改为 str
String getLabel() 返回当前显示的文本字符串
获得和设置字体的方法
void setFont(Font f) or setFont(String description) 将字体设置为 Java font 对象,或者设置成“族-风格-大小”形式的字符串
Font getFont() 返回当前字体
检索标签及其字体几何属性的方法
double getWidth() 以当前字体显示时,返回标签的水平宽度
double getHeight() 返回 GLLabel 对象的高度,它被定义为其字体的高度
double getAscent() 返回当前字体字符扩充到基线以上的距离
double getDescent() 返回当前字体字符扩充到基线以下的距离

图 9-9 GLLabel 类导出的有用方法

许多形状类,包括 GLLabel,导出包含初始坐标的构造函数的一个版本,再导出不包含初始坐标的构造函数的另一个版本。创建对象时如果知道位置,就使用包括坐标的构造函数。如果

需要实现某种计算来确定位置，最方便的方法是创建没有指定位置的构造函数，实现必要的计算来弄清楚它的位置，然后使用包含对象的 *x* 和 *y* 坐标的 add 方法，将它添加到画布。本章稍后将介绍这种方法的几个示例。

创建 GLabel 后，使用 getLabel 和 setLabel 方法可以检索或改变其内容。例如，假设要在屏幕上包含 GLabel 来记录用户在交互式游戏中的分数，首先可以定义名为 scoreLabel 的 GLabel，如下所示：

```
GLabel scoreLabel = new GLabel("Score: 0");
```

如果将此 GLabel 添加到画布上的合适位置，它将以参数字符串指定的格式显示分数。然而，随着游戏继续，用户的分数也会改变。如果程序将分数记录在名为 score 的实例变量中，进行如下调用，就可以更新标签：

```
scoreLabel.setLabel("Score: " + score);
```

setFont 和 getFont 方法能够控制用来显示标签的字体。从第 2 章可以知道，可以将字体指定为字符串，该字符串由用连字号分开的 3 部分(字体族，风格和磅值)组成。例如，如果将 scoreLabel 的字体设置为 24 磅粗体 sans-serif 字体，可以调用

```
scoreLabel.setFont("SansSerif-bold-24");
```

也可以将 Java Font 对象作为 setFont 的参数，但是字符串形式通常更利于阅读。

调用方法 getWidth、getHeight、getAscent 和 getDescent，可以获得有关 GLabel 几何参数的信息。getWidth 方法返回标签在水平尺寸上的像素值。注意，该结果取决于字体属性和存储在 GLabel 中的字符串值；其他 3 个方法的值与字符串值无关，它们只由字体确定。

这些属性有利于在特定位置使 GLabel 居中。例如，假设要重写第 2 章的 HelloProgram 示例，让消息在画布上居中。重写 run 方法如下：

```
public void run() {
    GLabel label = new GLabel("Hello, world");
    double x = (getWidth() - label.getWidth()) / 2;
    double y = (getHeight() + label.getAscent()) / 2;
    add(label, x, y);
}
```

为了水平居中 GLabel，必须将原点向左移动标签宽度一半的距离。该计算由 run 方法第二行中 *x* 的声明完成。在垂直方向居中标签有些诀窍。如果要让画布上的中线贯穿典型大写字母的中间，就需要设置基线，让它在中线以下上升半个字体，该计算在 *y* 的声明中表示。此行将字体向上调整，因为向下移动时，*y* 值增加。

如果您是一个追求审美细节的人，就会发现使用 getAscent 来垂直居中 GLabel 通常不能产生最佳结果。在画布上显示的大多数标签看起来低几个像素。原因是 getAscent 返回字体的最大上升，不是特定 GLabel 文本提升基线的距离。对于大多数字体而言，字符(特别是圆括号和重音符号)扩充到大写字母顶部，所以增加了字体上升，超出了它本身一个典型字符串的高度。如果要让它看起来完美，可以将垂直居中调整一两个像素。

9.3.2 GRect 类及其子类(GRoundRect 和 G3DRect)

如果 GLabel 是 acm.graphics 程序包中最特殊的一个类，那么 GRect 就是最传统的一个类。

它实现所有3个特殊接口——GFillable、GResizable 和 GScalable，除了整个 GObject 层次结构通用的那些方法之外，它们在其他方面没有定义方法。

和 9.3.1 小节中介绍的 GLabel 类一样，GRect 类导出构造函数的两个版本。一种到现在还在使用的版本，其形式如下：

```
new GRect(x, y, width, height)
```

它包含了左上角的坐标和矩形的尺寸。第二种形式是：

```
new GRect(width, height)
```

它创建指定大小的新矩形，并将它定位在原点。如果采用这种形式，当要将 GRect 添加到画布时，通常要指定其位置。要理解构造函数这两种形式之间的关系，有必要记住代码

```
GRect rect = new GRect(x, y, width, height);  
add(rect);
```

与

```
GRect rect = new GRect(width, height);  
add(rect, x, y);
```

有相同的效果。根据当时的语境，其中某个方法可能比其他方法方便。使两种方法都可用，就可以使用最适合于当前问题的一种。

如果回头看看图 9-1 中 acm.graphics 库的类图，就会发现 GRect 有两个子类——GRoundRect 和 G3DRect，它们表示两个稍有不同的矩形，这两个矩形只是在显示外观上与 GRect 不同。GRoundRect 类是圆角；G3DRect 类有阴影边界，使其具有三维效果。本书中没有使用这些类，如果想更好了解如何操作它们，可以自己试一试。

9.3.3 GOval 类

GOval 类几乎与 GRect 一样简单易懂，操作方式也一样。特别是从第 2 章开始使用 GOval 类之后，再没有深入探讨。

使用 GOval 类容易产生混淆的唯一方面是，假设点在图形之外，使用左上角作为参考位置对于椭圆的意义没有对于矩形的意义大。不知何故，当椭圆是圆形时，问题似乎更严重。在数学中，圆通常依据其圆心和半径来定义，而不是它包含的面积大小来定义。如图 5-4 中的 createFilledCircle 方法所示，可以定义方法，以恢复传统的数学解释。

尽管混淆有时来自于根据外切矩形来定义椭圆，但是在 acm.graphics 程序包中确定采用这种设计有其合理的理由。在标准 java.awt 程序包中的类使用包围矩形方法。与标准的 Java 模型保持一致可以很方便地在这两者之间转换。“愚蠢的一致性是无脑子的怪物”，尽管爱默生的这句评论可能正确，但是在特定一致性背后有足够的理由让它超出愚蠢的种类。

9.3.4 GLine 类

在 acm.graphics 程序包中，GLine 类用来构造线条画。GLine 类实现 GScalable(它让线条的起始点保持不变，线条从起始点开始延伸)，而不是 GFillable 或 GResizable。GLine 类也定义了其他几种方法，如图 9-10 所示。setStartPoint 方法允许客户改变线条的第一个端点但不改变

第二个端点；相反，`setEndPoint` 让客户改变第二个端点但不影响第一个端点。因此，这些方法在操作上与 `setLocation` 不同，它移动整个线条，而不改变其长度或方向。相应的 `getStartPoint` 和 `getEndPoint` 方法返回坐标作为 `GPoint`，它将单个 `x` 和 `y` 值结合为单个对象。

构造函数

```
new GLine(double x0, double y0, double x1, double y1)
```

创建连接点(x0,y0)和(x1,y1)的GLine 对象

单独获得和设置端点的方法

```
void setStartPoint(double x, double y)
```

将线条起点的坐标重新设置为点(x, y)，终点不变

```
GPoint getStartPoint()
```

返回线条起点的坐标

```
void setEndPoint(double x, double y)
```

将线条终点的坐标重新设置为点(x, y)，起点不变

```
GPoint getEndPoint()
```

返回线条终点的坐标

图 9-10 GLine 类导出的有用方法

`GLine` 类另一个值得强调的属性是它解释 `contains` 方法的方式，`contains` 方法是为所有 `GObject` 类的子类而定义的。对于 `GRect` 类和 `GOval` 类而言，由于形状有界定的内部，所以“容纳”有明显的含义。然而，在数学上，直线无限细，所以没有内部。然而实际中，如果点“足够接近”，以至于被认为是线条的一部分，那么将点定义为包含在线条内部就很有意义。在 `acm.graphics` 程序包中，此距离由 `GLine` 类中的常量 `LINE_TOLERANCE` 指定，通常定义为 1.5 个像素。

9.3.5 GArc 类

`GArc` 类用于显示弧。选择椭圆周长的一部分，就形成了弧。在 Java 中，指定界定椭圆(弧从该椭圆选取)的矩形的位置和尺寸可以定义弧的位置和尺寸。另外，指定弧开始的角和弧围绕周长移动时经过的角，就可以表示相应的椭圆部分。这两个值分别称为起始角和扫描角。和 `acm.graphics` 程序包里的所有角一样，起始角和扫描角用从 +x 轴逆时针旋转的角表示。因此，弧的完整几何图形由 6 个参数指定：`x`、`y`、`width`、`height`、`start` 和 `sweep`，这些参数的关系如图 9-11 所示。

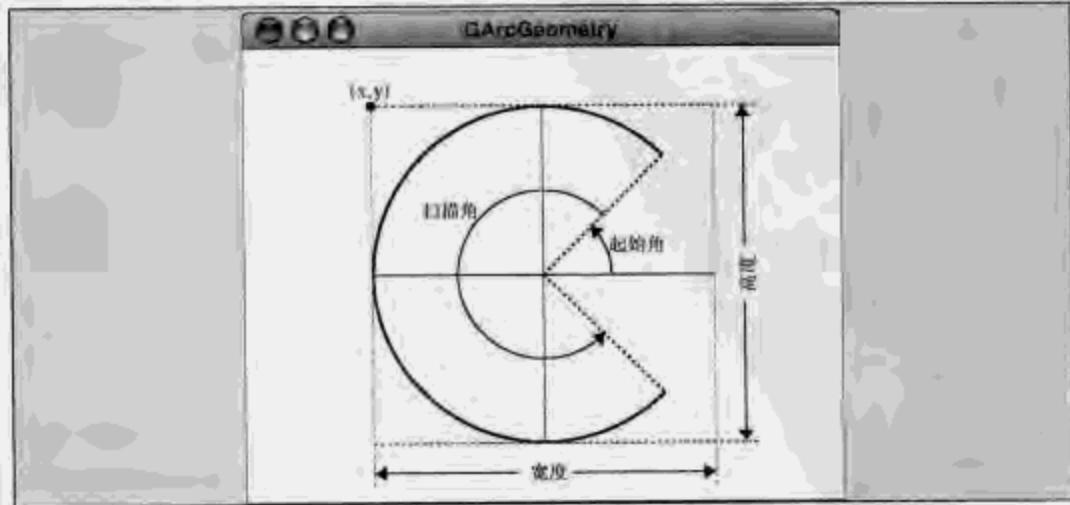


图 9-11 GArc 类的几何结构

然而, 通过示例更容易说明这些参数的效果。图 9-12 中的 4 个弧显示了其下面对应代码的结果。

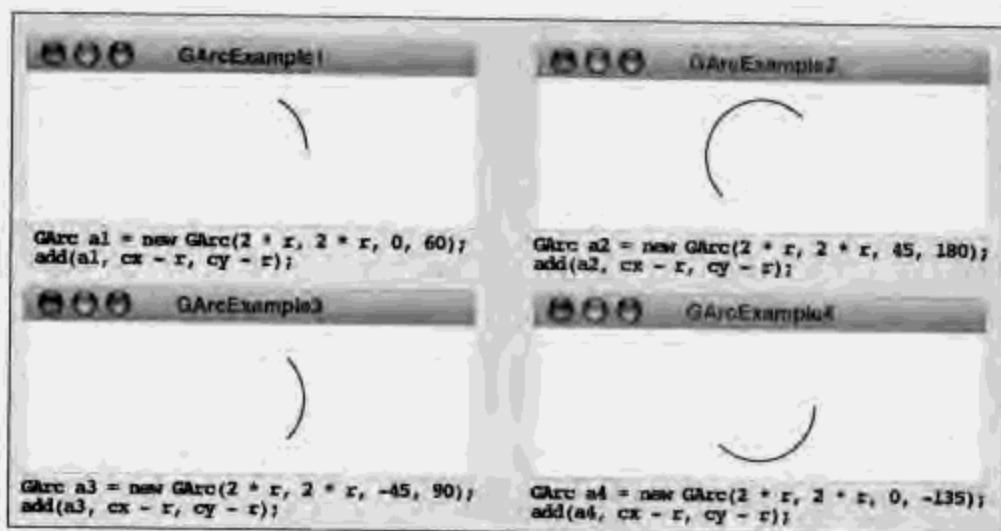


图 9-12 说明 start 和 sweep 参数的示例

这些代码片段使用不同的 start 和 sweep 值来创建和显示弧。每个弧的圆心为点(cx, cy), 半径为 r 像素。注意, start 和 sweep 的值可以为负数, 如图 9-12 中下面两个示例所示。负角表示以顺时针方向转动。

GArc 类实现 GFillable 接口, 即可以调用 setFilled 方法来控制是否填充弧。然而, 填充弧是什么意思还不明显。Java 设计师选择的弧填充的解释起初似乎有点混乱, 因为 GArc 未填充的版本不只是其填充版本的边界。如果显示未填充的 GArc 类, 显示的只是弧本身。如果对弧调用 setFilled(true), Java 会将弧的端点连接到弧中心, 然后填充该区域内部。图 9-13 所示为同一个 60° 弧未填充和填充版本的区别。



图 9-13 填充的弧和未填充的弧

产生该输出效果的 run 方法代码如下:

```

public void run() {
    double r = 50;
    GArc openArc = new GArc(2 * r, 2 * r, 0, 60);
    add(openArc, 0.3 * getWidth() - r, (getHeight() - r) / 2);
    GArc filledArc = new GArc(2 * r, 2 * r, 0, 60);
    filledArc.setFilled(true);
    add(filledArc, 0.7 * getWidth() - r, (getHeight() - r) / 2);
}

```

本示例最重要的是要说明，如果将 GArc 设置为填充，它的几何边界就变了。填充的弧是楔形区域，它有明确的内部。未填充的弧只是从椭圆周长选取的一部分。

正如线段一样，未填充的弧不包括任何点。因此，GArc 类 contains 方法的解释也取决于弧是否填充。对于未填充的弧而言，容纳说明弧点确实在弧上，与 9.3.4 小节对线上的点的描述中“紧闭”的解释相同。对于填充的弧而言，“容纳”说明包含在楔形内。

如果要使用 GArc 类显示楔形的轮廓，最简单的方法是创建弧，将它设置为填充，然后将填充颜色设置为背景色。这样就可以创建一个 60° 弧的轮廓，使用的 run 方法的代码如下：

```
public void run() {
    double r = 50;
    GArc outlinedArc = new GArc(2 * r, 2 * r, 0, 60);
    outlinedArc.setFilled(true);
    outlinedArc.setFillColor(Color.WHITE);
    add(outlinedArc, getWidth() / 2 - r, getHeight() / 2 - r);
}
```

以上代码在屏幕上产生的输出如图 9-14 所示。

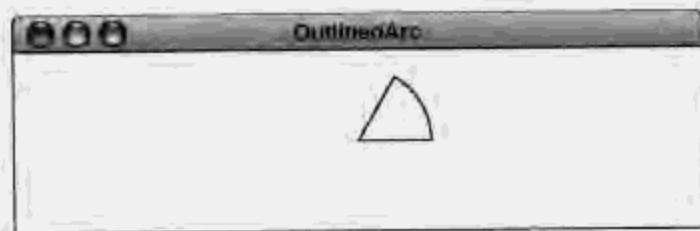


图 9-14 弧轮廓线

到目前为止在本章所见到的 GArc 示例都使用圆形弧，但也有可能产生椭圆形弧。创建这种效果有两种方法：一种方法是在构造函数中使用不同的宽度和高度值。另一种方法是利用 GArc 实现 GScalable 接口。弧是可测量的，因此可以创建圆形弧，然后将它伸展成椭圆形弧。例如，调用

```
outlinedArc.scale(4, 1);
```

可以将前面示例中的弧扩大 4 倍。如果将该尺度改变了的弧添加到画布(调整位置之后，让椭圆的中心仍然处于窗口中心)，就会看到如图 9-15 所示的图形。

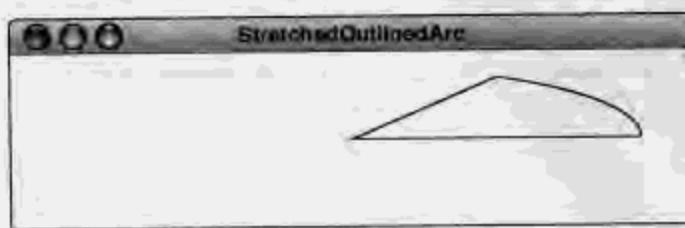


图 9-15 伸展的弧

此图形说明了一个有趣的属性，椭圆弧的起点和扫描角并不总是与出现在屏幕上的图形的几何学相一致。从 GArc 构造函数的参数可以知道，弧的扫描角是 60° ，只要弧是圆形的，楔形的两条半径线段形成的角就是 60° 。然而，如果弧是椭圆的，角就变了。在前面的示例运行中，画布中心的角很明示比 60° 要小。实际情况是，弧是椭圆周长的 $1/6(60 / 360)$ ，即使缩放比例减小了实际角的大小。实际上，考察弧缩放比例最简单的方法是确定圆上起点和扫描角的特定值，然后依比例确定每个坐标。找出椭圆上的对应端点。

GArc 类包括几种方法，这些方法超出了 GFillable 和 GScalabl 接口中定义的方法，如图 9-16 所示。

构造函数
<pre>new GArc(double x, double y, double width, double height, double start, double sweep) 创建新的由 6 个参数指定的 GArc 对象</pre>
<pre>new GArc(double width, double height, double start, double sweep) 在默认位置(0, 0)创建新的 GArc 对象</pre>
获得和设置起始角和扫描角的方法
<pre>void setStartAngle(double theta) 重新设置用来定义弧的起始角</pre>
<pre>double getStartAngle() 返回起始角的当前值</pre>
<pre>void setSweepAngle(double theta) 重新设置用来定义弧的扫描角</pre>
<pre>double getSweepAngle() 返回扫描角的当前值</pre>
检索弧端点的方法
<pre>GPoint getStartPoint() 返回弧起点的坐标</pre>
<pre>GPoint getEndPoint() 返回弧终点的坐标</pre>
检索或重新设置包含弧的框架矩形的方法
<pre>GRectangle getFrameRectangle() 返回限制椭圆(弧从该椭圆中选取)的矩形</pre>
<pre>void setFrameRectangle(GRectangle bounds) 重新设置限制弧的矩形</pre>

图 9-16 GArc 类导出的方法

9.3.6 GImage 类

GImage 类使用某种标准的图像格式，显示存储在包含编码数据的数据文件中的图像。最常见的两种格式是图形交换格式(GIF)和联合图像专家组(JPEG)格式。虽然大多数 Java 环境也能够显示其他格式的图像，但坚持使用最常见的格式可以使程序最方便易用。

显示图像的第一步是以标准格式创建图像文件。图像文件的名称应该以表示编码格式的扩展名结束，通常是.gif 或.jpg。然后将此文件存储在计算机的目录里(要使用它的 Java 程序也在该目录里)，或者存储在名为 images 目录的子目录里。在程序中，创建 GImage 对象，并将它添加到画布，方式和使用其他 GObject 子类一样。例如，有一个名为 MyImage.gif 的图像文件，使用下面的 run 方法可以将图像显示在画布的左上角。

```
public void run() {
    add(new GImage("MyImage.gif"));
}
```

相反，如果要让图像居中于画布，可以重写 run 方法，代码如下：

```
public void run() {
    GImage image = new GImage("MyImage.gif");
    double x = (getWidth() - image.getWidth()) / 2;
    double y = (getHeight() - image.getHeight()) / 2;
    add(image, x, y);
}
```

如这些示例所示，使用图像表示的机械细节不是特别复杂，因为 Java 完成了在屏幕上显示实际图像所需的工作。所要做的是将图像数据存入文件，然后将此文件的名称告诉 Java。

一个更有趣的问题是这些图像最初来自何处。一种可能是用户自己创建的图像。要这样做，需要使用数码相机或某种创建图像的软件。第二种可能是从 Web 上下载已有图像。只要图像出现在屏幕上，大多数 Web 浏览器都可以下载其相关图像文件。然而，如果真的要使用已有图像，就必须知道有关知识产权的可能限制。在 Web 上找到的大多数图像都受版权保护。按照版权法，要使用图像必须获得版权所有人的许可，除非使用图像满足“合理使用”的方针——遗憾的是，在数字时代，这种法律原则变得非常模糊。在“合理使用”的方针下，当然可以在课程论文中使用受版权保护的图像。另一方面，在没有得到版权使用许可(很可能赔偿)的情况下，不能将相同的图像用于商业出版的著作。

即使使用的图像符合“合理使用”的方针，标明其来源也很重要。通常是，在 Web 上找到要使用的图像时，首先要检查该 Web 站点是否说明了该图像的使用政策。Web 上许多最好的图像网站都对使用它们的图像有明确的方针。有些图像完全免费，有些可以免费引用，有些可以在某些语境中使用但不能在其他语境中使用，有些则完全限制使用。例如，美国国家航空和宇宙航行局的 Web 站点(<http://www.nasa.gov>)有许多有关太空探索的图像库。该 Web 站点解释说，只要在图像上包含引用“Courtesy NASA/JPL-Caltech”，就可以自由使用这些图像。

图 9-17 中 EarthImage 程序的代码说明了 GImage 类的用法，以及使用 GLabel 包括所要求的引用的策略。该图像是 1972 年 12 月阿波罗 17 上的宇航员前往月球途中看到的地球图像，它存储在名为 EarthFromApollo17.jpg 的图像文件里。EarthImage 程序将该图像读入 GImage 对象，然后将此对象及引用添加到画布上。结果产生如图 9-17 所示的显示效果。

EarthImage 程序也说明了另一个有用的方法，它可以为扩充 acm.program 程序包中 Program 类之一的任何程序设置窗口大小。Program 类的初始化检查特定子类是否定义了常量 APPLICATION_WIDTH 和 APPLICATION_HEIGHT。如果定义了，Program 类就用这些常量的值来确定程序窗口的大小；如果没有定义值，Program 类就创建一个默认大小的标准窗口。本示例中，设置了程序窗口的大小，除在底部留下 CITATION_MARGIN 像素用于引用外，地球的图像(恰好是 640×640 像素)填满了整个窗口。

```

/*
 * File: EarthImage.java
 *
 * This program displays an image of the earth from space.
 * The image appears through the courtesy of NASA/JPL-Caltech.
 */

import acm.graphics.*;
import acm.program.*;

public class EarthImage extends GraphicsProgram {
    public void run() {
        add(new GImage("EarthFromApollo17.jpg"));
        addCitation("Courtesy NASA/JPL-Caltech");
    }

    /* Adds a citation label at the lower right of the canvas */
    private void addCitation(String text) {
        GLabel label = new GLabel(text);
        label.setFont(CITATION_FONT);
        double x = getWidth() - label.getWidth();
        double y = getHeight() - CITATION_MARGIN + label.getAscent();
        add(label, x, y);
    }

    /* Private constants */
    private static final String CITATION_FONT = "SansSerif-10";
    private static final int CITATION_MARGIN = 13;

    /* Set the dimensions of the window */
    public static final int APPLICATION_WIDTH = 640;
    public static final int APPLICATION_HEIGHT = 640 + CITATION_MARGIN;
}

```

图 9-17 显示地球图像的程序



图 9-18 地球(1)

但如果事先不知道图像大小，同时又想填满屏幕上的可用区域，该怎么办呢？Java 中调整图像大小非常简单。GImage 类实现 GScalable 和 GResizable 接口，这说明应用缩放比例系数或调用有新高度和宽度的 setSize 都可以改变图像大小。因此，如下所示重写 run 方法，就可以确保地球图像及其引用可以填满屏幕。

```

public void run() {
    GImage image = new GImage("EarthFromApollo17.jpg");
    image.setSize(getWidth(), getHeight() - CITATION_MARGIN);
    add(image);
}

```

虽然上面的代码达到了用图像填满屏幕的目的，但它没有实现期望的审美效果。对于大多数图像而言，只有在各个方向以相同系数缩放图像，缩放比例才有用。在 x 和 y 方向以相同系数缩放图像可以保持其纵横比，即图像宽度除以其高度。如果允许纵横比过分变化，图像会在一个方向或另一个方向上伸展。例如，如果要在宽度远远大于高度的窗口中显示 EarthFromApollo17.jpg 图像，所看到的显示效果如图 9-19 所示。

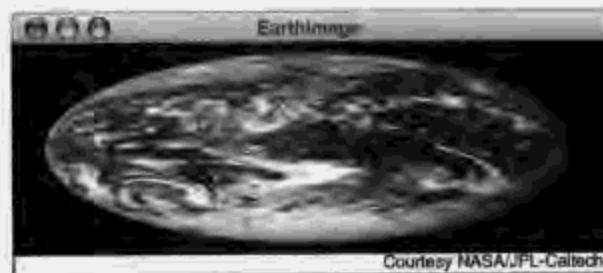


图 9-19 地球(2)

这张图片勉强能认得出是地球，但不能产生像“天堂中漂浮着的天球”这样的印象。

然而，有些缩放比例不同(至少在适度限制内)的图像，反而不影响其外观。例如，对于线图，改变它的比例只是改变每根轴上单元的大小。例如，假设名为 GlobalTemperatures.gif 的图像文件包含 1880-2005 年的年平均温度图，如图 9-20 所示。

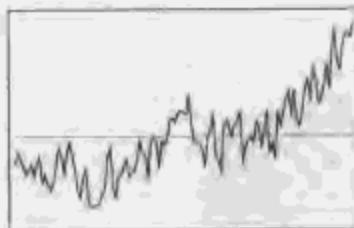


图 9-20 温度线(1)

这种类型的图像，改变纵横比不会产生太大差异。下面的示例显示了这种结果，它填满较大的窗口，同时为说明数据来源的引用留下了空间，如图 9-21 所示。

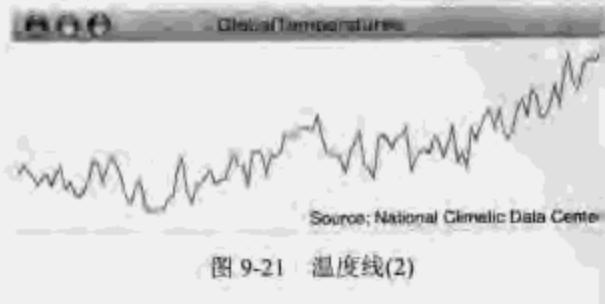


图 9-21 温度线(2)

`GImage` 类导出了一些有用的方法，这些方法超出了本章讨论的范围。在很大程度上，这些方法支持能够操作图像中单个像素的图像处理策略。第 11 章将介绍这些策略以及 `GImage` 导出的这些方法。

9.3.7 GPolygon 类

`GPolygon` 类可以显示多边形，多边形是一个数学名词，用来表示边由直线组成的闭合形状。组成多边形轮廓线的线段称为边界。一对边界的结合点称为顶点。多边形在现实中很常见。蜂房的每个单元都是六边形，这是有 6 条边的多边形的普通名称。停止标记是一个八边形，它有 8 条完全相等的边。然而，多边形不需要这么规则。例如，图 9-22 所示为 4 种适合一般定义的多边形。



图 9-22 4 种多边形

只要记住以下两点，使用 `GPolygon` 类就相对比较容易：

- 和其他形状类的构造函数不一样，`GPolygon` 类的构造函数不创建整个图形。相反，其构造函数创建一个不包含顶点的空 `GPolygon` 类。创建空的多边形之后，通过调用本节稍后描述的其他各种方法给它添加顶点。
- `GPolygon` 的位置没有定义在左上角。毕竟，许多多边形没有左上角。实际情况是，作为创建特定多边形的编程人员会选择参考点，用来定义多边形的位置。然后依据各顶点相对于参考点的位置来指定各顶点的坐标。这种设计的好处是如果在画布上移动 `GPolygon`，不需要再重新计算每个顶点的坐标。只移动参考点即可。因为顶点是相对于参考点定义的，所以重新画多边形的代码会将所有顶点转换到正确位置。

通过示例很容易说明如何创建 `GPolygon` 类。假设要创建一个表示菱形的 `GPolygon` 类，首先要确定将参考点放在哪里。对于大多数多边形而言，最方便的点是图形的几何中心。如果采用这种模型，就需要创建一个空的 `GPolygon` 类，然后给它添加 4 个顶点，指定每个顶点相对于中心坐标的坐标。假设菱形的宽度和高度存储在常量 `DIAMOND_WIDTH` 和 `DIAMOND_HEIGHT` 里，使用下面的代码可以创建菱形的 `GPolygon` 类：

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-DIAMOND_WIDTH / 2, 0);
diamond.addVertex(0, DIAMOND_HEIGHT / 2);
diamond.addVertex(DIAMOND_WIDTH / 2, 0);
diamond.addVertex(0, -DIAMOND_HEIGHT / 2);
```

执行下面的语句可以将菱形放在画布中央：

```
add(diamond, getWidth() / 2, getHeight() / 2);
```

图形窗口的显示如图 9-23 所示。

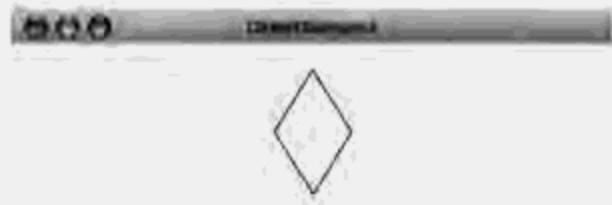


图 9-23 菱形

使用 `addVertex` 方法构造多边形时，相对于参考点表示每个顶点的坐标。在有些情况下，依据前一个顶点的坐标指定各顶点的坐标，这样更简单。为了使用这种方法，`GPolygon` 类提供了 `addEdge` 方法，除了参数指定从前一个顶点到当前顶点的位置转换之外，它与 `addVertex` 相同。所以进行下列调用，可以画出相同的菱形：

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-DIAMOND_WIDTH / 2, 0);
diamond.addEdge(DIAMOND_WIDTH / 2, DIAMOND_HEIGHT / 2);
diamond.addEdge(DIAMOND_WIDTH / 2, -DIAMOND_HEIGHT / 2);
diamond.addEdge(-DIAMOND_WIDTH / 2, -DIAMOND_HEIGHT / 2);
diamond.addEdge(-DIAMOND_WIDTH / 2, DIAMOND_HEIGHT / 2);
```

注意，仍然必须使用 `addVertex` 添加第一个顶点，但通过指定边界置换可以定义后面的顶点。

使用 `GPolygon` 类时，会发现有些多边形用对 `addVertex` 的连续调用进行定义比较方便，而有些用 `addEdge` 定义比较方便。然而，对于大多数多边形而言，更方便的是用极坐标定义边。`GPolygon` 类通过方法 `addPolarEdge` 支持这种风格。这种方法除了参数是边的长度，方向为用 `+x` 轴逆时针方向的度数表示之外，其他与 `addEdge` 相同。

有些图形知道其边的角，但需要用三角法计算顶点，`addPolarEdge` 方法能够很方便地创建这样的图形。例如，下面的方法使用 `addPolarEdge` 创建规则的六边形，其中每条边的长度都由参数 `size` 确定。

```
private GPolygon createHexagon(double side) {
    GPolygon hex = new GPolygon();
    hex.addVertex(-side, 0);
    int angle = 60;
    for (int i = 0; i < 6; i++) {
        hex.addPolarEdge(side, angle); angle += 60;
    }
    return hex;
}
```

总是用 `addVerte` 添加第一个顶点。这里，开始的顶点是六边形左边的顶点。接着，第一条边界从该点以 60° 角旋转。后面每条边长度都相同，但都是从前面一条边向右旋转 60° 角。添加全部 6 条边后，最后一条边在起始顶点结束，这样就闭合了多边形。定义这种方法之后，可以用它创建表示任何大小的规则六边形 `GPolygon`。例如，如果要将 `createHexagon` 的定义与 `run` 方法

```
public void run() {
    add(createHexagon(50), getWidth() / 2, getHeight() / 2);
}
```

结合起来，就会得到如图 9-24 所示的输出。

GPolygon 类实现 GFillable 和 GScalable 接口，而不是 GResizable。而且，对于所有 GObject 而言，都可以使用 setColor 方法设置 GPolygon 的颜色。例如，这些功能有助于绘制传统的停止标记，如图 9-25 所示。



图 9-24 六边形



图 9-25 停止标记

此程序的代码如图 9-26 所示。

```
import acm.graphics.*;
import acm.program.*;
import java.awt.*;

public class StopSign extends GraphicsProgram {

    public void run() {
        double cx = getWidth() / 2;
        double cy = getHeight() / 2;
        GPolygon sign = createOctagon(EDGE_LENGTH);
        sign.setFilled(true);
        sign.setColor(Color.RED);
        add(sign, cx, cy);
        GLabel stop = new GLabel("STOP");
        stop.setFont("SansSerif-bold-36");
        stop.setColor(Color.WHITE);
        add(stop, cx - stop.getWidth() / 2, cy + stop.getAscent() / 2);
    }

    /* Creates a regular octagon with the specified side length */
    private GPolygon createOctagon(double side) {
        GPolygon octagon = new GPolygon();
        octagon.addVertex(-side / 2, side / 2 + side / Math.sqrt(2));
        int angle = 0;
        for (int i = 0; i < 8; i++) {
            octagon.addPolarEdge(side, angle);
            angle += 45;
        }
        return octagon;
    }

    /* Private constants */
    private static final double EDGE_LENGTH = 50;
}

```

图 9-26 显示停止标记的程序

与 9.3.6 小节的 createHexagon 示例方法一样, 图 9-26 中的 StopSign 程序使用 createOctagon 方法来初始化所需形状的多边形。将创建多边形的代码与程序的余下部分分开, 是一种好的程序化分解示例。这种分解方式中, GPolygon 类的细节被封装在单个方法中。程序中的其他方法可以创建八边形, 你不必担心这些细节。

然而, 还有一种封装这种复杂性的方法, 更适合面向对象范例。这种方式不是定义创建特定多边形的方法, 而是定义 GPolygon 的子类, 它在其构造函数中起相同作用。这种策略如图 9-27 所示, 它定义了 GStar 类, 表示参考点在中心的五角星。构造函数代码自动从空的 GPolygon 类开始, 它由对 GPolygon 超类构造函数的隐式调用创建。GStar 构造函数所做的就是添加必需的顶点, 来创建五角星。

```
import acm.graphics.*;

/**
 * Defines a new GObject class that appears as a five-pointed star.
 */
public class GStar extends GPolygon {

    /**
     * Creates a new GStar centered at the origin with the specified
     * horizontal width.
     * @param width The width of the star
     */
    public GStar(double width) {
        double dx = width / 2;
        double dy = dx * GMath.tanDegrees(18);
        double edge = width / 2 - dy * GMath.tanDegrees(36);
        addVertex(-dx, -dy);
        int angle = 0;
        for (int i = 0; i < 5; i++) {
            addPolarEdge(edge, angle);
            addPolarEdge(edge, angle + 72);
            angle += 72;
        }
    }
}
```

图 9-27 显示五角星的 GPolygon 子类

遗憾的是, 在数学上, 五角星比菱形、六边形和八边形示例更复杂。构造函数的前两条线声明变量 `dx` 和 `dy`, 并将它们初始化为从星形中心到起始点(即最左边的顶点)各轴的距离。计算 `x` 方向上的距离比较容易, 因为起始点是星形宽度的一半(从中心向左的一半)。计算 `y` 方向上的距离需要一些三角形知识, 说明如图 9-28 所示。

五角星外围的每个点形成一个角, 它是整个圆的 $1/10$, 即 36° 。如果画一条线将角分为两半, 每一半都是 18° , 这条线会经过星形的几何中心, 形成图中所示的直角三角形。所以, `dy` 的值等于 `dx` 乘以 18° 的正切。

构造函数中的第三条线声明变量 `edge`, 并将它赋值为形成星形轮廓线线段的长度。计算该值所需的计算如图 9-29 所示。



图 9-28 五角星示例(1)



图 9-29 五角星示例(2)

要确定 `edge` 的值, 需要用水平线的整个长度减去其中的虚线部分, 这里用 `dx` 表示。虚线部分的长度(这里用希腊字母 Δ 表示)也可以使用三角形计算。这里, Δ 的值等于 `dy` 乘以 36° 的正切。

计算了这些初始值之后, `GStar` 构造函数余下的部分就相对简单明了。`for` 循环的每个周期都使用 `addPolarEdge` 给多边形添加两条线段。在每个周期结束时, 循环将 72 添加到 `angle` 的值, 让接下来的两条边界转动整个圆的 $1/5(72^\circ = 360^\circ / 5)$ 。

除了 `addVertex`、`addEdge` 和 `addPolarEdge` 方法之外, `GPolygon` 类导出另一种方法用来给多边形添加边。这种方法是 `addArc`, 如果定义多边形的边是直线的, 它似乎与多边形的数学概念相反。然而, `addArc` 用来将一系列短的边添加到 `GPolygon` 类, 模拟所需的弧。因此, `addArc` 可以创建可填充的形状, 这些形状可能有曲线边。`addArc` 的参数与 `GArc` 构造函数的参数相同, 指定椭圆(弧取自该椭圆)的宽度和高度, 以及弧的起始角和扫描角, 但没有明确指定弧的坐标。调用 `addArc` 方法时, 定位了近似弧边界的部分, 以便让它们连接到 `GPolygon` 中最近的顶点。

用示例很容易说明 `addArc` 方法的用法。假设要绘制门口, 门的上面一部分是半圆形弧, 运行结果如图 9-30 所示。



图 9-30 拱形门

这个门可以表示为有 3 条笔直边和一系列微小边的 `GPolygon`, 这些微小边弯曲成了拱门的顶部。下面的类定义可以实现这种形状作为 `GPolygon` 的子类:

```
public class GArchedDoor extends GPolygon {
    public GArchedDoor(double width, double height) {
        double lengthOfVerticalEdge = height - width / 2;
        addVertex(-width / 2, 0);
        addEdge(width, 0);
        addEdge(0, -lengthOfVerticalEdge);
        addArc(width, width, 0, 180);
        addEdge(0, lengthOfVerticalEdge);
    }
}
```

GArchedDoor 类的构造函数读取门的宽度和高度，其中高度从基部一直测量到弧的顶部。多边形的参考点是门基部的中心点。GPolygon 类的构造函数首先调用 addVertex 和左下角的坐标，然后使用 addEdge 创建门的底部边界和右边界。addArc 调用添加顶部的半圆弧，其直径等于门的宽度，起始角为 0°，扫描角为 180°。添加弧以后，构造函数调用 addEdge 添加门的左边界，这样就完成了多边形的轮廓。

虽然本节列出的方法对于本书使用的示例而言已经足够了，但 GPolygon 类仍然导出了其他几种方法。这些方法如图 9-31 所示。更多信息请参考 javadoc 页面。

构造函数
new GPolygon() 创建一个空的 GPolygon 对象，其参考点为(0, 0)
new GPolygon(double x, double y) 创建一个空的 GPolygon 对象，其参考点为(x,y)
给多边形添加边的方法
void addVertex(double x, double y) 在相对于多边形参考点的(x,y)添加一个顶点
void addEdge(double dx, double dy) 添加一个新顶点，其坐标从前一个顶点转移 dx 和 dy
void addPolarEdge(double r, double theta) 添加一个新顶点，其位置以相对于前一个位置的极坐标表示
void addArc(double arcWidth, double arcHeight, double start, double sweep) 添加一系列边界，模拟 GArc 构造函数风格中指定的弧
其他有用的方法
void rotate(double theta) 将多边形围绕参考点旋转角 theta 度(以度为单位)
void recenter() 调整多边形的顶点，让参考点位于其几何中心
GPoint getCurrentPoint() 返回最后添加到多边形的那个顶点的坐标，如果多边形为空，返回 null

图 9-31 GPolygon 类导出的方法

9.4 创建复合对象

acm.graphics 层次结构中还没有介绍的类是 GCompound 类，事实证明它非常有用，值得用一节来专门讨论。GCompound 类可以将几个 GObject 连接为单个单元，该单元本身就是 GObject。这种特性将抽象的概念扩充到了图形对象领域。方法允许将许多语句组合成单个单元，与此类似，GCompound 类也允许将图形对象放入有自身完整性的单个单元作为图形对象。

GCompound 类中可用的方法如图 9-32 所示。接下来的几节将用一系列示例来说明如何使用这些方法。

构造函数

```
new GCompound()
    创建新的不包含对象的 GCompound
```

从复合图形中添加和删除图形对象的方法

```
void add(GObject gobj)
    将图形对象添加到复合图形
void add(GObject gobj, double x, double y)
    在指定位置将图形对象添加到复合图形
void remove(GObject gobj)
    将指定的图形对象从复合图形中删除
void removeAll()
    将所有图形对象从复合图形中删除
```

其他方法

```
GObject getElementAt(double x, double y)
    返回包含指定点的最前面的对象。如果这个对象不存在，返回 null
GPoint getLocalPoint(double x, double y) or getLocalPoint(GPoint pt)
    返回局部坐标中相对于画布中 pt 的点
GPoint getCanvasPoint(double x, double y) or getCanvasPoint(GPoint pt)
    返回画布上相对于局部坐标中 pt 的点
```

图 9-32 GCompound 类导出的方法

9.4.1 简单的 GCompound 示例

要理解 GCompound 类如何运行，最好从简单的示例开始。假设要在画布上组合抽象的脸部，如图 9-33 所示。



图 9-33 脸

这张脸由几个独立的特征组成——用于头部的 GOval，用于眼睛的两个 GOval，用于嘴巴的 GRect 和用于鼻子的 GPolygon，需要将它们添加到合适位置。当然，可以将每个对象分别添加到画布上，和前面一样。然而，这样做很难将脸作为单元来操作，例如将脸移动到画布上的不同位置。如果单独添加每种特征，移动脸就需要移动每个特征。最好能够移动整张脸。

图 9-34 中的代码使用 GCompound 类来实现这一点。GFace 类扩充 GCompound 来创建包含所需特征的脸对象。用 GFace 构造函数创建这些特征，然后添加到合适的位置。构造函数以这种形式编码，即每种特征的大小依据脸整体的宽度和高度表示，这样较小的脸部眼睛也相对较小。

构造函数创建各种特征之后，接着依次添加它们，使用如下调用：

```
add(nose, 0.50 * width, 0.50 * height);
```

```

/*
 * File: GFace.java
 *
 * This file defines a compound GFace class.
 */

import acm.graphics.*;

public class GFace extends GCompound {

    /** Creates a new GFace object with the specified dimensions */
    public GFace(double width, double height) {
        head = new GOval(width, height);
        leftEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
        rightEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
        nose = createNose(NOSE_WIDTH * width, NOSE_HEIGHT * height);
        mouth = new GRect(MOUTH_WIDTH * width, MOUTH_HEIGHT * height);
        add(head, 0, 0);
        add(leftEye, 0.25 * width - EYE_WIDTH * width / 2,
            0.25 * height - EYE_HEIGHT * height / 2);
        add(rightEye, 0.75 * width - EYE_WIDTH * width / 2,
            0.25 * height - EYE_HEIGHT * height / 2);
        add(nose, 0.50 * width, 0.50 * height);
        add(mouth, 0.50 * width - MOUTH_WIDTH * width / 2,
            0.75 * height - MOUTH_HEIGHT * height / 2);
    }

    /* Creates a triangle for the nose */
    private GPolygon createNose(double width, double height) {
        GPolygon poly = new GPolygon();
        poly.addVertex(0, -height / 2);
        poly.addVertex(width / 2, height / 2);
        poly.addVertex(-width / 2, height / 2);
        return poly;
    }

    /* Constants specifying feature size as a fraction of the head size */
    private static final double EYE_WIDTH      = 0.15;
    private static final double EYE_HEIGHT     = 0.15;
    private static final double NOSE_WIDTH     = 0.15;
    private static final double NOSE_HEIGHT    = 0.10;
    private static final double MOUTH_WIDTH   = 0.50;
    private static final double MOUTH_HEIGHT  = 0.03;

    /* Private instance variables */
    private GOval head;
    private GOval leftEye, rightEye;
    private GPolygon nose;
    private GRect mouth;
}

```

图 9-34 使用 GCompound 定义的“图形脸”类

该语句添加鼻子，让它居于脸部中央。然而，重要的是要明白，`add` 调用没有将鼻子添加到画布，而是添加到 `GCompound`。到目前为止使用的 `add` 方法都是在 `GraphicsProgram` 中定义的。因为 `GFace` 类扩展了 `GCompound`，所以它使用在其超类里定义的 `add` 方法。除了将对象作为整体之外，将对象添加到 `GCompound` 直观上与添加到画布类似。只移动 `GCompound` 就可以移动 `GCompound` 中的所有元素。

9.4.2 GCompound 坐标系

和 GPolygon 类一样, GCompound 类也定义自己的坐标系, 其中所有坐标值都相对于参考点表示。这种设计可以定义 GCompound 类而不必知道它会出现在画布的什么位置。GCompound 的组件绘制之后, 会被转换到合适的位置。如果要在 GCompound 的局部坐标系和画布的坐标系之间来回转换, 可以使用方法 getCanvasPoint 和 getLocalPoint, 图 9-32 中的 GCompound 方法列表中包含这两种方法。

GCompound 类保持维护自己的坐标系, 这一点非常有用, 因为它允许改变图形对象的参考点, 这在某种程序上简化了常见的数学和物理计算。例如, 考察如何在画布上表示弹跳的球。一般的解决方案是使用 GOval 类, 使用相等的宽度和高度, 让它看起来是个圆。然而, 这种策略的问题是 GOval 的参考点位于左上角。如果要执行涉及球的所有物理计算, 最好将球的位置定义为其中心。

将球的参考点从角改变为中心最简单的方法是使用 GCompound 类。例如, 图 9-35 中的代码定义了一个新的 GBall 类, 其位置总是位于球的中心。

```
/*
 * File: GBall.java
 *
 * -----
 * This file defines a GCompound class that represents a ball.
 */

import acm.graphics.*;

/**
 * This class defines a GCompound subclass that represents a ball
 * whose reference point is the center rather than the upper
 * left corner.
 */
public class GBall extends GCompound {

    /** Creates a new ball with radius r centered at the origin */
    public GBall(double r) {
        GOval ball = new GOval(2 * r, 2 * r);
        ball.setFilled(true);
        add(ball, -r, -r);
        markAsComplete();
    }

    /** Creates a new ball with radius r centered at (x, y) */
    public GBall(double r, double x, double y) {
        this(r);
        setLocation(x, y);
    }
}
```

图 9-35 使用 GCompound 创建由其中心定义的球

9.4.3 使用 GCompound 的对象分解

在第 5 章讨论分解的时候, 主要示例是绘制有 3 节车厢的火车, 如图 9-36 所示。

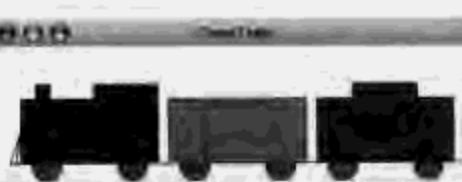


图 9-36 火车

从前面的讨论中可以知道, DrawTrain 程序提供许多将问题分解为一系列简单问题的方式, 这些方式不只是作为减少问题复杂性的策略, 而且可以作为利用结构内普通元素的方式。

虽然单个方法的逐步细化在面向对象中非常重要, 但是它不是唯一可用的分解策略。许多情况下, 包括 DrawTrain 示例, 通过创建类的层次结构来分解问题非常有效, 该结构反映了对象间的关系。

如果将火车图形的元素作为对象考察, 车厢就属于 3 种不同的类: 火车头、货车车厢和守车。统一这些类的是它们都是火车车厢, 共享更一般类的某些属性。以这种方式看待问题暗示了图 9-37 所示的类层次结构。

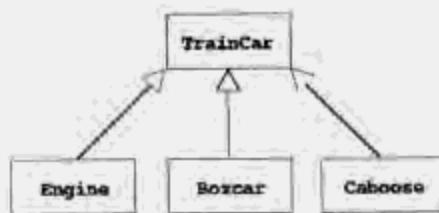


图 9-37 火车的类层次结构

每个具体的类(Engine、Boxcar 和 Caboose)都是抽象类 TrainCar 的子类, TrainCar 类定义了所有火车车厢的共享属性。

然而, 该类图没有显示 TrainCar 类本身的超类。要理解超类是什么, 重要的是理解要用对象实现什么操作(这些对象是 TrainCar 类的实例)。虽然其他操作对火车车厢有意义, 但要做的事情之一是将它显示在 GCanvas 上, 以便它可以显示在屏幕上。因此, TrainCar 一定是某种类型的 GObject。而且, 因为每节火车车厢由组成图片的图形对象构成, 所以 TrainCar 的超类最自然的选择就是 GCompound, 因为它是可以包含其他 GObject 的 GObject 的子类。因此, 更完整的类层次结构图如图 9-38 所示。

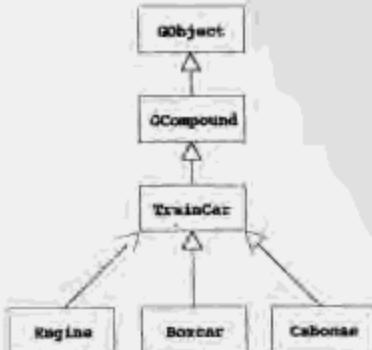


图 9-38 火车完整的类层次结构

在实现层次结构底部的类时，可以利用不同火车车厢类型的共性，第5章中定义drawCarFrame方法就很有用。TrainCar类的构造函数可以添加所有火车车厢共有的图形对象，Engine、Boxcar和Caboose类中的构造函数可以添加完成图形所需的其他对象。为了便于理解它如何运行，Boxcar和TrainCar类的定义如图9-39和图9-40所示。Engine和Caboose的相关定义留在练习13中完成。

```
/*
 * File: Boxcar.java
 *
 * -----
 * This file defines the boxcar class as a subclass of the more
 * general TrainCar class.
 */

import acm.graphics.*;
import java.awt.*;

/**
 * This class represents a boxcar. Like all TrainCar subclasses,
 * a Boxcar is a graphical object that you can add to a GCanvas.
 */
public class Boxcar extends TrainCar {

    /**
     * Creates a new boxcar with the specified color.
     * @param color The color of the new boxcar
     */
    public Boxcar(Color color) {
        super(color);
        double xRightDoor = CONNECTOR + CAR_WIDTH / 2;
        double xLeftDoor = xRightDoor - DOOR_WIDTH;
        double yDoor = -CAR_BASELINE - DOOR_HEIGHT;
        add(new GRect(xLeftDoor, yDoor, DOOR_WIDTH, DOOR_HEIGHT));
        add(new GRect(xRightDoor, yDoor, DOOR_WIDTH, DOOR_HEIGHT));
    }

    /* Dimensions of the door panels on the boxcar */
    private static final double DOOR_WIDTH = 18;
    private static final double DOOR_HEIGHT = 32;
}
```

图9-39 Boxcar类的代码

定义了这些类之后，使用如下所示的run方法，就可以创建具有3节车厢的火车，并让它位于窗口底部的中间：

```
public void run() {
    double trainWidth = 3 * TrainCar.CAR_WIDTH + 4 * TrainCar.CONNECTOR;
    double x = (getWidth() - trainWidth) / 2;
    double y = getHeight();
    double dx = TrainCar.CAR_WIDTH + TrainCar.CONNECTOR;
    add(new Engine(), x, y);
    add(new Boxcar(Color.GREEN), x + dx, y);
    add(new Caboose(), x + 2 * dx, y);
}
```

```

/*
 * File: TrainCar.java
 *
 * -----
 * This file defines the abstract superclass for all train cars.
 */

import acm.graphics.*;
import java.awt.*;

/** This abstract class defines what is common to all train cars */
public abstract class TrainCar extends GCompound {

    /**
     * Creates the frame of the car using the specified color.
     * @param color The color of the new boxcar
     */
    public TrainCar(Color color) {
        double xLeft = CONNECTOR;
        double yBase = -CAR_BASELINE;
        add(new GLine(0, yBase, CAR_WIDTH + 2 * CONNECTOR, yBase));
        addWheel(xLeft + WHEEL_INSET, -WHEEL_RADIUS);
        addWheel(xLeft + CAR_WIDTH - WHEEL_INSET, -WHEEL_RADIUS);
        double yTop = yBase - CAR_HEIGHT;
        GRect r = new GRect(xLeft, yTop, CAR_WIDTH, CAR_HEIGHT);
        r.setFilled(true);
        r.setFillColor(color);
        add(r);
    }

    /**
     * Adds a wheel centered at (x, y)
     */
    private void addWheel(double x, double y) {
        GOval wheel = new GOval(x - WHEEL_RADIUS, y - WHEEL_RADIUS,
                               2 * WHEEL_RADIUS, 2 * WHEEL_RADIUS);
        wheel.setFilled(true);
        wheel.setFillColor(Color.GRAY);
        add(wheel);
    }

    /**
     * Dimensions of the frame of a train car
     */
    protected static final double CAR_WIDTH = 75;
    protected static final double CAR_HEIGHT = 36;

    /**
     * Distance that the bottom of a train car rides about the track
     */
    protected static final double CAR_BASELINE = 10;

    /**
     * Width of the connector, which overlaps between successive cars
     */
    protected static final double CONNECTOR = 5;

    /**
     * Radius of the wheels on each car
     */
    protected static final double WHEEL_RADIUS = 8;

    /**
     * Distance from the edge of the frame to the center of the wheel
     */
    protected static final double WHEEL_INSET = 16;
}

```

图 9-40 TrainCar 类的代码

这种方法除了以下两点不同之外，其他的则与第 5 章的方法相同：

- 命名常量 CAR_WIDTH 和 CONNECTOR 在 TrainCar 类中定义，因此必须标记类的名称，如 TrainCar.CAR_WIDTH。
- 不是调用方法来画单个车厢，程序中的最后 3 条语句调用相应的构造函数来创建单个车厢，然后将每节车厢添加到画布。

然而，这种 run 方法的实现方式存在一些不足：代码必须计算每节火车车厢的坐标。执行计算的表达式占据了代码的一半，阅读起来很困难。更重要的是，如果要给火车添加新车厢，就必须改变这些表达式。最好将这些计算集成到创建图形对象的代码中。最有效的策略是定义新的 Train 类，它可以包含单个 TrainCar 对象，如 9.4.4 小节所述。

9.4.4 嵌套 GCompound 对象

GCompound 类最重要的特征之一是它可以包含其他 GCompound 对象。这样它就可能组合具有层次结构的图形结构。画火车的问题提供了特别具有说服力的示例。它说明了这种层次结构是多么有用。同样，各种火车类的客户对画完整火车的兴趣超过了画单个车厢的兴趣。定义 Train 类作为包含单个 TrainCar 对象的 GCompound，就可以将火车作为整体，充分简化编程过程。

这种设计的基本思想是定义 Train 类，该类的构造函数创建空的火车。创建 Train 对象之后，调用 append 方法可以将车厢添加到火车尾部。这样，要创建由火车头、绿色货车车厢和守车 3 节车厢的火车，可以使用如下代码：

```
Train train = new Train();
train.append(new Engine());
train.append(new Boxcar(Color.GREEN));
train.append(new Caboose());
```

要让火车位于窗口底部中间，可以利用 GObject 类中的 getWidth 方法可用于 GCompound 对象这一点，就像它用于其他图形对象一样。因此，只需要知道火车的长度，然后用画布中心的坐标减去火车宽度的一半即可。因此，将火车置于基线中间的代码如下：

```
double xc = getWidth() / 2;
add(train, xc - train.getWidth() / 2, getheight());
```

在实现 Train 类本身的过程中也可以利用 getWidth 方法。给火车添加第一节车厢时，其参考点是复合坐标系中的(0, 0)。添加后面的车厢时，要确保新车厢添加到了火车尾部。为此，可以读取现有火车的宽度，减去连接器的宽度，因为连接器在邻近车厢上重叠。这种理解就是实现 Train 类所需的，如图 9-41 所示。

使用 Train 类最大的好处不是不用计算单个车厢的坐标，而是可以将整个火车作为整体操作。例如，这样就可以很容易将火车变成动画，让它在屏幕上移动。例如，下面的 run 方法创建了具有 3 节车厢的火车，让它居中窗口，单击时，火车就会向左移动，直到从窗口消失。

```
public void run() {
    double xc = getWidth() / 2;
    Train train = new Train();
    train.append(new Engine());
    train.append(new Boxcar(Color.GREEN));
    train.append(new Caboose());
    add(train, xc - train.getWidth() / 2, getheight());
    waitForClick();
    while (train.getX() + train.getWidth() >= 0) {
        train.move(-DELTA_X, 0);
        pause(PAUSE_TIME);
```

```

/*
 * File: Train.java
 *
 * -----
 * This file defines the Train class, which can contain any
 * number of train cars linked end to end.
 */

import acm.graphics.*;

/**
 * This class defines a GCompound that represents a train.
 * The primary operation is append, which adds a TrainCar
 * at the end of the train.
 */
public class Train extends GCompound {

    /**
     * Creates a new train that contains no cars. Clients can add
     * cars at the end by calling append.
     */
    public Train() {
        /* No operations necessary */
    }

    /**
     * Adds a new car to the end of the train.
     * @param car The new train car
     */
    public void append(TrainCar car) {
        double width = getWidth();
        double x = (width == 0) ? 0 : width - TrainCar.CONNECTOR;
        add(car, x, 0);
    }
}

```

图 9-41 Train 类的代码

常量 PAUSE_TIME 和 DELTA_X 控制动画的速度，这和前面的示例中一样。例如，如果要让火车每时步移动 2 个像素，让 while 循环各周期间暂停 20ms，可以这样定义这些常量：

```

private static final double PAUSE_TIME = 20;
private static final double DELTA_X = 2;

```

如果回头看看 DrawTrain 程序以前的实现方式，很快就会发现使用这些早期设计很难让火车变成动画。而利用 GCompound 类提供的性能，在单个方法调用里就可以很轻易地操纵整个火车。

9.5 小结

本章详细探讨了 acm.graphics 程序包，形成了将整个程序包当作工具的集中收集器这一认识。同时，还从整体上考察了图形程序包的设计和程序包基于的假设、约定及隐喻。本章介绍

的重点是：

- 成为程序包设计基础的假设、约定和隐喻表示其概念模型。在有效使用程序包之前，必须理解其基本模型。
- acm.graphics 程序包模型的基本部分是坐标系。acm.graphics 程序包遵守 Java 中的标准图形程序包约定，用像素指定坐标，将原点置于画布的左上角。这种坐标系不同于高中几何课程中使用的笛卡尔平面坐标，笛卡尔坐标系的原点在左下角。
- acm.graphics 程序包的基础是 GObject 类，它是可以在画布上显示的所有对象的一般超类。GObject 类本身是抽象类，也就是说没有对象的主类是 GObject。在画布上创建图形图像时，实际使用的是 GObject 类的子类，称为形状类，包括 GArc, GImage, GLabel, GLine, GOval, GPolygon, GRect, GRoundRect 和 G3DRect。
- 每个形状类都从所有图形对象共享的 GObject 类中继承了一组方法。另外，每个形状类都包含定义其行为的其他方法。通过实现一个或更多接口 GFillable, GResizable 和 GScalable，有些形状类也可以共享一般行为。
- 要显示图形对象，需要将它添加到 GCanvas，GCanvas 是 GObject “拼贴画”的背景。虽然用户可以创建自己的 GCanvas 对象，并且独立于 acm.program 程序包单独使用它们，但在大多数情况下，程序包会自动提供 GCanvas 作为 GraphicsProgram 的一部分。
- 使用 GCompound 类型可以将单个对象组合成可以作为单元操作的较大结构。

9.6 复习题

- 本章为什么将 acm.graphics 程序包中使用的图形架构描述为“拼贴画”模型？
- 在 acm.graphics 程序包中，用什么单位指定坐标？
- acm.graphics 坐标系中原点在哪里？
- 本章强调“GObject 是抽象类”。这有什么意义？
- 不要看本章的图形，画图表示 GObject 类和形状类 GArc, GImage, GLabel, GLine, GOval, GPolygon, GRect, GRoundRect 及 G3DRect 形成的类层次结构。
- GFillable 接口定义了什么方法？哪些形状类实现 GFillable？
- GResizable 和 GScalable 接口有何不同？哪些形状类是可测量的，但不是可调整大小的？
- 依据画布上显示的 GLabel 的几何特征，说明下列每个方法返回的值的意义：getWidth, getHeight, getAscent 和 getDescent。
- 描述 GArc 类的构造函数中 start 和 sweep 参数的意义。
- 在 GArc 类中，acm.graphics 程序包如何解释填充？
- 描述下列调用 GArc 构造函数产生的弧：
 - new GArc(2.0, 2.0, 0, 270);
 - new GArc(2.0, 2.0, 135, -90);
 - new GArc(2.0, 2.0, 180, -45);
 - new GArc(3.0, 1.0, -90, 180);
- 如果 GArc 构造函数的 sweep 参数是负数，表示什么意思？
- 对于 GLine 类而言，方法 setLocation 和 setStartPoint 有何不同？

14. 写 Java 语句，作为单个 GPolygon 创建图 9-42 所示多边形。

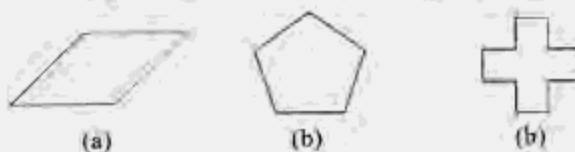


图 9-42 多边形

15. 如何获得 GraphicsProgram 使用的画布中心的坐标？

16. 用自己的话描述 GCompound 类的用途。

9.7 编程练习

1. 下载或扫描一张你最喜欢的图片或漫画到图像文件中，然后编写 GraphicsProgram，将它和表示来源的引用显示在屏幕上。

2. 编写 GraphicsProgram 程序，在画布中间画一个填充为黑色的正方形。程序运行后，为了让它变得更生动，让正方形每秒钟都改变一种颜色，新颜色通过调用 RandomGenerator 类中的 nextColor 方法随机选择。

3. 编写 GraphicsProgram 程序，让它为每种颜色名称 RED、ORANGE、YELLOW、GREEN、CYAN、BLUE 和 MAGENTA 创建 GLabel。然后将这些标签随机放在屏幕上。然而，每个标签的颜色应该从列出的颜色中随机选择，因此 GREEN 的 GLabel 应该是除了绿色之外的任何颜色。文本是一种意思，而颜色是另一种意思时，有人就很难确定标签的颜色。

4. 使用图 9-27 中的 GStar 类作为模型，定义新类 GDiamond、GTrapezoid 和 GTShape，这些类扩充 GPolygon，可以产生第 9.3.7 小节图 9-22 所示的其他多边形示例。问题是理解每个构造函数适合什么参数。

5. 编写 GraphicsProgram 程序，画一个万圣节南瓜，如图 9-43 所示。



图 9-43 万圣节南瓜

头是橙色的圆，眼睛、鼻子和嘴巴是多边形，主体是 GRect 类。在程序中使用命名常量定义各种特征的大小。

6. 除了 jack-o'-lantern 之外，南瓜还能做成南瓜饼。编写 GraphicsProgram 程序，画一个划分为相等楔形部分的南瓜饼图片，其中划分的份数由常量 N_PIECES 指定。每一个楔形应该是单独的 GArc，里面填充为橙色，轮廓线为黑色。例如，图 9-44 所示为 N_PIECES 等于 6 时的图片。

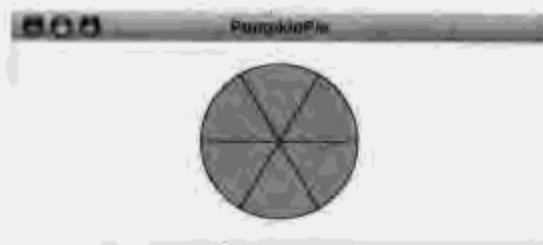


图 9-44 南瓜饼

7. 画心形图形的方法之一是，在边成对角的正方形顶部画两个半圆，如图 9-45 所示。

使用类 `GArc` 和 `GLine`，写 `GraphicsProgram`，使用此构造函数在屏幕上画一个心形。程序应该显示心形，不用画形成正方形顶部的内部线，因此，输出如图 9-43 所示。

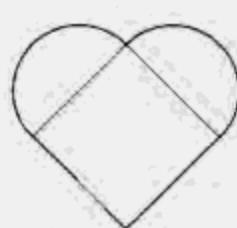


图 9-45 心形(1)

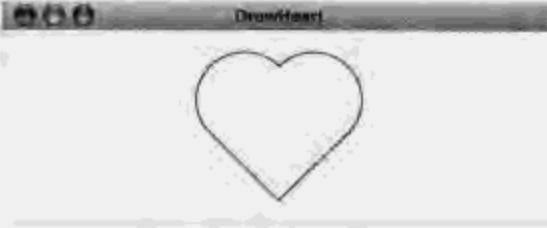


图 9-46 心形(2)

8. 重写前面练习中的 `DrawHeart` 程序，使其画整个心形作为 `GPolygon`，它包含传统的直线边和 `addArc` 方法产生的弧近似。这种风格的优点是心形可填充，即可以产生填充为红色的心形，如图 9-47 所示。



图 9-47 填充为红色的心形

9. 编写 `GraphicsProgram` 程序，画如图 9-48 所示的简单日历。

SUN	MON	TUE	WED	THU	FRI	SAT
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

图 9-48 日历

程序应该使用下面的命名常量来控制日历的格式:

```
/* The number of days in the month */
private static final int DAYS_IN_MONTH = 31;

/* The day of the week on which the month starts */
/* (Sunday = 0, Monday = 1, Tuesday = 2, and so on) */
private static final int DAY_MONTH_STARTS = 5;
```

程序应该正确生成显示月份天数所需的行数。这里，在有 31 天的月份里，从星期五开始，日历需要 6 行；如果生成的日历是非闰年的 2 月并以星期日开始，那么日历只需要 4 行。日历显示中用于表示天数的方框应该填满可用空间。

10. 第 5 章介绍了函数 `combinations(n, k)`，它返回从 n 个不同对象中选择 k 项的结果。使 `combinations` 函数形象化的经典方法之一是帕斯卡三角形(Pascal's Triangle)，它以 17 世纪法国数学家 Blaise Pascal 的名字命名，其实中国数学家在 2000 多年前就已经知道这种方法。三角形的最上面一行包含项目 `combinations(0, 0)`。下一行包含 `combinations(1, 0)` 和 `combinations(1, 1)`。模式由此继续， n 随着向下移动而增加， k 沿三角形从左向右移动而增加，如图 9-49 所示。

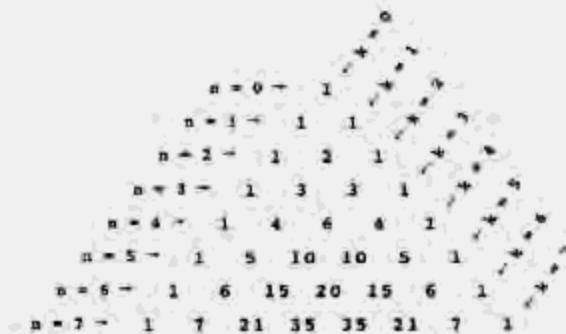


图 9-49 帕斯卡三角形(1)

三角形中，左边和右边的值是 1，三角形内部的值是上面左右对角的两个值之和。

编写 `GraphicsProgram` 程序，显示帕斯卡三角形的前面几行，如图 9-50 所示。

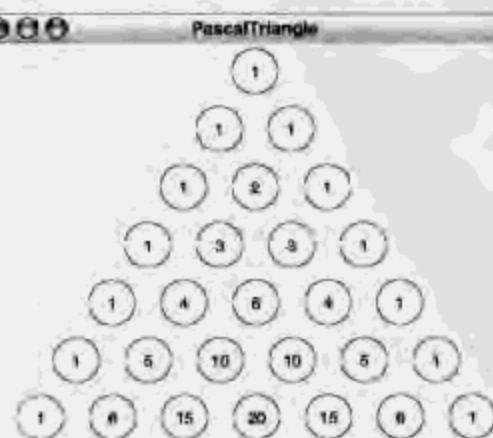


图 9-50 帕斯卡三角形(2)

顶上的圆圈应该位于窗口顶部以下几个像素的位置，且水平居中。只要新出现的行里的所有圆圈适合画布大小，程序就应该生成三角形的其他行。一旦行里的圆圈超出了窗口边界或底部，程序即停止。

11. 编写 GraphicsProgram 程序，画刺绣棉被，如图 9-51 所示。

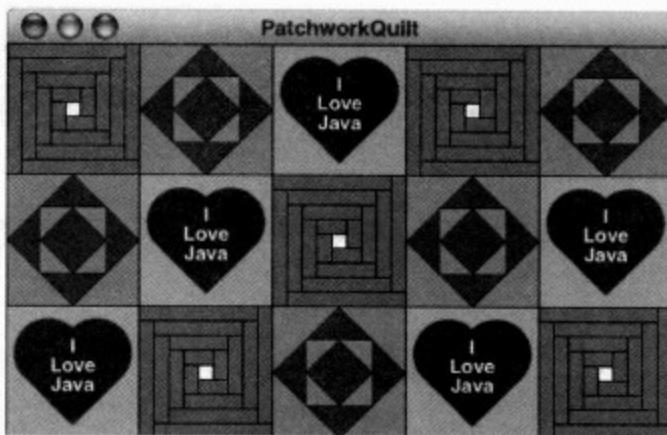


图 9-51 刺绣棉被

棉被由 3 种不同类型的块组成，如图 9-52 所示。



图 9-52 3 种基本元素

- 小木屋块。棉被中的第一个块使用传统缝被子中的流行模式。小木屋块由一系列框架组成，每一个框架都嵌套在外面一个较大的框架内。每个框架依次由 4 个矩形组成(本示例中是绿色的)，排列成一个正方形。
- 嵌套正方形块。第二个块也是一种传统模式，它堆叠颜色交互变化的正方形，后一个正方形相对于前一个正方形旋转 45°，然后改变其大小，让它刚好完全在前一个正方形内部。本图中，正方形填充为蓝绿色和洋红。
- I Love Java 块。这一块不是传统模式，它需要使用更多的 GObject 类。这种模式中的每个正方形都包含添加在粉红背景上的红色心形。心形中间是文字 I Love Java，每个单词一行。

组织实现方式特别有效的方式之一是定义名为 QuiltBlock 的抽象类，它扩充 GCompound。每个棉被块都可以定义为 QuiltBlock 的子类，与单个火车车厢是更一般抽象类的子类一样。

12. 在 Java 中，使用填充的 GArc 很容易画 PacMan 系列的标题特征。第一步，编写 GraphicsProgram 程序，在画布最左边添加一个 PacMan 图形，如图 9-53 所示。

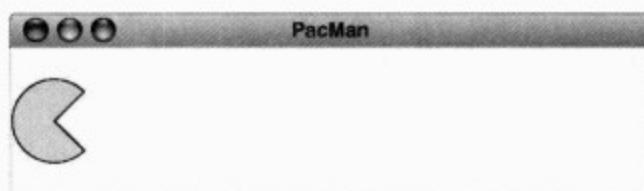


图 9-53 PacMan(1)

完成这一步后，添加代码，让 PacMan 图形向右移动，直到消失于画布边界。移动过程中，改变起始角和扫描角，让嘴巴看起来张开又合上，如图 9-54 所示。

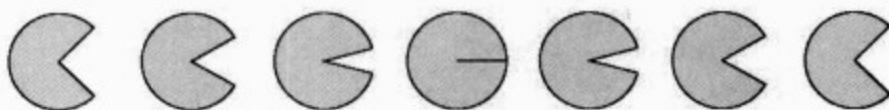


图 9-54 PacMan(2)

13. 编写 Engine 和 Caboose 类的定义，完成面向对象的 DrawTrain 程序的实现方式。
14. 如果将第 6 章练习 5 中介绍的赌博机程序从 ConsoleProgram 改变为 GraphicsProgram，那么它就会变得更有趣。假设有如图 9-55 所示的图像文件。

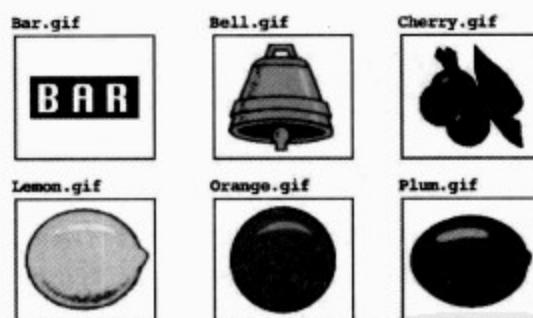


图 9-55 赌博机示例图像

除了这 6 种图像之外，名为 Empty.gif 的图像文件也很有用，它是同等大小的空方框。
重写赌博机程序，让它使用这些图像及一些 GLabel 对象来显示机器每次旋转后的结果。
程序开始时应该放置 3 个包含 Empty.gif 图像的方框，其显示如图 9-56 所示。

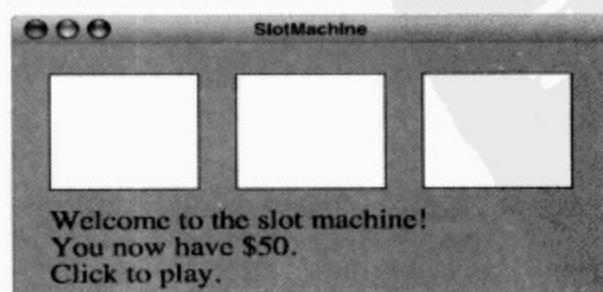


图 9-56 赌博机图像(1)

单击(调用图 9-2 中介绍的 waitForClick 方法可以检测)时，应该生成 3 个随机符号，并将

它们放在方框内，同时更新 `GLabel`，以便让用户知道游戏过程。例如，单击后，可以看到如图 9-57 所示的配置。



图 9-57 赌博机图像(2)

每次旋转前，重新将方框设置为空，在显示每个新符号后调用 `pause`，这样可以增加游戏的悬念。

15. 定义新的 `GLens` 类来表示凸透镜，如图 9-58 所示。



图 9-58 凸透镜

从几何学上来说，凸透镜由两个较大圆的交叉部分形成，如图 9-59 所示。



图 9-59 凸透镜的形成示例

每个圆的半径都是透镜宽度和高度的函数。运用勾股定理及一些几何知识，很容易计算出其关系。为了避免麻烦，给出半径 r 的公式：

$$r = \frac{h^2 + w^2}{4w}$$

知道半径之后，使用图 9-60 所示的几何关系，可以计算出每个圆圆心的位置：

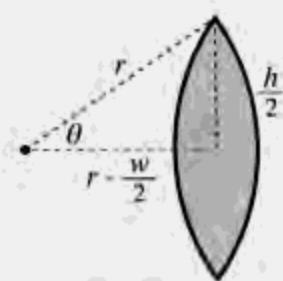


图 9-60 圆心的几何关系

根据这种实现方式，Glens 类应该扩充 GPolygon，并使用 addArc 方法来构造轮廓。构造函数应该用凸透镜的高度和宽度作为参数。要计算角 θ ，可以运用三角学或调用图 9-5 中所示的 GMath.angle 方法。编写测试程序，让每个凸透镜的宽度和高度取不同的值，在显示器上画几个透镜。

16. 可以使用前面练习中的 Glens 类来说明凸透镜的成像原理。光线与水平轴平行进入透镜后，会经过一个称为透镜焦点的点，从透镜中心到焦点的距离称为焦距。这些光线继续传播，在焦点的另一边会形成一个相对于观察者倒立的图像，如图 9-61 所示。



图 9-61 凸透镜成像示例

编写 GraphicsProgram 产生此图。假设有两个图像文件——Candle.gif 和 InvertedCandle.gif，它们包含图右边的蜡烛图像，和垂直方向上倒立的图像，其大小与原始图像大小相同。程序应该按比例决定 InvertedCandle.gif 图像，以便它有正确的大小(给定观察者的位置)。第 11 章将会学习如何倒转图像，到时就可以使用单个图像文件解决这个问题。

第 10 章

事件驱动程序

I claim not to have controlled events, but confess plainly that events have controlled me.

—Abraham Lincoln, letter to Albert Hodges, 1864



Alan Kay

Alan Kay 在美国空军服役时接触了计算机编程，由于他在编程技能测试中取得了高分，因此得以为早期的 IBM1401 计算机编程。Kay 退役以后，先取得了大学学位，而后在犹他州大学取得了博士学位。他的博士论文 “Reactive Engine” 扩充了 Ivan Sutherland 计算机图形的早期版本，讨论了使用编程语言——特别是 SIMULA 首先提出的面向对象的思想——创建灵活的图形系统的方法。1971 年，Kay 在 Xerox 具有传奇色彩的 Palo Alto 研究中心(Palo Alto Research Center)从事研究工作，该中心开创了许多推动现代计算的基本思想。Kay 与他的同事 Adele Goldberg 和 Dan Ingalls 一道，共同开发了 Smalltalk，这是第一种支持交互式图形的面向对象编程语言。2003 年 Kay 获得了 ACM 图灵奖。

第 4 章介绍了一种让图形程序变成动画的简单策略，它让显示随着时间的变化而变化。如果要编写程序，让这些程序像每天使用的计算应用程序那样运行，就要学习如何让这些程序对用户采取的行为产生响应，从而实现与它们的交互。

在某些情况下，前面几章里的 ConsoleProgram 示例可以归类为交互式，因为它们大多数都需要用户输入。然而，那种交互风格至少在一个重要方面与现代应用程序中使用的交互不同。在 ConsoleProgram 示例中，只有在执行过程中某些定义良好的点——最常见的是程序调用像 `readInt` 这样的方法，然后等待响应时——才能给程序提供输入。这种风格的交互称为同步交互。

因为用户的输入与程序运行同步。相反，现代用户界面都是异步的，因为在任何时候它们都允许用户发出请求，一般通过鼠标或键盘来触发特定动作。与程序运行异步发生动作，例如单击鼠标或从键盘上输入，通常称为事件。通过响应这些事件来运行的交互式程序称为事件驱动程序。本章的主要目的是介绍如何写简单的事件驱动程序。

除了事件驱动以外，许多现代应用程序也提供图形用户界面(简写为 GUI，发音与 *gooey* 类似)，它允许用户通过操作屏幕上的交互式组件来控制程序。典型的交互式组件包含可以单击的按钮，可以输入数据的字段，允许调整某些设置的滑块，以及弹出菜单(它允许从一列选项中选取)。总之，这样的交互式组件称为交互器(interactor)，因为它们能够让用户与运行的程序以方便的方式进行交互。Java 库包括多种支持基于 GUI 应用程序开发的交互器类，本章将介绍其中最重要的几个。

10.1 Java 事件模型

和理解第 9 章描述的 `acl.graphics` 程序包一样，要了解事件驱动程序如何运行，首先要理解基本的概念模型。Java 中，对象表示的每个事件都是 `java.util` 程序包中 `EventObject` 类的子类。`EventObject` 的子类表示特殊类型的事件。例如，本章将介绍下面的事件类。

- `MouseEvent`: 用来表示鼠标的动作，例如将它从一个位置移动或拖动到另一个位置，或者单击鼠标键。
- `KeyEvent`: 表示在键盘上按键。
- `ActionEvent`: 用来表示用户采取的用户界面动作，例如单击屏幕上的按钮。

因为这些事件对象包含不同的信息，所以应用于各子类的大多数方法对于该类而言是特定的。例如，如果发生 `MouseEvent` 事件，就需要知道屏幕上鼠标的坐标，以及关于鼠标动作(例如，单击、移动、拖动等)的某些指示。如果是 `KeyEvent` 事件，就要知道按的是哪个键。对于 `ActionEvent` 事件而言，要能够确定是哪个交互器生成了事件，以便决定如何响应。提供这些信息的方法将在后面几节描述。

在 Java 事件模型中，事件对象本身不执行任何动作。实际上，事件被传递给其他某个对象，该对象负责对那种特定类型的事件作出响应。这种对象称为侦听器。可以预料，不同类型的侦听器对应不同类型的事件。这样，需要有鼠标侦听器对鼠标事件作出响应，键侦听器对键盘事件作出响应，动作侦听器对用户界面动作(例如按按钮)作出响应。

与事件不同，各种侦听器类型不作为 Java 中的类实现。相反，每个侦听器类型都定义为接口。从第 9 章对 `GFillable`、`GResizable` 和 `GScalable` 接口的讨论中可以知道，接口是定义特定行为的方法的集合。任何为这些方法提供定义的类，通过在 `implements` 从句中包括接口名称作为类标题的一部分，可以声明它实现该接口。例如，如果要写对鼠标事件作出响应的 `GraphicsProgram`，可以使用标题行

```
public class MyProgram extends GraphicsProgram
    implements MouseListener
```

然后实现 `MouseListener` 接口请求的每种方法。

凑巧的是，使用 `acl.program` 程序包时，定义是多余的。`Program` 类声明自身为鼠标事件、键事件和动作事件的侦听器，并为相应接口中所需的每个方法提供默认定义。每种方法的默认版

根本没有用,但是可以在程序中重写默认实现方式,以便让方法实现一些有意义的动作。这种设计极大地简化了响应事件的过程,因为不必实现不需要的侦听器方法。然而,仍需要导入事件类的定义,它们在java.awt.event程序包定义。因此,大多数交互式程序都会包含下面的导入行。

```
import java.awt.event.*;
```

10.2 简单的事件驱动程序

在详细讨论各种事件类和侦听器接口之前,有必要介绍一个简单的事件驱动程序示例,以说明基本事件模型。图 10-1 中所示的 DrawStarMap 程序使用鼠标事件,当用户在画布上单击时,在当前鼠标位置画星星。

例如,如果在画布左上角附近单击,程序就会以当前鼠标指针位置为中心画星星,如图 10-2 所示。

```
import acm.graphics.*;
import acm.program.*;
import java.awt.*;
import java.awt.event.*;

/**
 * This program creates a five-pointed star every time the
 * user clicks the mouse on the canvas.
 */
public class DrawStarMap extends GraphicsProgram {

    /* Initializes the mouse listeners */
    public void init() {
        addMouseListeners();
    }

    /* Called whenever the user clicks the mouse */
    public void mouseClicked(MouseEvent e) {
        GStar star = new GStar(STAR_SIZE);
        star.setFilled(true);
        add(star, e.getX(), e.getY());
    }

    /* Private constants */
    private static final double STAR_SIZE = 20;
}
```

图 10-1 说明简单鼠标事件的程序



图 10-2 画星星示例

如果继续在其他位置单击，也会在相应位置画星星。例如，可以画大熊座的图片，这个星座通常称为过山车(Big Dipper)。要做的就是画每颗星星时单击一次，如图 10-3 所示。



图 10-3 大熊座图片

图 10-1 中的代码很短，只有两种方法和一些常量定义。即便如此，该程序也与到目前为止看到的其他程序存在显著不同，这一点值得详细讨论。

`DrawStarMap` 程序与其他程序之间最明显的不同是 `DrawStarMap` 没有 `run` 方法。在 `run` 方法的位置，是 `init` 方法。`init` 方法用来指定程序开始前需要执行的初始化代码。`run` 方法——如果它存在——指定程序要做什么。事件驱动程序启动时，通常不会采取特别主动的行为。对于事件驱动程序而言，通常模式是设置画布的一些初始配置，然后等待事件的发生。这种类型的操作正好由 `init` 方法完成。

`DrawStarMap` 程序中的 `init` 方法如下所示。

```
public void init() {
    addMouseListeners();
}
```

对 `addMouseListeners` 的调用将程序注册为画布内发生的所有鼠标事件的监听器。假设 `Program` 类已经定义了所需的类，并将自身声明为 `MouseListener`，那么 `Program` 的所有子类都可作为鼠标监听器。要做的就是重定义对于此特定应用程序而言至关重要的所有方法。这里，当用户单击鼠标时，程序都会将星星添加到画布上。为此，整个 `DrawStarMap` 程序必须为 `mouseClicked` 方法编写定义，在程序窗口的任何位置单击鼠标时，都会调用 `mouseClicked` 方法。

对于 `DrawStarMap` 程序而言，`mouseClicked` 方法——它必须是公有的，因为监听器方法从 Java 运行系统中调用——有如下实现方式：

```
public void mouseClicked(MouseEvent e) {
    GStar star = new GStar(STAR_SIZE);
    star.setFilled(true);
    add(star, e.getX(), e.getY());
```

在很大程度上，代码简单明了，没有使用超出所见过的图形方法的方法。第一条语句根据图 9-27 列出的 `GStar` 类的定义，创建 `GStar` 对象。接下来的一条语句确保会填充星星，而不是星星只有轮廓线。最后一条语句将星星添加到画布。

`mouseClicked` 的这种实现方式唯一的新特征是 `MouseEvent` 参数，它提供有关在哪里单击鼠标的信息。该信息存储在 `MouseEvent` 对象 `e` 里，调用 `e.getX()` 和 `e.getY()` 可以检索它。本示

例中，目标是让星星正好出现在鼠标指示的地方，因此这些值正好是设置星星的位置所必需的。

10.3 响应鼠标事件

DrawStarMap 程序中的 mouseClicked 方法只是可以用来响应鼠标事件的几种监听器方法之一。能够响应鼠标事件的监听器方法的完整列表如图 10-4 所示。每种方法都允许响应特定类型的鼠标动作，其中大多数可能是使用计算机时常见的。例如，拖动鼠标就是在移动鼠标的同时按下按钮。如果要将对象从一个地方移动到另一个地方，应用程序就会使用拖动。一般来说，在要移动的对象上按住鼠标键，然后将它拖动到新的位置。

MouseListener 接口	
void mousePressed(MouseEvent e)	按住鼠标按钮时调用
void mouseReleased(MouseEvent e)	释放鼠标按钮时调用
void mouseClicked(MouseEvent e)	“单击”（在短时间内按下和释放）鼠标按钮时调用
void mouseEntered(MouseEvent e)	鼠标进入画布时调用
void mouseExited(MouseEvent e)	鼠标退出画布时调用
MouseMotionListener 接口	
void mouseMoved(MouseEvent e)	释放按钮移动鼠标时调用
void mouseDragged(MouseEvent e)	按下按钮移动鼠标时调用

图 10-4 响应鼠标事件的标准监听器方法

10.3.1 MouseListener 和 MouseMotionListener 接口

从图 10-4 中的标题中可以发现，Java 为响应鼠标事件定义了两种不同的监听器接口。第一种是 MouseListener，它可以监听大部分涉及鼠标键的用户动作；第二种是 MouseMotionListener，它在鼠标移动时，用来跟踪鼠标。区分这两种类型监听器的原因是鼠标动作生成的事件比面向按钮动作生成的事件多。如果应用程序只由鼠标单击驱动，而不必跟踪鼠标本身，则在应用程序不必响应它没有兴趣的常见动作事件时，其运行就会更有效。在程序中调用 addMouseListeners 会激活两种类型的监听器，用户不必再为监听器方法来自两个不同的接口这一点而担心。在 Program 类的语境中，两者看起来是一样的。

10.3.2 重写监听器方法

如 10.3.1 小节所述，Program 类通过定义图 10-4 中监听器方法的各种实现方式，将自身声明为 MouseListener 和 MouseMotionListener。然而，Program 类提供的实现方式却毫无用处。例如，mouseClicked 的标准定义就是

```
public void mouseClicked(MouseEvent e) {  
    /* Empty */
```

)

因此，除非采取相反的动作，否则程序就会忽略鼠标单击及其他所有鼠标事件。然而，如果要为特定事件改变行为，就要为那种方法添加新定义。新定义会取代原始定义，也会被调用。没有重写的其他方法会默认继续进行它们要做的动作，即什么也不做。因此，只需要重写程序实际使用的方法即可。

图 10-4 中的每种方法将 `MouseEvent` 类型的对象作为其参数，`MouseEvent` 是一个定义为 Java 标准窗口系统工具箱的类。和侦听器接口本身一样，`MouseEvent` 类也在 `java.awt.event` 程序包中。

虽然 Java 的 `MouseEvent` 类包含一些用来设计复杂用户界面的方法，但本文只使用其中两种方法。假设 `MouseEvent` 存储在名为 `e` 的变量里，那么调用 `e.getX()` 和 `e.getY()` 就可以确定鼠标的位置——鼠标指针尖上的那个点。能够检测鼠标事件发生的位置，这是编写许多鼠标驱动事件所必需的，下面两小节将用示例说明这一点。

10.3.3 画线程序

鼠标交互的第一个示例是简单的画线程序，它以商业画图应用程序采取的方式运行——至少直线是这样。要在画布上绘制直线，需要在起点按住鼠标键，将鼠标拖动到另一个端点。这样做的时候，线条本身保持在画布上的更新，以便将起点与鼠标的当前点连接起来。

例如，在屏幕上某个地方按住鼠标键，然后将它向右拖动 1inch，在此过程中不要松开按钮，就会得到如图 10-5 所示的图。



图 10-5 画线示例(1)

如果不释放鼠标，向下移动鼠标，显示的线条会跟随鼠标移动，就会得到如图 10-6 所示的图。



图 10-6 画线示例(2)

释放鼠标时，线条就会停留在那里。如果在同一点再按下鼠标键，再将鼠标拖动到新线条的端点，这样可以继续画其他线段，如图 10-7 所示。



图 10-7 画线示例(3)

同样，可以将鼠标移动到一个全新的位置，在画布的其他位置画一条不相连的线条。

用鼠标拖动线条时，连接起点和鼠标当前位置的线条会随着鼠标的移动伸长、缩短和改变方向。如果将起点与鼠标指针用伸长的弹性线连接起来，那么这种效果就是我们想要的，这种技术称为橡皮条。

使用橡皮条画线条的程序代码如图 10-8 所示。尽管程序执行了一项非常有趣的任务，但代码却极其简短——程序中 3 种方法的主体总共只包括 4 行代码，但每种方法都值得仔细研究。

和图 10-1 中 DrawStarMap 程序的情况一样，DrawLines 程序用 init 方法代替 run 方法。同样，唯一所需的初始化是让程序作为鼠标事件的侦听器，这通过调用 addMouseListeners 完成。

用户按住鼠标键时，调用 mousePressed 方法。在画线程序中，mousePressed 方法体创建了新的 GLine 对象，该对象在鼠标的当前位置开始和结束。GLine 在画布上就是一个点，它存储在私有实例变量 line 里，这意味着类里的其他方法可以访问它。特别是，按住鼠标键时拖动鼠标会调用 mouseDragged 方法，它将线条的端点重新设置为鼠标当前的位置。

```
import acm.graphics.*;
import acm.program.*;
import java.awt.event.*;

/* This class allows users to draw lines on the canvas */
public class DrawLines extends GraphicsProgram {

    /* Initializes the program by enabling the mouse listeners */
    public void init() {
        addMouseListeners();
    }

    /* Called on mouse press to create a new line */
    public void mousePressed(MouseEvent e) {
        line = new GLine(e.getX(), e.getY(), e.getX(), e.getY());
        add(line);
    }

    /* Called on mouse drag to reset the endpoint */
    public void mouseDragged(MouseEvent e) {
        line.setEndPoint(e.getX(), e.getY());
    }

    /* Private instance variables */
    private GLine line;
}
```

图 10-8 使用鼠标创建线条的程序

10.3.4 在画布上拖动对象

图 10-9 中的 DragObjects 程序提供了一个稍微复杂的事件驱动程序示例，它使用鼠标重新定位显示的对象。示例开始时创建了两个图形对象——一个红色的矩形和一个绿色的椭圆，与第 2 章的示例类似——然后将这些对象添加到画布。然后它调用 addMouseListeners 将程序注册为鼠标事件的监听器。

```

import acm.graphics.*;
import acm.program.*;
import java.awt.*;
import java.awt.event.*;

/** This class displays a mouse-draggable rectangle and oval */
public class DragObjects extends GraphicsProgram {

    /* Initializes the program */
    public void init() {
        GRect rect = new GRect(100, 100, 150, 100);
        rect.setFilled(true);
        rect.setColor(Color.RED);
        add(rect);
        GOval oval = new GOval(300, 115, 100, 70);
        oval.setFilled(true);
        oval.setColor(Color.GREEN);
        add(oval);
        addMouseListeners();
    }

    /* Called on mouse press to record the coordinates of the click */
    public void mousePressed(MouseEvent e) {
        last = new GPoint(e.getPoint());
        gobj = getElementAt(last);
    }

    /* Called on mouse drag to reposition the object */
    public void mouseDragged(MouseEvent e) {
        if (gobj != null) {
            gobj.move(e.getX() - last.getX(), e.getY() - last.getY());
            last = new GPoint(e.getPoint());
        }
    }

    /* Called on mouse click to move this object to the front */
    public void mouseClicked(MouseEvent e) {
        if (gobj != null) gobj.sendToFront();
    }

    /* Private instance variables */
    private GObject gobj;           /* The object being dragged */
    private GPoint last;            /* The last mouse position */
}

```

图 10-9 在画布上画对象的程序

图 10-9 中定义的第一个监听器方法是 mousePressed，当按住鼠标键时，调用该方法，如前面的示例所示。在这个应用程序中，mousePressed 方法如下：

```

public void mousePressed(MouseEvent e) {
    lastX = e.getX();
    lastY = e.getY();
    gobj = getElementAt(lastX, lastY);
}

```

前两条语句只是在变量 lastX 和 lastY 中记录鼠标的 x 和 y 坐标。从程序中可以知道，这些变量被声明为对象的实例变量而不是大多数方法里的局部变量。这说明稍后拖动对象时还需要这些值。因为方法返回时，局部变量就消失了，所以必须将这些值整体存储在与对象相关的实例变量里。

mousePressed 中的最后一条语句检查画布上什么对象包含鼠标的当前位置。这里，重要的是认识到有两种可能性。第一，可能是在对象上按住鼠标键，这意味着准备开始拖动它。第二，可能是在画布没有对象可以拖动的其他位置按住鼠标键。getElementAt 方法着眼于指定位置，返回它在那里找到的对象。如果在那个位置有多个对象，它选择堆栈顺序中在其他对象前面的对象。如果那个位置没有对象，getElementAt 返回特定值 null，表示不存在对象。其他方法将核对该值，以确定是否有可以拖动的对象。

mouseDragged 方法的代码如下：

```

public void mouseDragged(MouseEvent e) {
    if (gobj != null) {
        gobj.move(e.getX() - lastX, e.getY() - lastY);
        lastX = e.getX();
        lastY = e.getY();
    }
}

```

if 语句用来检查是否有可以拖动的对象。如果 gobj 的值是 null，说明没有可以拖动的对象，方法的余下部分就会被跳过。如果有对象，就需要将它在各方向上移动一定距离。距离不取决于鼠标的绝对位置，而是取决于它从最后一次跟踪其位置的地方移动了多远。因此，move 方法参数的位置是——对于 x 和 y 组件而言——鼠标现在的位置减去它曾经的位置。移动鼠标后，必须重新记录它的坐标，以便在调用 mouseDragged 时正确更新其位置。

图 10-4 中指定的最后一种侦听器方法是 mouseClicked，代码如下：

```

public void mouseClicked(MouseEvent e) {
    if (gobj != null) gobj.sendToFront();
}

```

此方法的目的是，通过在对象上单击，让用户能够将该对象移动到前面，从而将它从画布上其他对象的下面移出来。方法的主体也很容易阅读，如下面的句子一样：

如果当前对象存在，就将它发送到画布前面。

这里唯一的问题是如何初始化保存当前对象的实例变量 gobj。如果 mouseClicked 事件由自身产生，在图形对象上单击就不可能设置此变量。Java 运行时系统与 mouseClicked 事件一起生成 mousePressed 和 mouseReleased 事件，答案即取决于这一点。所以，gobj 变量由 mousePressed 设置，就像要拖动它一样。mousePressed 和 mouseReleased 事件都优先于 mouseClicked 事件，因为在鼠标键弹起来之前，系统不知道发生了单击的动作。

10.4 响应键盘事件

到目前为止的交互式示例都用鼠标作为事件源。按下键盘上的键也可以产生事件，它可以用相同方法使用。按键生成 KeyEvent 事件，然后被传递到将自身注册为 KeyListener 的任何对象。将键事件安排到程序里最简单的方法是，通过调用 addKeyListeners，将程序本身注册为键监听器。addKeyListeners 调用与前面示例中的 addMouseListeners 调用的运行方式相同。调用 addKeyListeners，由键盘输入而生成的所有事件都被发送到程序，并触发对图 10-10 中所示方法的调用。

对于键盘而言，图 10-10 所示的方法支持两种不同的规定。keyPressed 和 keyReleased 方法提供低级别的控制，在键盘键按下和弹起来的时候，都会调用监听器。因此，这些方法适用于按键时间长短起主要作用的应用程序。keyTyped 方法提供的控制级别稍高一点，它可用于使用键盘输入文本的应用程序。

<code>void keyPressed(KeyEvent e)</code>	按下键时调用
<code>void keyReleased(KeyEvent e)</code>	释放键时调用
<code>void keyTyped(KeyEvent e)</code>	"输入"（按下再释放）键时调用

图 10-10 响应键盘事件的标准监听器方法

所调用的用来从 KeyEvent 中摘录信息的方法取决于使用哪种风格。在 keyPressed 和 keyReleased 方法中，通过调用 KeyEvent 上的 getKeyCode 方法，就可以知道按下了哪个键。然而，返回值不是字符，而是整数代码，即 Java 事件模型设计师所说的虚拟键。KeyEvent 为大多数常见虚拟键代码定义的常量名称如图 10-11 所示。

<code>VK_A through VK_Z</code>	<code>VK_F1 through VK_F12</code>	<code>VK_UP</code>
<code>VK_0 through VK_9</code>	<code>VK_NUMPAD0 through VK_NUMPAD9</code>	<code>VK_DOWN</code>
<code>VK_COMMA</code>	<code>VK_BACK_SPACE</code>	<code>VK_LEFT</code>
<code>VK_PERIOD</code>	<code>VK_DELETE</code>	<code>VK_RIGHT</code>
<code>VK_SLASH</code>	<code>VK_ENTER</code>	<code>VK_PAGE_UP</code>
<code>VK_SEMICOLON</code>	<code>VK_TAB</code>	<code>VK_PAGE_DOWN</code>
<code>VK_EQUALS</code>	<code>VK_SHIFT</code>	<code>VK_HOME</code>
<code>VK_OPEN_BRACKET</code>	<code>VK_CONTROL</code>	<code>VK_END</code>
<code>VK_BACK_SLASH</code>	<code>VK_ALT</code>	<code>VK_ESCAPE</code>
<code>VK_CLOSE_BRACKET</code>	<code>VK_META</code>	<code>VK_PRINTSCREEN</code>
<code>VK_BACK_QUOTE</code>	<code>VK_NUM_LOCK</code>	<code>VK_INSERT</code>
<code>VK_QUOTE</code>	<code>VK_SCROLL_LOCK</code>	<code>VK_HELP</code>
<code>VK_SPACE</code>	<code>VK_CAPS_LOCK</code>	<code>VK_CLEAR</code>

图 10-11 KeyEvent 中定义的主要键常量

使用 keyTyped 方法时，调用 KeyEvent 上的 getKeyChar 方法可以确定键盘上输入的实际字符。这里，getKeyChar 方法自动识别修饰符键，如 Shift，因此按下 Shift 键，并按 A 键，样就产生了所需的大写字母字符 "A"。如果使用 keyPressed 和 keyReleased 方法，getKeyChar 方法就不可用。如果要考虑修饰符键，需要调用 KeyEvent 类中的其他方法，这些方法超出了本文

讨论的范围。

但至少还可以看到一个键侦听器的示例。扩充图 10-9 中的 DragObjects 程序就很有必要，这样，用鼠标拖动它或者使用箭头键，都可以移动当前选择的对象。这些箭头键通常更适用于精密调整。程序要做的唯一改变是将行

```
addKeyListeners();
```

添加到 init 方法的后面，然后包含如下 keyPressed 方法的定义。

```
public void keyPressed(KeyEvent e)
    if (gobj != null) {
        switch (e.getKeyCode()) {
            case KeyEvent.VK_UP: gobj.move(0, -1); break;
            case KeyEvent.VK_DOWN: gobj.move(0, +1); break;
            case KeyEvent.VK_LEFT: gobj.move(-1, 0); break;
            case KeyEvent.VK_RIGHT: gobj.move(+1, 0); break;
        }
    }
}
```

这种方法从事件读入键代码，然后用 switch 语句选择合适的动作。如果代码匹配某个箭头键的虚拟键值，方法会调用 move，将当前对象沿相应方向移动一个像素，其他所有键代码都会被忽略。

10.5 创建简单的 GUI

Arthur 听了一会儿，但他还是不明白 Ford 所说的绝大部分内容，于是他开始变得迷惑起来，他的手指沿着不太熟悉的计算机台移动，随后按了一下邻近面板上那个引人注目的红色按钮。面板亮了起来，并显示一句话“请不要再按这个按钮”。

—Douglas Adams, *Hitchhiker's Guide to the Galaxy*, 1979

既然知道了如何使用事件和侦听器来响应鼠标和键盘事件，那么就可以讨论如何使用 Java 提供的 GUI 交互器设计图形用户界面。然而，开始之前，有必要通过一个简单示例来阐述基本思想。

假设要写程序来模仿本节开头 *Hitchhiker's Guide to the Galaxy* 中的小插图。要在屏幕上放置一个按钮，可以通过颜色来标记它，按钮如下所示。



然而，使用这样的按钮之前，需要回答下面几个问题：

- 如何构造表示该按钮的对象？
- 如何在屏幕上放置该按钮？
- 如何设置，以便单击按钮时程序能作出适当响应？

第一个问题最简单。表示屏幕上按钮的标准 Java 类称为 JButton。这个类——以及本章定义的其他大多数交互器——是称为 Swing(下一节将详细讨论)的通用交互器库的一部分。最常见的形式是，JButton 类的构造函数将按钮上的标签作为字符串。因此，可以创建按钮，再将

它指派给局部变量，代码如下：

```
 JButton redButton = new JButton("Red");
```

如何在屏幕上放置按钮就比较复杂了。本章稍后将会介绍使用布局管理器来安排交互器的方法，在 GUI 设计方面，这种方法很有意义。此时，最好的方法是使用 Program 类提供的资源，这样就很容易将交互器添加到程序窗口的边框。

例如，假设要将该按钮添加到窗口底部，Java 描述为 SOUTH 边框。执行如下语句，就可以将按钮添加到相应边框：

```
 add(redButton, SOUTH);
```

4 个方向(NORTH、SOUTH、EAST 和 WEST)的名称在 Program 类中定义为常量。

如果边框区域从未添加过对象，边框就不会出现在屏幕上。然而，如果添加一个交互器到边框区域，Program 类将在合适边框的中心显示该交互器。因此，如果创建 ConsoleProgram，并将 redButton 添加到 SOUTH 边框，那么窗口的显示如图 10-12 所示。



图 10-12 HitchhikerButton 示例(1)

如果将多个交互器添加到同一个边框，Program 类会排列这些交互器，让它们沿相应的轴形成直线。NORTH 和 SOUTH 边框是水平轴，EAST 和 WEST 边框是垂直轴。这样这些交互器就位于相应区域的中央。

本章中的许多程序将它们的交互器沿 SOUTH 边框安装，和在 HitchhikerButton 示例中一样。沿底边的交互器的集合作为这种程序的图形用户界面。在这些示例中，在底部包含这些交互器的边框区域称为控件栏。

要考虑的最后一个问题时用户单击按钮时如何响应。至于 Java 架构中的其他事件，适当的策略是声明倾听按钮的监听器。这里，需要实现 actionPerformed 方法，其标题行如下：

```
 public void actionPerformed(ActionEvent e)
```

编写响应动作事件的代码包含两个部分。第一，必须将程序注册为动作事件的监听器。在 init 方法中包含下面的调用可以实现这一点，它将程序作为动作监听器添加到程序窗口中当前安装的每个按钮：

```
 addActionListener(this);
```

然后必须实现 actionPerformed，让它执行所需的动作。虽然示例中只有一个按钮，但屏幕上通常会有多个按钮。actionPerformed 的代码必须明白是哪个按钮触发了事件。最简单的策略是调用事件上的 getActionCommand，它返回按钮名称。可以使用 equals 方法来核对每个按钮名称。

图 10-13 显示了程序的完整实现方式。如果运行程序，并单击按钮，就会在屏幕上看到如图 10-14 所示消息。

```

import acm.program.*;
import java.awt.event.*;
import javax.swing.*;

/*
 * This program puts up a button on the screen, which triggers a
 * message inspired by Douglas Adams's novel.
 */
public class HitchhikerButton extends ConsoleProgram {
    /* Initializes the user-interface buttons */
    public void init() {
        add(new JButton("Red"), SOUTH);
        addActionListeners();
    }

    /* Responds to a button action */
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Red")) {
            println("Please do not press this button again.");
        }
    }
}

```

图 10-13 HitchhikerButton 示例程序



图 10-14 HitchhikerButton 示例(2)

如果忽略了指令，再次单击按钮，就会再次执行 actionPerformed 方法，这样就会重复在屏幕上显示消息，如图 10-15 所示。



图 10-15 HitchhikerButton 示例(3)

10.6 Swing 交互器层次结构

Java 编程人员用来建立图形用户界面的大多数交互器都是 Swing 库的一部分，它是 1997 年添加到 Java 的。Swing 程序包中基本类都是名为 javax.swing 的更大程序包的一部分，这说明使用这些类的程序需要包含行

```
import javax.swing.*;
```

在介绍性的文本中，讨论所有 Swing 没有意义。本书重点介绍图 10-16 所示的交互器类，

该图也显示了不同类之间的层次结构关系。单个交互器类将在下面几节介绍。如果要了解 Swing 类的更多信息，请参考 javadoc 页。

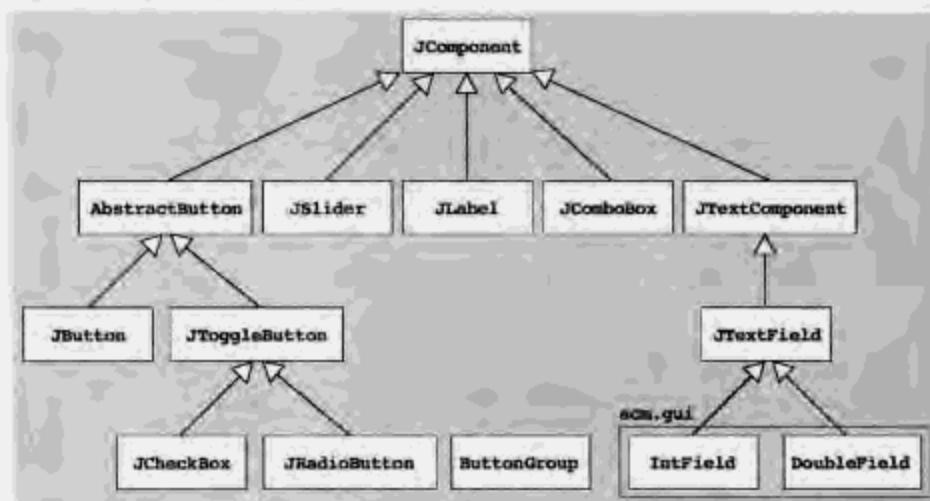


图 10-16 本文使用的交互器的类层次结构

10.6.1 JButton 类

JButton 类用来创建按钮，当用鼠标单击这些按钮时，会触发某种动作。在 HitchhikerButton 程序中已经讨论过 JButton 的示例，当单击按钮时，该示例使用 JButton 在屏幕上打印警告消息。那个示例说明了使用按钮的标准模式，它由下面几步组成：

- (1) 调用 JButton 构造函数以及要在按钮中显示的字符串来创建新按钮。
- (2) 将 JButton 添加到用户界面，该用户界面就是将按钮添加到边框的控件栏。
- (3) 将程序指派为每个 JButton 类的 ActionListener。调用按钮的 add ActionListener 方法可以完成指派，但对于程序中的所有按钮而言，在 init 方法结尾调用 add ActionListener 可以简化执行步骤。
- (4) 实现 actionPerformed 方法，它调用 ActionEvent 上的 getActionCommand 来比较按钮名称和返回的字符串，以检查按钮名称。

看看这种模式的第二个示例。接下来几段将介绍如何将按钮添加到图 10-1 中的 DrawStarMap 程序，程序通过删除所有显示的星星来清除屏幕。将下面的行添加到 init 方法，可以完成使用按钮工作的编程模式的前三步：

```

add(new JButton("Clear"), SOUTH);
addActionListeners();
  
```

第一行创建标记为“Clear”的按钮，并将它添加到程序窗口底部的控件栏，创建的用户界面如图 10-17 所示。



图 10-17 画星座图(1)

第二行确保程序成为窗口中所有按钮的动作监听器，尽管示例中只有一个按钮。

最后一步通过实现 `addActionListeners` 方法，指定对按钮的响应。一旦发现按钮的名称是“Clear”，要做的就是调用 `removeAll` 方法，它会删除画布上的所有图形对象。因此，`addActionListeners` 的实现方式如下：

```
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("Clear")) removeAll();
}
```

在有些应用程序中，程序运行时，改变 JButton 上的标签很有必要。要这样做，可以调用 `setText` 方法及新的按钮名称，相反，调用 JButton 上的 `getText` 可以返回其当前标签。

虽然本文中的 JButton 示例通常标记为字符串，但是 JButton(实际上包括层次结构中扩充 AbstractButton 的所有类)类的 Swing 实现方式也支持使用包含虚拟图标按钮。要创建这样的按钮，需要用 ImageIcon 对象取代 JButton 构造函数中的按钮名称。例如，如果有名为 RedSquare.gif 的图像文件，它包含一个小的红色正方形。用下面的代码取代第一行，就可以修改 HitchhikerButton 程序：

```
ImageIcon icon = new ImageIcon("RedSquare.gif");
JButton button = new JButton(icon);
button.setActionCommand("Red");
add(button, SOUTH);
```

如果运行上面的程序代码，单击按钮，就会看到如图 10-18 所示的程序窗口。

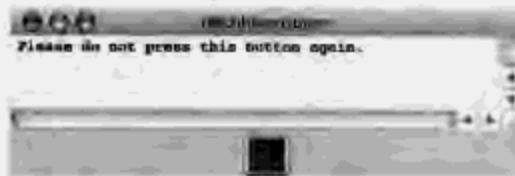


图 10-18 HitchhikerButton 示例(4)

注意，按钮的图标版本没有标签，因此需要调用 `setActionCommand` 来明确设置动作命令。

10.6.2 JToggleButton 类

JButton 类创建为效果而执行的按钮。每次单击 JButton 时，都生成动作事件，并将它传递给等待该事件的所有监听器。虽然按下鼠标键时，JButton 的外观发生了改变，但一旦释放鼠标键，它就会回到以前的状态。而单击 JToggleButton 会改变其状态。第一次单击选择按钮，并在屏幕上突出它；再次单击时，取消选择。因此，每次单击按钮时，它就在已选择和未选择状态间来回转换。可以调用 `isSelected` 来确定是否选择了 JToggleButton。`isSelected` 方法返回 Boolean 值，可以用该值来控制程序运行。

同样，JToggleButton 类本身不会广泛使用。大多数需要转换按钮开/关的应用程序会使用它的主要子类——JCheckBox 或 JRadioButton，接下来的两小节会介绍这两个类。

10.6.3 JCheckBox 类

转换按钮最常见的形式是 JCheckBox 类。因为它扩充 JToggleButton，所以 JCheckBox 类

继承了其超类的行为。因此，单击 JCheckBox 一次，就会将它打开，再单击一次就将它关闭。JCheckBox 类与其超类间唯一的不同是显示对象的方式。JToggleButton 与 JButton 看起来一样，除了按钮被选择时会突出显示之外。选择 JCheckBox 时，它显示为一个包含复选标记的近似方形的小方框。JCheckBox 的标签在方框右边。

可以使用 JCheckBox 类将其他特征添加到 DrawStarMap 程序。按照程序现在的情况看，当这些星星出现在屏幕上时，它们全部都是填充的。如果要引起对特殊星星的注意，可以让一些填充，另一些显示轮廓。因为是否填充星星的决定只有两种可能——填充或不填充——所以 JCheckBox 就是在用户界面上表示此选项的合适方法。

要实现这一扩充，需要声明 JCheckBox 变量，并用它保存星星是否填充这一状态。在程序中，需要初始化 init 方法中的 JCheckBox 变量，但创建新的星星时，需要在 mouseClicked 方法中引用该变量。如果要让变量在一种方法里初始化，在另一种方法里使用，就必须将它声明为实例变量而不是局部变量。因此，声明位于所有方法之外，代码如下：

```
private JCheckBox fillCheckBox;
```

init 方法必须创建实际的复选框，将它指派给实例变量，并添加到控件栏交互器的集合。如果要确保复选框开始时处于“选中”状态，可以将下面的行添加到 init 方法：

```
fillCheckBox = new JCheckBox("Filled");
fillCheckBox.setSelected(true);
add(fillCheckBox, SOUTH);
```

这些语句创建的控件栏如图 10-19 所示。



图 10-19 画星座图(2)

要做的改变是修改 mouseClicked 方法，以便只有选择 fillCheckBox 时，它才将星星设置为填充。重写对 setFilled 的调用可以实现这一目标，代码如下：

```
star.setFilled(fillCheckBox.isSelected());
```

这种改变能够创建一种显示，这种显示中有些星星是轮廓，有些是填充的。例如，如果要画突出织女星(天琴座最显眼的星星，也是夜空中最明亮的星星之一)的天琴座图形，可以通过在图上单击来创建织女星，然后在创建其他星星之前取消选中复选框，生成的图像如图 10-20 所示。



图 10-20 天琴座图形

10.6.4 JRadioButton 类和 ButtonGroup 类

遗憾的是，已填充和未填充的星星并没有提供创建有用星座图的灵活性。真正需要的是能够显示不同大小的星星。一种可能是让最亮的星星大一点，最暗的星星小一点，让处于中间范围的星星保持中等大小。假设现在有 3 种可能的大小，那么就无法使用复选框，而是需要从 3 个选项中选择。

在较少选项中进行选择的策略之一是使用单选按钮，它是汽车收音机上的控件名称，该收音机通过按下 5 个按钮中的某一个来选择一种预设状态。单选按钮的定义特征是，选择是唯一的，也就是说每次只能选择一个按钮。选择一个单选按钮会自动清除前面的选择。

Swing 中使用两个类实现单选按钮：JRadioButton 和 ButtonGroup。交互器本身就是 JRadioButton，与 JCheckBox 类似。唯一不同是 JRadioButton 画的是圆圈而不是方框，使用点而不是复选标记来标记已选择。ButtonGroup 类用来指定一组单选按钮，在这组单选按钮中每次只能选择其中之一。

可以使用下面的步骤创建一组单选按钮：

- (1) 为每个选项创建一个 JRadioButton 类，并将它存储在实例变量里。
- (2) 创建一个新的 ButtonGroup 类，它起初是空的。
- (3) 使用 ButtonGroup 类里的 add 方法，将按钮添加到组。
- (4) 调用要首先激活的按钮上的 setSelected。
- (5) 将每个单选按钮添加到用户界面。

将该策略应用到 DrawStarMap 程序，产生的代码如下：

```
smallButton = new JRadioButton("Small");
mediumButton = new JRadioButton("Medium");
largeButton = new JRadioButton("Large");
ButtonGroup sizeGroup = new ButtonGroup(); sizeGroup.add(smallButton);
sizeGroup.add(mediumButton);
sizeGroup.add(largeButton);
mediumButton.setSelected(true);
add(smallButton, SOUTH);
add(mediumButton, SOUTH);
add(largeButton, SOUTH);
```

如果将这些语句添加到 init 方法来取代 fillCheckBox 代码，可以创建一个如图 10-21 所示的控件栏。



图 10-21 画星座图(3)

在程序中，要做的变化是改变 mouseClicked 方法的首行，让它使用当前选择的大小代替常量 STAR_SIZE。如果通过写私有辅助方法来分解程序，那么可以用

```
GStar star = new GStar(getCurrentSize());
```

取代首行，然后像下面这样实现 getCurrentSize(假设名称 SMALL_SIZE、MEDIUM_SIZE 和 LARGE_SIZE 定义为合适的值)：

```
private double getCurrentSize() {
    if (smallButton.isSelected()) return SMALL_SIZE;
    if (largeButton.isSelected()) return LARGE_SIZE;
    return MEDIUM_SIZE;
}
```

进行这种改变后可以画出如图 10-22 所示的双子座图片。



图 10-22 双子座图片

两颗较大的星星中，右边是北河二，左边是北河三，它们以古典神话中著名的双胞胎命名。底部附近的中等星是 Almeisan，阿拉伯语的意思就是“闪亮的星星”。

10.6.5 JSlider 类和 JLabel 类

有时会觉得，星星大小只有 3 种选项，这是无法提供画星座图所需的尽可能详细的精度。最好能在控件栏添加一个交互器，让它能够画 1 个像素那么小或者 50 个像素那么大的星星。为此，需要不同类型的交互器，它允许产生连续的变化。在 Swing 库中，这种交互器最常见的选择就是 JSlider 类，它用可以从一种设置拖动到另一种设置的控制旋钮实现线性变化。

虽然 JSlider 构造函数有许多不同的形式，但最简单的形式采用 3 个整数参数：最小值、最大值和初始值。因此，要创建 JSlider，让它在 1 和 50 之间变化，初始值为 16，就可以调用 new JSlider(1, 50, 16)，并将结果存储在声明为 JSlide 的实例变量里。当然，实际应用中最好将值 1, 50 和 16 声明为命名常量，以便它们可以随着程序的改变而改变。如果将 sizeSlider 添加到控件栏，产生的用户界面如图 10-23 所示。

滑块虽然很有吸引力，但在说明其目的方面，不能给用户提供太多指导。将一些解释性标签添加到控件栏会有所帮助，如图 10-24 所示。



图 10-23 画星座图(4)



图 10-24 画星座图(5)

要实现这一目标，需要使用 JLabel 类。

虽然 JLabel 类包含在交互器层次结构里，但它是一种有趣的交互器，因为它完全不能移动，对任何事件都没有反应。它的目的只是将标签添加到其他需要它们的交互器，如滑块。

在控件栏中创建这些交互器的代码如下：

```
add(new JButton("Clear"), SOUTH);
sizeSlider = new JSlider(MIN_SIZE, MAX_SIZE, INITIAL_SIZE);
add(new JLabel(" Small"), SOUTH);
add(sizeSlider, SOUTH);
add(new JLabel("Large"), SOUTH);
```

第一个标签前面的空格增加了按钮与标签的距离，因为这样做有助于用户知道标签与滑块匹配，而不是与按钮匹配。

如前面示例所示，实现该扩充的最后一步是改变创建星星的代码，以确定当前大小。使用滑块，这种改变就很容易，因为存储在滑块里的值就是要用作尺寸的值。这样，唯一要做的事就是改变辅助方法 getCurrentSize 的代码，代码如下：

```
public double getCurrentSize() {
    return sizeSlider.getValue();}
```

10.6.6 JComboBox 类

DrawStarMap 程序中另一个有意义的交互器是 JComboBox 类，它适用于提供的选项太多，以至于不能使用单选按钮的语境，这种情况下，即便使用滑块值，范围也不连续。JComboBox 类是一个包含可用选项菜单的小交互器。在 JComboBox 上按下鼠标键会弹出菜单，允许用户选择一个可用选项。因为允许在一组选项中选择，所以 JComboBox 交互器通常称为选择器。

使用 JComboBox 最简单的编程模式包含下面几步：

- (1) 创建空的 JComboBox，并将它存储在实例变量里。
- (2) 调用 add 方法，将每个选项名添加到 JComboBox。
- (3) 调用 JComboBox 上的 setSelectedItem 来设置初始选项。
- (4) 将 JComboBox 添加到用户界面。

DrawStarMap 程序中 JComboBox 类的一个应用程序是使用它选择星星颜色。下面的方法将实例变量 colorChooser 初始化为空的 JComboBox，并给它添加 7 种颜色：

```
private void initColorChooser() {
    colorChooser = new JComboBox();
    colorChooser.addItem("White");
    colorChooser.addItem("Red");
    colorChooser.addItem("Yellow");
    colorChooser.addItem("Orange");
    colorChooser.addItem("Green");
    colorChooser.addItem("Blue");
    colorChooser.addItem("Black");
    colorChooser.setEditable(false);
    colorChooser.setSelectedItem("White");}
```

倒数第 2 行对 `setEditable` 的调用不让用户在颜色字段中输入新值。这条语句很重要，因为程序除了使用列表明确包含的值之外，没打算使用其他值。最后一行将 `JComboBox` 中当前已选项设置为字符串“White”。

虽然白色似乎是星星的自然色，但它在前面的几个程序中使用的白色背景下看不见。要画白色的星星，需要调用

```
setBackground(Color.GRAY);
```

将 `GCanvas` 的背景色改变为表示夜空的颜色。

剩下的唯一一件事就是从存储在 `JComboBox` 中的当前值中，找到设置星星颜色的方法。遗憾的是，`JComboBox` 中没有能够自动将字符串值转换到相应颜色的方法。最直接的方法是从 `JComboBox` 中读入颜色名，然后使用辅助方法将它与颜色列表对比，代码如下：

```
private Color getCurrentColor() {
    String name = (String) colorChooser.getSelectedItem();
    if (name.equals("Red")) return Color.RED;
    if (name.equals("Yellow")) return Color.YELLOW;
    if (name.equals("Orange")) return Color.ORANGE;
    if (name.equals("Green")) return Color.GREEN;
    if (name.equals("Blue")) return Color.BLUE;
    if (name.equals("Black")) return Color.BLACK;
    return Color.WHITE;
}
```

第一行的类型强制转移必不可少，因为 `JComboBox` 类里的值不一定非要是字符串，实际上它可以是任何对象(`JComboBox` 类可以存储任何类的对象，这种能力可能以更面向对象的方法解决此问题，如练习 6 所述)。

`JComboBox` 的值改变时，可以编写程序来检测，不过本示例中不需要。选择 `JComboBox` 中的值时，它会生成动作事件，侦听器能够对此事件作出响应。

完成此扩充，就很容易画猎户星座，其中巨大的红色“参宿四”以红色显示在左上角。右下角的另一颗大星星是“参宿七”，如图 10-25 所示。



图 10-25 猎户星座图片

为了帮助理解本章描述的扩充，图 10-26 给出了 `DrawStarMap` 程序当前版本的代码。

```
import acm.graphics.*;
import acm.program.*;
import java.awt.*;
import java.awt.event.*;

/**
 * This program creates a five-pointed star every time the
 * user clicks the mouse on the canvas. This version includes
 * a JButton to clear the screen, a JSlider to choose the size,
 * and a JComboBox to choose the color.
 */
public class DrawStarMap extends GraphicsProgram {

    /* Initializes the graphical user interface */
    public void init() {
        setBackground(Color.GRAY);
        add(new JButton("Clear"), SOUTH);
        sizeSlider = new JSlider(MIN_SIZE, MAX_SIZE, INITIAL_SIZE);
        add(new JLabel(" Small"), SOUTH);
        add(sizeSlider, SOUTH);
        add(new JLabel(" Large "), SOUTH);
        initColorChooser();
        add(colorChooser, SOUTH);
        addMouseListeners();
        addActionListeners();
    }

    /* Initializes the color chooser */
    private void initColorChooser() {
        colorChooser = new JComboBox();
        colorChooser.addItem("White");
        colorChooser.addItem("Red");
        colorChooser.addItem("Yellow");
        colorChooser.addItem("Orange");
        colorChooser.addItem("Green");
        colorChooser.addItem("Blue");
        colorChooser.addItem("Black");
        colorChooser.setEditable(false);
        colorChooser.setSelectedItem("White");
    }

    /* Returns the current color */
    private Color getCurrentColor() {
        String name = (String) colorChooser.getSelectedItem();
        if (name.equals("Red")) return Color.RED;
        if (name.equals("Yellow")) return Color.YELLOW;
        if (name.equals("Orange")) return Color.ORANGE;
        if (name.equals("Green")) return Color.GREEN;
        if (name.equals("Blue")) return Color.BLUE;
        if (name.equals("Black")) return Color.BLACK;
        return Color.WHITE;
    }
}
```

图 10-26 DrawStarMap 程序当前版本的代码

```

/* Returns the current size */
private double getCurrentSize() {
    return sizeSlider.getValue();
}

/* Called whenever the user clicks the mouse */
public void mouseClicked(MouseEvent e) {
    GStar star = new GStar(getCurrentSize());
    star.setFilled(true);
    star.setColorgetCurrentColor());
    add(star, e.getX(), e.getY());
}

/* Called whenever an action event occurs */
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("Clear")) {
        removeAll();
    }
}

/* Private constants */
private static final int MIN_SIZE = 1;
private static final int MAX_SIZE = 50;
private static final int INITIAL_SIZE = 16;

/* Private instance variables */
private JSlider sizeSlider;
private JComboBox colorChooser;
}

```

图 10-26 (续)

10.6.7 JTextField 类、IntField 类和 DoubleField 类

图 10-16 中唯一没有讨论的类是那些扩充 `JTextComponent` 的类。在图的显示之外, `JTextComponent` 层次结构包含一组强大的扩充类, 如编辑窗口(该窗口让用户可以显示和修改格式化文本)。然而, 这些更复杂的交互器超出了本书的范围。本书只限于讨论 `JTextField` 类及 `acm.gui` 程序包中定义的与其紧密相关的 `IntField` 和 `DoubleField` 子类。这些类分别允许接受单行文本、整数值和浮点值输入。在某种程度上说, 这些类就是从第 2 章开始使用的 `readLine`、`readInt` 和 `readDouble` 方法的交互器版本。例如, 如果要编写可读入字符串的基于 GUI 的程序, 只需创建 `JTextField` 对象, 然后将它添加到控件栏即可。用户可以将文本输入该字段, 向程序提供它所需信息。

虽然可以使用 `DrawStarMap` 程序中的 `JTextField`(见练习 7), 但还是有必要看一个新的示例。因为大多数现代计算机都使用大量字体, 能在屏幕上看看特殊字体效果的应用程序应该非常有用。

图 10-27 中的 `FontSampler` 程序刚好能实现这一点。它首先为字体创建 `JTextField`, 然后将它添加到控件栏, 同时添加说明字段是什么的 `JLabel`。然后它以 Java 运行系统用于 `GLabel` 对象的默认字体, 显示使用字母字符的字符串。结果如图 10-28 所示。

```

import acm.program.*;
import acm.graphics.*;
import java.awt.*;
import java.awt.event.*;

/**
 * This program allows the user to type in a font name and
 * then displays a line of text using that font.
 */
public class FontSampler extends GraphicsProgram {

    public void init() {
        fontField = new JTextField(MAX_FONT_NAME);
        fontField.addActionListener(this);
        add(new JLabel("Font"), SOUTH);
        add(fontField, SOUTH);
        lastY = 0;
        lastLabel = new GLabel(TEST_STRING);
        addGLabel(lastLabel);
    }

    /* Called when any action event is generated */
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == fontField) {
            GLabel label = new GLabel(TEST_STRING);
            label.setFont(lastLabel.getFont());
            label.setFont(fontField.getText());
            addGLabel(label);
            lastLabel = label;
        }
    }

    /* Adds a GLabel on the next line, adjusting for different sizes */
    private void addGLabel(GLabel label) {
        lastY += label.getHeight();
        lastY += lastLabel.getDescent() - label.getDescent();
        add(label, LEFT_MARGIN, lastY);
    }

    /* Private constants */
    private static final int MAX_FONT_NAME = 30;
    private static final int LEFT_MARGIN = 3;
    private static final String TEST_STRING =
        "The quick brown fox jumped over the lazy dog.';

    /* Private instance variables */
    private JTextField fontField;
    private GLabel lastLabel;
    private double lastY;
}

```

图 10-27 以用户选择的字体显示标准字符串的程序

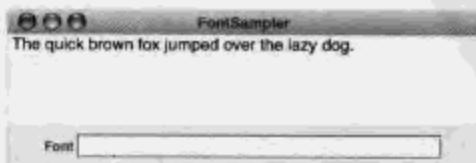


图 10-28 字体示例(1)

JTextField 是控件条中长长的矩形方框，它由语句

```
fontField = new JTextField(MAX_FONT_NAME);
```

创建。

其中 MAX_FONT_NAME 是常量，表示可以在字段中输入的最大字符数。该参数的值确定 JTextField 方框的宽度，通常比输入数据时需要的要宽，原因是实现方式为要求的字符数量保留足够的空间(假设它们都和 m 字符的宽度一样，它的水平扩充在所有字母字符中最大)。因为大多数字符比 m 狹窄，所以在字段末尾通常有大量剩余空间。

运行包含 JTextField 的程序时，可以在字段中输入字符。按 Enter 键来表示输入结束时，JTextField 会向相关监听器发送一个动作事件。因为 addActionListeners 方法只对按钮而言将程序添加为监听器，所以必须进行显式调用，将程序指派为每个 JTextField 的监听器，如下面的调用所示。

```
fontField.addActionListener(this);
```

本示例中，actionPerformed 方法的代码使用了与前面按钮示例中不同的策略来指定事件的来源。除了检查特定动作命令字符串之外，这种实现方式还检查动作事件的来源，以确定它是否是文本字段。虽然可以将这种方法应用于按钮，但通常会扩大程序，因为必须将每个按钮都存储在实例变量里。这里，检查来源比较容易，因为 JTextField 生成的动作事件通常与对按钮有效的动作命令无关。

动作事件发生并确定事件来自于 fontField 的交互器时，程序执行下列语句：

```
GLabel label = new GLabel(TEST_STRING);
label.setFont(lastLabel.getFont());
label.setFont(fontField.getText());
addGLabel(label);
lastLabel = label;
```

其中辅助方法 addGLabel 的代码如下：

```
private void addGLabel(GLabel label) {
    lastY += label.getHeight();
    lastY += lastLabel.getDescent() - label.getDescent();
    add(label, LEFT_MARGIN, lastY);
```

该代码的结果是将新的 GLabel 添加到画布上低于最近标签的一行，使用一些计算来确保这些行是分隔开的。表示标签 y 坐标的 lastY 变量，它的计算基本原理本节稍后讨论。标签以 fontField 的交互器中指定的字体出现。因此，如果输入 Serif-14，然后按 Enter 键，显示会将新的一行添加到画布，这一行设置为 14 磅 Serif 字体，如图 10-29 所示。

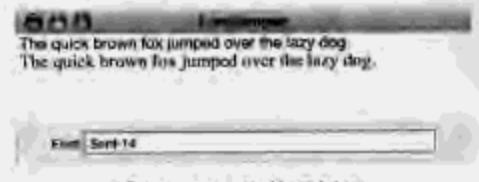


图 10-29 字体示例(2)

虽然显示标签的许多代码是 `acm.graphics` 库提供的工具的简单应用，但一些细节也值得注意。第一，程序需要保持最后一个标签的基线的 `y` 坐标，以便知道如何定位下一个标签。为此程序使用实例变量 `lastLabel` 和 `lastY` 来保存最近添加到画布的 `GLabel` 及其基线位置。

首先，似乎不需要记住用来正确定位下一个标签的最后一个标签。毕竟，在只使用一种字体的程序中，通过添加字体高度，就可以从一行移动到另一行。然而，在这个程序中，每一行上的字体都不同。连续行之间的间隔不但取决于当前行使用的字体，而且取决于字符在前一行基线之下下降的距离，程序需要考虑到这一点。图 10-27 中的代码通过调整新的基线，包含从 `lastLabel` 而不是从新标签的字体下降距离，以实现基线计算。

值得解释的程序的另一项内容是设置新标签字体的代码，它由对 `setFont` 的两个连续调用组成。第一个对 `setFont` 的调用看起来好像无关，如果第二个调用再次改变字体。这种实现策略的原因是第二个对 `setFont` 的调用——使用字符串参数——不需要指定新字体的各种要素。例如，如果字符串省略了磅值大小或风格，`GLabel` 就使用其当前字体的值。因此，如果用户输入一种新的字体族名，而没有提供磅值信息，图 10-27 中 `FontSampler` 程序的实现方式只会改变字体族，而不会改变磅值。在本示例中，可以将字体设置为 `Lucida Blackletter`，但不改变磅值，如图 10-30 所示。

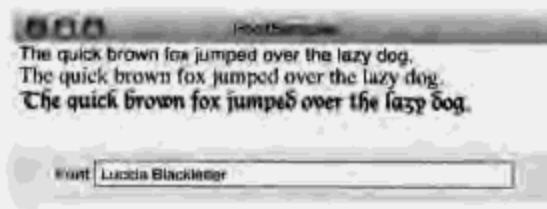


图 10-30 字体示例(3)

为了简化从文本字段读入数字数据的过程，`acm.gui` 程序包包含了两个新的类——`IntField` 和 `DoubleField`，它们允许用户输入相应类型的数字值。这些类扩充了 `JTextField`，而且还提供了其他一些方法，隐藏了涉及数字转换和检查输入错误的复杂性。`DoubleField` 可用的最有用方法如图 10-31 所示：除了参数和结果类型要进行适当改变之外，`IntField` 方法与此类似。本章稍后将介绍这些类的示例。

构造函数
DoubleField() 创建没有初始值的 <code>DoubleField</code> 对象
DoubleField(double value) 创建有指定初始值的 <code>DoubleField</code> 对象
设置和取得字段值的方法
void setValue(double value) 设置字段的值，并更新显示
double getValue() 返回字段中的值。如果值不在范围之内，就会出现错误或重试
控制格式的方法
void setFormat(String format) 用 <code>DecimalFormat</code> (详情请参考 javadoc)风格，设置字段内字符串的格式
String getFormat() 返回当前格式的字符串

图 10-31 DoubleField 类中定义的方法

10.7 管理组件布局

虽然前面交互式示例中使用的控件栏模型在许多应用程序中都会取得很好的效果，但大多数 GUI 应用程序不将其交互器放置在窗口边框。人们真正想要的是可以将交互器放置在应用程序窗口的任何位置，以创建适用于特定应用程序的综合图形用户界面。

10.7.1 Java 窗口层次结构

在创建允许交互器出现在应用程序窗口内任何位置的程序之前，先需要了解一下 Java 如何在计算机屏幕上显示窗口，以及如何安排其内部内容。学习 Java 程序包的结构时，最初几步之一就是理解相关类的层次结构。为了理解 Java 的窗口系统，需要理解的最重要的类如图 10-32 所示。

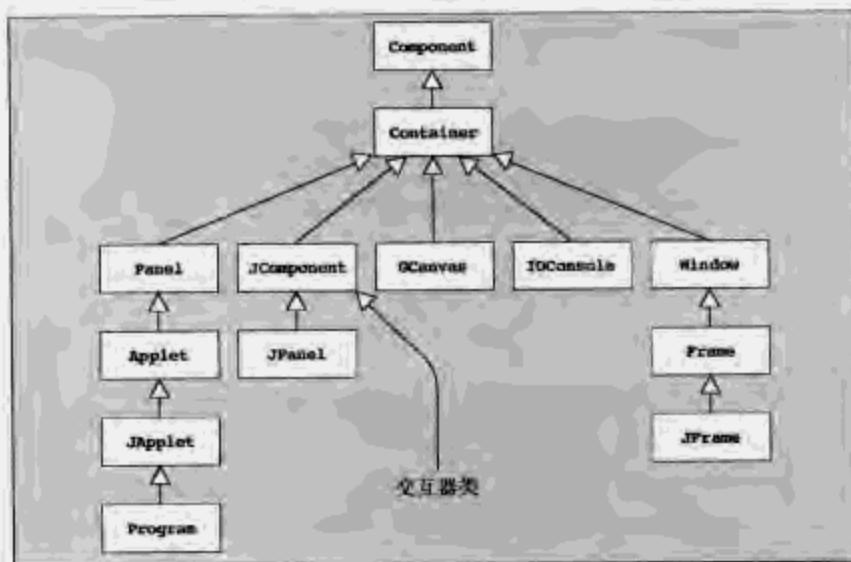


图 10-32 窗口系统中部分类的层次结构

图 10-32 中许多结构虽然都是新的，但有些类以前还是见过。左边一列底部的 `Program` 类是所有程序的超类，它有两个直系父类：`JApplet` 和 `Applet`。然而，层次结构继续向上，就是一连串名为 `Panel`、`Container` 和 `Component` 的类。下一列中的 `Jcomponent` 类是所有像 `JButton` 这样的 Swing 交互器的基础，也 `JPanel` 类的基础，它在创建交互器层次结构时非常有用。和 `Panel` 及 `JComponent` 一样，`GCanvas` 和 `IOConsole` 类也扩充 `Container`，因此它们都是 `Container` 类和 `Component` 类。虽然这些类超出了本书讨论的范围，但由 `Window`、`Frame` 和 `JFrame`(用来创建大多数卓越的 Java 应用程序)组成的类层次结构部分也是以 `Component` 和 `Container` 类开始的类层次结构的一部分。

Component 类和 Container 类对于全面理解 Java 窗口系统至关重要，尽管很少有机会直接使用它们。Component 类表示可以出现在窗口中的所有 Java 类。如图 10-32 中的层次结构所示，每个 Applet 类都是 Component 类，图中其他类也一样。Container 类是 Component 类的特殊类型，它可以包含其他 Component 类。例如，Program 类是 Container 类的子类，这说明可以将

组件添加到程序。Program 类的标准子类也可以这样做。运行 GraphicsProgram 类时，它创建 GCanvas，然后将 GCanvas 添加到程序窗口，以便让 GCanvas 填满可用空间。运行 ConsoleProgram 时，过程与此类似，唯一不同的是 ConsoleProgram 用 IOConsole 而不是 GCanvas 填充程序窗口。

Component 和 Container 类之间的关系与 acm.graphics 程序包中 GObject 和 GCompound 类之间的关系类似，如果容器内组件的思想看起来容易产生混淆，那么弄清楚这一点会有所帮助。所有可以出现在 GCanvas 的类都是 GObject 的子类；GCompound 是 GObject 的特殊子类，它可以包含其他 GObject。同样，可以出现在窗口系统里的类都是 Component；Container 是 Component 的特殊子类，它可以包含其他 Component。

然而，当讨论组件在容器内如何排列的问题时，就不能再用 GObject 和 GCompound 的相似性。将 GObject 添加到 GCompound，需要明确指定 GObject 的坐标。而将 Component 添加到 Container 时，通常不指定位置而是指定容器用来排列其组件的策略，参见 10.7.2 小节。

10.7.2 布局管理器

当 Java 设计师创建第一版 Java 窗口工具箱时，他们认识到，作为现代 GUI 设计基础的图形窗口必须能够改变大小。在典型的窗口环境中，拖动窗口边界，或者单击让窗口填满屏幕的按钮，都可以改变窗口大小。如果窗口大小可以改变，那么编程人员在容器内指定组件的位置就没有意义，因为每个组件的最佳位置及其大小取决于显示它的窗口的大小。

为了让应用程序更容易响应大小的改变，Java 设计师采取使用布局管理器的策略。布局管理器是负责在容器内排列组件的类。排列取决于下面几个因素：

- 布局管理器所使用的策略。每个布局管理器采取一种特殊的策略，用来安排容器内的组件。不同的布局管理器使用不同的策略；作为 GUI 设计师的任务就是选择布局管理器，让布局管理器能够以适合应用程序的方式排列其组件。
- 容器内可用空间的大小。布局管理器的主要任务是有效利用容器内的空间。随着容器大小的改变，布局通常也会改变。
- 每个组件的预定义大小。每个组件都有预定义大小，它是假设没有大小约束时组件理想的空间大小。例如， JButton 定义其预定义大小，让它宽得足以显示其标签的整个文本，同时两边还有一些剩余空间。一般来说，当决定如何排列组件时，布局管理器会考虑这些预定义大小。
- 将组件添加到其容器时，为组件指定的所有约束。将组件添加到其容器的 add 方法采用可选的第二参数，用来将有关组件可选位置的信息转移到布局管理器。本章稍后有关 BorderLayout 和 TableLayout 管理器这两节将会介绍约束的示例。

接下来几节将简要介绍 java.awt 程序包中最常见的布局管理器及其用法的简单示例。10.7.3 小节中 BorderLayout 管理器的示例说明如何将布局管理器指派给单独创建的 JPanel 类。后面几节中的示例将把新的布局管理器指派给程序本身。

10.7.3 BorderLayout 布局管理器

虽然隐藏了细节，但我们已经见过运行的标准布局管理器之一。指派到所有 Program 子类的窗口都使用 BorderLayout 沿窗口边框创建控件栏。容器使用 BorderLayout 时，被划分为 5 个区域，如图 10-33 所示。

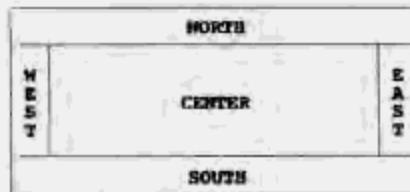


图 10-33 边框示意图

将组件添加到 BorderLayout 管理的容器时，必须提供该区域的名称作为 add 方法的第二参数，以此指定希望将组件添加到哪个边框。区域名称——NORTH、SOUTH、EAST、WEST 和 CENTER——在 BorderLayout 类中都定义为常量。为了方便起见，也在 Program 类里定义这些常量，这样就可以在程序语境中使用它们，而不必提供 BorderLayout 作为限定符。

BorderLayout 管理器安排容器内的组件时，按下面的步骤依次进行：

(1) 沿 NORTH 和 SOUTH 边框改变组件大小，让它们水平扩充到整个容器宽度，给每个边框提供其预定义大小表示的垂直空间。

(2) 沿 EAST 和 WEST 边框改变组件大小，让它们垂直扩充到考虑了已经指派的边框之后容器剩余的高度。每个组件的预定义大小确定这些区域的宽度。

(3) 改变 CENTER 组件的大小，让它占满指派边框区域之后剩下的全部空间。

如果遗漏了某些组件，就根本没有空间指派给这些区域。

下面的程序说明了 BorderLayout 管理器和 JPanel 类的用法。JPanel 类是容器类，通常用于在布局管理器的控制下集合组件。

```
public class BorderLayoutExample extends Program {
    public void init() {
        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.add(new JButton("NORTH"), BorderLayout.NORTH);
        panel.add(new JButton("SOUTH"), BorderLayout.SOUTH);
        panel.add(new JButton("WEST"), BorderLayout.WEST);
        panel.add(new JButton("EAST"), BorderLayout.EAST);
        panel.add(new JButton("CENTER"), BorderLayout.CENTER);
        add(panel);
    }
}
```

首先要注意的就是程序直接扩展了 Program 类。到目前为止，在本书中所见到的示例都扩充了 Program 子类之一，如 GraphicsProgram 或 ConsoleProgram。如 10.7.2 小节所述，这些子类在程序窗口自动安装组件。如果要建立自己的 GUI 应用程序，就要自己汇编程序窗口的内容。

init 方法的第一行创建新的 JPanel 类。和第 9 章 GCompound 类的情况一样，新的 JPanel 最初是空的。要将想要的内容放在一起，可以使用 add 方法每次一个组件的方式将组件集合到面板。这样做之前，需要给面板指派一个布局管理器，让它控制排列组件的方法。本示例中，init 方法的第二行将布局管理器设置为 BorderLayout 对象。接下来几行将 5 个 JButton 类添加到容器，因此每个按钮上的标签对应于指派给它的边框区域。最后，init 方法中的最后一条语句将整个面板添加到程序窗口，它默认占满整个空间。

运行 BorderLayoutExample 程序时, JPanel 的布局管理器将根据 BorderLayout 策略排列其组件, 产生的程序窗口如图 10-34 所示。



图 10-34 BorderLayout 示例的输出

NORTH 和 SOUTH 按钮水平延伸到整个面板, WEST 和 EAST 按钮垂直延伸, 占满了剩余的空间。CENTER 按钮在两个方向上都延伸。

此时, 可能想知道本章前面添加到控件条的交互器为什么没有像图 10-34 中的按钮一样延伸到整个空间。答案是即使 Program 类使用 BorderLayout 管理器来排列边框区域, 实际上仍然有一点剩余。在 Program 中, 每个边框区域都是一个 JPanel 类, 这说明它可以保存多个组件。JPanel 延伸到整个空间, 但 JPanel 内单个交互器的排列由其布局管理器控制, 而不是由程序的布局管理器控制。Program 类重写 add 方法, 让它将组件添加到前面某个边框区域, 这些区域要求有相应的 JPanel 对象。

要有效使用布局管理器, 需要知道布局过程何时发生。在 Java 中, 调用容器中的 validate 方法, 可以调用容器的布局管理器。对 validate 的调用在程序的 init 方法之后或改变程序窗口大小时自动发生, 因此不需要显式调用该方法, 除非在程序的其他地方改变容器的内容。

10.7.4 FlowLayout 布局管理器

另一种常见的布局管理器是 FlowLayout, 它可能最容易使用。FlowLayout 布局管理器排列容器内的组件时, 它从窗口的最上面开始, 使用每个组件指定的预定义大小, 水平排列组件。当下一个组件不适合当前行时, FlowLayout 布局管理器会移动到下一行, 继续放置在那一行放置其他组件, 直到空间再次用尽为止。FlowLayout 布局管理器默认居中窗口内的每行组件。

要理解 FlowLayout 的运行原理, 有必要考察一个简单示例。下面的程序使用其 init 方法安装 FlowLayout 布局管理器, 然后将 6 个 JButton 对象添加到程序窗口:

```
public class FlowLayoutExample extends Program {
    public void init() {
        setLayout(new FlowLayout());
        add(new JButton("Button 1"));
        add(new JButton("Button 2"));
        add(new JButton("Button 3"));
        add(new JButton("Button 4"));
        add(new JButton("Button 5"));
        add(new JButton("Button 6"));
    }
}
```

这个程序采取的方法与 BorderLayout 示例不同。它没有创建新的 JPanel 类, 本示例中的 init 方法只是将新的布局管理器指派给程序本身。

运行 FlowLayoutExample 的结果取决于窗口大小。图 10-35 所示为一种可能的运行结果。

给定窗口大小，FlowLayout 布局管理器将前 4 个按钮排列在布局的最上面一行。此时，该行再没有空间放置第 5 个按钮，因此布局管理器移动到下面一行放置后两个按钮。在每一行都居中，产生的布局如样本运行所示为一种可能的运行结果。

如果改变窗口大小，布局管理器会自动重新放置按钮。例如，如果让窗口变窄一点，则每一行只能放置 3 个按钮，如图 10-36 所示。



图 10-35 FlowLayout 示例的输出(1)



图 10-36 FlowLayout 示例的输出(2)

相反，如果让窗口变宽一点，最终可以将所有按钮放置在最上面一行。

FlowLayout 布局管理器的主要优点是它产生的布局易于阅读，同时也易于使用。不足之处是几乎不能控制布局。特别是 FlowLayout 布局管理器不能指定行边界出现的位置，而这在有些应用程序中至关重要。

作为 FlowLayout 管理器发生错误的示例，可以假设将滑块作为用户界面的交互器之一，和本章前面的 DrawStarMap 程序中一样。通常情况是，程序中的滑块与两个 GLabel 类有关，两边各有一个，它们让用户知道滑块在控制什么。在 DrawStarMap 程序中，标签和滑块如下所示。



如果使用 FlowLayout 布局管理器来排列这些交互器，对于有些窗口来说，其布局中行之间的边界居于一个标签和滑块之间，破坏了视觉效果。

10.7.5 GridLayout 布局管理器

哪些交互器被指派给特殊行？控制这一点的方法之一是使用 GridLayout 布局管理器，它在二维栅格里排列组件。调用 GridLayout 构造函数时，指定行列数。例如，构造函数

```
new GridLayout(2, 3)
```

创建的栅格有 2 行 3 列。当 GridLayout 布局管理器排列容器内的组件时，它先填满一行，然后移动到下一行。延伸每一个组件来填满栅格内的空间，栅格本身会延伸填满容器内的全部可用空间。因此，在 2 行 3 列的栅格里，每个组件都占据可用水平空间的 1/3，垂直空间的 1/2。

作为 GridLayout 运行方法的示例，init 方法

```
public void init() {
    setLayout(new GridLayout(2, 3));
    add(new JButton("Button 1"));
    add(new JButton("Button 2"));
    add(new JButton("Button 3"));
    add(new JButton("Button 4"));
    add(new JButton("Button 5"));
}
```

```

    add(new JButton("Button 4"));
}

```

产生的布局如图 10-37 所示。

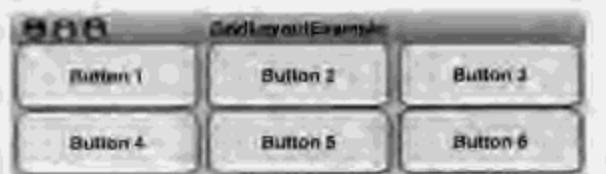


图 10-37 GridLayout 的示例输出

如果改变窗口大小，按钮会在一个方向上延伸，但栅格的总体结构会保持完整。

10.7.6 标准布局管理器的不足

虽然前面几节介绍的标准布局管理器在许多情况下都很有用，但要设计的程序超出了最简单 GUI 布局时，则无法提供所需的灵活性。要创建商业应用程序中使用的那种复杂布局，就要使用更复杂的布局管理器。遗憾的是，Java 提供的大多数通用布局管理器(Java.awt 程序包中的 `GridBagLayout` 类)都过于复杂，不适合于初学编程的人员学习。而且，它设计得不好，有经验的 Java 编程人员都会避免使用它。

`GridBagLayout` 不适合于介绍性文本，这一点让人进退两难。容易学习的布局管理器不够强大，不足以创建有趣的布局。相反，有强大功能的布局管理器又非常复杂，不便于介绍。要解决这个问题，ACM Java 工作组(ACM Java Task Force)开发了一种新的布局管理器，称为 `TableLayout`，它适合在灵活的二维栅格里排列组件。`TableLayout` 管理器具有 `GridBagLayout` 的全部功能，同时降低了其复杂性。然而，`TableLayout` 类的功能非常多，需要单独用一节介绍。

10.8 使用 TableLayout 类

正如 10.7 节结尾所述，使用现有布局管理器面临的困难让 ACM Java 工作组设计了一种新的布局管理器，称为 `TableLayout`，它可以很方便地设计复杂布局。在某些方面，`TableLayout` 类与 `GridLayout` 类相同，特别是在二维栅格中排列交互器时。与 `GridLayout` 的不同之处主要有以下几个方面。

- `TableLayout` 中的单元可以改变大小。行和列默认具有足够空间，可以保存行和列中最大的项。
- `TableLayout` 管理器根据组件的预定义大小计算栅格的大小，通常不会扩充栅格来填满可用空间。
- `TableLayout` 管理器可以将约束与单个组件联系起来，以指导布局过程。约束的使用将在 10.8.3 小结介绍。

接下来的几节将介绍 `TableLayout` 管理器的运行原理以及在程序中有效使用它的方法。

10.8.1 比较 GridLayout 与 TableLayout

要了解 GridLayout 和 TableLayout 管理器之间的不同，最简单的方法就是在只改变布局管理器的条件下，运行相同的程序。用来说明 GridLayout 的程序示例建立了一个 2 行 3 列的栅格布局，然后将 6 个 JButton 类添加到程序窗口。将程序调整为使用 TableLayout，只需要将 setLayout 调用从

```
setLayout(new GridLayout(2, 3));
```

改变为

```
setLayout(new TableLayout(2, 3));
```

即可。运行修改后的程序，产生的按钮排列如图 10-38 所示。



图 10-38 TableLayout 示例的输出

如果将此运行结果与前面 GridLayoutExample 产生的运行结果进行对比就会发现，按钮恢复到了它们的预定义大小，不再延伸填满程序窗口。

10.8.2 使用 TableLayout 创建温度转换器

要更好理解 TableLayout 提供的灵活性，有必要考察一个更复杂的示例，如图 10-39 所示。

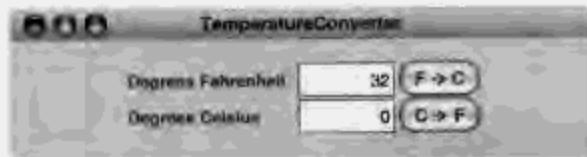


图 10-39 温度转换器(1)

用户在两个数字字段中输入值，然后单击相应的按钮，就可以向两个方向中任意一个方向转换温度。例如，要找出 20° 对应的华氏温度，用户只需要在 Degrees Celsius 方框中输入值 20，然后单击标记为 C > F 的按钮，就会产生下面的显示，如图 10-40 所示。

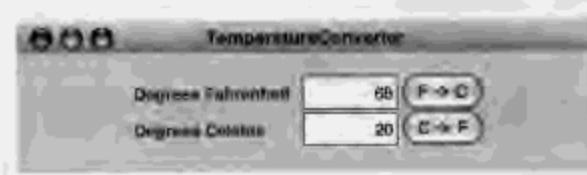


图 10-40 温度转换器(2)

温度转换器 TemperatureConverter 程序代码如图 10-41 所示。

```
import acm.graphics.*;
import acm.gui.*;
import acm.program.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This program allows users to convert temperatures back and forth
 * from Fahrenheit to Celsius.
 */
public class TemperatureConverter extends Program {

    /* Initializes the graphical user interface */
    public void init() {
        setLayout(new TableLayout(2, 3));
        fahrenheitField = new IntField(32);
        fahrenheitField.setActionCommand("F -> C");
        fahrenheitField.addActionListener(this);
        celsiusField = new IntField(0);
        celsiusField.setActionCommand("C -> F");
        celsiusField.addActionListener(this);
        add(new JLabel("Degrees Fahrenheit"));
        add(fahrenheitField);
        add(new JButton("F -> C"));
        add(new JLabel("Degrees Celsius"));
        add(celsiusField);
        add(new JButton("C -> F"));
        addActionListeners();
    }

    /* Listens for a button action */
    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if (cmd.equals("F -> C")) {
            int f = fahrenheitField.getValue();
            int c = GMath.round((5.0 / 9.0) * (f - 32));
            celsiusField.setValue(c);
        } else if (cmd.equals("C -> F")) {
            int c = celsiusField.getValue();
            int f = GMath.round((9.0 / 5.0) * c + 32);
            fahrenheitField.setValue(f);
        }
    }

    /* Private instance variables */
    private IntField fahrenheitField;
    private IntField celsiusField;
}
```

图 10-41 基于 GUI 的温度转换器程序

和前面示例中一样，init 方法负责建立图形用户界面。这里，init 方法首先调用

```
setLayout(new TableLayout(2, 3));
```

创建一张表，两行水平运行，3 列垂直运行。布局管理器准备就绪后，init 方法的其余部分创建交互器，并将它们添加到表中，从左至右填充每一行，然后从上到下填充每一列。在

TemperatureConverter 示例中，对 add 的调用使用行

```
add(new JLabel("Degrees Fahrenheit"));
add(fahrenheitField);
add(new JButton("F -> C"));
```

创建表的华氏温度行，然后使用行

```
add(new JLabel("Degrees Celsius"));
add(celsiusField);
add(new JButton("C -> F"));
```

创建相应的摄氏温度行。

如果看看该代码产生的运行结果就会发现，表中各交互器的大小都根据栅格制定的约束进行了调整。JLabel 对象大小不同，但 TableLayout 管理器确保第一列中有足够的空间保存两个标签中较长的一个。每个组件默认延伸填满其栅格单元，尽管单元没有延伸填满容器。

创建 TableLayout 对象时，可以使用 0 代替行数或列数来表示未指定的值，但不能同时代替两者。例如，调用

```
setLayout(new TableLayout(0, 7));
```

指定的表有 7 列，行数是表中表示组件所需的行的数量。可以使用这样的布局来创建日历，它总是有 7 列表示天数，但行数依据月份不同而有所改变。

TemperatureConverter 的代码调用 addActionListeners 方法，分别将程序指定为其内部所有按钮的动作监听器。分别调用 addActionListener 和 setActionCommand，让用户能够通过单击相应的按钮或在交互器内按 Enter 键触发转换。每个动作都生成 ActionEvent 事件，它的动作命令是字符串 “F -> C” 或者 “C -> F”，这取决于哪个按钮或交互器生成了事件。actionPerformed 方法实现所需的转换，然后更新相应字段的值，以响应这些动作事件。

10.8.3 指定约束

虽然 TableLayout 管理器的默认行为本身很有用，但它最重要的特征是，添加新的组件时，可以指定其他约束。例如，这种约束可以设置包含表单元格的行和列的最小宽度和高度，指定组件随着单元格大小改变而改变的方式，或指示特定的单元格横跨几行或几列。

将组件添加到 TableLayout 栅格时，可以用字符串的形式指定这些约束，该字符串作为第二参数传递给 add 方法。这种字符串的格式是一条或多条下面这种形式的规范：

```
constraint=value
```

其中，*constraint* 是某个可用约束选项的名称，*value* 是该选项对应的值。例如，如果要将当前表中行的最小高度指定为 50 像素，可以指定

```
"height=50"
```

来完成。同样，如果要让组件占据 3 列栅格，可以指定以下约束字符串：

```
"gridwidth=3"
```

在相同字符串里可以包含多个 *constraint*=*value* 条目，中间至少用一个空格隔开，这样可

以在对 add 的同一个调用里包含多个约束。图 10-42 描述了一些约束，将约束添加到使用 TableLayout 策略管理的容器时，可以指定它们。本书只使用前两个选项(gridwidth/gridheight 和 width/height)，其他约束可以自己研究。

gridwidth=columns or gridheight=rows 表示该表应该横跨指定的行数或列数
width=pixels or height=pixels width 规范表示这一列的宽度应该是指定的像素。如果给同一列里的单元格指定了不同的宽度，列宽就定义为其中的最大值。如果没有 width 规范，那么列宽就是首选宽度中最大的一个。height 规范与行高的解释相同
weightx=weight or weighty=weight 如果表的总体大小比内含的要小，TableLayout 通常会将表在可用空间内居中。然而，如果有单元格是非零 weightx 或 weighty 值，那么就按照指定的高度比例沿对应的轴分布多余空间。高度解释为浮点值
fill=fill 表示如果预定义大小比单元格小，应该如何调整单元格中组件的大小。合法的值是 NONE、HORIZONTAL、VERTICAL 和 BOTH，表示应该沿哪个轴进行延伸。默认为 BOTH
anchor=anchor 如果组件没有填满特定的轴，anchor 规范表示应该将组件放置在单元格里的什么位置。默认值是 CENTER，但可以使用所有标准方向(NORTH、SOUTH、EAST、WEST、NORTHEAST、NORTHWEST、SOUTHEAST 或 SOUTHWEST)

图 10-42 TableLayout 类中可用的约束选项

10.8.4 使用 TableLayout 创建简单的计算器

TableLayout 类广泛用于 GUI 设计。为了理解它在较大示例的语境中如何运行，假设要创建一个应用程序，用来模仿使用整数值的传统四功能计算器。计算器的布局和运行如图 10-43 所示，它显示了使用计算器相加整数 25 和 17 的步骤。

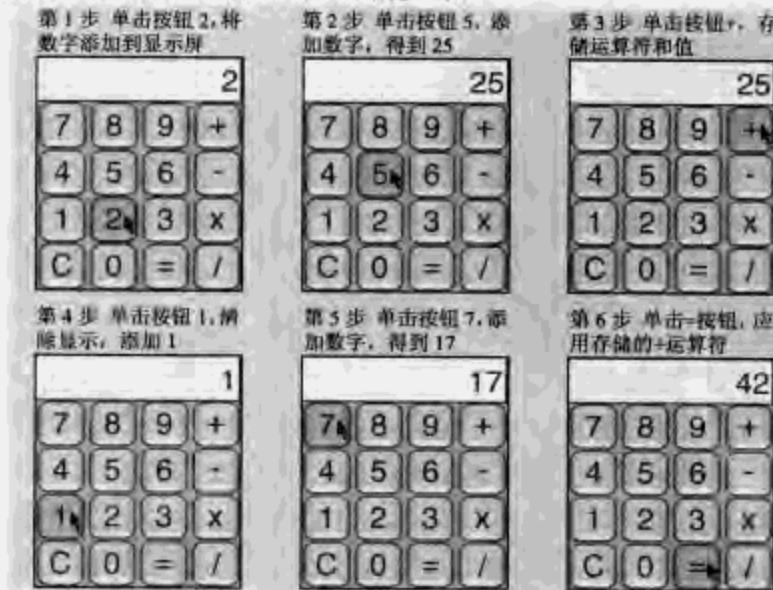


图 10-43 计算 25+7 的步骤

该程序非常复杂，提出了好几个设计问题。与本章关系最密切的问题是如何排列按钮和交互器。按钮本身形成 4×4 的栅格，其中每个按钮都是正方形。为了方便改变计算器的大小，有必要将正方形大小定义为常量，例如可以调用 `BUTTON_SIZE`。

计算器顶部的数字显示也是栅格的一部分。其垂直尺寸与按钮相同，但在水平轴上它横跨了全部 4 列。因此，该应用程序的 `init` 方法如下。

其中，`CalculatorDisplay` 是仍未定义的类，它实现数字显示。

```
public void init() {
    setLayout(new TableLayout(5, 4));
    display = new CalculatorDisplay();
    add(display, "gridwidth=4 height=" + BUTTON_SIZE);
    Add the 16 buttons that occupy the next four rows.
    Enable the action listeners for those buttons.
}
```

将显示添加到程序窗口的方法使用表达式。

```
"gridwidth=4 height=" + BUTTON_SIZE
```

来指定 `TableLayout` 约束。此表达式使用连接运算符`+`，将常量 `BUTTON_SIZE` 的值插入约束字符串。如果 `BUTTON_SIZE` 的值是 40，约束规范就等于

```
"gridwidth=4 height=40"
```

它表示计算器显示应该横跨 4 列，高度为 40 像素，如图 10-43 所示。

另一个重要设计是如何将按钮表示为 Java 对象。强制方法可以让 16 个按钮成为 `JButton` 类本身的实例。如果这样做，可以实现 `actionPerformed` 侦听器方法，以便检查每个可能的动作命令，如果需要还可以作出响应。然而，该方法没有考虑到按钮属于不同逻辑组这一点。有 10 个数字按钮(0~9),4 个运算符按钮(+、-、× 和 ÷)，两个不属于这两个种类的按钮(C 和 =)。这种结构关系说明按钮非常符合图 10-44 所示的类层次结构。

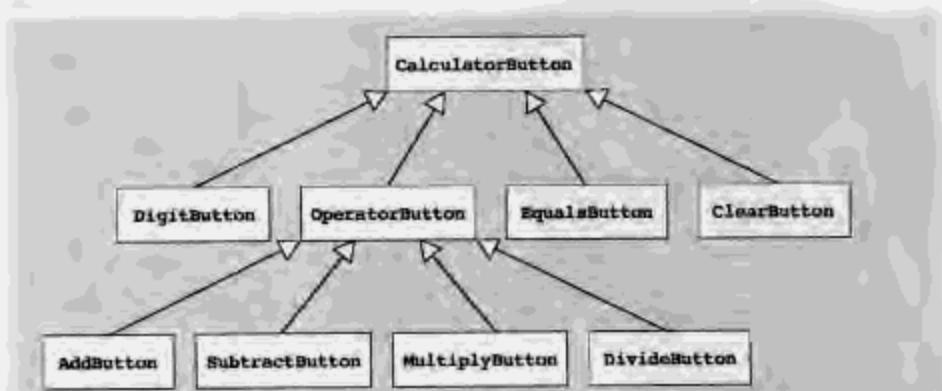


图 10-44 实现计算器按钮的程序包私有类

使用类层次结构可以写出 `actionPerformed` 方法更简单的实现方式。如果每个按钮都是单独类的实例，那么就不再需要写 `if` 语句来检查各种动作命令，而可以采取一种更加面向对象的方法。第一步是使用 `getSource` 确定动作事件的来源，很可能是按钮导致了该事件。如果来源是 `CalculatorButton` 层次结构中某个按钮的实例，可以让该按钮指定它自己的响应。`actionPerformed`

方法要做的就是调用导致事件的按钮上的特定方法——可以称为 action。层次结构中的每个类都可以定义自己的 action 来执行所需的操作。例如，DigitButton 类定义 action，将数字添加到显示中的值，AddButton 类定义 action，执行加法等。

Calculator 程序的完整实现方式如图 10-45 所示，它占满了下面 4 个页面。

```
import acm.gui.*;
import acm.program.*;
import acm.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/** This program implements a simple four-function calculator */
public class Calculator extends Program {

    /* Initializes the user interface */
    public void init() {
        setLayout(new TableLayout(5, 4));
        display = new CalculatorDisplay();
        add(display, "gridwidth=4 height=" + BUTTON_SIZE);
        addButtons();
        addActionListeners();
    }

    /* Called on each action event; the response is determined by the button */
    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source instanceof CalculatorButton) {
            ((CalculatorButton) source).action(display);
        }
    }

    /* Adds the buttons to the calculator */
    private void addButtons() {
        String constraint = "width=" + BUTTON_SIZE + " height=" + BUTTON_SIZE;
        add(new DigitButton(7), constraint);
        add(new DigitButton(8), constraint);
        add(new DigitButton(9), constraint);
        add(new AddButton(), constraint);
        add(new DigitButton(4), constraint);
        add(new DigitButton(5), constraint);
        add(new DigitButton(6), constraint);
        add(new SubtractButton(), constraint);
        add(new DigitButton(1), constraint);
        add(new DigitButton(2), constraint);
        add(new DigitButton(3), constraint);
        add(new MultiplyButton(), constraint);
        add(new ClearButton(), constraint);
        add(new DigitButton(0), constraint);
        add(new EqualsButton(), constraint);
        add(new DivideButton(), constraint);
    }

    /* Private constants and instance variables */
    private static final int BUTTON_SIZE = 40;
    private CalculatorDisplay display;
}
```

图 10-45 计算器程序的实现

```

/*
 * This class defines the display for the calculator.
 *
 * Implementation notes:
 * This class does most of the work for the calculator application and keeps
 * track not only of the number currently in the display but also the previous
 * operator button (op) and the previous value from the display (memory), which
 * will become the left operand of the operator. When a new operator button is
 * pressed, this class calculates the new value of the display by applying
 * that operator to the value in memory and the current value in the display.
 *
 * It is also important to note that the behavior of digit buttons depends on
 * whether an operator button is clicked. If the last click was an operator
 * button, the digit buttons must clear the display to start a new value. If
 * not, the digit is added to the end of the existing value. The code uses the
 * boolean variable startnewValue to record this state.
 */
class CalculatorDisplay extends IntField {

    /* Creates a new calculator display that is not directly editable by the user */
    public CalculatorDisplay() {
        setEditable(false);
       setFont(new Font("SansSerif", Font.PLAIN, 24));
        setValue(0);
        startnewValue = false;
        op = null;
    }

    /* Adds a digit to the display, clearing the old value if startnewValue is set */
    public void addDigit(int digit) {
        int value = (startnewValue) ? 0 : getValue();
        setValue(10 * value + digit);
        startnewValue = false;
    }

    /* Sets a new operator, applying the previous one if one exists */
    public void setOperator(OperatorButton button) {
        if (op == null) {
            memory = getValue();
        } else {
            memory = op.apply(memory, getValue());
            setValue(memory);
        }
        op = button;
        startnewValue = true;
    }

    /* Private instance variables */
    private OperatorButton op;          /* The last operator button pressed */
    private int memory;                /* The value to which the operator is applied */
    private boolean startnewValue;     /* Set after an operator to start a new value */
}

```

图 10-45 (续)

```
/*
 * This abstract class is the superclass for every calculator button. Every button
 * must define an action method, which is called whenever the button is clicked.
 */
abstract class CalculatorButton extends JButton {

    /* Creates a new CalculatorButton with the specified name */
    public CalculatorButton(String name) {
        super(name);
        setFont(new Font("SansSerif", Font.PLAIN, 24));
    }

    /* Called when the button is clicked (every subclass must implement this method) */
    public abstract void action(CalculatorDisplay display);
}

/*
 * This class is used for each of the digit buttons. The action consists of
 * adding the digit used as a label on the button, which is returned by getText().
 */
class DigitButton extends CalculatorButton {

    /* Creates a new DigitButton for the digit n */
    public DigitButton(int n) {
        super(" " + n);
    }

    /* Adds this digit to the display */
    public void action(CalculatorDisplay display) {
        display.addDigit(Integer.parseInt(getText()));
    }
}

/*
 * This abstract class is the superclass of the various operator buttons.
 * Each concrete subclass must override the apply method.
 */
abstract class OperatorButton extends CalculatorButton {

    /* Creates a new OperatorButton with the specified name */
    public OperatorButton(String name) {
        super(name);
    }

    /* Informs the display that this operator button has been clicked */
    public void action(CalculatorDisplay display) {
        display.setOperator(this);
    }

    /* Applies this operator (every subclass must implement this method) */
    public abstract int apply(int lhs, int rhs);
}
```

图 10-45 (续)

```

/*
 * The classes AddButton, SubtractButton, MultiplyButton, and DivideButton
 * are the same except for their label and the implementation of apply.
 */
class AddButton extends OperatorButton {
    public AddButton() { super("+"); }
    public int apply(int lhs, int rhs) { return lhs + rhs; }
}

class SubtractButton extends OperatorButton {
    public SubtractButton() { super("-"); }
    public int apply(int lhs, int rhs) { return lhs - rhs; }
}

class MultiplyButton extends OperatorButton {
    public MultiplyButton() { super("x"); }
    public int apply(int lhs, int rhs) { return lhs * rhs; }
}

class DivideButton extends OperatorButton {
    public DivideButton() { super("/"); }
    public int apply(int lhs, int rhs) { return lhs / rhs; }
}

/*
 * The EqualsButton class displays the current value. As it happens, this
 * operation can be implemented simply by setting the operator to null.
 */
class EqualsButton extends CalculatorButton {
    public EqualsButton() {
        super("=");
    }

    public void action(CalculatorDisplay display) {
        display.setOperator(null);
    }
}

/*
 * The ClearButton class resets the calculator by setting the operator to
 * null and the display value to 0.
 */
class ClearButton extends CalculatorButton {
    public ClearButton() {
        super("C");
    }

    public void action(CalculatorDisplay display) {
        display.setOperator(null);
        display.setValue(0);
    }
}

```

图 10-45 (续)

虽然图 10-45 中的许多代码完全遵守了本节前面描述的设计，但下面几个方面还是值得注意。

- 程序包私有类。本文到目前为止，类都是指定为 `public`。使用类层次分解来定义问题结构时，通常许多这样的类只在实现方式内部有意义，就像私有辅助方法只在定义它们的类中有意义一样。Java 允许在类定义中删除关键字 `public`，以支持这种“辅助类”。

这样的类称为程序包私有类，它们只有在定义它们的程序包中可用。Java 要求所有的公有类必须定义在单个源文件里；相反，单个源文件可以包含多个程序包私有类。

- 检查对象是否为类的实例。将来的实现者可能要将 `CalculatorButton` 层次结构之外的新按钮添加到计算器，为此 `actionPerformed` 的实现方式在调用 `action` 方法之前，应该检查事件源是不是真正的 `CalculatorButton`。调用内置的 `instanceof` 运算符，Java 可以实现这样的检查。`instanceof` 运算符将值放在左边，将类名称放在右边。如果值是指定类的合法实例且不等于 `null`，`instanceof` 运算符就返回 `true`。
- 抽象方法。定义抽象类时，通常要指定存在没有编写实现方式的特定方法。本示例中，抽象类 `CalculatorButton` 用来指定存在 `action` 方法，它将该方法的确切实现方式让给它的各个子类完成。指定关键字 `abstract`，并用分号代替主体，就可以声明这种方法。

10.9 小结

本章介绍了编写交互式程序的几种方法，特别是涉及图形用户界面(简称 GUI)的那些方法。本章介绍的重点有：

- 现代应用程序都能对用户任何时候——不仅仅是程序要求它们的时候——采取的动作作出响应。在程序正常顺序流之外发生的用户动作称为事件。响应这些事件的程序称为事件驱动程序。
- 在 Java 中，程序通过指定对象作为特定事件类型的监听器，来响应事件。事件监听器必须实现一个或多个 `java.awt.event` 程序包中定义的接口。本章讨论的监听器接口有 `MouseListener`、`MouseMotionListener`、`KeyListener` 和 `ActionListener`。
- `Program` 类通过给每种所需的方法提供空定义来实现上述 4 个接口。如果要改变某个方法的动作，只需为程序类里的那个方法提供新定义即可。新的定义将重写默认的实现方式。
- 事件驱动程序倾向于包括 `init` 方法而不是 `run` 方法。`init` 方法指定程序开始前必须执行的初始化代码。`run` 方法指定程序运行时采取的动作。
- 实现图 10-4 中定义的一种或多种监听器方法，然后调用 `addMouseListeners` 作为 `init` 方法的一部分，可以响应 `GraphicsProgram` 中的鼠标事件。同样，实现图 10-10 中定义的监听器方法，然后调用 `addKeyListeners`，可以响应键盘事件。
- `javax.swing` 和 `acm.gui` 程序包定义了许多响应用户动作的类。这些类的实例通常称为交互器。本文使用了下面的交互器类：

<code>JButton</code>	触发特定动作
<code>JCheckBox</code>	指定选项为选择或者未选择
<code>JRadioButton</code>	从一组选项中选择唯一选项
<code>JSlider</code>	在连续范围内调整参数
<code>JLabel</code>	添加交互式消息来指导用户
<code>JComboBox</code>	弹出选择菜单
<code>JTextField</code>	允许用户输入文本
<code>IntField</code>	允许用户输入整数
<code>DoubleField</code>	允许用户输入浮点值

- 单击 JButton 类的实例上单击时，会生成动作事件。定义 actionPerformed 方法可以响应该事件，actionPerformed 方法实现 ActionListener 接口。调用 init 方法中的 addActionListeners，可以将程序作为动作侦听器添加给窗口内的所有按钮。
- 确定哪个按钮导致了动作事件的方法有两种。一种方法是调用事件的 getSource，它返回导致事件的对象。另一种方法是调用事件的 getActionCommand，然后使用结果字符串来确定响应。对于 JButton 而言，动作命令默认定义为按钮标签。
- 将交互器添加到程序最简单的方法是将它们添加到程序窗口边框中的控件栏。
- 排列窗口里的交互器通常要使用布局管理器，它负责排列容器内的组件。本章介绍了 java.awt 程序包中标准的 BorderLayout、FlowLayout 和 GridLayout 布局管理器，以及 acm.gui 中更加灵活的 TableLayout 管理器。TableLayout 管理器可以指定控制单个组件布局的约束。约束规范中支持的选项如图 10-42 所示。
- 在许多应用程序中，定义交互器行为最好的方法是创建层次结构，在此层次结构中，各子类通过重写层次结构中所有类通用的方法定义来定义自己的行为。
- 使用类层次分解设计应用程序时，层次结构中的辅助类通常只与实现方式有关，对客户则没有用。在那种情况下，通常使用程序包私有类，它不使用关键字 public 定义。程序包私有类可以作为公有类包含在相同的源文件里，它们应用于该公有类。

10.10 复习题

1. 定义术语“事件”和“事件驱动”。
2. 什么是事件侦听器？
3. MouseEvent、KeyboardEvent 和 ActionEvent 在什么程序包定义？
4. 判断题：Program 类定义了一组毫无用处的侦听器方法。
5. 在 Program 类中，init 方法和 run 方法在功能上有什么不同？
6. Java 事件模型用来响应鼠标事件的接口是哪两个？产生这种差别的原因是什么？
7. 判断题：MousePressed 事件总是在 MouseClicked 事件前面，而 MouseClicked 事件总是在 MouseReleased 事件前面。
8. 描述方法 addMouseListeners、addKeyListeners 和 addActionListeners 的结果。
9. 用一两句话描述使用下列交互器的环境：JButton、Jcheckbox、JradioButton、Jslider、Jlabel、JComboBox、JtextField、IntField 和 DoubleField。
10. 单击 JButton 对象时，会生成什么类型的事事件？
11. 构造新的 JButton 时，用作其默认动作命令的值是什么？
12. 如果将多个交互器添加到位于程序窗口 SOUTH 边框的控件栏中，会发生什么情况？
13. acm.graphics 程序包中的哪些类与 Java 窗口层次结构中的 Component 和 Container 类相似？
14. 什么是布局管理器？
15. 分别描述 BorderLayout、FlowLayout 和 GridLayout 布局管理器使用的布局策略。
16. BorderLayout 类中定义了 5 个区域，它们的名称是什么？

17. BorderLayout 管理器排列其组件时，它将角空间定位到左右边框还是上下边框？

18. 描述 TableLayout 类中 width 和 gridwidth 选项之间的不同。

19. 对于下列每个 init 方法，画出完成初始化过程之后程序窗口的略图。

(1) public void init() {

```
    add(new JButton("Button 1"), SOUTH);
    add(new JButton("Button 2"), SOUTH);
    add(new JCheckBox("Finished"), SOUTH);
}
```

(2) public void init() {

```
    setLayout(new GridLayout(2, 1));
    add(new JButton("Button 1"));
    add(new JButton("Button 2"));
}
```

(3) public void init() {

```
    setLayout(new TableLayout(3, 2));
    add(new JButton("Button 1"), "gridwidth=2");
    add(new JButton("Button 2"), "gridheight=2");
    add(new JButton("Button 3")); add(new JButton("Button 4"));
}
```

20. 什么是程序包私有类？

10.11 编程练习

1. 修改第 9 章编程练习 3 中的 RandomColorLabel 程序，让它在某个 GLabel 上按下鼠标键时，临时将颜色重新设置为与其名称相匹配的颜色。释放鼠标键应该为标签任意选择一种新颜色。

2. 编写 GraphicsProgram 程序，在窗口移动或拖动鼠标指针时，该程序使用 GLabel 显示鼠标指标坐标。GLabel 应该总是出现在鼠标指针当前位置的左边，如图 10-46 所示。



图 10-46 鼠标跟踪器

如果移动鼠标指针，标签应该跟随它移动，并且在移动的同时更新坐标值。

3. 重写 9.4 节的 DrawFace 程序，让眼睛有圆形的瞳孔，并让它总是看着鼠标指针的位置。例如，如果鼠标指针在眼睛以下脸的右边，那么眼睛应该向右下看，如图 10-47 所示。



图 10-47 DrawFace 示例

移动鼠标时，眼睛应该跟随鼠标指针的位置。只要鼠标指针在脸部之外就不要紧，但是单独计算每只眼睛中瞳孔的位置非常重要。例如，如果鼠标指针在两只眼睛之间移动，瞳孔应该看着相反的方向，这样看起来就是斜视眼。

4. 除了 DrawLines 程序生成的直线画以外，交互式绘画程序还可以在画布上画其他形状。在典型的绘画应用程序中，在某个角按下鼠标，然后朝它相对的角拖动，可以创建矩形。例如，如果在图 10-48 左图中的位置按下鼠标，然后向图 10-48 右图中所示的方向拖动，可以创建矩形。

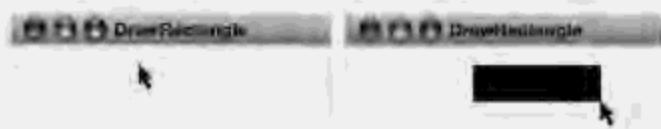


图 10-48 画矩形的示例(1)

矩形随着鼠标的拖动而变大。释放鼠标时，矩形就在那个位置完成。可以用相同的方法添加更多矩形。

虽然该练习的代码很短，但有一点很重要。上面的示例中，最初在矩形的左上角按住鼠标键。然而，除了向右下角拖动鼠标外，如果向其他方向拖动，程序也应该能够完成矩形。例如，向左拖动鼠标时也应该能够画矩形，如图 10-49 所示。

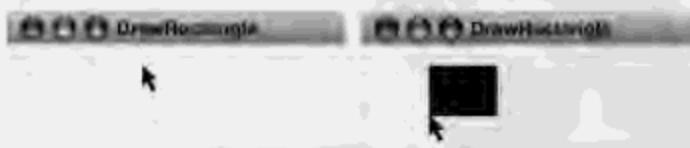


图 10-49 画矩形的示例(2)

5. 使用 DrawLines 和 DrawRectangle 程序作为起点，创建一个更精细的绘画程序，它在画布左边有由 5 个形状——填充的矩形、矩形轮廓、填充的椭圆、椭圆轮廓和直线——组成的调色板，如 10-50 图所示。



图 10-50 绘画示例

在调色板中的某个正方形上单击，可以选择该形状作为绘画工具。因此，如果在调色板中间填充的椭圆上单击，程序应该画填充的椭圆。在调色板之外的空白区域单击和拖动，应该画出当前选择的形状。在现有形状内单击和拖动，应该在画布上移动该形状。在形状上单击但没有拖动，应该将形状移动到堆栈顺序的最前面。

6. 图 10-26 中 DrawStarMap 程序的代码包含 JComboBox，它允许用户选择新星星使用的颜色。在目前情况下，从 JComboBox 使用的名称中确定实际颜色，这一过程需要一长串 if 语句，这些语句用来寻找不同的颜色名，并返回相应的 Color 对象。除了单调乏味之外，这种策略还有其他，就是添加新颜色需要改变程序的两个部分：initColorChooser 方法(用来创建颜色选择器)中的代码，以及解释颜色名的 getCurrentColor 的代码。如果有人要添加新的颜色而没有改变这两个方法，程序就不能正确运行。

JComboBox 中的选项不必是字符串，这一点有利于提供更有效的实现方式，但解决方案不能一蹴而就。如果将 Java 颜色值直接放进 JComboBox，就可以通过取得当前项并将它投射到 Color 来设置当前颜色，这样就从总体上减少了 if 语句。问题是 JComboBox 选项的标签不再特别容易理解，因为 JComboBox 类使用每一项的 `toString` 方法来创建标签。不是看到选择列表中的 RED，而是看到字符串

```
java.awt.Color[r=255,g=0,b=0]
```

很了解 Java 并理解颜色模型的人可能明白这种颜色就是红色，但大多数用户会根本不知道。要解决这个问题，可以定义 LabeledColor 类，它扩充标准的 Color 类，还会给 `toString` 方法的结果确定某个名称。如果使用扩充类，通过写

```
colorChooser.addItem(new LabeledColor(Color.RED, "Red"));
```

就可以将颜色添加到颜色选择器。

实现 LabeledColor 类并将它合并到 DrawStarMap。

7. 将下列扩展合并到 DrawStarMap 应用程序。

- 让用户能够拖动画布上的星星。如果按下鼠标键事件发生在现有星星之外，它应该创建一个新的星星，和在程序的原始版中一样。然而，如果按下鼠标键的事件发生在现有星星内部，那么拖动鼠标就能够让星星随之移动，和图 10-9 中 DragObjects 程序一样。
- 添加一个键侦听器，以便按下箭头键，就能将当前的星星朝相应方向移动 1 个像素。另外，按 Delete 键应该可以将当前星星从画布上完全删除。
- 将 JTextField 添加到控件条，它允许用户输入星星名称。用户在名称结尾按 Enter 键时，DrawStarMap 程序应该将名称作为 GLabel 添加到最近一颗星星的右边。标签应该使用

颜色选择器指定的颜色，与星星的颜色相同，除非用户进行了显式改变。确保可以在画布上拖动 GLabel，和拖动星星一样。

8. 写 GraphicsProgram 的 init 方法，创建如图 10-51 所示的底部显示的控件栏。



图 10-51 速度控件栏

9. 将前面练习中的速度控件栏合并到第 4 章练习 15 的 BouncingBall 程序的代码中。程序应该使用 run 方法来实现动画循环(而不应该真正移动球)，直到用户单击“开始”按钮为止。虽然第 14 章将会介绍用于控制动画的更好策略，但实现该程序最简单的方法是声明标记变量。如果标记是 true，程序在每个动画周期都移动球。如果标记是 false，程序就不移动球。然而，在任何一种情况下，程序都要调用 pause 方法，确保程序没有让计算机在每个 CPU 周期都忙碌。

控件栏中滑块的目的是让用户能够改变动画的速度。例如，当控件始终设置在滑块缓慢的一边时，程序就会暂停相对较长的时间，让球缓慢移动。如果将控件移动到滑块快速的一端，暂停时间就会缩短。如果要让球更快地移动，还需要改变每个时步中球移动的距离。

10. 编写程序来玩经典的 Breakout 游戏，它由 Steve Wozniak 于 1976 年开发，Steve Wozniak 后来成为了 Apple 的创始人之一。在 Breakout 中，目标是用跳动的球打击每块砖来清除它们。

Breakout 游戏的最初配置如图 10-52 所示。屏幕顶部有颜色的矩形表示砖块，两行红色的，两行橙色的，两行绿色的，两行青色的。底部稍微大一点的矩形是划桨。划桨在垂直方向上处于固定位置，但它可以随着鼠标的移动在屏幕上来回移动，直到移动到空间的边界为止。

完整的 Breakout 游戏由 3 个回合组成。每个回合，球都从窗口中心向屏幕底部以任意角发出。球从划桨和墙壁上反弹。这样，再次反弹之后——一次从划桨，一次从墙壁，球的轨迹如图 10-53 所示。(注意，用来表示球路径的点线不会出现在屏幕上)。

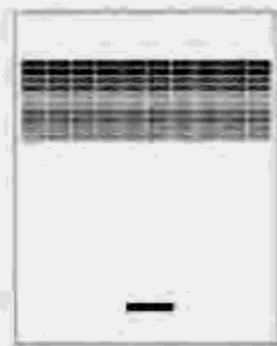


图 10-52 Breakout 的最初配置

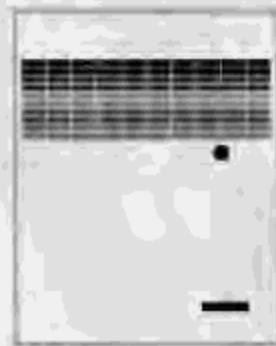


图 10-53 球的轨迹

从图 10-53 可以得知，球与底部一排的某个砖块发生碰撞。发生这种情况时，球会反弹，就像发生其他碰撞一样，但砖块会消失。图 10-54 表示碰撞之后以及移动划桨对准即将来临的球的情况，游戏以这种方法继续，直到出现下面的条件之一为止：

- 球打到了下面的墙壁上，这说明划桨没有接住它。在这种情况下，这一回合结束，开始使用下一个球，如果没有耗尽分配的 3 个回合的话。如果耗尽了，游戏即以失败结束。
- 清除了最后一块砖。在这种情况下，游戏立即结束，可以胜利退出。

某一列中的所有砖块都被清除之后，路径会向顶上的墙壁敞开。发生这种可喜的情况时，球通常在墙壁顶部和砖块最上面一行来回反弹几次，这时用户不必用划桨打球。这种情况是对“突破”的回报，这也是游戏名的含义。图 10-55 所示为第一个球突破墙障后不久的情况。球在回到开放通道之前会继续清除更多砖块。

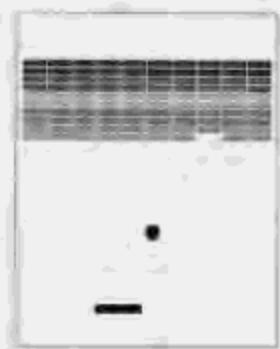


图 10-54 碰撞之后

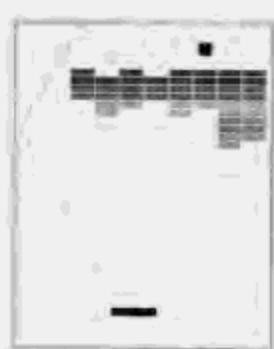


图 10-55 突破

即使突破是玩家经历中很令人兴奋的时候，但程序中不必做特殊的事情来让它发生。注意到这一点非常重要。这个游戏一直使用相同的规则。从墙壁反弹，清除砖块，满足物理规律。

该实现方式唯一需要解释的是如何确定球是否已经与砖块或划桨发生了碰撞。最简单的策略是调用 `getElementAt(x, y)`，它返回包含画布上点(x, y)的对象，特别是 x 和 y 坐标。如果位置上没有图形对象，`getElementAt` 返回常量 `null`。如果有多个对象，`getElementAt` 总是选择堆栈顺序中最前面的一个。

假设球不是单个点，那么只核对中心的坐标是不够的。在这个程序中，最简单的策略是核对正方形(球与该正方形内切)的 4 个顶点。GOval 依据其外切矩形来定义，这一点说明，如果球的左上角位于点(x, y)，那么其他角的位置就如图 10-56 所示。



图 10-56 球外切矩形的坐标

这些点的好处是处于球之外——这说明 `getElementAt` 不能返回球本身——但它们非常接近，足以说明碰撞是否发生。因此，对于这 4 个点，需要在各个位置调用 `getElementAt` 来确定那里是否有对象。如果返回的值不是 `null`，就不必再寻找了。可以将该值当作碰撞与之发生的 GObject。如果 `getElementAt` 在全部 4 个角上都返回 `null`，表示没有发生碰撞。

11. 虽然美国国会在 1866 年批准法律声明“在所有合同、交易和法院活动中使用公制度量衡”是合法的，但传统的英制单位在美国仍然盛行。所以，美国人仍然必须学习英制单位的转

换规则，至少要知道左边一些熟悉的规则，以及右边一些不常见的规则：

12inch = 1foot	6ft = 1fathom
3inch = 1yard	16.5ft = 1rod
5280inch = 1mile	22yard = 1fur

为了让用户理解这些单位，编写在各种单位间相互转换的 LengthConverter 程序。程序的用户界面如图 10-57 所示。



图 10-57 LengthConverter 程序的输出示例(1)

顶上一行的每个选择器应该让用户能够选择所有 7 种可能的单位：inch、foot、yard、fathom、rod、furlog 和 mile。要实现转换，用户应该选择单位，在数字字段输入数字，然后单击相应的转换按钮。例如，如果要将 18inch 转换为英尺，应该将左边的选择器改变为 foot，在右边数字字段里输入值 18，然后单击字段下面的按钮。程序应该使用 JComboBox 设置来确定转换系数，然后计算新的结果并存储在另一个数字字段里，如图 10-58 所示。



图 10-58 LengthConverter 程序的输出示例(2)

和图 10-41 中的 TemperatureConverter 程序一样，该程序也应该允许用户通过按 Enter 触发转换。

程序允许用户选择两个不同的单位，实现这一点最简单的策略是将输入字段的值转换为英寸，然后执行第二次转换，得到最终结果。要实现该策略，需要在代码中包含一些机制，来确定每个单位转换到相等英寸数的系数。一种策略是用一连串 if 语句明确编码该信息。然而，可以使用练习 6 中的策略，将转换系数与选择项存储在一起。

12. 扩展 Calculator 程序，让它使用浮点数而不使用整数。解决这个问题包括决定设计(例如将小数点键添加到哪里)以及实现方式的变化，这些变化要求至少改变现有结构的几个部分。

第 11 章

数组与 ArrayList 类

Little boxes on a hillside, little boxes made of ticky-tacky

Little boxes, little boxes, little boxes all the same

There's a green one and a pink one and a blue one and a yellow one

And they're all made out of ticky-tacky and they all look just the same

—Malvina Reynolds, “Little Boxes,” 1962



Bob Frankston and Dan Bricklin

现代计算中，数组结构最明显的一个应用是电子数据表，它使用二维数组存储表格数据。第一个电子数据表是 VisiCalc，它由 Software Arts Incorporated 于 1979 年发布。Software Arts Incorporated 是一家新的小公司，由 MIT 毕业生 Dan Bricklin 和 Bob Frankston 创建。事实证明 VisiCalc 是一个很受欢迎的应用程序，这导致很多大公司也着手开发竞争产品，包括 Lotus 123 以及 Microsoft Excel。

到目前为止，本书许多程序已经使用了单个数据项。然而，计算的真正力量来源于使用数据集合的能力。本章介绍了数组的思想，数组是相同类型值的有序集合。数组在编程中非常重要，因为现实中这样的集合很常见。当需要表示一组值而且要让这些值形成序列时，数组在其解决方案中就可以发挥作用。

同时，数组本身却变得越来越不重要，因为 Java 的库程序包包含许多类，这些类可以实现数组所能实现的所有功能，甚至更多。因为这些类表示单个数据值的集合，所以发挥数组传统作用的这些类称为集合类。Java 编程人员越来越依赖这些集合类来表示存储在数组内的各种数据。然而，如果理解了数组模型，会比较容易理解集合类，数组模型是许多集合类的概念基础。由于在编程早期就已经使用过数组，所以本章从讨论它们开始。然而，这一点不能确保理解了数组处理的细节，就能提供理解 Java 集合类运行方式所需的直观认识。

为了让用户预先了解集合类，本章还介绍了 ArrayList 类，它是与数组本身最相似的集合

类。第 13 章将更全面地介绍 Java 的集合类。

11.1 数组简介

数组是单个值的集合，它有两个明显特征。

- 数组是有序的。必须按顺序枚举单个值：第一个在这里，第二个在那里，等等。
- 数组是同类的。存储在数组里的每个值必须是同一种类型。因此，可以定义整数数组或浮点数数组，但不能定义两种类型混合的数组。

从整体的观点来看，最好将数组作为一序列方框，每个方框表示数组里的一个数据值。数组里的每个值称为元素。例如，下面的方框表示有 5 个元素的数组。



每个数组都有两个应用于整个数组的基本属性：

- 元素类型，表示元素里可以存储什么值。
- 长度，即数组包含的元素数量。

在 Java 中，可以在不同时间指定这些属性。声明数组变量时，可以定义元素类型；创建初始值时，可以设置数组长度。然而大多数情况下，会在相同声明里指定这两种属性，如 11.1.1 小节所述。

11.1.1 数组声明

和 Java 中的其他变量一样，在使用数组之前必须先声明。数组声明最常见的形式如下边的语法框所示。

例如，声明

```
int[] intArray = new int[10];
```

声明名为 intArray 的数组，它有 10 个元素，每个元素都是 int 类型。可以画一行有 10 个方框的图来表示该声明，整个集合的名称是 intArray。

数组声明的典型语法

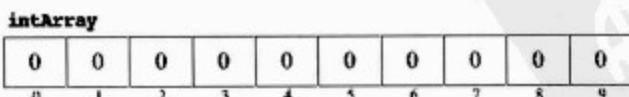
```
type[] identifier = new type[length];
```

其中：

type 是每个元素的类型；

name 是声明为数组变量的名称；

length 是分配为数组一部分的元素数量。



Java 创建新数组时，它将每个元素初始化为该类型的默认值。对于数字而言，默认值是 0，这就是 0 成为 intArray 的每个元素的原因。其他所有类型也有默认值，例如，boolean 的默认

值是 false，对象的默认值是 null。

数组中每个元素用称为索引的数字值标出。Java 中，数组的索引数总是从 0 开始，一直累加到比数组长度少 1 个数。因此，在有 10 个元素的数组中，索引数是 0、1、2、3、4、5、6、7、8 和 9，如上面的 intArray 数组。

虽然可以使用整数值——例如前面示例中的 10——来指定数组长度，但更常见的是使用命名常量。例如，假设要求定义能够保存体育竞赛(如体操或花样滑冰)分数的数组，在这些比赛中，分数由裁判小组给定。每位裁判按 0~10(10 为最高分)的等级为表演打分。因为分数可能包含小数点，如 9.9，所以数组的每个元素必须是 double 类型。而且，因为裁判数可能随应用程序的改变而改变，所以最好使用命名常量声明数组长度。这里，数组 scores 的声明如下所示：

```
private static final int N_JUDGES = 5;
double[] scores = new double[N_JUDGES];
```

该声明引入了称为 scores 的新数组，它有 5 个元素，如下所示。

scores				
0	1	2	3	4
0.0	0.0	0.0	0.0	0.0

在 Java 中，用来指定数组长度的值不一定是常量。如果要让体育分数程序更通用，可以由用户读入裁判数，代码如下：

```
int nJudges = readInt("Enter number of judges: ");
double[] scores = new double[nJudges];
```

11.1.2 数组选择

虽然很多情况下可以作为单个实体使用数组值，但通常也要操作单个元素。确定数组内特定元素的过程称为选择。要选择单个元素，首先要提供数组名，然后按方括号内的索引查找。结果是选择表达式，其形式如下：

```
array [ index ]
```

在程序内，选择表达式和简单的变量一样。可以在表达式中使用它，特别是可以给它赋一个值。因此，如果第一位裁判(裁判#0，因为 Java 从 0 开始计算数组元素)给选手打了 9.2 分，那么写赋值语句：

```
scores[0] = 9.2;
```

就可以将这个分数指派给第一个元素。该赋值语句的结果如下所示。

scores				
0	1	2	3	4
9.2	0.0	0.0	0.0	0.0

因此，使用语句

```
scores[1] = 9.9;
scores[2] = 9.7;
```

```
scores[3] = 9.0;
scores[4] = 9.6;
```

就可以继续指派其他 4 位裁判的分数。

执行这些语句后产生的结果如下所示。

scores				
9.2	9.9	9.7	9.0	9.6
0	1	2	3	4

数组元素的索引与该元素的值不同，使用数组的过程中，理解这种差别至关重要。例如，数组中第一个方框的索引是 0，而它的值是 9.2。可以改变数组中的值，但不能改变索引数，记住这一点也很重要。

索引值不一定是常量，它实际上可以是赋值为整数或其他标量类型的表达式，数组选择的功能即来源于这一点。许多情况下，选择表达式是 for 循环的索引变量，这样就可以方便地依次在数组的每个元素上执行操作。例如，使用下面的语句，可以将 scores 数组里的每个元素设置为 0.0：

```
for (int i = 0; i < nJudges; i++) {
    scores[i] = 0.0; }
```

11.1.3 简单数组的示例

图 11-1 中的 GymnasticsJudge 程序是数组操作的简单示例。程序要求用户输入每位裁判的分数，然后显示平均分。

```
/*
 * File: GymnasticsJudge.java
 *
 * -----
 * This file reads in an array of scores and computes the
 * average.
 */

import acm.program.*;

public class GymnasticsJudge extends ConsoleProgram {

    public void run() {
        int nJudges = readInt("Enter number of judges: ");
        double[] scores = new double[nJudges];
        for (int i = 0; i < nJudges; i++) {
            scores[i] = readDouble("Score for judge " + i + ": ");
        }
        double total = 0;
        for (int i = 0; i < nJudges; i++) {
            total += scores[i];
        }
        double averageScore = total / nJudges;
        println("The average score is " + averageScore);
    }
}
```

图 11-1 求体操分数组平均值的程序

使用 11.1.2 小节中的数据运行 GymnasticsJudge 程序，产生的运行结果如图 11-2 所示。



图 11-2 GymnasticsJudge 程序的运行结果

11.1.4 改变索引范围

在 Java 中，每个数组中的第一个元素都在索引位置 0。然而，在许多应用程序中，这种设计会让用户产生混淆。对于使用 GymnasticsJudge 程序的非技术人员而言，询问 0 号裁判可能让人莫名其妙。在现实世界中，倾向于从 1 开始对元素编号。因此，对这个程序而言，让用户输入 1~5 号裁判的分数可能更合理。

有两种标准方法可以改变用户看到的索引值：

- 在内部使用索引 0~4，但从用户输入数据或显示输出数据时，给每个索引值加 1。如果采用这种方法，GymnasticsJudge 程序唯一需要改变的是读入输入值的 readDouble 语句，将它改变为

```
scores[i] = readDouble("Score for judge " + (i + 1) + ":" );
```

- 用其他元素声明数组，让它的索引编号为 0~5，然后完全忽略元素 0。使用这种方法，其 run 方法如下：

```
public void run() {
    int nJudges = readInt("Enter number of judges: ");
    double[] scores = new double[nJudges + 1];
    for (int i = 1; i <= nJudges; i++) {
        scores[i] = readDouble("Score for judge " + i + ":" );
    }
    double total = 0;
    for (int i = 1; i <= nJudges; i++) {
        total += scores[i];
    }
    double averageScore = total / nJudges;
    println("The average score is " + averageScore);
}
```

为了分配其他元素，run 方法使用 nJudges + 1 声明 scores 数组来指定数组大小。

第一种方式的优点是内部数组索引仍然从 0 开始，使用依赖这种假设的现有方法就相对容易。不足是程序需要两组不同的索引：用户使用的外部索引和编程人员使用的内部索引。即使用户理解其一致性并且熟悉索引模式，考察两组索引也会让程序过程变得复杂。第二种方法的优点是编程人员的索引与用户的索引一致。

11.1.5 对象的数组

数组的元素不局限于原始类型(例如 `GymnasticsJudge` 示例中使用的 `double`)，它可以是任何 Java 类的对象。例如，假设要将数组声明为存储班级前 5 名学生，其中单个元素是 6.4 节中定义的 `Student` 类的实例。这样的数组可以声明如下：

```
Student[] topStudents = new Student[5];
```

这种声明的结果引入了数组，该数组中的 5 个元素都能够保存 `Student` 对象。然而，这些元素被初始化为 `null`，`null` 是所有对象的默认值。执行 `topStudent` 声明的数组如下所示。

topStudents				
0	1	2	3	4
<code>null</code>	<code>null</code>	<code>null</code>	<code>null</code>	<code>null</code>

可以将新的值赋给 `topStudents` 数组的元素，和使用原始值的数组一样。例如，执行语句

```
topStudents[0] = new Student("Hermione Granger", 314159);
```

会用与 `Hermione Granger` 对应的新学生对象取代初始元素中的 `null`。

11.1.6 在图形程序中使用数组

从本章结尾的一些练习中可以看出，数组在 `GraphicsProgram` 也很有用。`acm.graphics` 程序包本身的实现方式在一些语境中也使用数组。例如，`GPolygon` 的顶点在内部存储为 `GPoint` 对象的数组。在应用级别创建 `GPoint` 对象的数组通常也很有意义。

例如，图 11-3 中的 `YarnPattern` 程序只使用 `GLine` 对象就创建了美丽的图案。每个 `GLine` 对象连接两个存储在数组里的 `GPoint`。图案模型来自于现实中可以实现的过程。首先取一块矩形木板，沿边界排列钉子，让它们沿 4 条边界均匀分布。`YarnPattern` 程序的代码使用常量 `N_ACROSS` 和 `N_DOWN` 定义尺寸。钉子的总数——假设定义了 `N_ACROSS` 和 `N_DOWN`，它们不会将角计算两次——可以定义如下：

```
private static final int N_PEGS = 2 * N_ACROSS + 2 * N_DOWN;
```

生成图案的过程是围绕钉子缠绕一些有颜色的纱线，从左上角的钉子开始，然后扩展到围绕周边固定距离的钉子。距离由参数 `DELTA` 给定，通常比 `N_ACROSS` 或 `N_DOWN` 大，但比它们的和小。从这颗钉子开始，过程每次在钉子数组中向前移动 `DELTA` 步，直到纱线循环回到起点为止。给定常量值的 `YarnPattern` 程序的输出如图 11-4 所示。改变这些常量，可以创建其他完全由直线组成的有趣图案。

```

import acm.graphics.*;
import acm.program.*;
import java.awt.*;

/*
 * This program creates a pattern that simulates the process of winding a piece
 * of colored yarn around an array of pegs along the edges of the canvas.
 */
public class YarnPattern extends GraphicsProgram {

    public void run() {
        initPegArray();
        int thisPeg = 0;
        int nextPeg = -1;
        while (thisPeg != 0 || nextPeg == -1) {
            nextPeg = (thisPeg + DELTA) % N_PEGS;
            GPoint p0 = pegs[thisPeg];
            GPoint p1 = pegs[nextPeg];
            GLine line = new GLine(p0.getX(), p0.getY(), p1.getX(), p1.getY());
            line.setColor(Color.MAGENTA);
            add(line);
            thisPeg = nextPeg;
        }
    }

    /* Initializes the array of pegs */
    private void initPegArray() {
        int pegIndex = 0;
        for (int i = 0; i < N_ACROSS; i++) {
            pegs[pegIndex++] = new GPoint(i * PEG_SEP, 0);
        }
        for (int i = 0; i < N_DOWN; i++) {
            pegs[pegIndex++] = new GPoint(N_ACROSS * PEG_SEP, i * PEG_SEP);
        }
        for (int i = N_ACROSS; i > 0; i--) {
            pegs[pegIndex++] = new GPoint(i * PEG_SEP, N_DOWN * PEG_SEP);
        }
        for (int i = N_DOWN; i > 0; i--) {
            pegs[pegIndex++] = new GPoint(0, i * PEG_SEP);
        }
    }

    /* Private constants */
    private static final int DELTA = 67;      /* How many pegs to advance */
    private static final int PEG_SEP = 10;       /* Pixels separating each peg */
    private static final int N_ACROSS = 50;      /* Pegs across (minus one corner) */
    private static final int N_DOWN = 30;        /* Pegs down (minus one corner) */
    private static final int N_PEGS = 2 * N_ACROSS + 2 * N_DOWN;

    /* Private instance variables */
    private GPoint[] pegs = new GPoint[N_PEGS];
}

```

图 11-3 在画布上使用直线创建精美图案的程序

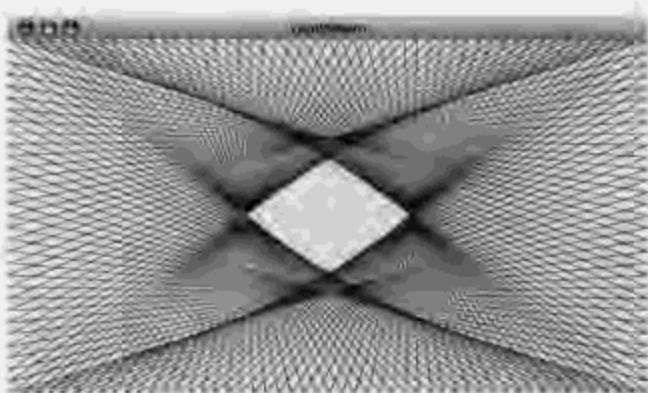


图 11-4 YarnPattern 程序的输出

11.1.7 ++和--运算符的区别

YarnPattern 程序介绍了 Java 的一个新特征，如果阅读现有 Java 代码，肯定会发现这个新特征。这个特征出现在语句

```
pegs[pegIndex++] = new GPoint( . . . );
```

中，它在 initPegArray 方法中以 GPoint 构造函数的不同参数出现了 4 次。

到目前为止，本文中的程序只是作为操作数执行++和--运算符，它们从应用到的变量里加 1 或减 1。凑巧的是，这些运算符比在示例中看到的还要有用。

首先，可以用两种方法写这两个运算符。运算符在它应用的操作数之后，像大家熟悉的

```
x++
```

或者在操作数之前，像

```
++x
```

第一种形式，运算符在操作数之后，这种形式称为后缀形式，第二种形式称为前缀形式。

如果单独执行++运算符——就像它是单独语句或作为 for 循环的递增运算符一样——前缀和后缀运算符的结果一样。如果将这两个运算符作为较长表达式的一部分使用，那么就会出现不同。和所有的运算符一样，++运算符返回值，但该值取决于运算符相对于操作数的位置。这两种情况如下所示：

- `x++` 首先计算 `x` 的值，然后递增。在执行递增运算符之前，返回到包围表达式的值是原始值。
 - `++x` 首先递增 `x` 的值，然后将新值作为++运算的值。
- 运算符与此类似，除了是递减值而不是递增值之外。

图 11-5 中的程序说明了++运算符的前缀形式与后缀形式之间的区别，解释了其操作。前 4 条语句说明了++运算符前缀形式的运算，接下来 4 条语句说明了其后缀形式的运算。程序的输出如图 11-6 所示。

所以，语句

```
pegs[pegIndex++] = new GPoint( . . . );
```

的结果是在 pegIndex 的当前值指定的索引位置将 GPoint 值赋给 pegs 数组的元素，然后递增 pegIndex，让它表示循环下一个周期里元素的位置。

```
import acm.program.*;

/** This program illustrates the prefix and postfix forms of the ++ operator */
public class IncrementOperatorExample extends ConsoleProgram {
    public void run() {
        int x = INITIAL_VALUE;
        println("If x initially has the value " + x + ", evaluating ++x");
        int result = ++x;
        println("changes x to " + x + " and returns the value " + result + ".");
        x = INITIAL_VALUE;
        println("Conversely, if x has the value " + x + ", evaluating x++");
        result = x++;
        println("changes x to " + x + " but returns the value " + result + ".");
    }
    /* Private constants */
    private static final int INITIAL_VALUE = 5;
}
```

图 11-5 说明应用++运算符结果的程序

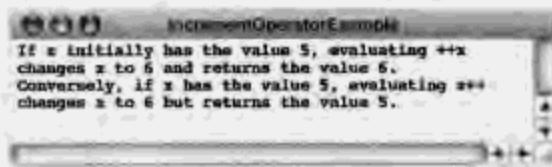


图 11-6 递增运算符示例的输出

11.2 数组的内部表示法

虽然本章前面的各种示例让用户对如何在程序中使用数组有了一定认识，但这些程序都过于简单，不足以说明使用数组时会遇到的某些重要问题。特别是到目前为止还没有哪个程序包含将数组作为参数或返回数组作为结果的方法。为了能编写更复杂的使用数组的程序，需要了解如何将数组信息在一种方法和另一种方法之间来回传递。

在理解数组数据如何在两个方法间传递的细节之前，有必要先理解数组在内存中的表示方法。Java 中，作为参数将数组——或对象，如第 7 章所示——传递给方法与传递简单变量有根本不同。如果理解数组在计算机内部的表示方法，就可以理解 Java 如何将数组作为参数处理。如果不理解内部表示法，Java 的方法就似乎完全让人难以理解。

第 7 章已经简要介绍了内存工作的方法。特别是已经知道对象存储在称为堆的内存区域，每个对象都用其内存里的地址标识。Java 中，在内部表示数组的方法与对象存储在内存里的方法完全一样。所以，数组值的内部表示法由一些对所有对象通用的标准信息组成，接着是存储堆的连续位置里数组元素所需的内存。

为了说明这是怎么回事，可以考察发布局部变量声明

```
double[] scores = new double[N_JUDGES];
```

时会出现什么情况，其中 `N_JUDGES` 定义为常量 5。虽然通常将这种声明模式作为单独的运算，但它实际上由两个不同的部分组成。等号左边的声明引入了名为 `scores` 的变量，它和所有局部变量一样，存储在堆栈中。然而，堆栈不包括实现元素的空间。`double` 数组由等号右边的初始器创建。表达式

```
new double[N_JUDGES]
```

分配堆栈中的数组对象(该对象包含通用于所有对象的信息)、数组长度(这里是 5)及放置 5 个 `double` 类型值的足够空间。5 个值本身消耗内存 40 个字节，可以计算如下：

8 字节/元素 × 5 元素 = 40 字节

`scores` 变量本身足以存储堆中数组对象的地址。因此内存的内部图片如图 11-7 所示。Student 数组示例中的状态更加令人费解。图 11-8 显示了执行语句

```
Student[] topStudents = new Student[5];
topStudents[0] = new Student("Hermione Granger", 314159);
```

后的内存布局。

这个示例包含 3 个对象：数组本身，对 `Student` 构造函数调用的结果，以及用来记录 `Hermione` 姓名的字符串。这些对象都存储在堆上面，如图 11-8 所示。String 对象——这里将它作为简单的字符数组用图解法表示——的内部表示法不完全正确，但这种表示法的细节超出了本文的范围。该图包含的重要思想是分配给字符串的内存都在堆上面。

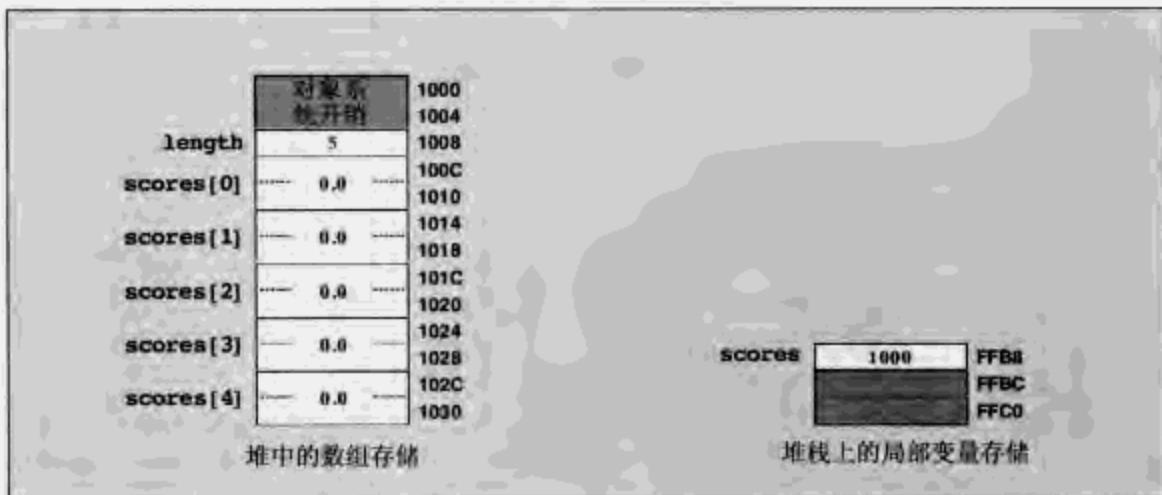
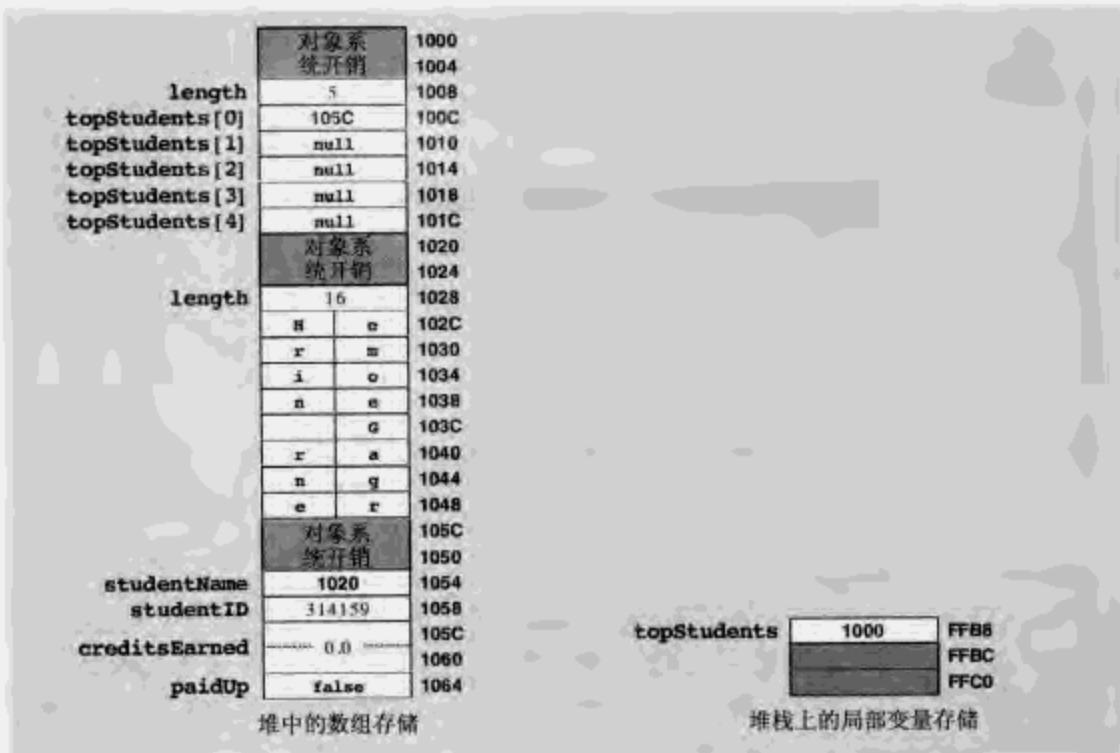


图 11-7 `scores` 数组的内存布局

如果关心的是对数组中的值有一个直观印象，那么这些内存图已经包含了所需的信息。对于抽象级别而言，简单的方框图就足够了。然而，在理解如何赋数组值以及如何作为参数传递数组方面，更精确的简化模型就很有用，这在 11.3 节介绍。

图 11-8 `topStudents` 数组的内存布局

11.3 数组作为参数传递

从第 5 章可以知道，编写大型程序的关键是将它分解为方法，每个方法都小到可以作为单元理解。单个方法通过将参数从一种方法传递到另一种方法来通信。如果大型程序涉及数组，分解程序就要求定义将整个数组作为参数传递的方法。在 Java 中，作为参数传递数组的操作与数组在内存内的表示法紧密相关，因此看起来很神秘。既然已经知道了内部表示法，那么就可以学习数组参数的运行方式及有效使用它们的方法。

将数组作为参数使用涉及的问题最好通过简单示例来说明。假设要求写实现下列步骤的程序：

- (1) 读入一列 5 个整数。
- (2) 倒转该列中的元素。
- (3) 显示次序倒转后的列。

如图 11-9 所示为程序的运行结果。

为了理解将数组作为参数传递的过程，有必要将这个程序分解为 3 个方法，这些方法对应于程序运行的 3 个阶段：读取输入值并将它们存储在数组里，倒转数组中的元素，显示结果。如果采用这种分解方法，程序的 `run` 方法其结构如下：

```
public void run() {
    int[] array = new int[N_VALUES];
    println("This program reverses an integer array.");
```

```
readArray(array);
reverseArray(array);
printArray(array);
```

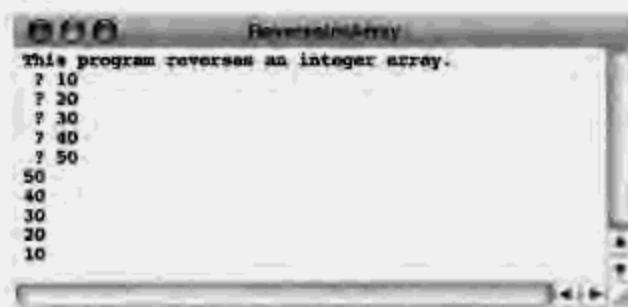


图 11-9 倒转整数的程序运行结果

虽然这种分解方法很直观，但它只有在 Java 允许方法改变数组元素——这些数组元素作为参数传递给这些方法的值时才能正确运行。如果传递像整数这样的原始值，这就意味着方法不可能改变它。然而，从第 7 章可以知道，将对象作为参数传递给方法意味着只复制了引用，因此方法与调用者有效共享了对象。所以分解 ReverseArray 程序过程中出现的重要的问题是，将数组作为原始值看待还是作为对象看待。答案是 Java 将所有数组定义为对象，这意味着数组元素在调用方法和被调用的方法间共享。

在详细介绍 Java 如何对待数组参数之前，有必要在假设的分解 ReverseArray 程序的过程中扩充一些方法。最简单的方法是 printArray，它的实现方式如下：

```
private void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        println(array[i]);
    }
}
```

数组参数的声明和数组变量的定义非常相似，只是没有指定初始值的表达式。值由调用者提供，因此方法唯一需要知道的是参数是整数数组这一点。printArray 方法的主体使用 for 循环来转换数组的每个元素，然后调用 println 显示值。

这种实现方式中唯一的新特征是用作 for 循环上限的表达式：

```
array.length
```

Java 中的每个数组都有 length 字段，它说明数组包含多少个元素。实际上，数组长度表示为对象内的字段，而不是方法。这一点在有时肯定会让产生混淆，特别是在数组和其他 Java 类之间建立逻辑连接时。例如，如果将数组在概念上看成对象序列，就像字符串是字符序列一样，那么数组使用 length 字段，而 String 类使用 length 方法，这不是很奇怪吗？

常见错误

确定数组长度的语法与确定字符串长度的语法不同，很容易忘记这一点。Java 中，每个数组对象都有 length 字段，它不是您调用的方法。

在 `printArray` 中, Java 如何实现将数组传递给方法这一过程并不重要。如果 Java 复制了整个数组, 结果正是我们想要的, 因为 `printArray` 只是查找数组中的值, 并没有打算改变它们。这种情况不同于 `readArray`, 它必须给元素赋新值。然而, 此代码非常类似 `printArray` 的代码:

```
private void readArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        array[i] = readInt(" ? ");
    }
}
```

这里, `readArray` 方法的正确执行取决于能够在数组元素中存储新值。如果查看图 11-5, 就会发现作为参数 `array` 传递的值只是堆中数组的地址, 并不包含数组元素本身。对于所有作为参数传递的对象都是这样, 地址被复制到 `readArray` 内相应的参数变量里, 但继续引用堆中的相同对象。

在 `ReverseArray` 程序中没有注意到的一点是 `reverseArray` 方法本身。基本算法很简单: 要倒转数组, 可以用最后一个元素取代第一个元素, 用倒数第二个元素取代第二个元素, 依次类推, 直到交换完所有元素为止。因为数组从 0 开始编号, 所以 `n` 项数组的最后一个元素的索引是 `n - 1`, 倒数第二个元素的索引是 `n - 2`, 依次类推。实际上, 给定任何整数索引 `i`, 从末尾出现 `i` 个元素的数组元素总是在索引位置

`n - i - 1`

因此, 为了倒转 `array` 的元素, 要做的就是, 从数组开头直到中间(其索引位置是 `n/2`), 为每个索引值 `i` 交换 `array[i]` 和 `array[n - i - 1]` 中的值。只要到了中间, 数组第二部分的元素已经有了正确的值, 因为 `for` 的每个周期正确重新配置了两个数组元素——每一半中各一个元素。在伪代码中, `reverseArray` 的实现方式如下:

```
private void reverseArray(int[] array) {
    for (int i = 0; i < array.length / 2; i++) {
        Swap the values in array[i] and array[n - i - 1];
    }
}
```

即使超出了本示例的范围, 交换数组中两个值的操作也非常有用。为此, 可以将该操作定义为单独的方法, 并且用单个方法调用取代 `reverseArray` 中剩余的伪代码。

虽然值得一试, 但交换两个元素的方法调用不能写成

`swapElements(array[i], array[n - i - 1]);`



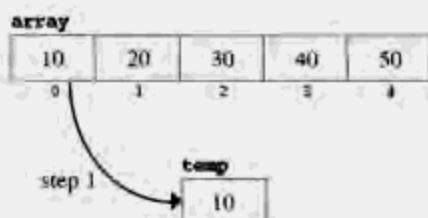
在该调用中, `swapElements` 的参数都是单个数组元素。数组元素和简单变量一样, 因此被复制给相应的形参。`swapElements` 方法可以交换这些值的局部副本, 但不能对调用参数永久赋值。要避免这个问题, 可以完全将整个数组, 以及表示交换位置的两个索引传递给实现交换操作的方法。例如, 调用

```
swapElements(array, i, n - i - 1);
```

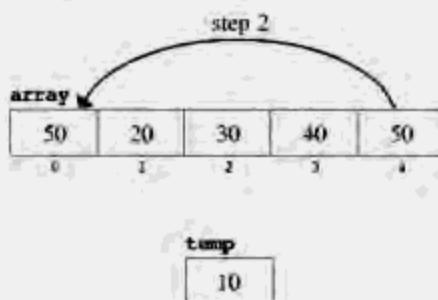
可以交换 array 中索引位置 i 和 $n - i - 1$ 的元素，这正是取代 reverseArray 中伪代码所需要的。

实现 swapElements 比预想的要复杂。不能简单地将一个元素赋给另一个元素，因为这样目标元素的原始值会丢失。解决这个问题最简单的方法是使用局部变量来临时保存两者其中的某个值。如果保存其中一个元素的值，那么就可以直接赋另一个值，之后可以从临时变量复制第一个值。假设要交换 array 中位置 0 和位置 4 上的值，该策略需要 3 步。

(1) 将 array[0]里的值存储在临时变量里，如下所示。

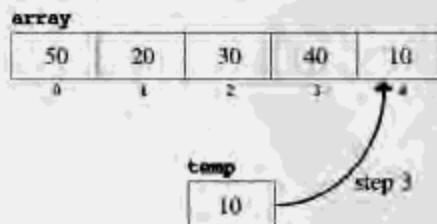


(2) 将 array[4]里的值复制到 array[0]，保持下面的配置。



因为前面已经将 array[0]原来的值存储在 temp 里，所以信息没有丢失。

(3) 将 temp 中的值指派给 array[4]，如下所示。



这 3 步策略是如下 swapElements 实现方式的基础：

```
private void swapElements(int[] array, int p1, int p2) {
    int temp = array[p1];
    array[p1] = array[p2];
    array[p2] = temp;
}
```

该方法是 ReverseArray 程序的最后一部分，完整的 ReverseArray 程序如图 11-10 所示。

```
import acm.program.*;

/**
 * This program reads in an array of five integers and then displays
 * those elements in reverse order.
 */
public class ReverseIntArray extends ConsoleProgram {

    public void run() {
        int[] array = new int[N_VALUES];
        println("This program reverses an integer array.");
        readArray(array);
        reverseArray(array);
        printArray(array);
    }

    /* Reads the data into the array */
    private void readArray(int[] array) {
        for (int i = 0; i < array.length; i++) {
            array[i] = readInt(" ? ");
        }
    }

    /* Prints the data from the array, one element per line */
    private void printArray(int[] array) {
        for (int i = 0; i < array.length; i++) {
            println(array[i]);
        }
    }

    /* Reverses the data in the array */
    private void reverseArray(int[] array) {
        for (int i = 0; i < array.length / 2; i++) {
            swapElements(array, i, array.length - i - 1);
        }
    }

    /* Exchanges two elements in an array */
    private void swapElements(int[] array, int p1, int p2) {
        int temp = array[p1];
        array[p1] = array[p2];
        array[p2] = temp;
    }

    /* Private constants */
    private static final int N_VALUES = 5;
}
```

图 11-10 倒转整数数组的程序

11.4 使用数组制作表格

程序的数据结构通常反映了数据在应用程序的现实域中的组织。如果要编写程序来解决包含一组值的问题，使用数组来表示这一组值就很直观。例如，在图 11-1 的 GymnasticsJudge 程序中，问题包含一组分数——5 名裁判每人打出一个分数。因为单个分数在应用程序的概念域里形成了一列，所以使用数组来表示程序里的数据就没有什么奇怪的。数组元素直接对应于列

里的单个数据项。因此, scores[0]对应于#0 裁判的分数, scores[1]对应于#1 裁判的分数, 依次类推。

- 一般来说, 应用程序包含可以用

```
a0, a1, a2, a3, a4, ..., an-1
```

形式表示的数据时, 数组就是基本表示法的自然选择。对于编程人员而言, 将数组元素的索引作为下标引用也很常见, 这也说明数组用来保存数据这一点。在数学中, 这些数据通常使用下标。

然而, 应用程序域中的数据与程序中的数据之间的关系采用不同形式时, 数组的用法就很重要。对于有些应用程序而言, 不是将数据存储在数组的连续元素里, 而是使用数据来生成数组索引, 这样更有意义。这些索引可以选择数组中的元素, 该数组记录了数据的某些统计属性。

要理解这种方法的运行原理, 以及它与数组的传统用法有何不同, 需要考察具体示例。假设要编写程序从用户读入文本行, 并记录 26 个字母出现的频率。当用户输入空行表示输入结束时, 程序应该显示一张表, 这张表应该显示输入数据中每个字母出现的次数。

为了生成这种字母频率表, 程序必须逐个字符搜索文本的每一行。每出现一次字母, 程序必须更新运行的计数, 该计数记录了到目前为止该字母在输入中出现的频率。问题有吸引力的地方是设计保留 26 个字母计数所需的数据结构。

定义 26 个单独变量——nA, nB, nC, 这样一直到 nZ——然后使用 switch 语句来检查全部 26 个字母(如下所示), 这样不使用数组也可以解决这个问题。

```
switch (Character.toUpperCase(ch)) {
    case 'A': nA++; break;
    case 'B': nB++; break;
    case 'C': nC++; break;
    ...
    case 'Z': nZ++; break;
}
```

这个过程导致程序很长, 重复太多。更好的方法是将 26 个单独变量结合到数组, 然后使用字符代码来选择数组里合适的元素。每个元素都包括表示字母当前计数的整数, 字母都对应于数组里的索引。如果调用数组 letterCounts, 可以写

```
int[] letterCounts = new int[26];
```

来声明它。该声明为有 26 个元素的整数数组分配空间, 如下所示。



每次字母在输入中出现, 都要递增 letterCounts 中相应的元素。寻找需要递增的元素就是使用第 8 章介绍的字符算法, 将字符转换为范围为 0~25 的整数。CountLetterFrequencies 程序的代码如图 11-11 所示。

```
import acm.program.*;

/**
 * This program creates a table of the letter frequencies in a
 * paragraph of input text terminated by a blank line.
 */
public class CountLetterFrequencies extends ConsoleProgram {

    public void run() {
        println("This program counts letter frequencies.");
        println("Enter a blank line to indicate the end of the text.");
        initFrequencyTable();
        while (true) {
            String line = readLine();
            if (line.length() == 0) break;
            countLetterFrequencies(line);
        }
        printFrequencyTable();
    }

    /* Initializes the frequency table to contain zeros */
    private void initFrequencyTable() {
        frequencyTable = new int[26];
        for (int i = 0; i < 26; i++) {
            frequencyTable[i] = 0;
        }
    }

    /* Counts the letter frequencies in a line of text */
    private void countLetterFrequencies(String line) {
        for (int i = 0; i < line.length(); i++) {
            char ch = line.charAt(i);
            if (Character.isLetter(ch)) {
                int index = Character.toUpperCase(ch) - 'A';
                frequencyTable[index]++;
            }
        }
    }

    /* Displays the frequency table */
    private void printFrequencyTable() {
        for (char ch = 'A'; ch <= 'Z'; ch++) {
            int index = ch - 'A';
            println(ch + ":" + frequencyTable[index]);
        }
    }

    /* Private instance variables */
    private int[] frequencyTable;
}
```

图 11-11 生成字母频率表的程序

11.5 数组初始化

和其他变量一样，数组变量也可以声明为局部变量或实例变量。在这两种情况下，也可以用非常简便的语法给数组指定一组初始值。这种语法中，指定初始值的等号后面紧接着用花括

号包围起来的每个元素的初始值。例如，声明

```
int[] digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

引入了名为 digits 的局部变量，其中的 10 个元素都被初始化为它们自己的索引数。

第二个示例是，假设要写需要数组的程序，该数组包含所有人口超过 1 000 000 的美国城市的名称，数据来源于 2000 年人口普查。可以使用下面的代码将该数组声明为命名常量：

```
private static final String[] US_CITIES_OVER_ONE_MILLION = {
    "New York",
    "Los Angeles",
    "Chicago",
    "Houston",
    "Philadelphia",
    "Phoenix",
    "San Diego",
    "San Antonio",
    "Dallas",
    ...
}
```

注意，US_CITIES_OVER_ONE_MILLION 数组最后一个初始表达式后面是逗号。逗号是可选的，但通常最好包括它。这样做可以在末尾添加新的城市，而不必改变现有项。例如，如果大城市增长率继续按现在的轨迹发展，到 2010 年，加利福尼亚州的 San Jose 将会加入此行列。要将它添加到列的结尾，只需要将行

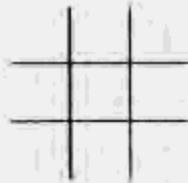
```
"San Jose",
```

添加到 initializer 列就行了，不需要重新在前面的条目后面添加逗号。

11.6 多维数组

Java 中，数组元素可以是任何类型。特别是数组的元素本身可以是数组。数组的数组称为多维数组。多维数组最常见的形式是二维数组，它通常用于表示数据，这些数据中的单个条目形成了分为行和列的矩形结构。这种类型的二维结构称为矩阵。三维或更多维数组在 Java 中也是合法的，但很少见。

作为二维数组的示例，可以假设要表示 tic-tac-toe 游戏作为程序的一部分。用户很可能知道，tic-tac-toe 游戏在由 3 行和 3 列组成的棋盘上进行，如下所示。



游戏者轮流在空正方形中放置字母 X 和 O，希望在水平方向、垂直方向和对角上连成 3 个相同的符号。

要表示 tic-tac-toe 棋盘，最明显的策略是使用 3 行 3 列的二维数组。虽然可以定义枚举类

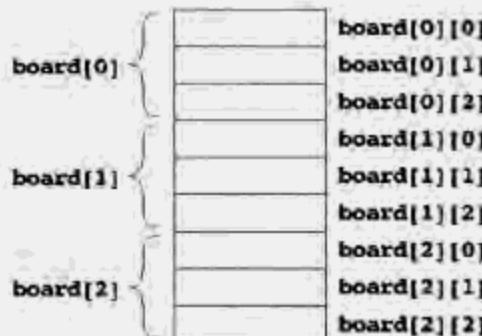
型来表示每个方块可能的内容——空、X和O，但这里使用char作为元素类型，使用字符'X'和'O'表示每个方块的合法状态，这样可能更简单。tic-tac-toe棋盘的声明如下：

```
char[][] board = new char[3][3];
```

给定声明之后，提供两个单独的索引——一个指定行数，另一个指定列数——可以引用表示棋盘上单个方块的字符。在这种表示法中，每个数变化的范围是0~2，棋盘上各位置的名称如下：

board[0][0]	board[0][1]	board[0][2]
board[1][0]	board[1][1]	board[1][2]
board[2][0]	board[2][1]	board[2][2]

在内部，Java将变量board表示为3个元素的数组，每个元素又都是有3个字符的数组。分配给board的内存由9个字符组成，其分布如下：



在board数组的二维图中，假定第一个索引表示行数。然而，这种选择很武断，因为矩阵的二维几何图形完全是概念上的；在内存中，这些值形成一维的一列。如果要让第一个索引表示列，第二个索引表示行，唯一需要改变的方法是依赖概念几何的那些方法，例如显示棋盘当前状态的方法。然而，按照内部排列，如果看看元素在内存里如何排列，就会发现第一个索引值总是没有第二个索引值变化快。因此在内存里，board[0]的所有元素都出现在board[1]的所有元素之前。

11.6.1 将多维数组传递给方法

多维数组在方法间的传递和单维数组在方法间的传递一样。方法头里的参数声明与变量的原始声明一样，要包含索引信息。例如，下面的方法显示board数组的当前状态：

```
private void displayBoard(char[][] board) {
    for (int row = 0; row < 3; row++) {
        if (row > 0) println(" + + ");
        println(" | |");
        print(" - -");
    }
}
```

```

        for (int col = 0; col < 2; col++) {
            if (col > 0) print(" | ");
            print(board[row][col]);
        }
        println();
        println(" ---");
    }
}

```

DisplayBoard 中的许多代码用来格式化输出，以便棋盘以一种易于阅读的形式出现。

11.6.2 初始化多维数组

可以对多维数组进行静态初始化，就像对单维数组一样。为了强调整体结构，用来初始化每个内部数组的值通常包含在附加的花括号里。例如，声明

```

static double identityMatrix[3][3] = {
    { 1.0, 0.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 1.0 }
};

```

声明了一个 3×3 的浮点数矩阵，并将它初始化为包含下面的值：

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

这个特殊的矩阵在数学应用程序中经常出现，称为单位矩阵。

11.7 图像处理

在现代计算中，二维数组最重要的应用程序之一出现在计算机图形领域。从第 2 章的介绍可以知道，图像由单个像素构成：图 2-16 提供了屏幕的放大图，显示了像素创建整体图像的方法。它与本章的连接点是安排好的像素，让它们形成二维数组。

11.7.1 图像的表示方式

在 Java 中，图像是矩形数组，其中图像整体是一系列行，每一行是一系列单个像素值。数组中每个元素的值表示出现在屏幕上对应像素位置的颜色。从第 9 章可以知道，Java 中可以通过确定每个原始颜色的强度来指定颜色。每个强度的范围是 0~255，因此它占用 8 位字节。颜色存储在一个 32 位整数里，它包含红色、绿色和蓝色强度值，以及颜色透明度的测量标准，它用希腊字母阿尔法(α)表示。对于大多数图像中使用的不透明色而言， α 的值总是 255，用二进制表示是 11111111，用十六进制表示是 FF。

例如，下面显示了这 4 个字节，它们形成颜色 PINK，它由 Java 使用 255、175 和 175 作为红、绿和蓝色组件定义。将这些值转换为二进制形式。

<i>alpha</i>	<i>red</i>	<i>green</i>	<i>blue</i>
11111111	11111111	10101111	10101111

Java 将有关颜色的所有信息装进一个 32 位整数中，这说明可以将图像存储为 int 类型的二维数组。在外部，数组的每个元素都是图像的整行。为了与 Java 的坐标系统保持一致，图像的行由顶部开始从 0 编号。图像上从左到右，每一行都是表示像素值的 int 类型的数组。

11.7.2 使用 GImage 类操作图像

acm.graphics 程序包中的 GImage 类导出一些方法，它们可以方便地进行基本图像处理。给定 GImage 对象，通过调用 getPixelArray，可以获得像素值的二维数组。因此，如果变量 image 包含 GImage，可以调用

```
int[][] pixelArray = image.getPixelArray();
```

来检索它的像素数组。

图像的高度等于像素数组的行数。宽度是任意一行中元素的数量，每一行的长度都与矩形图像的长度相等。因此，可以初始化变量来保存像素数组的高度和宽度，代码如下：

```
int height = pixelArray.length; int width = pixelArray[0].length;
```

GImage 类也包含一个构造函数，用于从二维数组中创建一个新的 GImage 对象。

GImage 类的这些新功能可以让用户编写操作图像的程序，方法与 Adobe Photoshop™ 这样的商业系统一样。一般策略由下面 3 步组成：

- (1) 使用 getPixelArray 来获得像素值的数组。
- (2) 实现操作数组里的值所需的转换。
- (3) 调用 GImage 构造函数，从修改后的图像中创建一个新对象。

下面的方法定义使用这种模式从垂直方向翻转图像：

```
private GImage flipVertical(GImage image) {
    int[][] array = image.getPixelArray();
    int height = array.length; for (int p1 = 0; p1 < height / 2; p1++) {
        int p2 = height - p1 - 1;
        int[] temp = array[p1];
        array[p1] = array[p2];
        array[p2] = temp;
    }
    return new GImage(array);
}
```

本示例中，实际上不需要考察单个像素。要做的就是使用与图 11-10 所示的 reverseArray 方法中相同的策略，倒转行的顺序。

flipVertical 方法非常有利于更好地创建第 9 章练习 16 的解决方案。在那个练习中，任务是写程序，模仿使用透镜创建图像。现在不需要两个图像文件——一个是原始形式，一个是倒转后的形式，可以只使用原始对象的图像文件，然后调用 flipVertical 来创建倒转后的图像。

11.7.3 位操作

然而，大多数图像操作都不像 `flipVertical` 那样简单。许多情况下，必须分解表示单个像素的整数，以便获得它们的红色、绿色和蓝色组件。要这样做，需要 Java 提供的低级别运算符，它们用来操作内存中字的位。这些运算符称为位运算符，如图 11-12 所示。它们采用任何标量类型的值，将它们解释为一系列位，这些位对应它们在硬件层次的基本表示法。

<code>x & y</code>	逻辑与。当 x 和 y 都有 1 位时，结果有 1 位
<code>x y</code>	逻辑或。当 x 或者 y 有 1 位时，结果有 1 位
<code>x ^ y</code>	异或。x 和 y 的位不相同时，结果有 1 位
<code>-x</code>	逻辑非。x 有 0 位，结果有 1 位；反之亦然
<code>x << n</code>	左移位。将 x 中的位向左移动 n 位，右边用 0 填充
<code>x >> n</code>	右移位(算术)。向右移动 n 位，保持第一个位不变
<code>x >>> n</code>	右移位(逻辑)。向右移动 n 位，左边用 0 填充

图 11-12 Java 中的位运算符

要理解位运算符的行为，有必要考察一些专门示例。假设已经声明变量 x 和 y，并将它们初始化如下：

```
int x = 0xACCEDED;
int y = 0xDEFACEDE;
```

作为位模式，其中的初始值则完全没有意义，仅仅是用字母 A~F 能写的最长单词中的两个。如果将十六进制数字转换为基本的位模式，就会发现这些变量在机器内部如下所示。

x	00001010	11001100	11101101	11101101
y	00001101	11101111	10101100	11101101

&、|、和[^] 运算符将图 11-11 中指定的逻辑运算应用于其操作数的每个位。例如，& 运算符只有在两个操作数都有 1 位时，产生的结果才有 1 位。例如，如果执行表达式 `x & y`，Java 计算的结果如下所示。

x	00001010	11001100	11101101	11101101
y	00001101	11101111	10101100	11101101
x & y	00001000	11001100	10101100	11101101

| 运算符在两个操作数中一个或两个都有 1 位时，它的结果才包含 1 位，除了这一点之外，其他方面与& 运算符相同，计算如下所示。

x	00001010	11001100	11101101	11101101
y	00001101	11101111	10101100	11101101
x y	00001111	11101111	11101101	11101101

只有当两个操作数中的位不同时，[^] 运算符产生的结果才包含 1 位，如下所示。

x	00001010	11001100	11101101	11101101
y	00001101	11101111	10101100	11101101
x ^ y	00000111	00100011	01000001	00000000

`-` 运算符只使用一个操作数，然后倒转它每个位的状态。因此，如果将`-`运算符应用于`x`中的位模式，结果如下所示。

x	00001010	11001100	11101101	11101101
-x	11110101	00110011	00010010	00010010

在编程过程中，应用`-`运算符称为取其操作数的补数。

`<<` 运算符的形式如下：

```
value << number of bits
```

该运算符将指定值中的位向左移动右边指定的位数。因此，表达式`x << 1`产生一个新的值，其中`x`值中的每一位都向左移动一个位置，如下所示。

x	00001010	11001100	11101101	11101101
x << 1	00010101	10011001	11011011	11011010

从右端移进来的位总是 0。

向右移位更加复杂。在 Java 中，向右移位的运算符有两种形式。`>>>` 运算符实现逻辑移位，其中出现在两端的新位总是 0。`>>` 实现计算机科学家所谓的算术移位，其中字最左端的位不会改变。这个位称为高次位，同时也称为符号位，因为它标记负数。

右移位的两种形式之间存在的差别不能用`x`的当前值来说明，因为它的高次位是 0。不管使用哪个右移位运算符，从 0 开始的移位模式的结果都一样。语句

```
x = Color.PINK.getRGB();
```

给粉红像素的位模式`x`指派一个新值，从本章前面可以知道，它的二进制表示法如下所示。

11111111	11111111	10101111	10101111
----------	----------	----------	----------

给`x`指派这个新值有利于说明`>>` 和`>>>` 运算符之间的不同，如下所示。

x	11111111	11111111	10101111	10101111
x >> 8	11111111	11111111	11111111	10101111
x >>> 8	00000000	11111111	11111111	10101111

两种运算符都将原始字中的位向右移动 8 个位置，效果就是将它们移动到下一个字节位置。`>>` 运算符用 1 位填补这个缺口，因为原始字中高次位是 1。`>>>` 运算符用 0 位填充缺口，它一直都这样做。

11.7.4 使用位操作分解像素组件

11.7.3 小节的示例看起来很晦涩难懂，因为它们和实际应用程序没有直接关系。介绍 Java 的位运算符不是要引起混淆，而是为了提供操作图像所需的工具。如果要使用颜色的单个组件，就要能够分解存储在整数字里像素值的组件。事实证明，位运算符正是所需的工具。

在 11.7.3 小节结尾，变量 `x` 被赋给颜色 `PINK` 的像素值，它的二进制形式如下所示。

11111111 11111111 10101111 10101111

那么，又该如何确定红色、绿色和蓝色组件的值呢？

分解蓝色组件最简单。要做的就是计算表达式 $x \& 0xFF$ 的值，如下所示。

x	11111111	11111111	10101111	10101111
0xFF	00000000	00000000	00000000	11111111
	00000000	00000000	00000000	10101111

只有两个操作数都有 1 位时，结果才包含 1 位。然而，另一种考察表达式的方法是，在结果中可能是 1 的位都是在常量 0xFF 中是 1 的位。对于这些位，结果由 x 的值确定。所以结果是，0xFF 中的 1 位从结果中所需的 x 中选择位。在计算机科学中，这种运算称为遮蔽。

可以使用遮蔽来分解红色和绿色组件，但必须将结果向右移位到字的结尾。例如，可以将 x 的值移动 16 位，然后使用 0xFF 作为掩码，这样来确定红色组件。这种操作如下所示。

x >> 16	1111111111111111	1111111111111111	1111111111111111	1111111111111111
0xFF	0000000000000000	0000000000000000	0000000000000000	1111111111111111
	0000000000000000	0000000000000000	0000000000000000	1111111111111111

使用下面的声明，可以计算存储在变量 x 里的像素值的 4 个组件：

```
int alpha = (x >> 24) & 0xFF;
int red = (x >> 16) & 0xFF;
int green = (x >> 8) & 0xFF;
int blue = x & 0xFF;
```

要向另一个方向——从单个组件向整个像素值——转换时，可以使用 | 运算符。在通常情况下，透明度值是 0xFF，可以从变量 red、green 和 blue 中创建完整像素值，代码如下：

```
int pixel = (0xFFE << 24) | (red << 16) | (green << 8) | blue;
```

11.7.5 创建灰度图像

为了说明操作单个像素值这一过程，一个很重要的应用程序是将图像从彩色转换为灰度格式。灰度是一种格式，它里面的所有像素要么是黑色，要么是白色，或者是灰色的中间色。例如，本书中的图片都是彩色图像，但在打印到页面之前，需要将它们转换为灰度格式。这种转换在出版社由软件自动实现，就像黑白打印机实现彩色图像打印一样。

在很大程度上, `createGrayscaleImage` 方法的实现方式符合一般图像操作程序使用的标准模式。首先可以从原始图像中检索像素数组, 然后修改单个像素值。最后, 从修改后的像素数组中创建一个新图像。本示例的目标是用新的近似于该颜色透明亮度的灰色阴影取代每个像素。在计算机图形中, 透明亮度称为发光度。如果用一个函数来计算给定红、绿、蓝组件像素的发光度, 可以如下编写 `createGrayscaleImage` 方法的余下部分:

```
private GImage createGrayscaleImage(GImage image) {
    int[][] array = image.getPixelArray();
    int height = array.length;
    int width = array[0].length;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int pixel = array[i][j];
            int red = (pixel >> 16) & 0xFF;
            int green = (pixel >> 8) & 0xFF;
            int blue = pixel & 0xFF;
            int xx = computeLuminosity(red, green, blue);
            pixel = (0xFF << 24) | (xx << 16) | (xx << 8) | xx;
            array[i][j] = pixel;
        }
    }
    return new GImage(array);
}
```

虽然通过平均像素的红、绿、蓝组件可以计算发光度, 但实际上可以比这做得更好。灰度转换的目标是产生灰色阴影, 该阴影近似于每个像素对于眼睛的亮度。事实证明, 发光度并不是平均地取决于颜色组件, 它更多地由像素中绿色的多少来控制, 而不是由红色或蓝色的多少来控制。毕竟, 红色和蓝色让图像显得更暗淡, 而绿色让图像变得更亮。美国负责电视信号的标准委员会采用的发光度公式是:

$$\text{luminosity} = 0.299 \text{ red} + 0.587 \text{ green} + 0.114 \text{ blue}$$

Java 中使用下面的函数很容易编码这种计算:

```
private int computeLuminosity(int r, int g, int b) {
    return GMath.round(0.299 * r + 0.587 * g + 0.114 * b);
```

11.7.6 通过平均使图像变得平滑

数字图像操作可以充分提高图像质量, 特别是原始图像的分辨率很低时。许多来自太空探测器的图像特意使用低分辨率, 以便将需要传回地球的数据最小化。例如, 太空探测器不是发送每像素 32 位的全色图像, 而是选择发送 4 位灰度图像, 这样就将传送时间缩短了 7/8。

最近访问土星的 Cassini 探测器就是这么做的。图 11-13 上图所示的图像是 2005 年 1 月 23 日, Cassini 在距离土星大约 280 万 km 的位置用广角照相机拍摄的。因为图像使用的是 16 灰度标准, 所以这幅图像有斑点, 那是沿行星移动时看到的大致等高线。数字过滤可以删除这些瑕疵, 如图 11-13 下图所示, 它里面的大致等高线消失了。

虽然生成下面这幅图像所使用的平滑技术超出了本书的范围, 但可以编写简单的数字过滤

器来了解这些技术。过滤器用每个像素上、下、左、右像素的发光度来平均自己的发光度。生成图像的毛边比较少，因为用不连接像素的值平滑了该像素值。

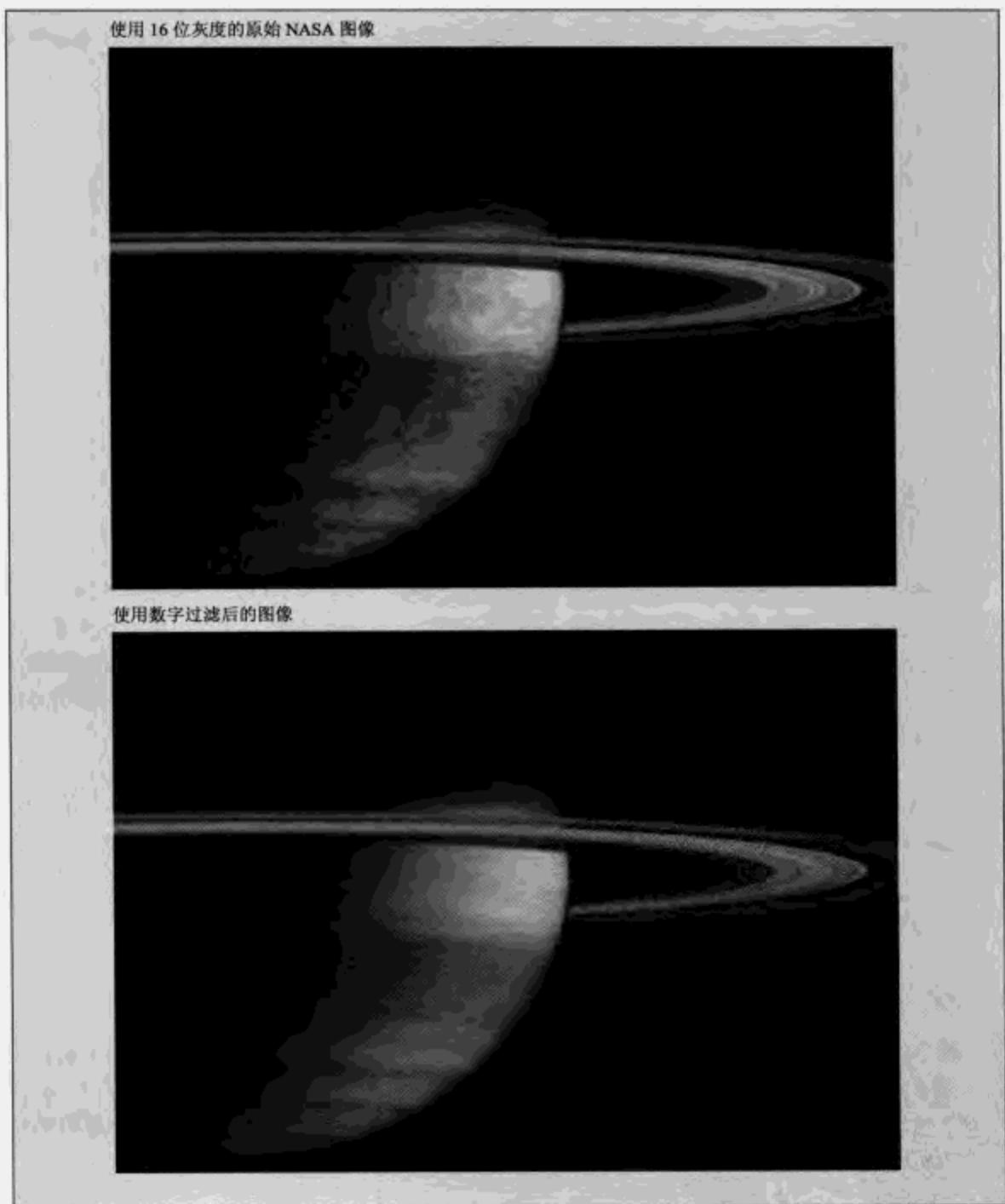


图 11-13 Cassini 太空探测器拍摄的土星图像

applyAveragingFilter 方法的代码如图 11-14 所示。代码存在的唯一问题是必须区别对待图像的边界，避免选择像素数组范围之外的元素。

```
/*
 * Creates a new image by applying an averaging filter to the original.
 * Each pixel in the original image is replaced by a grayscale pixel with the
 * average luminosity of the current pixel and its four immediate neighbors.
 */
private GImage applyAveragingFilter(GImage image) {
    int[][] array = image.getPixelArray();
    int height = array.length;
    int width = array[0].length;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int xx = averageNeighborLuminosity(array, i, j);
            array[i][j] = (0xFF << 24) | (xx << 16) | (xx << 8) | xx;
        }
    }
    return new GImage(array);
}

/*
 * Computes the average luminosity of the pixel at array[i][j] and its four
 * immediate neighbors (up, down, left, and right).
 */
private int averageNeighborLuminosity(int[][] array, int i, int j) {
    int sum = getLuminosity(array, i, j);
    int count = 1;
    if (i > 0) {
        sum += getLuminosity(array, i - 1, j);
        count++;
    }
    if (i < array.length - 1) {
        sum += getLuminosity(array, i + 1, j);
        count++;
    }
    if (j > 0) {
        sum += getLuminosity(array, i, j - 1);
        count++;
    }
    if (j < array[0].length - 1) {
        sum += getLuminosity(array, i, j + 1);
        count++;
    }
    return GMath.round((double) sum / count);
}

/* Determines the luminosity of the pixel at array[i][j] */
private int getLuminosity(int[][] array, int i, int j) {
    int pixel = array[i][j];
    int red = (pixel >> 16) & 0xFF;
    int green = (pixel >> 8) & 0xFF;
    int blue = pixel & 0xFF;
    return GMath.round(0.299 * red + 0.587 * green + 0.114 * blue);
}
```

图 11-14 applyAveragingFilter 方法的代码

11.7.7 隐藏复杂性

前面几小节已经介绍了位操作的许多细节——比想象的还要多。现在，需要找到一种方法来隐藏这些细节，让用户不必再考察它们。常用方法是定义将这些细节封装到其实现方式中的方法，同时遵守第 5 章介绍的信息隐藏原则。定义隐藏位操作细节的方法似乎是一个好的开端。

设计方法时要考虑客户的利益，这一点很重要。在图像操作程序中，客户对代码最低级别的移位和遮蔽的细节不感兴趣。客户真正想做的是从像素值中获得红、绿、蓝组件，然后将相同的值放在一起形成像素。如果这些确实是客户想要的操作，可以考虑创建下面的一般方法来隐藏位操作涉及的复杂性：

```
public static int getRed(int pixel) {  
    return (pixel >> 16) & 0xFF;  
}  
public static int getGreen(int pixel) {  
    return (pixel >> 8) & 0xFF;  
}  
public static int getBlue(int pixel) {  
    return pixel & 0xFF;  
}  
public static int createRGBPixel(int r, int g, int b) {  
    return (0xFF << 24) | (r << 16) | (g << 8) | b;  
}
```

这些方法声明为 `public`，因为它们可以满足外部客户及编写代码的人的需要。毕竟，为了写这些实现方式，这些方法的作者必须知道一切。而使用这些方法的客户根本不必知道这些细节。方法声明为 `static`，因为它们不依赖它们类中的任何实例变量。所有客户都可以调用这些方法，而不必创建类(这些方法定义于该类)的对象。

这些定义实际上是 `GImage` 类的一部分，这说明写像

```
int red = GImage.getRed(pixel);
```

这样的调用就可以使用它们，而不必写像

```
int red = (pixel >> 16) & 0xFF;
```

这种难以理解的调用。介绍这种实现方式的目的是让用户了解幕后发生的情况。

11.8 ArrayList 类

虽然数组是一个重要的编程概念，但面向对象语言通常包含库类。库类的使用减少了对数组的需求。Java 也不例外。`java.util` 程序包包含许多类，它们一起组成了 Java 集合架构(Java Collections Framework)，这将在第 13 章介绍。本节介绍 `ArrayList` 类，一方面因为它可以预先了解将来的吸引力，另一方面因为它提供了有用的工具。`ArrayList` 类是框架中广泛使用的类之一。

`ArrayList` 类提供对数组传统操作的等价操作，此外还包含几种新的操作，它们让 `ArrayList` 类更易于使用。而且 `ArrayList` 类允许将新元素添加到列的结尾，甚至是将它们插入两个现有元素之间。相反，如果没有分配新的数组并将其旧数组中的所有元素复制到新数组，就不可能改变现有数组的大小。

`ArrayList` 类一些最常用的方法如图 11-15 所示。然而，方法头与前面看到的不同。有些方法在通常包含类型名称的位置使用了符号`<T>`。例如，图 11-15 中的第一种方法的方法头如下

所示：

```
public void add(<T> element)
```

从描述中可以知道，这种方法将新元素添加到 ArrayList 类的结尾，但方法头没有回答 element 参数必须使用什么类型这个问题。答案是参数的类型取决于 ArrayList 的类型，ArrayList 的类型在声明时就已经确定了。如果声明指定 ArrayList 类包含字符串值，那么<T>语法表示的元素类型就是 String；另一方面，如果声明指定 ArrayList 包含 GPoint 值，那么<T> 语法应该表示 GPoint。

boolean add(<T> element)	在 ArrayList 后面添加一个新元素；返回值总是 true
void add(int index, <T> element)	在 index 指定的位置前面将元素插入 ArrayList
<T> remove(int index)	删除指定位置的元素，返回它的值
boolean remove(<T> element)	删除指定元素的第一个实例；如果找到匹配的内容，值为 true
void clear()	从 ArrayList 中删除所有元素
int size()	返回 ArrayList 中的元素数量
<T> get(int index)	返回指定索引位置的对象
<T> set(int index, <T> value)	将指定索引位置的元素设置为新值，并返回原来的值
int indexOf(<T> value)	返回第一个指定值的索引，如果没有，就返回 -1
boolean contains(<T> value)	如果 ArrayList 包含指定值，返回 true
boolean isEmpty()	如果 ArrayList 不包含元素，返回 true

图 11-15 ArrayList 类中的重要方法

定义指定对象类型的类，这种能力是 Java 5.0 中最重要的新特征。在这些类中，符号<T> 是类型参数。像 ArrayList 这样使用这种特征的类在 Java 中称为通用类(通常简称为 generics)。尽管这种类在计算机科学更通用的语境中称为模板或参数化类。坚持使用通用类，非常有助于编写易于阅读和维护的代码。另一方面，实现通用类的过程也充满了陷阱。本书只讨论通用类的用法，将创建它们的问题留到更高级的课程中讨论。

在 Java 5.0 中，要指定存储在 ArrayList 中值的类型，可以在尖括号内写类的名称，然后在 ArrayList 类出现的地方写它的名称。例如，声明

```
ArrayList<String> stringList = new ArrayList<String>();
```

可以构造一个空的字符串 ArrayList。这样，Java 编译器就知道存储在变量 stringList 里的 ArrayList 对象是受限制的，它只能包含字符串。而且，将图 11-15 中任何方法应用到 stringList 变量时，编译器都要确定指定为<T>类型的参数实际上是 String 类型。相反，如果使用声明

```
ArrayList<GPoint> vertices = new ArrayList<GPoint>();
```

创建 `ArrayList` 类, 编译器会检查 `ArrayList` 方法, 以确保这些参数都是 `GPoint` 类型。

用于 `ArrayList` 的类型参数必须是 Java 类, 不能是原始类型。例如, 下面的声明形式是不合法的:

```
ArrayList<int> intList = new ArrayList<int>();
```



正确的做法是为每个原始类型使用第 7 章定义的包装器类。将前面非法声明重写为

```
ArrayList<Integer> intList = new ArrayList<Integer>();
```

可以实现其结果。

Java 5.0 使用第 7 章介绍的装箱和拆箱技术, 在原始类型与对应的包装器类间自动来回转换, 这样充分加强了这种定义类型的值。Java 策略的效果是可以像

```
intList.add(17);
```

或

```
int firstValue = intList.get(0);
```

一样写语句, 而不用编写任何类型强制转换。存储在 `IntList` 里的值是 `Integer` 类型, 但这个值可以自由地与 `int` 类型相互转换。

从第 7 章的介绍中可以知道, Java 的装箱和拆箱策略也有某些缺陷。实际上, 限制使用方法的方式可以避免这些问题。这些方法返回是包装器类成员的对象, 例如前面语句中的 `get` 方法。如果总是将这个值赋给适当原始类型的变量, 而从不在其他表达式中使用它, 那么在危险出现之前就会强制进行转换。

如果没有 Java 5.0, 又该怎么办呢? 尽管不能再指定元素类型选项, `ArrayList` 类仍很有用。在 Java 5.0 之前的版本中, 定义 `ArrayList` 类来保存 `Object` 类型的值。因为 `Object` 是每个 Java 对象的根本超类, 所以可以将任何类型的对象都放入 `ArrayList` 类。在大多数情况下, 使用 `get` 方法从 `ArrayList` 中选择对象时, 都需要使用类型强制转换, 这几乎不会使代码复杂多少。同样, 必须自己编码原始类型的装箱和拆箱, 这有点棘手, 但从概念上说并不难。没有 Java 5.0 的通用性能时, 缺少的重要性能是让编译器检查正在使用每个 `ArrayList` 来存储类型匹配 `ArrayList` 声明的值。参数化类型有助于加强类型化的管理, 在强类型化中, 编译器可以检测所有类型方面的不匹配。

使用 `ArrayList` 代替数组, 至少从整体的角度来看, 代码的总体结构没变。声明数组对象、初始化其元素、提供索引数来选择元素的高级操作, 不会随着模型的改变而改变。

然而, 从简化的角度来看, 代码明显改变了, 特别是在语法方面。最重要的区别是 `ArrayList` 是 Java 类, 这意味着作为方法调用来调用它的每个操作。数组符号中的括号消失了。这样, 要检索特定索引位置的 `ArrayList` 的值时, 需要调用 `get` 方法而不是使用索引。同样, 不能使用赋值语句来改变 `ArrayList` 元素的值, 而使用数组时却可以; 必须调用 `set` 方法。其他改变大多数是名称的改变, 并不重要。特别是, 用来返回 `ArrayList` 长度的方法称为 `size` 而不是 `length`, 很容易忘记这一点。

在语法风格方面, 使用 `ArrayList` 导致的区别如图 11-16 和图 11-17 中的程序所示。这两个程序都更新了图 11-10 中的 `ReverseIntArray` 程序, 它们使用 `ArrayList` 类而不是数组来存储单个元素。图实际上是参数化类 `ArrayList<Integer>`, 图 11-16 说明了使用 Java 的一般机制来指定这一

点，从而实现程序的方法。图 11-17 说明在没有 Java 5.0 性能的情况下，编写相同程序的方法。

图 11-16 和图 11-17 中的程序使用终止条件表而不是预先定义固定的项数来表示输入结束，这样改进了原来的实现方式。这种改变是评论的，因为 ArrayList 可以动态发展。用户输入新值，readArrayList 的代码就将它添加到 ArrayList 的结尾。给新对象分配内存所需的所有内部操作都完全隐藏在 ArrayList 实现方式之内。

```
import acm.program.*;
import java.util.*;

/**
 * This program reads in a list of integers and then displays that list in
 * reverse order. This version uses an ArrayList<Integer> to hold the values.
 */
public class ReverseArrayList extends ConsoleProgram {
    public void run() {
        println("This program reverses the elements in an ArrayList.");
        println("Use " + SENTINEL + " to signal the end of the list.");
        ArrayList<Integer> list = readArrayList();
        reverseArrayList(list);
        printArrayList(list);
    }

    /* Reads the data into the list */
    private ArrayList<Integer> readArrayList() {
        ArrayList<Integer> list = new ArrayList<Integer>();
        while (true) {
            int value = readInt(" ? ");
            if (value == SENTINEL) break;
            list.add(value);
        }
        return list;
    }

    /* Prints the data from the list, one element per line */
    private void printArrayList(ArrayList list) {
        for (int i = 0; i < list.size(); i++) {
            int value = (Integer) list.get(i);
            println(value);
        }
    }

    /* Reverses the data in an ArrayList */
    private void reverseArrayList(ArrayList list) {
        for (int i = 0; i < list.size() / 2; i++) {
            swapElements(list, i, list.size() - i - 1);
        }
    }

    /* Exchanges two elements in an ArrayList */
    private void swapElements(ArrayList list, int p1, int p2) {
        int temp = list.get(p1);
        list.set(p1, list.get(p2));
        list.set(p2, temp);
    }

    /* Private constants */
    private static final int SENTINEL = 0;
}
```

图 11-16 倒转整数 ArrayList 的程序(Java 5.0 风格)

```

import acm.program.*;
import java.util.*;

/**
 * This program also reverses a list of integers. This version uses ArrayList as
 * it existed before Java 5.0 and must therefore do its own boxing and unboxing.
 */
public class ReverseArrayList extends ConsoleProgram {
    public void run() {
        println("This program reverses the elements in an ArrayList.");
        println("Use " + SENTINEL + " to signal the end of the list.");
        ArrayList list = readArrayList();
        reverseArrayList(list);
        printArrayList(list);
    }

    /* Reads the data into the list */
    private ArrayList readArrayList() {
        ArrayList list = new ArrayList();
        while (true) {
            int value = readInt(" ? ");
            if (value == SENTINEL) break;
            list.add(new Integer(value));
        }
        return list;
    }

    /* Prints the data from the list, one element per line */
    private void printArrayList(ArrayList list) {
        for (int i = 0; i < list.size(); i++) {
            Integer valueAsInteger = (Integer) list.get(i);
            println(valueAsInteger.intValue());
        }
    }

    /* Reverses the data in an ArrayList */
    private void reverseArrayList(ArrayList list) {
        for (int i = 0; i < list.size() / 2; i++) {
            swapElements(list, i, list.size() - i - 1);
        }
    }

    /* Exchanges two elements in an ArrayList */
    private void swapElements(ArrayList list, int p1, int p2) {
        Object temp = list.get(p1);
        list.set(p1, list.get(p2));
        list.set(p2, temp);
    }

    /* Private constants */
    private static final int SENTINEL = 0;
}

```

图 11-17 倒转整数 ArrayList 的程序(Java 5.0 之前)

11.9 小结

本章介绍了两种 Java 用来表示数据列的策略：语言级别工具数组和 java.util 程序包中的 ArrayList 类。这两种策略在许多方面很类似，但也有各自不同属性，这些属性让每种策略能够

适应不同的应用程序。数组内置在 Java 中，由语言的语法支持。ArrayList 类更强大，被更紧密地结合进 Java 类层次结构，尽管使用 ArrayList 的语法稍微有点麻烦。

本章介绍的重点是：

- 和大多数编程语言一样，Java 使用数组来表示有序和同类数据的集合。
- 数组里的单个值称为元素，由数字索引表示。Java 中，所有数组的索引数都从 0 开始。
- 数组中元素的个数称为数组的长度。Java 中，选择 length 字段可以确定数组的长度。
- 创建数组变量的过程分为两步：声明数组变量和为元素分配空间。在大多数应用程序中，将这些操作结合为单个声明行最简单的方法是使用例证形式

```
type[] name = new type [length];
```

- 它创建一个名为 *name* 的数组变量，它有 *length* 个元素，每个元素都是 *type* 的实例。
- 引用数组中单个元素的过程称为选择，它通过在数组名之后的方括号里写表达式表示。
- + 和 - 运算符可以写为前缀形式，即在操作数前面；也可以写为后缀形式，即在操作数后面。对于变量而言，两种形式的结果相同，变量是递增还是递减取决于使用的运算符。不同之处在于，这些运算符的后缀形式将变量的原始值返回给包围表达式的语境。
- Java 数组是对象，这说明作为参数给方法传递数组会将数组的引用复制到相应的形参中。因为这种引用使用相同的对象作为调用者，所以方法返回之后，会保留对方法内数组元素所做的所有改变。
- 使用初始器可以指定数组中的初始值，初始器由包含在花括号内的元素组成。
- 数组可以使用多个索引声明，称为多维数组。在 Java 中，多维数组通常作为数组的数组对待。第一个索引选择最外面数组的元素，第二个索引选择那个子数组里的元素，依次类推。
- 图像数据可以表示为像素值的二维数组，这些像素值将红、绿、蓝和颜色的透明度组件结合进一个 32 位整数字里。
- 使用 Java 的位运算符(&、|、^、~、<<、>> 和 >>>)，可以检索像素值的单个组件。
- java.util 中的 ArrayList 类实现所有基本的数组操作，也可以用一些特别有用的性能进行扩充，其中最显著的能力是动态扩充列的大小。
- 在 Java 5.0 中，在尖括号内写类的名称，如 ArrayList<String>，可以指定 ArrayList 中的元素类型。如果没有 Java 5.0，使用类型强制转换和包装器类可以达到相同效果。

11.10 复习题

- 数组的两个典型属性是什么？
- 定义下列术语：元素、索引、元素类型、数组长度和选择。
- 编写声明创建下列数组变量：
 - 由 100 个 double 类型值组成的 doubleArray 数组。
 - 由 16 个 boolean 类型值组成的 inUse 数组。
 - 由 50 个字符串组成的 lines 数组。
- 编写变量声明和所需的 for 循环，创建和初始化下面的整数数组：

squares										
0	1	4	9	16	25	36	49	64	81	100
0	1	2	3	4	5	6	7	8	9	10

5. 如何确定数组长度?

6. 本文介绍了两种方法来表示数组, 在这两种方法中, 约定的、现实中的索引值从 1 而不是 0 开始。请问是哪两种方法? 这两种方法之间有什么不同?

7.11.1A 小节有一个示例, 它使用下列代码行:

```
scores[i] = readDouble("Score for judge " + (i + 1) + ":" );
```

表达式 $i+1$ 外面的圆括号必不可少吗? 为什么?

8. ++ 运算符的前缀形式和后缀形式有何不同?

9. 判断题: 数组在内存中的表示方法与原始值在内存中的表示方法一样。

10. swapElements 方法中变量 temp 的作用什么?

11. 如何声明和初始化数组变量 powersOfTwo——它包含 2 的前 8 个幂(1, 2, 3, 8, 16,

32, 64 和 128)——让整个初始化过程出现于声明行中?

12. 什么是多维数组?

13. 画图说明像素的颜色组件在 32 位整数字中的排列方法。

14. 假设变量 p 和 q 初始化如下(其中像字母 O 的十六进制数字是数字 0):

```
int p = 0xC0FFEE;
int q = 0xD000DAD;
```

表达式 $p \& q$ 、 $p | q$ 、 $p ^ q$ 和 $\sim p$ 的值什么?

15. 在学习编程的班级里, 一位聪明的同学声称, 在不使用临时变量的情况下, 也可以交换整数变量 x 和 y 里的值。要做的就是实现如下赋值:

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

她说得对吗?

16. 运算符 $>>$ 和 $>>>$ 之间有何不同?

17. 写一条 Java 表达式, 使用位运算符来分解 Java 颜色 ORANGE 的绿色组件。

18. 使用 ArrayList 类而不使用 Java 数组有哪些优点?

19. 描述下列 ArrayList 方法的结果: size, add, set, get 和 remove。

20. 创建 ArrayList<boolean> 合法吗?

11.11 编程练习

1. 因为单独某位竞赛裁判可能有偏见, 所以在计算平均分之前会去掉最高分和最低分。编写程序, 读入 7 名裁判所打的分数, 计算去掉最高分和最低分之后剩余的 5 个分数的平均值。

2. 在统计学中, 数据值的集合通常称为分布。统计分析的主要目的是将全部数据压缩进表示分布属性的汇总统计表。最常见的统计方法是平均数, 即传统的平均值。分布的平均数通常用希腊字母 μ 表示。

写方法 `mean(array)`, 返回 `double` 类型数组的平均数。将该方法合并到图 11-1 的 `GymnasticsJudge` 程序中来测试它。

3. 另一种常见的统计方法是标准偏差, 它表示分布中的单个值与平均数存在多大不同。要计算元素为 x_1, x_2, \dots, x_n 的标准偏差, 需要实现以下几步:

(1) 像练习 2 中那样, 计算分布的平均数。

(2) 查看分布中的各数据项, 计算每个数据值与平均数之间差分的平方。将这些值添加到连续总数。

(3) 从步骤(2)中取出总数, 用它除以数据项数。

(4) 计算结果数的平方根, 它就是标准偏差。

在数学形式中, 标准偏差(σ)的公式如下:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (\mu - x_i)^2}{n}}$$

其中 μ 表示平均数。希腊字母 Σ 表示量的和, 这里这些量是平均数和每个数据值之间差分的平方(注意: 统计学家使用这个公式来计算完全数据分布的标准偏差; 要计算基于样本的标准偏差, 公式应该除以 $n - 1$)。

写方法 `stdev(array)`, 读入一个 `double` 类型的数组, 返回该数组包含的数据分布的标准偏差。

4. 魔方是整数二维数组, 其中行、列和对角相加都得到相同的值。最著名的魔方之一出现在 Albrecht Dürer 的雕版画 Melencolia I 中(如图 11-18 所示), 这幅雕版图右上角的铃铛下面有一个 4×4 的魔方。在 Dürer 的魔方中, 全部 4 行、4 列和对角相加都等于 34。



图 11-18 Albrecht Dürer 的雕版画 Melencolia I 中的魔方

而且, Dürer 排列的正方形, 底下一行中间两列显示的是 1514, 正是他创作该雕版画的年份。

一个更熟悉的示例是下面的 3×3 魔方, 其中每行、每列和对角相加都等于 15, 如图 11-19 所示:

图 11-19 3×3 魔方

实现方法 `isMagicSquare`, 来测试 $N \times N$ 数组是否包含魔方。

5. 最近几年, 一个新的称为 Sudoku 的逻辑难题在全球非常流行。在 Sudoku 中, 从 9×9 的数字栅格开始, 其中有些单元已经填充了 1~9 之间的数字, 难题的任务就是用 1 到 9 之间的数字填充每个空格, 让每个数字在每行、每列和每个较小的 3×3 正方形中只出现一次。每个 Sudoku 难题都是精心构造的, 所以只有一种解决方案。例如, 假设难题如图 11-20 左图所示, 唯一的解决方案如图 11-20 右图所示。

	2	4	5	8				
4	1	8			2			
6		7		3	9			
2		3		9	6			
	9	6	7	1				
1	7		5			3		
9	6		8			1		
2			9	5	6			
	8	3	6	9				

3	9	2	4	6	5	8	1	7
7	4	1	8	9	3	6	2	5
6	8	5	2	7	1	4	3	9
2	5	4	1	3	8	7	9	6
8	3	9	6	2	7	1	5	4
1	7	6	9	5	4	2	8	3
9	6	7	5	8	2	3	4	1
4	2	3	7	1	9	5	6	8
5	1	8	3	4	6	9	7	2

图 11-20 Sudoku 逻辑难题及其解决方案

虽然生成或解决 Sudoku 难题的算术策略超出了本书的范围, 但可以编写方法, 让它检查假设的解决方案是否符合 Sudoku 规则, 在行、列和 3×3 正方形中有没有重复的值。编写方法

```
private boolean checkSudokuSolution(int[][] grid)
```

实现这种检查, 如果栅格是正确的解决方案, 返回 `true`。

6. 在 B.C.E 第三世纪, 希腊天文学家 Eratosthenes 开发了一种算法, 用来寻找上限为 N 的所有素数。要应用这种算法, 首先要写出介于 2 和 N 之间的一列整数。例如, 如果 N 等于 20, 那么首先写下面的列:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

然后可以给列里的第一个数画上圆圈, 表示已经找到了一个素数。将一个数标记为素数时, 可以查看列剩下的数, 删除所有那个数的倍数, 因为这些倍数本身不可能是素数。这样, 执行算法的第一个周期后, 可以将数字 2 圈起来, 删除 2 的倍数, 如下所示:

(2) 3 ✕ 5 ✕ 7 ✕ 9 ✕ 11 ✕ 13 ✕ 15 ✕ 17 ✕ 19 ✕

将列里既没有删除也没有被圈起来的第一个数圈起来，然后删除它的倍数，重复这个过程，就可以完成该算法。本示例中，可以将 3 作为素数圈起来，删除列中所有 3 的倍数，结果如下所示。

(2) (3) ✕ 5 ✕ 7 ✕ ✕ ✕ 11 ✕ 13 ✕ ✕ ✕ 17 ✕ 19 ✕

最后，列里的数要么被圈起来了，要么被删除了，如下所示：

(2) (3) ✕ (5) ✕ (7) ✕ ✕ ✕ (11) ✕ (13) ✕ ✕ ✕ (17) ✕ (19) ✕

被圈起来的数是素数，删除的数是合数。这种算法称为 Eratosthenes 滤网。编写 ConsoleProgram 程序，它使用 Eratosthenes 滤网来生成 2~1000 之间的素数。

7. 统计学家寻找描述随着时间变化的量的行为时，如某一年的全球平均温度、月失业出生率、每日股市指数或者类似的数据值，最有用的表示格式是线图，在线图中，先在 x/y 格上划分数据值，然后用直线连接每对相邻的点。在这样的图形中，x 轴表示时间，y 轴表示指标的值。例如，图 11-21 所示的数据点表示 Google 公司自 2004 年 8 月最初公开发行股票以来的月平均股票价格。

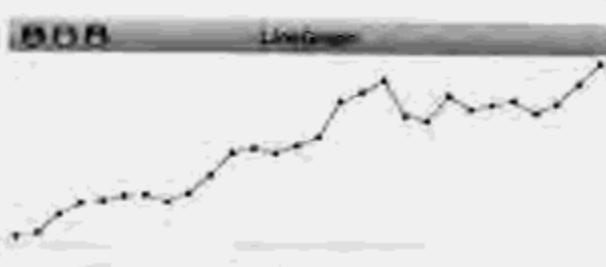


图 11-21 线图

编写名为 LineGraph 的 GraphicsProgram 程序，它生成图 11-21 所示的图形。程序应该包含 drawLineGraph 方法，它读入双精度数数组，并在画布上划分它。drawLineGraph 的实现方式应该调整 x 和 y 的刻度，除了给每个边界留页边之外，让数组中的数据点填满窗口内的空间。计算 x 轴的刻度很简单：每个数据点之间的距离等于可用的水平空间(除去页边之外)除以数据点的数量。标记图形 y 轴的刻度有点复杂。为了确保所有点在窗口中可见，必须确定数组中的最大值和最小值。最小值在底下，最大值在上面。其他每个值必须按比例衡量，以便让它出现在最小值和最大值之间的合适位置。

8. 1844 年 5 月，Samuel F. B. Morse 将消息“What hath God wrought!”用电报从华盛顿发送到巴尔的摩(Baltimore)，宣告电子通信时代的开始。为了使用单音的 presence or absence 来交流信息，Morse 设计了一种编码系统，该系统中的字母和其他符号都表示为长短音的编码序列，这些短音和长音通常称为点和长划。在摩斯(Morse)代码中，字母表的 26 个字母表示为下列代码：



在程序中，通过声明有 26 个元素的数组，然后将字符序列对应存储为相应的数组条目，可以存储这些代码。

编写 `ConsoleProgram` 程序，通过用户输入读入字符串，然后将字符串中的每个字母转换为其对应的摩斯代码，使用句点表示点，使用连字号表示长划。在输入中遇到空格时，调用 `println` 隔开输出的单词，忽略其他所有标点字符。程序应该能够产生如图 11-22 所示的结果：

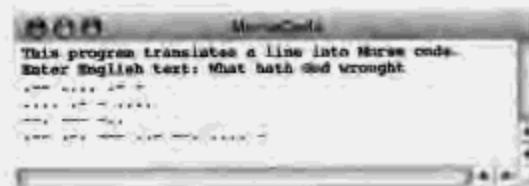


图 11-22 摩斯代码的运行示例(1)

19世纪电报的发明极大地促进了社会变革，就像20世纪90年代Internet的发展也极大地促进了社会变革一样。在1998年《The Victorian Internet》一书中，英国新闻记者Tom Standage描述了电报如何引发了全新的商业模式(包括新奇的犯罪机会)、在线传奇文学、对密码术兴趣的复苏以及对其发起人坚定不移的信任，新的技术将改变世界。

9. 重写练习 8 中的 MorseCode 程序，让它用图形显示其结果，使用 JTextField 从用户读入消息。图 11-23 显示了 1912 年 4 月 15 日 Titanic 在沉没时发送的部分消息。



图 11-23 摩斯(Morse)代码的运行示例(2)

消息中的 COD 是遇险信号的另一种形式，它最终被更常见的 SOS 完全取代了。

10. 编写交互的 GraphicsProgram，它在计算机上显示 Nim 游戏。在最简单的版本中，两名游戏者使用桌子上的一堆 11 枚硬币。游戏者轮流从堆中拿走一枚、两枚、3 枚硬币。取走最后一枚硬币者为输家。

程序应该首先显示 11 枚硬币，并让它们在窗口上居中，如图 11-24 所示。



图 11-24 Nim 游戏(1)

如果在这一行最后 3 枚硬币中的任意一枚上按下鼠标，该硬币和它右边的所有硬币都变成红色。因此，如果在倒数第 3 枚硬币上按下鼠标键，显示如图 11-25 所示。



图 11-25 Nim 游戏(2)

在窗口其他位置——甚至是它左边的硬币上——按鼠标键，都会被忽略。释放鼠标键时，红色硬币应该消失，计算机继续运行。计算机会将 1 个、2 个或 3 个硬币变成黄色，并且暂停 1s，然后消除它们。因此，选择了图 11-25 中的红色硬币之后，计算机会继续选择接下来 3 个硬币，如图 11-26 所示：



图 11-26 Nim 游戏(3)

问题是设计计算机游戏的人，让它尽可能好地玩这个游戏。例如，如果考察黄色硬币删除之后的位置，如果它总是按对键，就会总是计算机获胜。该如何推广整个游戏策略？

11. 图 11-12 中的位运算符似乎难以理解，主要因为它们不像传统的算术运算符那么常见。可视化这些运算符结果的方法之一是设计一种应用程序，可以在屏幕上解释这些运算符的行为。

编写名为 BitwiseOperatorDemo 的交互式程序，它显示 3 行，每行 8 个按钮，以及一个 JComboBox，它的元素是运算符 &、| 和 ^。最初状态如图 11-27 所示。

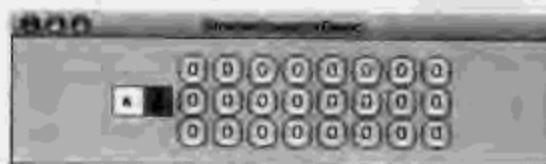


图 11-27 位运算符演示(1)

程序的基本思想是，将 JComboBox 里显示的运算符应用于上面两行按钮，在底下一行产生位模式。演示的最初状态显示将&运算符应用于 0 位产生 0 位，这是对的，但不是那么有

意思。

程序的交互应该是可以改变运算符和上面两行按钮的位，然后马上看到产生的结果。单击包含 0 的一个按钮，应该将那个按钮改变为 1，并相应更新底下一行的按钮。例如，如果单击每个输入行里右边的按钮(假设运算符是 &，单击一个位不会改变结果)，显示会变成如图 11-28 所示。



图 11-28 位运算符演示(2)

如果将 JComboBox 中的运算符改变为其他运算符，程序应该应用相应的位运算符。改变运算符和单个位，这种能力允许使用这 3 个运算符设置自己喜欢的任何配置。例如，下面的运行显示了在包含每个可能的位的输入模式下，^ 运算符的结果，如图 11-29 所示。



图 11-29 位运算符演示(3)

12. 编写方法 flipHorizontal，它与本章介绍的 flipVertical 方法相同，除了它是在水平方向上倒转图片之外。因此，如果有包含图 11-30 左图所示图像(Jan Vermeer 的 The Milkmaid, c. 1659)的 GImage，在图像上调用 flipVertical 可以返回新的 GImage，如图 11-30 右图所示。

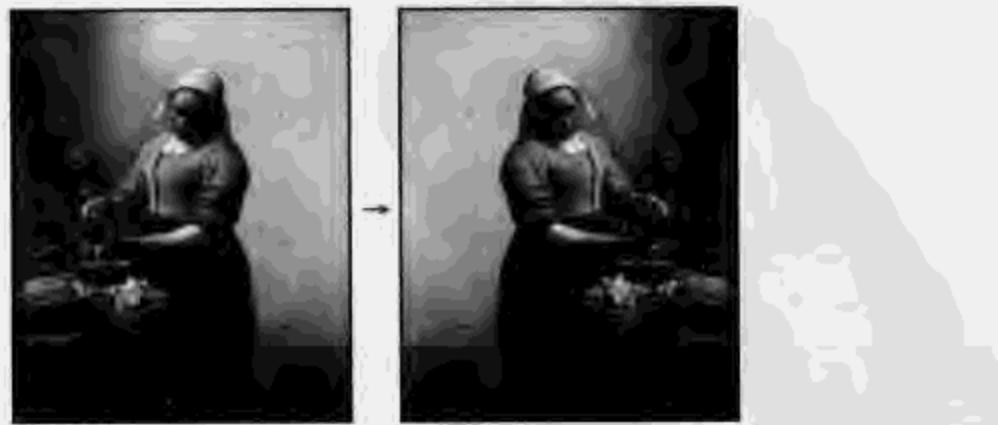


图 11-30 Jan Vermeer 的 The Milkmaid, c. 1659 图像

13. 编写方法 rotateLeft，读入 GImage，然后产生新的 GImage，其原始图像向左旋转了 90°。例如，如果最初是第 9 章练习 16 中使用的蜡烛，调用 rotateLeft 会将图像转换为新的图像，如

图 11-31 所示。

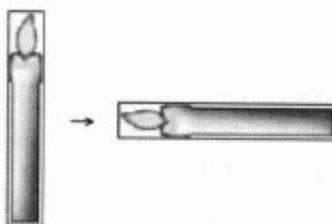


图 11-31 蜡烛图像翻转示例

14. 在 Dan Brown 的畅销影片《达·芬奇密码》中，英国艺术历史学家 Sir Leigh Teabing(由 Sir Ian McKellen 饰演)使用计算机重新排列达芬奇的名画《最后的晚餐》，来揭示传说的隐藏含义。虽然从图像中选择不规则区域超出了本书的范围，但 Teabing 实现的操作与可用图像操作工具中的矩形剪切操作相似。

编写 GraphicsProgram 程序，它在屏幕上显示图像(图像名可以指定为常量)，然后允许用户拖动鼠标从图像中选择区域。只要鼠标按钮没有弹起，程序就应该画剪切矩形的轮廓。用户释放鼠标之后，程序应该删除原始图像，用屏幕上矩形内部构造的新图像取代它。例如，假设用户释放鼠标前图像如图 11-32 所示。



图 11-32 剪切图像示例(1)

如果用户释放鼠标，程序应该从矩形内部提取像素，然后在窗口左上角显示较小的图像，如图 11-33 所示。

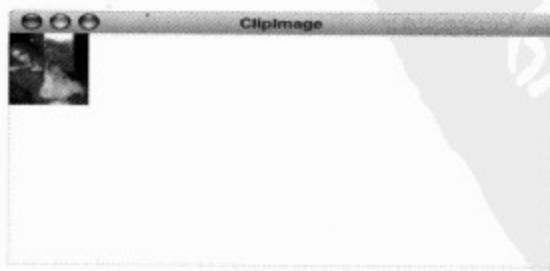


图 11-33 剪切图像示例(2)

15. 编写方法 makeColorNegative，让它创建 GImage 的底片像。在彩色负片中，新图像的

各颜色组件应该是它原来 0~255 标准值的相反值。因此，组件值 0 变成了 255，值 200 变成了 55。例如，如果有包含图 11-34 左图(文森特·梵高的《星夜》, 1889)的 GImage，调用 makeColorNegative 会返回如图 11-34 右图所示的 GImage。

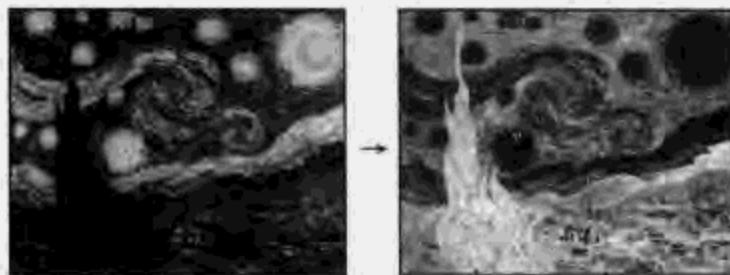


图 11-34 梵高的《星夜》(Starry Night)及其底片像



第 12 章

搜索与排序

"I weep for you," the Walrus said:

"I deeply sympathize."

With sobs and tears he sorted out

Those of the largest size.

—Lewis Carroll, Through the Looking Glass, 1872



C. A. R. Hoare

Charles Antony Richard (Tony) Hoare 爵士是牛津大学计算机科学的名誉退休教授，是英国剑桥 Microsoft 研究试验室的高级研究员。1956 年，Hoare 在牛津大学获得了哲学学位，之后他为新兴的计算机科学所深深吸引，于是选择这一方向攻读研究生。Hoare 在攻读研究生期间，开发了一种高效的称为 Quicksort 的排序算法，这种算法直到今天还在使用。20 世纪 60 年代，他率先为 Algol 60 创建了第一个商用编译器。Algol 60 是一种编程语言，它是后来很多语言(包括 Java)的重要模型。1980 年，Hoare 教授获得了 ACM 图灵奖。

第 11 章已经介绍了许多基本的数组操作，讨论了在各种应用程序中使用数组的方法。然而，第 11 章省略了两种重要的数组操作，这两种操作值得专门用一章来详细介绍。

- 搜索：找出数组中特定元素的过程。
- 排序：排列数组元素让它们以明确顺序存储的过程。

因为搜索和排序与数组紧密相关，所以本章在某种程序上是数组讨论的延续。然而，本章的另一个中心主题不仅与第 11 章有关，而且与第 5 章算法方法的讨论有关。因为搜索和排序有许多不同的策略——有效性也存在很大差异——这些操作提出了许多有趣的算法问题。

12.1 搜索

如本章前言所述，搜索问题就是寻找数组中的元素。最简单的策略——尽管不是最有效——存在于下面的忠告这中，这条忠告是 Lewis Carroll 所著的《Alice's Adventures in Wonderland》中，King of Hearts 对 White Rabbit 提出来的：

从头开始，一直向前，直到终点，再停下来。

将这条非正式语句转换为搜索算法并不难。要做的唯一修改是，算法如果找到了它搜索的元素，也应该停止。因此，可以将 Lewis Carroll 的搜索算法表达得更完整：

从头开始，找到了寻找的元素或到了终点就停下来。如果找到了元素，就报告它的位置；如果到了终点，说明元素不存在。

因为过程从头开始，以直线方式搜索数组元素，所以这种算法称为线性搜索。

12.1.1 在整数数组中搜索

将这种非正式的描述转换为 Java 方法，需要给方法规范添加一些细节。例如，可以假设要求编写方法 linearSearch，在整数数组中查找整数 key。这种方法的标题行如下：

```
private int linearSearch(int key, int[] array)
```

该方法的结果是数组元素键第一次出现的索引，如果它在数组中不存在，结果是 -1。linearSearch 的实现方式实际上不过是 Lewis Carroll 方法的转换：

```
private int linearSearch(int key, int[] array) {
    for (int i = 0; i < array.length; i++) {
        if (key == array[i]) return i;
    }
    return -1;
}
```

for 循环从头开始搜索，直到数组结尾为止。找到键，返回值；如果到了整个循环结尾，就说明没有找到。

12.1.2 搜索表

作为讨论不同搜索算法的序幕，本节介绍一种更复杂的搜索应用程序，以便讨论提出的问题。假设要在程序中表示图 12-1 所示的里程表。表中的单个记录项形成一个有 12 行 12 列的二维数组。矩阵中的每一项都是整数，它表示该行该列对应的城市之间的英里数。初始化二维数组所需的代码称为 mileageTable，如图 12-2 所示，它也包含初始化表中城市名称数组的代码。

	Atlanta	Boston	Chicago	Denver	Detroit	Houston	Los Angeles	Miami	New York	Philadelphia	San Francisco	Seattle
Atlanta	1108	708	1430	732	791	2191	663	854	748	2483	2625	
Boston	1108	0	994	1998	799	1830	3017	1520	222	315	3128	3016
Chicago	708	994	0	1021	279	1091	2048	1397	809	785	2173	2052
Denver	1430	1998	1021	0	1283	1034	1031	2107	1794	1739	1255	1341
Detroit	732	799	279	1283	0	1276	2288	1385	649	609	2399	2327
Houston	791	1830	1091	1034	1276	0	1541	1190	1610	1511	1911	2369
Los Angeles	2191	3017	2048	1031	2288	1541	0	2716	2794	2703	387	1134
Miami	663	1520	1397	2107	1385	1190	2716	0	1334	1230	3093	3303
New York	854	222	809	1794	649	1610	2794	1334	0	101	2930	2841
Philadelphia	748	315	785	1739	609	1511	2703	1230	101	0	2902	2816
San Francisco	2483	3128	2173	1255	2399	1911	387	3093	2930	2902	0	810
Seattle	2625	3016	2052	1341	2327	2369	1134	3303	2841	2816	810	0

Source: Rand McNally

图 12-1 美国城市里程表

```

/* Initializes the mileage table */
private int[][] mileageTable = {
    { 0, 1108, 708, 1430, 732, 791, 2191, 663, 854, 748, 2483, 2625},
    {1108, 0, 994, 1998, 799, 1830, 3017, 1520, 222, 315, 3128, 3016},
    { 708, 994, 0, 1021, 279, 1091, 2048, 1397, 809, 785, 2173, 2052},
    {1430, 1998, 1021, 0, 1283, 1034, 1031, 2107, 1794, 1739, 1255, 1341},
    { 732, 799, 279, 1283, 0, 1276, 2288, 1385, 649, 609, 2399, 2327},
    { 791, 1830, 1091, 1034, 1276, 0, 1541, 1190, 1610, 1511, 1911, 2369},
    {2191, 3017, 2048, 1031, 2288, 1541, 0, 2716, 2794, 2703, 387, 1134},
    { 663, 1520, 1397, 2107, 1385, 1190, 2716, 0, 1334, 1230, 3093, 3303},
    { 854, 222, 809, 1794, 649, 1610, 2794, 1334, 0, 101, 2930, 2841},
    { 748, 315, 785, 1739, 609, 1511, 2703, 1230, 101, 0, 2902, 2816},
    {2483, 3128, 2173, 1255, 2399, 1911, 387, 3093, 2930, 2902, 0, 810},
    {2625, 3016, 2052, 1341, 2327, 2369, 1134, 3303, 2841, 2816, 810, 0},
};

/* Initializes an array of city names corresponding to the entries in the table */
private String[] cityNames = {
    "Atlanta",
    "Boston",
    "Chicago",
    "Denver",
    "Detroit",
    "Houston",
    "Los Angeles",
    "Miami",
    "New York",
    "Philadelphia",
    "San Francisco",
    "Seattle",
};

```

图 12-2 初始化里程表及城市名的代码

既然有数据，下一个问题是如何写程序，让它读入两个城市名称，然后显示它们之间的距离，其运行结果如图 12-3 所示。

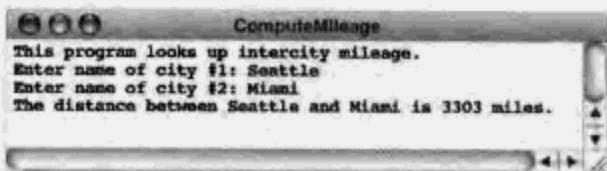


图 12-3 ComputeMileage 程序的输出

此程序的代码如图 12-4 所示。

```

import acm.program.*;

/**
 * This program uses a table of mileage data to calculate the distance
 * between cities in the United States.
 */
public class ComputeMileage extends ConsoleProgram {
    public void run() {
        println("This program looks up intercity mileage.");
        int city1 = getCity("Enter name of city #1: ");
        int city2 = getCity("Enter name of city #2: ");
        println("The distance between " + cityNames[city1]
            + " and " + cityNames[city2] + " is "
            + mileageTable[city1][city2] + " miles.");
    }

    /*
     * Prompts the user for a city name, reads in a string, and returns the
     * index corresponding to that city, if it exists. If the city name is
     * undefined, the user is given a chance to retry.
     */
    private int getCity(String prompt) {
        while (true) {
            String name = readLine(prompt);
            int index = linearSearch(name, cityNames);
            if (index != -1) return index;
            println("Unknown city name -- try again.");
        }
    }

    /*
     * Finds the first instance of the specified key in the array
     * and returns its index. If the key does not appear in the array,
     * linearSearch returns -1.
     */
    private int linearSearch(String key, String[] array) {
        for (int i = 0; i < array.length; i++) {
            if (key.equals(array[i])) return i;
        }
        return -1;
    }

    /* Include the definitions of mileageTable and cityNames from the text */
}

```

图 12-4 计算两个城市间距离的程序

`linearSearch` 中使用的线性搜索算法从数组开头开始，逐个搜索这一行的元素，直到找到匹配的元素或到达数组结尾。对于包含 12 个城市名的数组而言，寻找每个元素似乎不需要太多时间。但如果数组有上千个或数百万个元素，情况又怎样呢？在某些程序上，如果数组非常大，由于计算机要搜索所有元素，就会出现延迟。但真的有必要搜索每个元素吗？这个问题值得好好思考一下。

假设有人要您找出里程表中 Seattle 和 Miami 之间的距离。为了找出 Seattle 的记录项，需要从页面最上面开始一直向下寻找吗？可能不必这样。因为城市是以字母顺序排列的，所以知道 Seattle 一定在靠近列结束的某个位置。同样，Miami 可能出现在中间。这种机率很大，很快就可以找到这些值，而不必查找列中大多数城市名。

12.1.3 折半搜索

要利用 `cityNames` 数组按字母顺序排列的优势，需要使用不同的算法。为了尽可能具体地说明这个过程，假设要在包含下列值的数组中寻找 Miami：

cityNames	
0	Atlanta
1	Boston
2	Chicago
3	Denver
4	Detroit
5	Houston
6	Los Angeles
7	Miami
8	New York
9	Philadelphia
10	San Francisco
11	Seattle

这里不是像线性搜索那样需要从数组最上面开始搜索，如果选择靠近中间的某个元素开始，情况会如何？像下面这样求出索引范围端点的平均值，可以计算中间元素的索引：

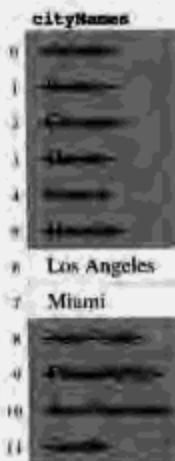
$$\frac{0+11}{2}$$

使用整数算法计算时，表达式的值是 5。

存储在 `cityArray[5]` 中的城市名称是 Houston。假设要寻找 Miami，这时应该怎么办呢？仍然没有找到 Miami，因此必须继续寻找。另一方面，Miami 一定在 Houston 之后，因为数组是按字母顺序排列的。因此，可以直接排除索引位置 0~5 的所有城市名，如下所示。



第一步就排除了一半的可能性。然而，真正的好消息是现在可以重复这样做。已经知道 Miami——如果它在列中——一定在 cityNames 的位置 6 和位置 11 之间。如果使用整数算法计算范围的中间值，可以得到索引值 8。字母表中，Miami 在 New York 之前，因此，可以再删除 4 个位置，如下所示。



在下一周期，可以查找位置 6 上的元素，它是使用整数算法平均 6 和 7 之后的结果。Miami 在 Los Angeles 之后，因此也可以删除 Los Angeles 的值。第 4 个周期中，只需要核对一个元素，它刚好就是在索引位置 7 上的 Miami。

这种算法——寻找有序数组的中间元素，在此基础上确定搜索哪一半数组——称为折半搜索。要实现这种算法，需要记录两个索引，这两个索引分别标记索引搜索限制范围的端点。在该方法中，这些索引存储在变量 lh 和 rh 里，它们分别表示左侧(下方)索引和右侧(上方)索引。最初，这些索引范围包含整个数组，但随着可能性的排除，它们越来越靠近。如果索引值在某个时候交叉，说明数组中不存在键值。

方法 binarySearch 的代码如图 12-5 所示。

```

/*
 * Finds an instance of the specified key in the array, which must be sorted
 * in lexicographic order. If the key exists, the method returns an index at
 * which that key appears, but this index will not necessarily be the first
 * if the same key appears multiply. If the key does not appear in the array,
 * binarySearch returns -1.
 */
private int binarySearch(String key, String[] array) {
    int lh = 0;
    int rh = array.length - 1;
    while (lh <= rh) {
        int mid = (lh + rh) / 2;
        int cmp = key.compareTo(array[mid]);
        if (cmp == 0) return mid;
        if (cmp < 0) {
            rh = mid - 1;
        } else {
            lh = mid + 1;
        }
    }
    return -1;
}

```

图 12-5 用于字符串数组的折半搜索算法的实现方式

12.1.4 搜索算法的相对效率

12.1.3 小节的介绍说明折半搜索算法比线性算法更有效率。即使这样，在没有使用某些定量方法比较两种算法的性能之前，很难说折半算法就有多好。对于搜索而言，显示算法性能最方便的衡量标准是调用 equals 或 compareTo 类比较键值与数组中某个元素的次数。

假设在包含 N 个元素的数组里执行线性搜索算法。方法会调用 equals 多少次？答案当然取决于键值在列中出现的位置。在最坏情况下——也就是键值在最后位置或根本不存在——linearSearch 将调用 equals N 次，数组中每个元素各一次。

运用 binarySearch 中使用的折半搜索算法，情况又怎么样呢？第一次调用 compareTo 后，算法马上可以排除一半数组元素，剩下 $N/2$ 个元素需要继续搜索。第 2 次调用之后，再排除这些元素的一半，就只剩下 $N/4$ 个元素。第 1 次，可能性都减少一半。最后，将整数 N 对半多次以后，就会以 1 结束，此时只需要一次比较就行了。出现这种情况需要的步骤数是得到 1 之前 N 除以 2 的次数，在下面的公式中用 k 表示：

$$N / \underbrace{2/2/2/\dots/2/2}_{k} = 1$$

乘以所有这些 2，得到等式：

$$N = 2^k$$

如果记得高中几何里的对数，那么可以将 k 的值表示为

$$k = \log_2 N$$

因此，搜索包含 N 个元素的数组，使用线性搜索，需要进行 N 次比较；使用折半搜索，则需要进行 $\log_2 N$ 次比较。

以数学形式表示这些算法的相对效率是一种有用的定量分析效率的方法。然而，对于大多数人而言，这样的公式不能传达如何比较这些算法的真正含义。因此，可以看一组数字。表 12-1 显示 N 取不同值时，最接近 $\log_2 N$ 的整数。

表 12-1 N 取不同值时 $\log_2 N$ 的近似值

N	$\log_2 N$
10	3
100	7
1000	10
1,000,000	20
1,000,000,000	30

从表 12-1 中的值可以看出，对于小的数组而言，两种策略都相当好。另一方面，如果有 1 000 000 000 个元素的数组，在最坏的情况下，线性搜索需要 1 000 000 000 次比较来搜索数组；而折半搜索算法只需要大约 30 次比较就能完成。很明显，这种比较次数的减少极大地提高了算法的效率。

但是，折半搜索算法相对于线性搜索而言存在两点不足。第一，折半搜索算法要求数组必须经过排序。如果数组没有经过排序，就必须使用线性搜索。同时，可以自己排序数组来确保数组元素按正确顺序排列。排序数组比搜索数组更有挑战性，也是本章后面讨论的主题。

第二，线性搜索算法优于折半搜索的原因是它更容易编码。写折半搜索算法要谨慎，要考虑到所有特殊情况。相反，典型的线性搜索实现方式中的 for 循环非常简单，不需要考察细节就可以把它写下来。如果程序要求搜索相对较小的数组，就没有必要使用折半搜索，以免增加复杂性——尽管它的效率比较高。只有当数组非常大，折半搜索的优点超过了其复杂性时，才使用它。

12.2 排序

在大多数商业应用程序中，计算机都用来处理非常简单的操作，例如相加一系列数或计算平均值——都是前面几章要解决的那种问题。然而，商业编程所需的一些重要操作却非常复杂。最重要的示例是排序，它是将一列值(通常表示为数组)排列为某种明确顺序的过程。排序的示例有：根据数值，将它们从小到大排列；或者按字母顺序排列一列名称。事实证明这两种操作非常类似。尽管细节上有所不同(一种使用数，另一种使用字符串)，但解决的问题却是相同的：如果有一列元素，要比较其中的两个，该如何重新排列这些元素，让它们以正确的顺序排列？

12.2.1 给整数数组排序

例如，考察给整数数组排序的问题。假设有一个整数数组，以随机顺序排列，如下所示：

31	41	59	26	53	58	97	93
0	1	2	3	4	5	6	7

这里要做的就是定义新方法，可以调用 sort，它将重新排列数组的元素，让数组从小到大排列，如下所示。

26	31	41	53	58	59	93	97
0	1	2	3	4	5	6	7

sort方法的标题行可能如下所示：

```
private void sort(int[] array)
```

然而，编写相应的实现方式比它想象的要灵活，特别是对寻找数据排序的有效策略感兴趣的话。和计算机科学中的许多问题一样，可以使用许多不同的算法。在高级计算机科学课程中，可以花几周甚至是几个月来学习不同的排序算法，但每种算法都有自身的优点和不足。然而这里，要学习计算机科学，最好从能够全面理解的一种算法开始。

12.2.2 选项排序算法

在众多排序算法中，最容易理解的就是选项排序算法。应用选项排序算法时，可以将数组元素一次一个地放入末位。第一步，找到记录项中最小的元素，将它放在开头。第二步，找到剩余元素中最小的那个，将它放在第2个位置。如果在整个数组中继续这一过程，最后的结果就是数组变得有序。

要理解选项排序方法，假设使用下面的数字数组，看看会发生什么：

31	41	59	26	53	58	97	93
0	1	2	3	4	5	6	7

因为最小的元素是位置3上的值26，所以将这个元素移动到位置0。和第11章ReverseIntArray程序一样，不能丢失原来在位置0上的值，因此最简单的方法是交换位置0和位置3上的值。这样，数组就变成如下状态：

26	31	59	31	53	58	97	93
0	1	2	3	4	5	6	7

交换之后，位置0上就是最小的值。

之后可以这样继续处理剩下的元素。下一步是使用相同策略，正确填充数组中的第2个位置。最小值(除了已经正确放置的值26之外)是31，它在位置3上。如果将它与索引位置1上的值进行交换，可以得到如下状态，其中前两个元素的值都是正确的：

26	31	59	41	53	58	97	93
0	1	2	3	4	5	6	7

在下一个周期，会将剩下元素中的最小值(41)交换到位置2：

26	31	41	59	53	58	97	93
0	1	2	3	4	5	6	7

如果继续，可以正确填充索引位置3、位置4，依次类推，直到数组完成排序为止。

要记录算法中的每一步填充了哪个元素，假设使用左手来依次指示每个索引位置。相对于

每个左手位置，可以使用右手来表示数组剩余元素中最小的一个。找到它之后，可以将两个值分别放在左右两只手的位置，然后交换它们。在实现方式中，变量 `lh` 和 `rh` 取代了左右手，它们保存数组中对应元素的索引数。

可以将这种直观印象转换为伪代码，代码如下：

```
for (each index position lh in the array) {
    Let rh be the index of the smallest value between lh and the end of the list
    Swap the elements at index positions lh and rh
}
```

用正确的 Java 语句取代该伪代码非常简单，主要是因为这两种操作很类似。`for` 循环的控制行是标准习语，可以在循环结尾调用 `swapElements` 方法完成交换操作。`swapElements` 方法定义为图 11-10 中 `ReverseIntArray` 程序的一部分。剩下的一步是找出最小值。按照逐步细化的原则，可以定义新方法来实现这种操作，并完成 `sort` 方法的编码，代码如下：

```
private void sort(int[] array) {
    for (int lh = 0; lh < array.length; lh++) {
        int rh = findSmallest(array, lh, array.length);
        swapElements(array, lh, rh);
    }
}
```

`findSmallest` 方法使用 3 个参数：数组和两个索引数，这两个索引数表示数组内的范围，在此范围内寻找最小值。为了与 Java 中使用两个索引指定范围的其他方法(例如 `String` 类中的 `substring` 方法)保持一致，`findSmallest` 方法从索引位置 `p1` 开始寻找最小元素，一直到索引位置 `p2` 之前为止。方法返回指定索引位置之间的数组中最小元素的索引——而不是值。

`findSmallest` 最简单的实现策略是查看数组，在每个循环周期记录当前最小值的索引，直到选择范围的结尾，该值就是整个范围内的最小值。如下面的代码所示，其中变量 `smallestIndex` 记录当前最小值的索引位置：

```
private int findSmallest(int[] array, int p1, int p2) {
    int smallestIndex = p1;
    for (int i = p1 + 1; i <= p2; i++) {
        if (array[i] < array[smallestIndex]) smallestIndex = i;
    }
    return smallestIndex;
}
```

扫描开始时，考察的第一个值自动成为当前最小值。因此，可以将 `smallestIndex` 初始化为起始索引位置，即参数 `p1` 的位置。依次查看每个位置时，必须确定当前值是不是比前面的最小值小。如果是，原来的值就不可能是整个列的最小值，就需要将 `smallestIndex` 的值更正为表示新位置的值。在找到更小值之前，该位置会保留它的值。

方法 `swapElements` 的实现方式与第 11 章 `ReverseIntArray` 的实现方式相同，本应用程序所需的方法有相同的结果，所以没有必要重写它。编写实现某些通用操作的方法时，最好记住它，以便将来使用。成功的编程人员总是尽量重复使用现有的代码，因为这样做可以省去从头开始编写和调试这些程序的麻烦。

复制 `swapElements` 代码就完成了整个选项排序算法的代码。`sort` 方法的代码如图 12-6

所示。

```
/*
 * Sorts an integer array into increasing order. The implementation uses
 * an algorithm called selection sort, which can be described informally
 * in English as follows:
 *
 * With your left hand, point at each element in the array in turn, starting
 * at index 0. At each step in the cycle:
 *
 * 1. Find the smallest element in the range between your left hand and the
 *    end of the array, and point at that element with your right hand.
 *
 * 2. Move that element into its correct index position by switching the
 *    elements indicated by your left and right hands.
 */
private void sort(int[] array) {
    for (int lh = 0; lh < array.length; lh++) {
        int rh = findSmallest(array, lh, array.length);
        swapElements(array, lh, rh);
    }
}

/* Returns the index of the smallest array element between p1 and p2 - 1 */
private int findSmallest(int[] array, int p1, int p2) {
    int smallestIndex = p1;
    for (int i = p1 + 1; i < p2; i++) {
        if (array[i] < array[smallestIndex]) smallestIndex = i;
    }
    return smallestIndex;
}

/* Exchanges the elements in an array at index positions p1 and p2. */
private void swapElements(int[] array, int p1, int p2) {
    int temp = array[p1];
    array[p1] = array[p2];
    array[p2] = temp;
}
```

图 12-6 选项排序算法的实现方式

12.2.3 评估选项排序的效率

选项排序算法有一些优点。第一，它相对容易理解。第二，它完成了任务。然而，还有效率更高的排序算法。遗憾的是，最好的排序算法所需的技术超出了您当前的编程知识水平。然而，虽然不能提高选项排序算法的效率，但也有能力对它进行评估。

一个有趣的问题是在给定输入数据时，执行选项排序花费了多少时间。有两种方法可以解决这个问题：

- 可以运行程序，然后计算它花费了多长时间。因为现代计算机中程序运行速度很快，通常在 1s 内就完成了，用秒表不可能测量出实耗时间，但使用计算机内部时钟可以得到结果。
- 可以更一般地考察程序的操作，形成对其运行方式的定性认识。

12.2.4 测量程序的运行时间

要确定运行程序花费了多长时间，最常见的方式是使用 `java.lang` 程序包里的 `System` 类来记录所需的总时间。`System` 类导出一个名为 `currentTimeMillis` 的静态方法，它返回当前时间。结果表示为 `long`，它记录系统时钟的当前设置与 1970 年 1 月 1 日午夜 12 点(定义为时间计算的基点)之间的毫秒数。这样，执行下面的计算就可以获得所需时间的大概值：

```
long start = System.currentTimeMillis();
// Perform some calculation ...
long elapsed = System.currentTimeMillis() - start;
```

然而，对于上述代码，需要注意以下几点。

- 短的计算通常在 1ms 内完成，即比较两次调用 `currentTimeMillis` 得出的实耗时间在有些情况下会返回 0。更严重的是，`currentTimeMillis` 的定义不能保证结果在毫秒级是精确的，因为有些系统的内部时钟不提供该级别的精度。
- `currentTimeMillis` 方法返回总实耗时间，要测量算法花费了多少实际处理时间，它不一定是最好的衡量标准。Java 运行时系统引入了一些系统开销。Java 中没有可靠方法能够将这些额外系统开销从计算中分解出来。而且，这些变量在 Java 中非常庞大。例如，如果 Java 运行时环境发现需要在计时过程中收回内存空间，则该过程花费的时间是平时的几倍。
- 执行某些计算所需的时间可能由数据决定。算法不同，执行计算所需的步骤可能取决于输入数据。要对典型性能进行合理评估，必须用不同的数据值重复实验。

也可以采取一些方式来最小化这些问题的影响。如果要测量计算时间，而这个时间又非常短暂，以至于系统时钟的精度显得非常关键，那么可以重复计算多次，测量总实耗时间，然后用总时间除以重复的次数。例如，如果排序相同数组 1000 次，从头到尾测量该过程的实耗时间，单次排序步骤的运行时间大概等于总实耗时间的 1%。同样，要最小化在运行中所有系统开销的影响，可以多次执行相同的试验，舍去所有不合适的值，然后求结果的平均值。

最后，由于测量过程中固有精度不够，所以不要用比要求更高的精度来估算算法的运行时间，这一点很重要。例如，在统计学上，如果只要求两位数字，那么对计时试验报告 7 位精度就不合适。

图 12-7 中的数据说明了这些原则，要求显示了计时试验的结果，该试验在不同大小的数据上重复调用 `sort`。对于元素的每个数值，表中的条目用毫秒表示测量的时间，这些时间是排序 10 个独立试验中整数所花费的时间。最后两列显示了这些试验的平均数和标准偏差，通常分别用希腊字母 μ 和 σ 表示(参见第 11 章)。每一行都有合理偏差，说明计时系统不像想象的那样精确。表中 4 个阴影项也非常不协调。例如，在排序 50 个整数的不同试验中，许多值出现在 40ms 附近，左右各有许多变化。然而，试验 7 显示 140ms 的运行时间，它几乎偏离平均数 3 个标准偏差。从统计学上看，其他阴影项也产生了严重偏离。为了确保这些计时测量结果——它可能反映了 Java 系统开销成本，而不是实际的计算时间——不歪曲计时数据，最好从计算中删除它们。

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	μ	σ
N = 10	.0021	.0025	.0022	.0026	.0020	.0030	.0022	.0023	.0022	.0025	.0024	.00029
20	.006	.007	.008	.007	.007	.011	.007	.007	.007	.007	.007	.00139
30	.014	.014	.014	.015	.014	.014	.014	.014	.014	.014	.014	.00013
40	.028	.024	.025	.026	.023	.025	.025	.026	.025	.027	.025	.0014
50	.039	.037	.036	.041	.042	.039	.140	.039	.034	.038	.049	.0323
100	.187	.152	.168	.176	.146	.146	.165	.146	.178	.154	.162	.0151
500	3.94	3.63	4.06	3.76	4.11	3.51	3.48	3.64	3.31	3.45	3.69	0.272
1000	13.40	12.90	13.80	17.60	12.90	14.10	12.70	81.60	16.00	15.50	21.05	21.33
5000	322.5	355.9	391.7	321.6	388.3	321.3	321.3	398.7	322.1	321.3	346.4	33.83
10000	1319.	1388.	1327.	1318.	1331.	1336.	1318.	1335.	1325.	1319.	1332.	20.96

图 12-7 选项排序算法的排序计时

表 12-2 总结了图 12-7 删除不合适记录项之后的数据，同时突出了测量方法中固有的不确定性：每个记录项所示的错误范围对应于平均数左右两个方向的标准偏差。如果数据是正态分布，使用两个标准偏差作为错误栏，可以确保平均数约 95% 的实际值会出现在指定范围内。

表 12-2 运行时间

N	运行时间	
10	0.0024	±0.0006
20	0.007	±0.0007
30	0.014	±0.0003
40	0.025	±0.003
50	0.039	±0.005
100	0.16	±0.03
500	3.7	±0.5
1000	14.3	±0.4
5000	346.0	±68.0
10000	1326.0	±21.0

该表也说明了一些有趣的结果。如果 N 值较小，选项排序运行相当快。然而，随着 N 值越来越大，选项排序算法急剧放慢。例如，如果数组包含 100 个值，sort 可以在约 0.1ms 的时间内给数组排序。如果是 10 000 项，选项排序就要运行 1s 多。商业应用程序通常需要排序 100 000 或 1 000 000 个值，或者更多。在这种范围内的数组中，选项排序会变得非常缓慢。

12.2.5 分析选项排序算法

要理解这些计时数为什么会这样，有必要考察一下算法的运行原理。例如考察 12.2.4 小节总结的选项排序的计时数据。 N 等于 50 时，算法需要运行 0.039ms。然而， N 翻一倍等于 100 时，算法需要 0.16ms，大约是 N 等于 50 时的 4 倍。图中其他数据的变化也产生了相同的级数。

将数据项的数值加倍——例如，从 500 到 1000 或从 5000 到 10 000——所需的时间大约增加至 4 倍。这种算法称为二次方算法，因为其运行时间随着输入大小的呈平方增长。

选项排序是二次方算法，如果理解它的运行原理，这样说就并不奇怪。在 8 个数的排序中，`sort` 方法的选项排序实现方式执行 8 次外部 `for` 循环。第一个周期找出 8 个数的最小值，下一个周期找出剩下 7 个数的最小值，依次类推。程序执行的运算数与要查找的值的个数成比例，这里是：

$$20 + 7 + 6 + 5 + 4 + 3 + 2 + 2$$

更一般的情况是，给定 N 个元素，执行选项排序算法所需的时间与下面的和成比例：

$$N + N-1 + N-2 + \dots + 3 + 2 + 1$$

应用下面的数学公式，可以更简洁地表示前 N 个整数的和：

$$N + N-1 + N-2 + \dots + 3 + 2 + 1 = \frac{N^2 + N}{2}$$

二次方行为即来源于 N^2 项。

选项排序在其表现上是二次方算法，这一点并不意味着所有排序算法都以这种效率级别实现。许多排序算法更快。最好的排序算法在于使用递归算法，第 14 章将简要介绍。然而，有一种具有历史影响的排序算法，可以作为算法和效率之间关系的有效演示，即使它在实际应用中存在某些缺点。

12.2.6 基数排序算法

从第1章对计算历史的简要回顾中可以知道，机械式计算器已经使用了数百年。虽然这样的机器没有现代计算机灵活，但也可以实现某些任务。美国在19世纪90年代的人口普查中曾经使用过编表打孔机。将人口普查调查表的每个答案打到卡上，然后由Herman Hollerith发明的机器制成表格，计算特殊穿孔出现的次数。这种设备开辟了数据处理领域，导致创建了像IBM这样的公司，这些公司依靠可以机械地将信息制成表格的机器，建立了大量成功的商业应用。

20世纪70年代用于数据处理的卡片(在选票计算机和其他一些应用程序中仍在使用)如图12-8所示。



图 12-8 穿孔卡片

标准的穿孔卡片将数据分为 80 列。数字值在卡片上用包含相应数字的矩形孔洞表示。这样，图 12-8 中的卡片包含第 1 列和第 2 列中的数 42。

在 20 世纪 40 年代, IBM 公司引入了一种称为排序器的机电设备。排序器是一个大型机器, 它的一端有一个备用箱, 运算器会将一堆堆穿孔卡片加载到其中。机器启动以后, 它会每次一张地从备用箱里取出卡片, 依据卡片特定列所打的值, 将它们分布到一组有限的输出区域——称为桶。例如, 如果排序器设置为查找列 1, 图 12-8 所示的卡片会被发送到#4 桶, 因为列 1 在数字位置 4 打孔。如果卡片按列 2 中的数字排序, 卡片会在#2 桶中结束。

可以用这种排序器将包含一位数的卡片排列为升序。可以取整堆卡片, 然后在排序器中运行它们。如果从#0 桶中选取卡片, 接着是#1 桶中的卡片, 然后是#2 桶中的卡片, 一直到#9 桶里的卡片, 卡片上的值就是排序的顺序。然而, 假设排序的数据包含多个位数。如何使用排序器将这些卡片按整个多位数字段排序?

IBM 排序器极其实用的基本点——基数排序算法的关键——是, 让这些多位数多次通过排序器, 每列中的数各一次, 这样可以对它们进行排序。窍门是排序操作必须从数的最后一个位数开始, 然后向第一位进行。例如, 要排序一组在列 1 和列 2 中包含两位数的卡片, 首先要使用列 2 中的数字对卡片排序, 收集表示卡片数值的每个桶, 然后使用列 1 中的数字排序。

最好用示例说明该过程。假设卡片上的数据包含下列 15 个值:

42, 25, 37, 58, 95, 25, 73, 30, 54, 21, 37, 45, 34, 43, 98

第一次通过排序器根据第 2 个数位(个位)的数字将值排列到 10 个桶, 结果如下所示。

					45				
					25				
					95				
					25				
					34				
					54				
					73				
					37				
					58				
					30				
					21				
					42				
0	1	2	3	4	5	6	7	8	9

然后可以依次从每个桶中选取卡片, 注意保持每个桶内卡片的顺序。该过程让卡片按下列顺序排列。

30, 21, 42, 73, 43, 54, 34, 25, 95, 25, 45, 37, 17, 58, 98

然后可以将这一堆新卡片发送到排序器, 这一次根据数第 1 个数位(十位)的数字(它在列 1 中打孔)划分卡片。这一过程的结果如下所示。

		25	37	45					
		25	34	43	58				
		30	42	54					
		17	21	37					
		73							
0	1	2	3	4	5	6	7	8	9

如果按顺序收集桶里的这些卡片, 这个序列是 17, 21, 25, 25, 30, 34, 37, 42, 43, 45, 54, 58, 73, 95, 98。正好从小到大排序。

相同的策略可用于任意长度的数, 只要从最后一个数位开始, 依次向前。算法有效, 因为每个桶内结束的卡片都与它们在原始堆栈中的相对顺序相同。因此, 最后一步根据第 1 个数位排序时, 值在每个桶里结束排序, 就像它们在前一步结尾一样。当时它们按下一个有效数字排列。

这种算法称为基数排序, 因为过程中的每一步将数据排序到基数(或者根)指定的桶中, 在桶中表示数据值。IBM 排序器使用十进制(基数为 10)形式的数, 因此, 将数据分布在 10 个

桶中。

编码基数排序算法比编码选项排序算法要困难得多，但如果选择好的分解方法，也不是太难。使用根排序，`sort`方法本身的伪代表形式如下所示。

```
private void sort(int[] array, int n) {
    for (each digit position starting at the right)
        Fill up the individual buckets with values from array
        Reassemble array by taking the contents from each bucket in turn
}
```

在本章练习 10 中有机会完成这种实现方式。

基数排序的运行时间与值乘以最大位数的积成比例，这个值远远小于选项排序所需的平方。因为位数与最大数的对数成比例，所以基数排序的复杂性与 $N \log N$ 成比例，这极大地提高了效率。查看 N 取不同值时这些函数得到的结果，可以理解它们的优点，如表 12-3 所示：

表 12-3 N 取不同值时 $N \log N$ 的值

N	N^2	$N \log N$
10	10	33
100	10 000	664
1000	1 000 000	9965
10 000	100 000 000	132 877

随着 N 逐渐增大，两列里面的数也都在增大，但 N^2 列的增长比 $N \log N$ 的增长列要快得多。所以基于 $N \log N$ 算法的排序算法在较大的数组中比较有用。

应用数学方法预测算法效率的过程称为算法分析，12.3 节将简要介绍这一点。如果继续研究计算机科学，就会更详细地学习分析算法性能的方法。这种知识是一种强大的工具，它可以评估哪种算法最适合特定应用程序。

12.3 评估算法效率

进行详细计时分析(例如本章前面的选项排序分析)产生的问题通常以信息过多而告终。虽然有时需要用公式预测程序运行的精确时间，但通常使用定性的方法。选项排序不适用于 N 值较大的情况，原因与运行于指定机器上特殊实现方式的精确计时特征毫无关系。问题很简单，也很重要。实际上，选项排序的问题是将输入数组的大小(即数组元素数量)翻一倍，选项排序算法的运行时间就会增加到 4 倍，这说明运行时间比数组中元素数量增长得更快。

对于算法效率最有价值的定性认识通常是那些有助于理解算法性能如何响应问题大小的认识。问题大小通常容易量化。对于操作数值的算法而言，通常用数本身表示问题的大小。对于操作数组的大多数算法而言，可以使用数组中的元素数。评估算法效率时，不管如何计算它，计算机科学家通常都使用字母 N 来表示问题的大小。随着 N 不断增大， N 和算法性能之间关系称为该算法计算的复杂性。尽管也可能将复杂性分析应用于其他问题(如所需的内存空间数量)，除非另作说明，否则本文所使用的复杂性评估都是指执行时间。

12.3.1 big-O 表示法

计算机科学家使用特殊符号来表示算法计算的复杂性。所谓的 big-O 表示法，由德国数学家 Paul Bachmann 于 1892 年提出，这远在计算机开发之前。表示法本身很简单，它的形式是字母 O，后面紧接着包含在圆括号里的公式。当它用来指定计算的复杂性时，公式通常是包含问题大小 N 的一个简单函数。例如，本章很快就会遇到 big-O 表达式

$$O(N^2)$$

它读作“N 平方的 big-oh”。

Big-O 表示法用来指定定性的近似值，它也是表示算法计算复杂性的概念。和它在数学中一样，big-O 表示法有明确的定义，但这里理解定义的细节并不重要。此时，对于用户来说——不管将自己当成编程人员还是计算机科学家——最重要的是直观理解 big-O 的含义。

12.3.2 big-O 的标准简化

随着 N 值增大，它的变化会如何影响算法的性能？使用 big-O 表示法来评估算法计算的复杂性时，目的就是提供对这一问题的定性认识。因为 big-O 表示法不是定量方法，所以它不仅应当而且值得减少圆括号内的公式，以便以最简单的形式表示算法的定性行为。使用 big-O 表示法时，可采取的最常见的简化如下：

(1) 如果有的项对总体的贡献随着 N 的增大而不再重要，删除所有这样的项。公式包含一些相加到一起的项时，随着 N 不断增大，某个项通常比其他项增长得快，并最终支配整个表达式。对于较大的 N 值而言，只有该项控制着算法的运行时间，所以可以完全忽略公式中的其他项。

(2) 删除所有常量因子。计算计算的复杂性时，主要关注的是运行时间如何随着 N 的函数而改变。常量因子对整个模式没有影响。如果购买的新机器比老机器快两倍，对于每个 N 值而言，在新机器上执行的所有算法都会比以前快两倍。然而，增长模式却没有变。所以，使用 big-O 表示法时可以忽略常量因子。

12.3.3 选项排序计算的复杂性

应用 12.3.2 小节的简化规则，得出用于选项排序计算的复杂性的 big-O 表达式。从 12.2.5 小节可以知道，包含 N 个元素的数组其选项排序算法的运行时间与

$$\frac{N^2 + N}{2}$$

成比例。虽然从数学上来说，在 big-O 表达式

$$O\left(\frac{N^2 + N}{2}\right)$$



中直接使用这个公式是正确的。但在实际应用中不会这么做，因为圆括号内的公式不是最简单的形式。

要简化这种关系，首先是要认识到公式实际上是两项之和，即

$$\frac{N^2}{2} + \frac{N}{2}$$

然后需要考察随着 N 值不断增大，其中每一项对总公式的贡献，如表 12-4 所示。

表 12-4 N 取不同值时各项的值

N	$\frac{N^2}{2}$	$\frac{N}{2}$	$\frac{N^2 + N}{2}$
10	50	5	55
100	5000	50	5050
1000	500 000	500	500 500
10 000	50 000 000	5000	50 005 000
100 000	5 000 000 000	50 000	5 000 050 000

随着 N 值的增加，包含 N^2 的项快速支配了包含 N 的项。所以，简化规则允许从表达式中删除较小的项。即使这样，也不会将选项排序计算的复杂性写成

$$O\left(\frac{N^2}{2}\right)$$


因为可以删除常量因子。用来表示选项排序复杂性最简单的表达式是：

$$O(N^2)$$

该表达式抓住了选项排序性能的本质。随着问题大小的增加，运行时间会按该增量的平方增加。因此，如果将数组的大小增加一倍，运行时间会增加到 4 倍。如果将输入值的数量乘以 10，运行时间会激增 100 倍。像表现 $O(N^2)$ 性能的选项排序这样的算法称为以二次方时间运行的算法。

12.3.4 根据代码结构预测计算的复杂性

如何确定方法

```
double sumArray(double[] array) {
    double total = 0;
    for (int i = 0; i < array.length; i++) {
        total += array[i];
    }
    return total;
}
```

计算的复杂性？哪个项计算数组元素的和？调用这种方法时，代码的有些部分只执行了一次，例如将 `total` 初始化为 0 的声明。这个初始化步骤要花费一定时间，但这个时间是常量，因为它不取决于数组大小。执行时间不取决于问题大小的代码称为在常量时间内运行，它在 big-O 表示法中表示为 $O(1)$ 。然而，有的同学会对名称 $O(1)$ 产生迷惑，因为圆括号内的表达式不取

决于 N 。实际上，不依赖 N 正是 $O(1)$ 表示法所想要的。随着问题大小的增加，执行运行时间是 $O(1)$ 的代码所需的时间与 1 增加时的方式相同；换句话说，代码的运行时间根本不会增加。

然而，`sumArray` 方法的其他部分会执行 N 次，在每个 `for` 循环周期执行一次。这些部分包括 `for` 循环中的表达式 `i++` 和语句

```
total += array[i];
```

它构成循环体。虽然这部分计算的单次执行会占用固定量时间，但这些语句共执行 N 次，说明它们的总体执行时间与数组大小成比例。这部分 `sumArray` 方法的计算复杂性是 $O(N)$ ，通常称为线性时间。

因此，`sumArray` 的总运行时间是常量部分所需时间与算法线性部分所需时间之和。然而，随着问题大小的增加，常量项变得越来越无关紧要。简化规则允许忽略随着 N 值增大而变得不重要的项，利用这一点，就可以说 `sumArray` 方法整体在 $O(N)$ 时间内运行。

仅仅通过观察代码的循环结构，就可以预测这种结果。在很多情况下，每个表达式和语句——除非它们包含必须单独计算的方法调用——在常量时间内运行。在计算的复杂性方面，真正重要的是执行这些语句的频率。对于许多程序而言，找到最经常执行的代码，然后确定它作为 N 的方法运行多少次，这样可以确定计算的复杂性。在 `sumArray` 方法中，循环体执行了 N 次。因为没有其他代码部分比它执行的次数更多，所以可以预测计算的复杂性就是 $O(N)$ 。

可以使用相同的策略分析图 12-6 中选项排序方法的性能。代码最经常执行的部分是 `findSmallest` 方法中语句

```
if (array[i] < array[smallestIndex]) smallestIndex = i;
```

中的比较语句。该语句嵌套在两个 `for` 循环中，一个在 `findSmallest` 方法本身，一个在调用它的 `sort` 方法中。从 12.2.5 节可知，语句执行了 $O(N^2)$ 次。

12.3.5 最坏情况与一般情况的复杂性

有些情况下，算法的运行时间不仅取决于问题大小，而且还取决于数据的特定特征。例如，考察本章开始介绍的方法

```
private int linearSearch(int key, int[] array) {
    for (int i = 0; i < array.length; i++) {
        if (key == array[i]) return i;
    }
    return -1;
}
```

因为实现方式中的 `for` 循环执行了 N 次，所以可以认为 `linearSearch` 的执行性能是 $O(N)$ 。

另一方面，有些对 `linearSearch` 的调用执行速度非常快。例如，假设要搜索的主要元素在数组的第一个位置。在这种情况下，`for` 循环的主体只运行一次。如果凑巧要搜索的值总是出现在数组开头，`linearSearch` 将以常量时间运行。

分析程序计算的复杂性时，通常对最小可能时间不感兴趣。一般来说，计算机科学家倾向于关注下面两种复杂性分析。

- **最坏情况下的复杂性。**最常见的复杂性分析是确定算法在最坏情况下的性能。这种分析很有用，因为它可以设置计算的复杂性的上限。如果分析最坏情况，可以确保算法的性能至少与分析显示的一样。有时会很幸运，但可以确保性能不会变得更差。
- **一般情况下的复杂性。**从实践的观点来看，如果在所有可能输入数据中平均算法的行为，那么考察算法的性能通常有用。特别是，如果没有理由假设问题的特定输入会出现异常，那么一般情况分析就可以提供实际性能的最好统计评估。然而，问题是一般情况分析通常难以实现，往往需要大量的数学知识。

`linearSearch` 方法的最坏情况就是数组中根本没有键值。当键值不在数组中时，方法必须完成 `for` 循环的 N 个周期，这说明其性能是 $O(N)$ 。如果键值在数组中，`for` 循环会平均执行次数的一半，这说明一般情况性能也是 $O(N)$ 。

有时算法的一般情况和最坏情况性能在定性方面的结果不同，这说明在实际中有必要同时考虑这两种性能特征。

12.4 使用数据文件

从 12.3 节可以知道，程序的运行时间可以随着要解决的问题的大小而改变。例如，线性搜索这样的算法适用于较小数组，随着数组的增加，它变得越来越不实用。对于大数组而言，特定算法可能或多或少有效。这一点提出了一个重要的现实问题：如果要用大数组测试程序，如何将数据读入这些数组？在本书前面几章的简单编程示例中，通常容易输入应用程序所需的数据值。随着数据量的增长，要求由用户输入数据变得越来越不可行。手工向相加两个数的程序输入数值非常简单，但手工填充包含 10 000 个元素的数组就很困难。

本章后面的部分将专门讨论从文件读入数据的问题，以及将数据写入新文件的反向过程。讨论的目的不是全面描述 Java 中可用的文件操作，而是提供编写应用程序——这些应用程序使用大量数据——所需的工具。

12.4.1 文件的概念

程序使用变量来存储信息：输入数据、计算结果及所有生成的中间值。然而，变量中的信息是暂时的。程序停止运行时，这些变量的值就会丢失。对于许多应用程序而言，能够以一种更永久的方式存储数据很重要。

要让信息在计算机上存储的时间比程序运行时间长，通常的方法是将数据收集到逻辑上一致的整体，然后将它作为文件（第 1 章介绍过这个概念）存储在永久存储媒介上。通常，文件存储在计算机内的硬盘上，也可以存储在可移动媒介上，如 CD 或 U 盘。在这两种情况下，操作的基本原则和模式是一样的。重点是存储在计算机上的永久数据对象——文档、游戏、可执行程序、源代码等——都以文件形式存储。

在大多数系统中，文件都有多种类型。例如，在编程领域，使用源文件、对象文件和可执行文件，每种文件都有不同表示法。用来存储程序使用的数据时，文件通常是文本，因此通常称为文本文件。可以将文本文件认为是存储在永久媒介上并且有文件名的一连串字符。文件名与它所包含的字符之间的关系和变量与其内容的关系一样。

例如，假设要收集一组最喜欢的莎士比亚的引用语，并且要将每条引用语存储在单独的文

件里,《哈姆雷特》中的引用语如下。

Hamlet.txt

```
To be, or not to be: that is the question.  
Whether 'tis nobler in the mind to suffer  
The slings and arrows of outrageous fortune,  
Or to take arms against a sea of troubles,  
And by opposing end them?
```

上面显示了文件的名称(这里是 Hamlet.txt),它在文件外面,就像变量图中名称在外面,而值在里面一样。

第二个引用语摘自《罗密欧与朱丽叶》舞台剧中朱丽叶的话:

Juliet.txt

```
What's in a name?  
That which we call a rose  
By any other name would smell as sweet.
```

计算机可以用两个单独的文件保存这些数据,因为文件名不同。

查看文件时,通常将它当成二维结构——由单个字符组成的一系列行。然而,文本文件在内部却表示为一连串字符。除了可以看到的打印字符之外,文件还包含标记每行结束的一些特殊字符或字符序列。遗憾的是,不同的操作系统使用不同的字符序列。但好在 Java 的输入输出操作隐藏了不同操作系统之间的差别,这样就很容易将文件作为一序列行来查看。

文本文件与字符串在许多方面都很类似,两者都是有序的字符序列。他们之间存在的两个重要区别是:

- 存储在文件里的信息是永久的。字符串变量的值存续的时间与变量存续的时间一样长。方法返回时,局部变量就会消失;对象消失时,实例变量也消失,但在程序退出之前,这通常不会发生。而存储在文件里的信息会保存到文件被删除为止。
- 文件通常可以按顺序阅读。从文件中读取数据时,通常从头开始,然后按顺序读取字符。读取一个字符后,继续读取下一个字符,直到文件结束。

12.4.2 阅读 Java 中的文本文件

如果要读取文本文件作为 Java 程序的一部分,首先可以构造一个对象,它从根本上说是 java.io 程序包中 Reader 类的实例,这个过程称为打开文件。和大多数 Java 的库类一样,Reader 类也是更复杂层次结构的一部分,该层次结构提供了多种适用不同场合的类。打开文件的过程包括选择 Reader 的特定子类。

阅读文本文件的过程中,需要使用的一个类是 FileReader,它通过查找文件系统中的特定名称来创建 Reader。例如,如果执行声明

```
FileReader rd = new FileReader("Hamlet.txt");
```

FileReader 类的实现工具会要求文件系统打开名为 Hamlet.txt 的文件,然后返回 FileReader 对象,该对象用来从该文件读入数据。然而,如果只有 FileReader 对象,就不能方便地阅读 Hamlet.txt 的内容。FileReader 类提供一组低级别的方法,它们支持一次一个地从文件读取字符,不允许以更大的单位读取数据。特别是,FileReader 类不能方便地每次阅读整行,而这通常是最有用的操作。为此,需要将 FileReader 类改为 BufferedReader 类。

`BufferedReader` 类的构造函数采用一种阅读器，并创建一个新的有其他功能的阅读器。然而，新阅读器与旧阅读器仍然从相同的源读取数据。虽然可以单独为 `FileReader` 类和 `BufferedReader` 类声明变量，但实际上除了创建 `BufferedReader` 类之外，其他时候不需要 `FileReader` 值。因此，通常是在同一个声明里调用这两个构造函数，代码如下：

```
BufferedReader rd
    = new BufferedReader(new FileReader("Hamlet.txt"))
```

声明并初始化变量 `rd` 之后，就可以调用 `BufferedReader` 类的方法从文件读取数据。调用 `read` 可以从文件读取单个字符，就像使用原本的 `FileReader` 对象一样。更重要的是，`BufferedReader` 允许调用 `readLine`，可以将整行作为字符串读入。

为了更好理解如何使用 `readLine` 方法，假设要阅读文件 `Antony.txt`，它包含的句子是《Julius Caesar》中 Marc Antony 葬礼演说的摘要，如下所示。

`Antony.txt`

Friends, Romans, countrymen,
Lend me your ears;
I come to bury Caesar,
Not to praise him.

编写声明

```
BufferedReader rd
    = new BufferedReader(new FileReader("Antony.txt"));
```

可以打开文件。该声明建立了文件和变量 `rd` 的值之间的链接。为了记录阅读了多少文件，阅读器保存一个内部文件指针，它标记下一个要阅读的字符。文件指针的位置使用竖线表示，尽管其信息全部存储在阅读器内，根本不会反映在实际文件中。打开文件时，文件指针从文件的第一个字符开始，如下所示。

|Friends, Romans, countrymen,
Lend me your ears;
I come to bury Caesar,
Not to praise him.

如果通过调用

```
line = rd.readLine();
```

阅读一行，字符串变量 `line` 会被设置为字符串

```
"Friends, Romans, countrymen,"
```

文件指针会穿过整个第一行，到达第二行开头，如下所示。

|Friends, Romans, countrymen,
Lend me your ears;
I come to bury Caesar,
Not to praise him.

调用

```
line = rd.readLine();
```

会第二次将 line 的值重新设置为字符串

```
"Lend me your ears;"
```

文件指针会移动到第 3 行的开头，如下所示。

```
Friends, Romans, countrymen,  
Lend me your ears;  
I come to bury Caesar,  
Not to praise him.
```

最后，读完文件全部 4 行之后，对 readLine 的调用会返回终止条件值 null 来表示文件中已经没有其他行。此时，应该调用

```
rd.close();
```

关闭阅读器，该调用中断阅读器与文件系统之间的连接。

12.4.3 异常处理

打开文件是一种示例操作，它有时会失败。例如，如果要求用户输入文件名，用户输入的名称不正确，FileReader 构造函数就找不到所需的文件。为了标记这种失败，Java 库中的方法通过抛出异常来做出响应，Java 用短语来描述报告正常程序流之外的异常条件这一过程。Java 方法抛出异常时，它的运行时系统停止执行代码，检查控制堆栈上是否还有其他方法——从当前方法开始，继续向堆栈后面查找，一直到调用当前方法的方法，调用那个方法的方法，等等——直到它找到一种表示要“捕获”这种异常的方法(如果异常被“抛出”)。如果没有捕获异常，程序会完全停止运行，Java 运行时系统向用户报告没有捕获异常。

Java 中发生的许多异常——例如除以 0 之类——称为运行时异常，可以在代码中任何位置发生。编写程序时，不必表示对运行时异常感兴趣。如果代码没有捕获它们，异常就会在控制堆栈上向后传播，如前一段所述。

然而，运行时异常层次结构之外的异常类则不同，情况也会不同。Java 设计师会要求客户找到程序包中方法抛出的异常，强制 java.io 程序包的客户检查不存在的输入文件等情况。因此，打开和阅读 Java 文件的代码是不完整的，除非它明确捕获了 IOException 类中的异常。为此，使用数据文件的代码必须出现在 try 语句内，该语句的一般形式如下边语法框所示。

try 语句的简化语法

```
try {  
    Code in which an exception might occur.  
} catch (type identifier) {  
    Code to respond to the exception  
}
```

其中

type 是异常的类型

identifier 是保存异常信息的变量的名称

在文件处理应用程序语境中，模板如下：

```
try {
    Code to open and read the file
} catch (IOException ex) {
    Code to respond to exceptions that occur
}
```

要理解如何使用异常处理来检查打开文件过程中的错误，最好是编写使用控制台程序的一般方法，该程序允许用户根据提示符输入文件名来选择文件。如果文件存在，方法返回可以阅读文件内容的 BufferedReader。如果文件不存在，方法显示一条消息，表示它没有找到指定文件，然后给用户提供另一次机会来输入文件名。这种方法的实现方式如下：

```
private BufferedReader openFileReader(String prompt) {
    BufferedReader rd = null;
    while (rd == null) {
        try {
            String name = readLine(prompt);
            rd = new BufferedReader(new FileReader(name));
        } catch (IOException ex) {
            println("Can't open that file.");
        }
    }
    return rd;
}
```

try 语句让程序能够检测 IOException 是否出现在主体执行内的某个位置，即使这种异常发生在 openFileReader 调用的某个库方法里。在丢失文件的情况下，FileReader 的构造函数抛出异常，但 try 语句里的 catch 代码块会捕获它。异常发生时，程序执行 println 语句给用户报告错误。而且，因为对 rd 变量的赋值从未完成，所以它的值仍然是 null，以导致 while 循环要求使用另一个文件名。

在像 openFileReader 这样的语境中，如果出现异常，相对容易确定该怎么办。在这种情况下，异常的原因几乎肯定是用户输入的文件名错误。异常出现在像 readLine 或 close 这样的调用上时，就很难对 IOException 做出响应。在这些情况下——当然这种情况很少见——可能的原因是文件系统中的错误。假设程序不能修复错误，最简单的策略是放弃响应异常的努力，让异常处理程序报告发生了不可恢复的错误这一事实。通常方法是抛出会传播到操作系统的运行时异常。acm.util 程序包定义的运行时异常子类(称为 ErrorException)专门为为此而设计，通过提供消息，例如

```
throw new ErrorException("Division by zero");
```

或者转向已经发生的其他一些异常，可以抛出错误异常。本章的文件操作程序通过在下面的语境中嵌入代码来阅读文件，执行第 2 种策略。

```
try {
    Code to open and read the file
} catch (IOException ex) {
    throw new ErrorException(ex);
}
```

12.4.4 倒转文件的程序

图 12-10 中的 ReverseFile 程序阅读数据文件然后以相反的顺序显示文件的行因此能够了解前面几节介绍的各种方法如何形成完整的应用程序。程序首先要求用户使用 12.4.3 小节介绍的 openFileReader 方法输入文件名。用户正确输入现有文件名之后，程序将整个文件读入数组，然后以相反顺序打印数组的行。

```
import acm.program.*;
import acm.util.*;
import java.io.*;
import java.util.*;

/** This program reverses the lines in a file */
public class ReverseFile extends ConsoleProgram {

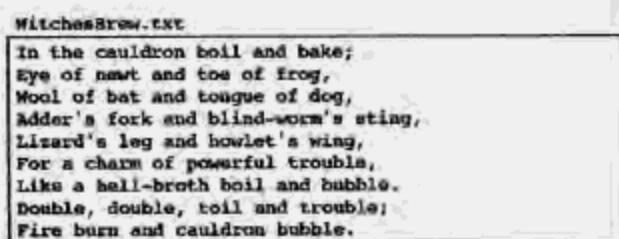
    public void run() {
        println("This program reverses the lines in a file.");
        BufferedReader rd = openFileReader("Enter input file: ");
        String[] lines = readLineArray(rd);
        for (int i = lines.length - 1; i >= 0; i--) {
            println(lines[i]);
        }
    }

    /*
     * Reads all available lines from the specified reader and returns an array
     * containing those lines. This method closes the reader at the end of the file.
     *
     * Implementation note:
     * This implementation uses an ArrayList internally to read the lines of the
     * file because doing so allows the list to expand dynamically. However,
     * because the definition of the method indicates that the method returns an
     * array rather than an ArrayList, the implementation copies the ArrayList
     * to an array after all the lines have been read.
     *
     * @param rd A BufferedReader for the input file
     * @return A string array containing the lines read from the reader
     */
    private String[] readLineArray(BufferedReader rd) {
        ArrayList<String> lineList = new ArrayList<String>();
        try {
            while (true) {
                String line = rd.readLine();
                if (line == null) break;
                lineList.add(line);
            }
            rd.close();
        } catch (IOException ex) {
            throw new ErrorException(ex);
        }
        String[] result = new String[lineList.size()];
        for (int i = 0; i < result.length; i++) {
            result[i] = lineList.get(i);
        }
        return result;
    }

    /* Include the definition of openFileReader from the text */
}
```

图 12-9 以相反顺序打印文件行的程序

可以使用 ReverseFile 程序来倒转下面文件中的行，它是“Macbeth”中女巫们策划的一部分：



如果运行该程序——第一次输入的文件名错误，后来输入正确——可以看到输出如图 12-10 所示，它几乎与原文一样：

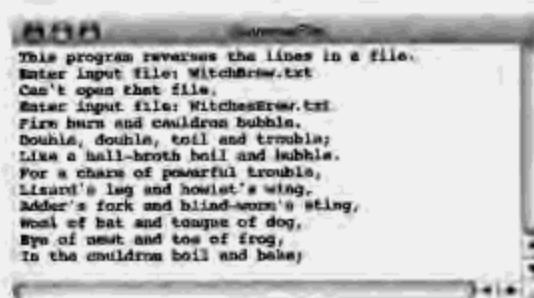


图 12-10 倒转文件输出示例

12.4.5 交互地选择文件

几十年前，能够在控制台窗口输入文件名似乎就足够了，但现代应用程序很少让用户手动输入文件名。现在使用的是文件对话框，它是一种交互式对话窗口，允许通过目录导航和鼠标选择文件。Java 库包括能够完成这种任务的类。

创建文件对话框时，需要创建 JFileChooser 类的实例。JFileChooser 类是 javax.swing 程序包的一部分。JFileChooser 构造函数最常见的形式没有参数，因此，需要创建的代码如下：

```
JFileChooser chooser = new JFileChooser();
```

这时，已经创建了选择器对象，尽管屏幕上什么也没有。要求用户输入文件名时，调用 showOpenDialog 方法。这种方法使用一个参数，该参数必须是显示在屏幕上的组件。如果从 Program 子类调用此方法，最简单的选项是使用 Program 对象本身，它可以用关键字 this 表示。showOpenDialog 方法也返回结果，但需要将该结果保存在 int 类型的变量里。因此，对 showOpenDialog 的典型调用如下：

```
int result = chooser.showOpenDialog(this);
```

使用该调用之后，就会在屏幕上弹出对话框，如图 12-11 所示。该对话框允许用户在文件系统里来回移动，最后选择文件。这里，用户选择了文件 Hamlet.txt。如果单击 Open 按钮，程序会打开 Hamlet.txt 文件；如果单击 Cancel 按钮，程序不会打开选择的文件。

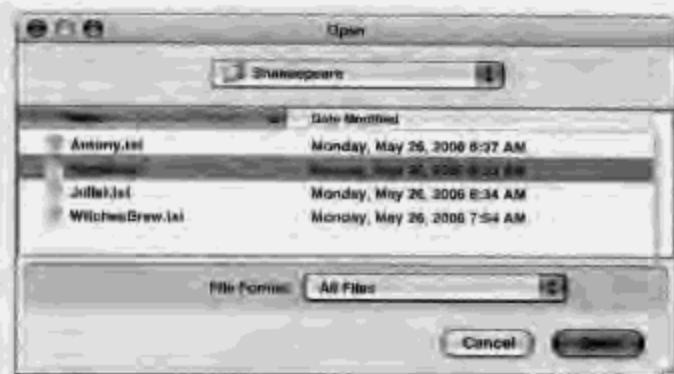


图 12-11 JFileChooser 对话框示例

要想知道单击了哪个按钮，可以检查 `showOpenDialog` 调用的结果。如果单击 `Open` 按钮，`showOpenDialog` 返回常量 `JFileChooser.APPROVE_OPTION`；如果单击 `Cancel` 按钮，方法返回 `JFileChooser.CANCEL_OPTION`。在打开文件前，程序应该确保 `result` 等于 `JFileChooser.APPROVE_OPTION`。

为了找出要打开的文件，需要向 `JFileChooser` 对话框查询信息。调用 `getSelectedFile` 方法可以实现这一点，它返回用户选择的文件。`getSelectedFile` 的结果不仅仅是文件名，而且还有 `File` 对象，与 `java.io` 程序包中定义的一样。使用 `File` 对象而不使用文件名的好处是 `File` 对象包括目录信息，这让 Java 能够打开文件，而不管它在文件系统的什么位置。幸运的是，`FileReader` 构造函数既接受 `File` 也接受 `String`。

如果将这些放在一起，可以用对下面方法的调用取代对图 12-10 中 `openFileReader` 的调用，下面的方法使用文件选择器打开输入文件。

```
private BufferedReader openFileReaderUsingChooser() {
    BufferedReader rd = null;
    JFileChooser chooser = new JFileChooser();
    int result = chooser.showOpenDialog(this);
    if (result == JFileChooser.APPROVE_OPTION) {
        try {
            File file = chooser.getSelectedFile();
            rd = new BufferedReader(new FileReader(file));
        } catch (IOException ex) {
            println("Can't open that file.");
        }
    }
    return rd;
}
```

和前面一样，如果单击 `Open` 按钮，方法返回与选择文件相关的打开的 `BufferedReader`。如果单击 `Cancel` 按钮，`openFileReader` 返回 `null`，说明这种情况下需要改变 `run` 方法。

12.4.6 使用 Scanner 类

虽然 `readLine` 方法可以从文件中阅读整行，但它没有真正解决阅读其他数据的问题。例如，如何在包含浮点数的文件中每行一个值地读取整数数组？最完美的策略之一是使用第 7 章介绍

的 Double.parseDouble 方法, 将每行转换为 double 类型。如果使用的是 Java 5.0 之前的版本, 这种策略的确是可以使用的唯一策略。为了简化读取各种 Java 原始类型数据值的问题, Java 5.0 的设计师们引入了一个新类——Scanner 类, 从而极大地简化了这些操作。

和 Java 5.0 中的许多类一样, java.util 程序包中的 Scanner 类导出大量方法——超过 70 种, 如果计算构造函数——这让全面掌握该类变得很困难。然而, 如果使用这些方法的子集, 就很容易掌握 Scanner 类。图 12-12 显示了 10 种方法, 它们在大多数应用程序中非常有用。

构造函数

new Scanner(Reader rd)
创建新的可以从阅读器读入值的 scanner

读取下一个最常见类型值的方法

String next()
返回下一个被空格分开的令牌
int nextInt()
如果令牌流下一个出现 int, 读取它; 如果不是, 扫描仪抛出异常
double nextDouble()
如果令牌流中出现的下一个令牌是 true 或 false, 读取 boolean, 否则扫描仪抛出异常
boolean nextBoolean()
测试特定类型的令牌是否存在方法

测试特定类型的令牌是否存在方法

boolean hasNext()
如果可以读取更多令牌, 返回 true
boolean hasNextInt()
如果此时在令牌流中可能读取 int, 那么读取 true
boolean hasNextDouble()
如果此时在令牌流中可能读取 double, 那么读取 true
boolean hasNextBoolean()
如果此时在令牌流中可能读取 Boolean, 那么读取 true

关闭扫描仪的方法

void close()
关闭扫描仪和基本阅读器

图 12-12 Scanner 类中的方法

使用 Scanner 类的基本范例是创建从特定源(通常是某种类型的 Reader)读取数据的实例。然后可以调用以 next 开头的各种方法来读取特定类型的数据值。例如, 如果要从 Scanner 读取 double 值, 可以调用 nextDouble 方法。这种方法会跳过所有空白字符, 将下一个令牌作为 double。如果成功, 就获得 double 类型作为方法的返回值。如果没有成功——很可能是因为当时在输入中文件没有包含合法的双精度数——Scanner 抛出 InputMismatchException。与 java.io 程序包中的异常不同, InputMismatchException 是 RuntimeException 的子类, 这说明不用强制捕获它。如果出现了这样的异常, 而没有采取步骤找到它, 程序会由于未找到异常错误而完全停止运行。使用 Scanner 类最大的好处之一是它不必考虑任何出现的 IOException, 因为在 Scanner 内部会自动捕获这种异常。

为了更好理解使用 Scanner 的方法, 下面的方法利用已经打开的 Reader, 一次一行地读取包含浮点值的文件:

```
private double[] readDoubleArray(Reader rd) {
    ArrayList<Double> doubleList = new ArrayList<Double>();
    Scanner scanner = new Scanner(rd);
```

```
        while (scanner.hasNextDouble()) {
            doubleList.add(scanner.nextDouble());
        }
        double[] result = new double[doubleList.size()];
        for (int i = 0; i < result.length; i++) { result[i] = doubleList.get(i);
    }
    scanner.close();
    return result;
}
```

方法返回文件中的 `double` 数组。如果要编写使用数字数组的程序，这种方法很有用。

12.4.7 输出文件

与文件相关的最后一个问题是，文件可以用来将数据写进文件系统，注意到这一点很重要。在许多方面，写数据比阅读数据容易，因为可以使用 `println` 方法。唯一不同是需要提供一个接收器，它必须是 `PrintWriter`。

从文件名中创建 `PrintWriter` 在许多方面与创建 `BufferedReader` 用于输出相反。可以先创建 `FileWriter`，然后用它来创建 `PrintWriter`。这种方法如下面的 `run` 方法所示，它创建一个包含消息“hello, world”的文件 `Hello.txt`：

```
public void run() {
    try {
        PrintWriter wr
            = new PrintWriter(new FileWriter("Hello.txt"));
        wr.println("hello, world");
        wr.close();
    } catch (IOException ex) {
        throw newErrorException(ex);
    }
}
```

从代码中可以知道，写文件也要求找到 `IOException`。可以使用 `JFileChooser` 类来选择要写的文件。唯一不同是调用 `showSaveDialog` 而不是 `showOpenDialog`。

12.5 小结

本章介绍了数组的两个重要操作——搜索和排序——每种操作本身都是一个有趣的算法问题。本章介绍的重点是：

- 线性搜索算法的运行方式是按顺序查找数组里的所有元素，直到找到所需的元素为止。
- 折半搜索算法比线性搜索算法效率更高，但要求数组元素必须按顺序排列。二进位算法的效率优势在于每个周期内都可以排除一半的潜在数组元素。
- 尽管二进位算法在理论上提高了效率，但线性搜索更容易编码，因此，当数组比较小时，选择线性搜索更合适。
- 排序算法的效率存在很大差异。对于包含较少元素的数组而言，像选项排序这样的简单算法就完全足够了。然而，对于较大的数组，这种算法就会变得没有效率。

- 许多算法问题可以表示为整数 N , 它表示问题的大小。对于操作数组的算法而言, 通常将问题的大小定义为元素数。
- 效率最有用的定性衡量标准是计算的复杂性, 它表示随着问题大小不断增加时, 问题大小与算法性能之间的关系。
- Big-O 表示法可以直观表示计算的复杂性, 因为它可以用最简单的形式, 突出复杂性关系最重要的方面。
- 使用 big-O 表示法时, 可以删除公式中随着 N 不断增大而变得不重要的所有项, 以及所有常量因子。
- 查看程序包含的循环的嵌套结构, 通常就可以预测程序计算的复杂性。
- 读取数据文件的方便策略是文件, 获得 BufferedReader 对象, 然后调用 readLine 方法, 作为字符串读入文件中的每一行。readLine 方法在文件结尾返回 null。
- 创建 PrintWriter 对象, 然后调用 println 方法将每行数据写到文件, 这样可以将数据写到文件。
- 使用 java.io 来读取或写入数据文件的代码必须包含 try 语句, 以找出文件处理过程中出现的异常。
- 对于应用程序而言, JFileChooser 类可以方便地让用户从交互式对话框中选择文件。
- 可以使用 Java 5.0 中的 Scanner 类从阅读器读取数字数据。

12.6 复习题

1. 定义术语“搜索”和“排序”。

2. 应该对 linearSearch 方法做什么改变, 以便它在 double 类型值的数组中找到匹配的值? 在同一个程序里可以同时包含 linearSearch 的两种形式吗?

3. 用简单语言描述线性搜索算法和折半搜索算法。

4. 判断题: 如果数据项数足够多, 折半搜索算法可以比线性搜索算法快几百万倍。

5. 应用折半搜索算法之前必须满足什么条件?

6. 描述选项排序算法包含的步骤。

8. sort 的选项排序实现方式中 for 循环控制行如下所示:

```
for (int lh = 0; lh < array.length; lh++)
```

如果将这一行改变为

```
for (int lh = 0; lh < array.length - 1; lh++)
```

方法还可以实现吗?

8. 可以调用什么方法确定当前时间(以毫秒为单位)?

9. “算法是二次方算法”, 这句话是什么意思?

10. 应用基数排序算法时, 在最重要的数位上还是在最不重要的数位上实现第一次排序?

为什么?

11. 计算

$$N + N-1 + N-2 + \dots + 3 + 2 + 1$$

之和的最简表达式是什么？

12. 用自己的话，定义计算复杂性的概念。

13. 本章介绍了哪两条用来简化 big-O 表示法的原则？

14. 从技术上说，选项排序在

$$O\left(\frac{N^2 + N}{2}\right)$$

时间内运行，这种说法正确吗？如果不对，有什么错误？

15. 从技术上说，选项排序在 $O(N^3)$ 时间内运行，这样说对吗？如果不对，它又错在哪里？

16. 解释最坏情况和一般情况复杂性之间的差别。一般来说，哪种衡量标准更难计算？

17. 根据 big-O 表示法，下列算法一般情况下计算的复杂性是什么：线性搜索，折半搜索，选项排序和合并排序？

18. 使用 big-O 表示法，将下列方法的计算复杂性表示为参数 n 的函数：

```
(1) private int Mystery1(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            sum += i * j;
        }
    }
    return sum;
}

(2) private int Mystery2(int n) {
    int count = 0;
    while (n > 0) {
        n /= 2;
        count++;
    }
    return count;
}

(3) private int Mystery3(int n) {
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < i; j++) {
            sum += j * n;
        }
    }
    return sum;
}
```

19. 本章代码使用了 BufferedReader 类中的什么方法，而更原始的 FileReader 类中不存在这种方法？

20. 什么是异常？

21. 异常处理通常包括两种不同的方法：一种方法检测异常条件，它出现时发送信号，而另一种方法响应该信号。Java 使用什么动词来描述这两种方法采取的动作？

22. 本章的一些程序调用了 BufferedReader 类中的一种方法，而这种方法在更原始的 FileReader 类中不存在。这个方法的名称是什么？

12.7 编程练习

1. 编写程序 GuessTheNumber, 它与用户玩一个猜数字的游戏, 用户可能是一名小学儿童。这名儿童先假定一个数, 然后回答一连串问题, 直到计算机正确猜出这个数为止。图 12-13 所示为该数是 17 时的情况。

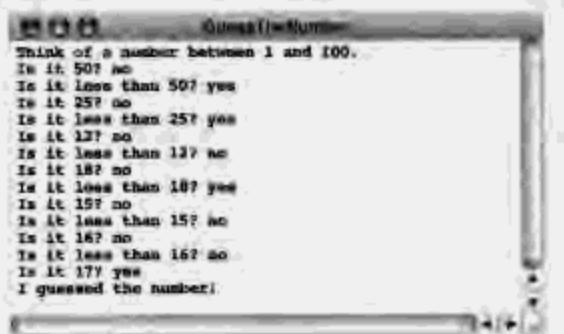


图 12-13 GuessTheNumber 的运行示例

2. 编写断言方法 `isSorted(array)`, 它考察一个整数数组, 如果数组以非降顺序排序, 方法返回 `true`。

3. 重写选项排序实现方式, 让 `sort` 方法可以使用 `ArrayList<Integer>` 和 `int` 类型的数组。方法的 `ArrayList` 形式只能使用 `ArrayList` 操作, 不仅要将 `ArrayList` 转换为数组并进行排序, 还要将它转换回来。

4. 修改选项排序算法的代码来产生调用 `alphabetize` 的方法。`alphabetize` 可以将字符串数组按词典顺序排序。

5. 在第 11 章的练习中, 要求写两个程序来计算常见的统计标准: 平均数和标准偏差。另一个重要的统计标准是中间数, 这个数据值占据了分布(分布的值已经按从低到高的顺序排列)中间元素的位置。如果分布包含偶数个值, 那么就没有中间元素, 标准约定是平均最靠近中点的两个值。

编写方法 `median(array)`, 返回 `double` 类型数组的中间数。实现方式不能假设数组是有序的, 也不能以任何方式改变数组, 尽管它可以创建数组的副本。

6. 除了平均数和中间数之外, 另一种统计方法是用来表示分布中最有代表性的元素——众数, 它是数组中出现最频繁的值。例如, 数组

65	84	95	75	82	79	82	72	84	94	86	90	84
0	1	2	3	4	5	6	7	8	9	10	11	12

中, 众数是 84, 因为它出现了 3 次。其他唯一一个出现次数超过 1 次的值是 82, 它只出现了两次。

编写方法 `mode(array)`, 让它返回 `double` 类型数组的众数。如果有几个值出现的频率相等, 而且它们比其他值出现的次数要多(这种分布称为多峰分布), 方法应该将这些值都作为众数返回。和练习 5 中一样, 实现方式不能假设数组是有序的, 也不能改变数组元素的顺序。

7. 许多算法问题与其解决方案结构中的排序有关。例如，根据随机键值给数组“排序”可以打乱该数组。这样做的方法之一是从选项排序算法开始，然后用选择随机位置这一步骤取代寻找最小值位置这一步骤。结果就会产生混乱的算法，其中每种输出配置的可能性都是相等的。

编写程序 `Shuffle`，它以随机排列的顺序显示 1~52 之间的整数。

8. 在介绍层次结构时，最著名的算法问题之一是荷兰国旗问题(Dutch National Flag)，它由 Edsger Dijkstra 首先提出。假设有包含 N 个元素的数组，每个元素是字符“R”、“W”、或“B”，它们表示荷兰国旗的颜色。起初，这些值在数组中是无序的，如下面的配置所示：

R	B	W	W	B	B	R	W	W	R	R	W	R	B	R
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

现在的任务是重新排列这些字符，让它们出现的顺序与荷兰国旗颜色排列的顺序相同：先是所有红色，接着是所有白色，最后是所有蓝色。

通过研究图 12-14 所示的程序运行示例来推理算法，它显示了每次交换两个位置后颜色的序列。

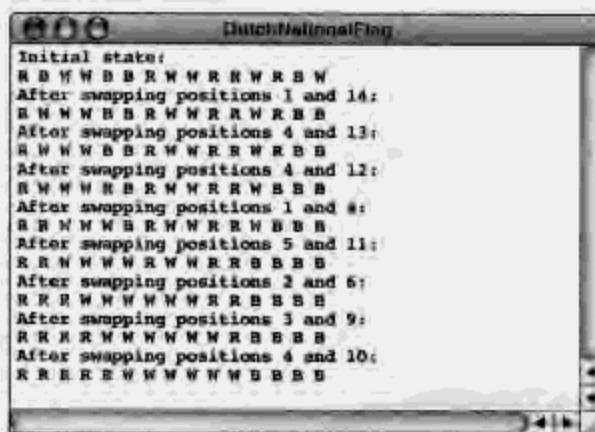


图 12-14 荷兰国旗问题的运行示例

编写程序实现算法，算法的初始状态是随机构造的。

9. 以现有的编程知识水平，除了选项排序之外，还有其他一些排序算法可用。有一种算法，它在实际生活中有一些实用程序，因为它在有序数组上能够快速运行，这种算法称为插入排序。

插入排序算法运行如下：依次查看数组中的每个元素，和选项排序算法一样。然而，在此过程中的每一步，目标不是找剩余值中的最小值并将它交换到正确位置，而是确保数组中到目前为止查看到的值相对于另一个值顺序正确。虽然随着处理的元素越来越多，这些值可能移位，但它们自身形成了有序的序列。

例如，如果再次考察选项排序中使用的数据，插入排序算法的第一个周期不会采取任何措施，因为一个元素的数组总是有序。

			有序的				
31	41	59		26	53	58	97
0	1	2		3	4	5	6

主循环接下来的两个周期也不会重新排列数组，因为序列 31-41-59 本身形成了有序的子数组。

第一个重要操作出现在下一个周期，此时需要将 26 放入此序列。要找到放置 26 的位置，需要回到前面的元素(这些元素相互间是按顺序排列的)寻找 26 的位置。每一步，都需要将其他元素移动一位，以便给 26 留出空间，本示例中这个位置是 0。因此，第 4 个周期之后排序如下所示。

			有序的				
26	31	41	59	53	58	97	93
0	1	2	3	4	5	6	7

后面的每一步，可以将下一个元素插入前面子数组中的合适位置，子数组在每一步结束后都是有序的。使用插入排序算法重新实现 sort 方法。

10. 完成基数排序算法的实现方式，让它会排序长度为 10 位数的非负整数数组。算法的伪代码版本参考第 12.2.6 小节。

11. 编写程序 WordCount 读取文件，然后报告它里面有多少行、多少字及多少字符。对于这个练习，程序不应该计算行尾字符，应该假设字由一序列连续的除了空白字符之外的字符组成。例如，如果文件 Lear.txt 包含《李尔王》中的一段：

```
Lear.txt
Poor naked wretched, whereso'er you are,
That hide the paling of this pitiless storm,
How shall your houseless heads and unfed sides,
Your loopt'd and window'd raggedness, defend you
From seasons such as these? O, I have ta'en
Too little care of this!
```

程序应该能够生成如图 12-15 所示的运行结果。

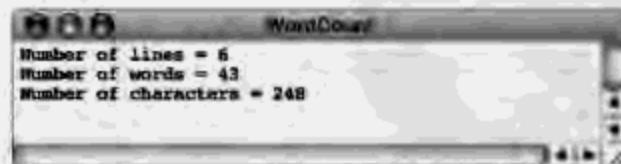


图 12-15 WordCount 的运行示例

12. 有些文件使用制表符将数据排列成列。然而，这样做可能给某些应用程序带来问题，这些应用程序不能直接使用制表符。对于这些应用程序而言，可以用到下一个制表符停顿点所需的空格数，代替输入文件中的制表符。在编程过程中，通常每隔 8 个空格设置制表符停顿点。例如，假设输入文件包含

```
abc---inopqr---xyz
```

其中，——表示制表符占用的空间，它依据在行中的位置不同而不同。如果每隔 8 个空格设置一个制表符停顿点，第一个制表符一定由 5 个空格取代，第二个被 3 个空格取代。

编写程序 Untabify，使用用户的 JFileChooser 并更新文件，以便用足够到达下一个制表符停顿点的空格取代所有制表符。要取代文件，最简单的方法是将整个文件读入内存。关闭它，修改文件的存储形式，重新打开相同的文件用于输出，然后写修改后的文本。

13. 有时出版商发现可以使用没有被实际单词分隔的布局和风格设计。所以，他们有时用随机字母取代原始字母来排版样本页面。结果文本有原文的间距和标点符号，但不能传达任何设计意思。以这种方法取代后的文本在出版术语称为希腊版式。

编写程序 Greek，它从输入文件中读取字符，进行适当的随机取代之后，将它们显示在控制台上。输入中的所有大写字符应该用随机大写字符代替，小写字符应该用随机小写字符代替，非字母字符保持不变。例如，假设输入文件包含下面 *Julius Caesar* 第一幕中 Cassius 与 Casca 之间的对话：

```
CassiusCasca.txt
CASSIUS. Did Cicero say anything?
CASCA. Ay, he spoke Greek.
CASSIUS. To what effect?
CASCA. Nay, an I tell you that, I'll ne'er look
you i' the face again; but those that understood
him smiled at one another and shook their heads;
but for mine own part, it was Greek to me.
```

程序应该生成如图 12-16 所示的输出。

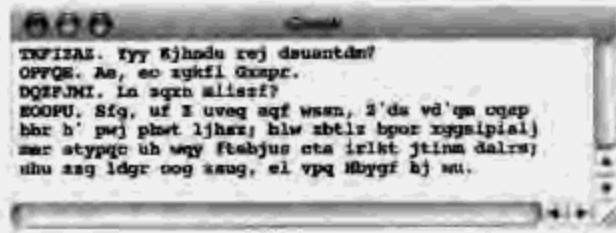


图 12-16 Greek 运行示例

14. 美国出版的第一个纵横拼字谜由 Arthur Wynne 设计，出现在 1913 年 12 月 21 日的《The New York World》上。该字谜的解决方案栅格如图 12-17 所示。



图 12-17 纵横拼字谜栅格(1)

使用空格表示边界周围完全看不见的方块，使用@字符表示内部的黑色方块，可以将此解决方案存储到数据文件里。因此该字谜的文件形式如图 12-18 所示。

```

FirstCrossword.txt
R
FUN
SALES
RECEIPT
MERE@FARM
DOVE@@@RAIL
MORE@@@DRAM
HARD@@@TIED
LION@SAND
EVENING
EVADE
ARE
D

```

图 12-18 字谜的文件形式

编写 GraphicsProgram 程序，它以这种形式读取数据文件，然后画黑色的纵横字谜栅格。文件中由空格表示的位置不应该出现，由@字符表示的位置应该作为填充的方块出现在栅格中，用字母标记的位置应该作为轮廓线方块出现在栅格中。如果方块在对角、向下或两个方向上的单词开头，方块应该包含字谜指定的数。图 12-19 所示图形程序的输出应该如图 12-19 所示。

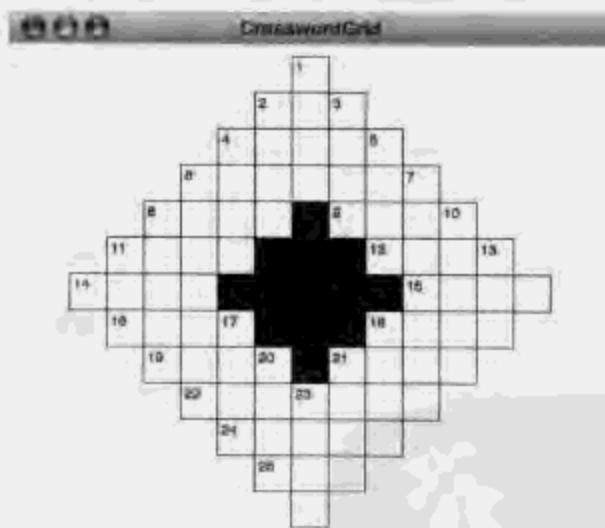


图 12-19 纵横拼字谜栅格(2)

第 13 章

数组与 ArrayList 类

I think this is the most extraordinary collection of talent, of human knowledge, that has ever been gathered at the White House—with the possible exception of when Thomas Jefferson dined alone.

—John F. Kennedy, dinner for Nobel laureates, 1962



Joshua Bloch

Joshua Bloch 第一次对编程感兴趣是在 1975 年，当时他有幸使用了分时 PDP-10 计算机。3 年后，他进入哥伦比亚大学学习，主修计算机科学，之后在 Carnegie Mellon 大学取得了博士学位。取得博士学位之后，Bloch 在 Transarc 公司工作，Transarc 是由他的论文导师创建的一家新公司。1994 年，他到加利福尼亚州，参加了 Sun Microsystems 的 Java 开发小组。就在那里，Bloch 设计和实现了本章介绍的 Java 集合架构(Java Collection Framework)。2004 年，Bloch 离开 Sun 公司，加入了 Google 公司，在那里他担任首席工程师和首席 Java 结构师。最近 Josh Bloch 和他的同事 Neal Gafter 写了一本非常好的 Java 错误集，书名是 *Java Puzzlers*，也由 Addison-Wesley 出版。

虽然使用 ACM Java 库或标准 Java 程序包中的许多有用类可以极大简化编程过程，但成为一名高效的编程人员不需要记住每个可用库程序包里的内容。大多数经验丰富的编程人员只熟悉自己经常使用的程序包和类。如果某个类大概每年只使用一次，那么实际需要它的时候就去查看它的文档，而不是记住它的全部细节。重要的是知道如何找到信息、如何理解它。

一般来说，有关 Java 程序包详细信息的最好资源是它基于 Web 的文档，通常是一组 javadoc 页面。从第 6 章可以知道，程序包的 javadoc 条目描述每个类，以及客户可以使用的公共条目。javadoc 基于 Web，这使导航变得非常容易。如果一个类里的方法要从其他类定义，那么就可以通过链接立即跳转到所需的文档。

然而，理解找到的文档却需要一些实践经验。作为一名新的编程人员，很快就会发现自己

被文档及其描述的类所淹没。信息太多就会带来信息过载问题，需要的信息包含在里面，但从大量细节中找到它却并不那么容易。为了避免这种情况，必须找到一种方法将各种信息安排到概念结构之中，该结构允许导航所有细节。

大多数 Java 程序包都有一个基本的概念模型，这有助于让混乱变得有序。如果没有理解这种模型就直接使用它，通常不可能成功。有必要花点时间研究是什么将程序包中不同的类绑在一起，让它们能够作为综合系统使用。

本章的目的是介绍标准 Java 库非常重要的一部分——Java 集合架构。它是一组综合类，这些类可以更方便地操作数据集合。虽然有些类——特别是 `ArrayList` 和 `HashMap`——可能是日常工具的一部分，但本章的目的不是作为一组单个类而是作为一个整体来查找 Java 集合架构。通过考察整体模型，不仅可以说知道如何使用这些类，而且可以说知道如何更好地设计自己的程序包。

13.1 `ArrayList` 类回顾

前面已经介绍过 `ArrayList` 类，它确实是 Java 集合架构中最常用的类。从第 11 章可以知道，`ArrayList` 类提供了可以应用于数组的基本操作，并且通过动态添加和删除元素，扩充了数组的标准行为。

13.1.1 动态分配的能力

能够添加新元素让 `ArrayList` 类比内嵌的数组机制更有用。例如，下面的代码摘自图 12-9 中 `readLineArray` 方法的实现方式，它创建了一个名为 `lineList` 的 `ArrayList` 类，并用文件的行填充它，该文件与存储在变量 `rd` 里的 `BufferedReader` 有关。

```
ArrayList<String> lineList = new ArrayList<String>();
try {
    while (true) {
        String line = rd.readLine();
        if (line == null) break;
        lineList.add(line);
    }
    rd.close();
} catch (IOException ex) {
    throw new ErrorException(ex);
}
```

虽然请求捕获异常的 `try` 语句让示例有点难以理解，但其余代码却简单明了。第一条语句创建新的字符串 `ArrayList`，它最初为空。`try` 语句里的 `while` 循环调用阅读器的 `readLine` 来读取文件的下一行。只要从 `readLine` 返回的行不是 `null` 终止条件，代码就会将行添加到 `lineList` 的后面。

新的一行出现时，`ArrayList` 类要为它留出空间。至少与单纯依赖数组的实现方式进行比较时，这一点体现了程序的重要简化，然而，所有只依赖数组的实现方式必须以某种方式确定数组的大小。如果事先知道文件里有多少行，就可以正确分配数组的大小。在没有这些信息的情况下，很难知道这种策略如何运行。

这个问题的解决方案在于 Java 能够实现动态分配，这意味着随着程序的运行，可以在堆里分配新的对象。基本思想是使用某个默认大小的数组，当数组空间用完时，扩展数组。图 13-1

中 readLineArray 修改后的实现方式是这种方法的一种实现方式。在这种实现方式中，代码首先分配包含 INITIAL_CAPACITY 元素的数组，当数组空间用尽时，将数组大小翻一倍。因为不可能改变现有数组对象的大小，所以 readLineArray 实现必须分配新的数组，它的大小是原来数组的 2 倍，然后将原来数组中的所有值复制到新数组，最后用新数组取代原来的数组。在图 13-1 所示的代码中，这一工作由方法 doubleCapacity 完成。

```
/*
 * Reads all available lines from the specified reader and returns an array
 * containing those lines. This method closes the reader at the end of the file.
 *
 * Implementation note:
 * This version of the implementation uses arrays rather than array lists
 * to read the lines. Because you cannot add new elements to an array in
 * the way you can with an ArrayList, this implementation reallocates the
 * array whenever it runs out of space by doubling the available capacity.
 * As in the original version, this method uses a second pass to copy the
 * contents of lineArray into a new array with the right number of elements.
 *
 * @param rd A BufferedReader for the input file
 * @return A string array containing the lines read from the reader
 */
private String[] readLineArray(BufferedReader rd) {
    String[] lineArray = new String[INITIAL_CAPACITY];
    int nLines = 0;
    try {
        while (true) {
            String line = rd.readLine();
            if (line == null) break;
            if (nLines + 1 >= lineArray.length) {
                lineArray = doubleCapacity(lineArray);
            }
            lineArray[nLines++] = line;
        }
        rd.close();
    } catch (IOException ex) {
        throw new ErrorException(ex);
    }
    String[] result = new String[nLines];
    for (int i = 0; i < nLines; i++) {
        result[i] = lineArray[i];
    }
    return result;
}

/*
 * Creates a string array with twice as many elements as the old array and
 * then copies the existing elements from the old array to the new one.
 */
private String[] doubleCapacity(String[] oldArray) {
    String[] newArray = new String[2 * oldArray.length];
    for (int i = 0; i < oldArray.length; i++) {
        newArray[i] = oldArray[i];
    }
    return newArray;
}

/* Private constants */
private static final int INITIAL_CAPACITY = 10;
```

图 13-1 只使用数组操作的 readLineArray 的实现

Java `ArrayList` 类的代码使用相同的策略，只是在一些次要细节上有所不同。然而，作为 `ArrayList` 类的客户，这些细节无关紧要。对用户而言，`ArrayList` 类如何让附加空间可用并不重要，重要的是它这么做了。

理解图 13-1 中代码的重要一点是，这种实现方式有两种不同方法可用来计算数组包含多少元素。从客户的观点来看，唯一重要的衡量标准是从文件读入的行数，它称为有效大小。`readLineArray` 修改后的实现使用变量 `nLines` 来记录有效大小。另一种衡量标准是给新元素(包括那些还不存在的元素)预留的空间。这种衡量标准称为数组的容量。程序必须确保有效大小不能超过数组容量。如果添加新元素会让数组超过其当前容量，实现方式就必须让数组变大以扩展空间。

13.1.2 区分表示法和行为

13.1.1 小节介绍实现细节的目的不是指明在代码中如何避免使用 `ArrayList` 类，而是增强用户对 `ArrayList` 类提供的灵活性的理解。大多数编程人员不想为内部表示法的细节费心。`ArrayList` 类是一种很不错的工具，它将这些细节保持在适度范围，让编程人员能够专心研究要解决的问题。

至少在一个方面，使用 `ArrayList` 类给客户完全提供了基本表示法。顾名思义，`ArrayList` 类建立在 Java 数组模型之上。然而，对于许多客户而言，这一点不太重要。对于他们而言，重要的是能够调用 `ArrayList` 类导出的方法，这些方法如图 11-15 所示。如果其他类导出相同的方法，如果这些方法有相同的效果，客户就可以方便地使用那个类。

在 Java 等面向对象编程语言中，所有类的本质特征是它的行为，特别是在类导出什么方法，这些方法对对象的明显状态有什么效果方面。客户依赖实现方式来确保每个方法如预期的那样动作，而该实现的细节无关紧要。同样，只要客户看到的行为保持不变，实现方式应该可以改变类的基本表示法。例如，如果不使用数组就能实现 `ArrayList` 的行为，那么实现这种策略很可能对客户没有影响。

然而，在 Java 中，定义类就给基本表示法强加了限制。定义新类时，该类不仅继承了其超类的行为，而且也继承了超类的实现和表示法。为了更有效地区分行为和表示法的细节，Java 使用一种称为接口的结构，它指定类必须实现的方法，而不必像前面那样约束策略。

从第 9 章开始就已经在使用接口，第 9 章介绍了接口 `GFillable`、`GScalable` 和 `GResizable` 来描述共享特定行为特征的类。例如，将自身声明为 `GFillable` 的类必须实现方法 `setFilled`、`isFilled`、`setFillColor` 和 `getFillColor`，因此允许客户依赖这种行为。实际上，这些方法的实现方式留给了单个类。

一般来说，使用接口可以提供更大的灵活性，同时比通过类扩充继承行为强加的约束要少。特别是接口可以指定不符合类层次结构的行为，在这种层次结构中，每个类只有一个超类。相反，单个类可以实现许多不同接口。这样做，可以与其他类共享行为，而不必依赖于扩充子类和继承。

在许多情况下，只使用类扩充不可能实现接口的效果。例如，考察 `acm.graphics` 程序包中的 `GFillable` 和 `GResizable` 接口。创建单个图形类扩充的抽象类，完全可以实现这些接口的效果。因此，可以创建名为 `FillableShape` 的抽象类，让 `GRect`、`GOval`、`GArc` 和 `GPolygon` 都扩充该类。同样，可以定义名为 `ResizableShape` 的抽象类，让 `GRect`、`GOval` 和 `GImage` 扩充类。如果要同时实现其中两个层次结构，就会出现问题。如果每个可调整大小的图形都是可填充的，可以让 `ResizableShape` 成为 `FillableShape` 的子类。但这种策略不行，因为 `GArc` 是可填充的，

但不能调整大小，而 `GImage` 可以调整大小，但不能填充。这种不对称说明 `FillableShape` 类和 `ResizableShape` 类都不是对方的子类。

Java 的接口机制允许类实现多个接口，从而解决了这个问题。例如，`ArrayList` 类实现称为 `List` 的一般接口，该接口指定定义 `ArrayList` 类行为的方法，但不约束其表示法。这种策略的好处在 13.3 节中非常明显。

13.2 HashMap 类

在全面介绍 Java 集合架构之前，有必要先介绍 `HashMap` 类，事实证明它在许多应用程序中都非常有用。`HashMap` 类实现映射的抽象思想，映射是键与值之间的结合关系。键是对象，它只会在映射里出现一次，所以可以用来确定值；值也是与特定键相关的对象。相同的值可以在映射里出现多次；然而，每个键是独一无二的。`ArrayList` 是 `List` 的特殊实现方式，与此类似，`HashMap` 也是更通用的 `Map` 接口的特殊实现方式。

映射的概念在现实中也有许多相似物。其中，最常见的是字典，它里面每个单词与对应的定义相关。在字典中，单词表示键，定义表示值。典型的大学数据库使用相同的策略来记录学生信息。在这种情况下，键是确定每个学生的唯一的 ID 号。对应的值可以是 `Student` 类的实例，这个 `Student` 类与第 6 章定义的 `Student` 类相似，都是用更多数据字段记录更详细的信息。

13.2.1 映射的简单示例

1963 年，美国邮局(现在的美国邮政管理局)引入了一组表示美国各州、地区和地方的二字母代码。这些代码(省略较小岛区)如图 13-2 所示。虽然可以向相反的方向转换，但本小节只考虑将二字母代码转换为州名的问题。因此，如果要将图 13-2 中的数据表示为映射，键就是二字母代码，值就是对应的州名。

AK	Alaska	IA	Iowa	MT	Montana	PR	Puerto Rico
AL	Alabama	IL	Illinois	NC	North Carolina	RI	Rhode Island
AR	Arkansas	IN	Indiana	ND	North Dakota	SC	South Carolina
AZ	Arizona	KS	Kansas	NE	Nebraska	SD	South Dakota
CA	California	KY	Kentucky	NH	New Hampshire	TN	Tennessee
CO	Colorado	LA	Louisiana	NJ	New Jersey	TX	Texas
CT	Connecticut	MA	Massachusetts	NM	New Mexico	UT	Utah
DC	District of Columbia	MD	Maryland	NV	Nevada	VA	Virginia
DE	Delaware	ME	Maine	NY	New York	VT	Vermont
FL	Florida	MI	Michigan	OH	Ohio	WA	Washington
GA	Georgia	MN	Minnesota	OK	Oklahoma	WI	Wisconsin
HI	Hawaii	MO	Missouri	OR	Oregon	WV	West Virginia
ID	Idaho	MS	Mississippi	PA	Pennsylvania	WY	Wyoming

图 13-2 美国各州的二字母代码

映射的目的是以某种方法实现键与值之间的关联，这种方法可以方便地添加新的键/值对或找到现有键的值。例如，如果用图 13-2 中的数据初始化映射，可以向映射查询代码 `HI` 相对应的值。假设已经正确地对映射进行了初始化，它就会将报告 `HI` 对应 Hawaii 州。

可以使用 `HashMap` 类来表示图 13-2 中的二字母邮递区号表。第一步是创建一个空映射，可以向这个空映射中添加单个数据项。从第 11 章对 `ArrayList` 的介绍可以知道，Java 5.0 允许

在尖括号内指定类型来指定集合包含什么类型。Map 要求指定两种类型的参数。第一个是用来表示键的类，第二个是用来表示值的类。在二字母州代码映射中，这两个类都是 String。创建空映射所需的声明如下所示：

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

该声明使用 HashMap 构造函数来创建空的将字符串映射为字符串的 HashMap 对象。然而，stateMap 变量更通用的类型是 Map<String, String>。在本章稍后将会知道，声明使用更通用的接口类型有好处，尽管需要构造特定类的初始值。

调用 put 方法以及键和值，可以将新的关联添加到映射。因此，要初始化变量 stateMap 来包含所需的键/值对，首先可以写

```
stateMap.put("AK", "Alaska");
```

然后使用语句

```
stateMap.put("WY", "Wyoming");
```

在约 50 行后结束。检索与键相关联的值的对应操作称为 get。如果调用

```
stateMap.get("NV");
```

方法返回字符串 “Nevada”。如果提供的键不在映射中，调用 get 方法会返回值 null。

作为工具使用 HashMap 类，必需简单介绍一下 get 和 put 方法，不必知道如何实现键与值之间的关联，虽然许多像 HashMap 这样的类的值来自于这种关联。作为客户来说，重要的是它能这样做。

然而，如果要更深入学习计算机科学，对基本实现方式心存好奇是一个好现象。例如，可能想知道 get 和 put 的效率如何，这样就可以知道它要花多长时间来查找指定键的值。事实证明，HashMap 类中使用的实现策略非常有效率，这种实现方式可以保证 get 和 put 方法的平均运行时间与映射中键的数量无关。如果使用第 12 章介绍的 big-O 表示法表达这个意思，get 和 put 方法都是 $O(1)$ ，这是希望能够实现的最好结果。

在讨论 HashMap 类为什么如此有效率之前，有必要更深入地研究一下将双字母州代码转换为对应的州名这个问题。如何尽可能有效率地查找这些代码呢？

13.2.2 探讨可能的实现策略

认识这种算法问题的方法之一是试验不同的实现策略，看看有没有什么方法可以让它变得更快。如果只关心将双字母州代码转换为州名，首先可以将数据放入两个数组，让一个数组里特定双字母代码的索引数匹配另一个数组里对应的名称。列出它们的元素就可以自动初始化数组。例如，下面的语句初始化双字母代码的常量数组：

```
private static final String[] STATE_CODES = {
    "AK", "AL", "AZ", "AR", "CA", "CO", "CT", "DE", "DC", "FL", "GA", "HI", "ID",
    "IL", "IN", "IA", "KS", "KY", "LA", "ME", "MD", "MA", "MI", "MN", "MS", "MO",
    "MT", "NE", "NV", "NH", "NJ", "NM", "NY", "NC", "ND", "OR", "OK", "OR", "PA",
    "PR", "RI", "SC", "SD", "TN", "TX", "UT", "VT", "VA", "WA", "WV", "WI", "WY"
};
```

可以用相同方法创建一个名为 STATE_NAMES 的数组，代码如下：

```
private static final String[] STATE_NAMES = {
    "Alaska", "Alabama", "Arkansas", "Arizona", "California", "Colorado",
    "Connecticut", "District of Columbia", "Delaware", "Florida", "Georgia",
    "Hawaii", "Iowa", "Idaho", "Illinois", "Indiana", "Kansas", "Kentucky",
    "Louisiana", "Massachusetts", "Maryland", "Maine", "Michigan", "Minnesota",
    "Missouri", "Mississippi", "Montana", "North Carolina", "North Dakota",
    "Nebraska", "New Hampshire", "New Jersey", "New Mexico", "Nevada", "New York",
    "Ohio", "Oklahoma", "Oregon", "Pennsylvania", "Puerto Rico", "Rhode Island",
    "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah", "Virginia",
    "Vermont", "Washington", "Wisconsin", "West Virginia", "Wyoming"
};
```

声明这些数组之后，就可以写方法 getStateName，它读取一个二字母代码，返回对应的州名，代码如下：

```
private String getStateName(String code) {
    int index = linearSearch(code, STATE_CODES);
    if (index == -1) {
        return null;
    } else {
        return STATE_NAMES[index];
    }
}
```

该代码使用第 12 章里的 linearSearch 方法在 STATE_CODES 数组中查找指定的代码。如果它存在，getStateName 方法返回 STATE_NAMES 中相应位置的元素。由于州名在两个个数组里的顺序都相同，所以那个元素会包含州名称，该州对应于 linearSearch 找到的代码。如果指定代码在数组中不存在，getStateName 方法返回 null。相关项在相同索引位置出现的数组称为平行数组。

这种实现方式计算的复杂性与 linearSearch 方法计算的复杂性相同。如第 12 章所述，在最坏情况下线性搜索方法必须查看数组中的每个元素，这样它的 big-O 表示法就是 $O(N)$ 。STATE_CODES 数组以字母顺序排列，因此可以使用折半算法，利用这一点在某种程度上可以提高性能。用对 binarySearch 的调用取代对 linearSearch 的调用可以将计算的复杂性减少为 $O(\log N)$ 。虽然那个级别的效率也非常适用于更大的数组，但它仍然达不到 HashMap 可以达到的 $O(1)$ 性能。怎样才能做得比折半搜索更好？

如果考察一下将二字母代码转换为州名的特定示例，也许就有了答案。学习 13.2.3 节之前自己先尝试一下。

13.2.3 实现 $O(1)$ 性能

在上面的特定问题中，利用美国各州邮递区号的长度只有两个字母这一点，可以实现所需的 $O(1)$ 性能。要做的就是将州名称存储在一个二维数组里，这个数组里的索引直接由二字母州代码中的字符计算。

实现这种策略的第一步是创建一个二维数组来保存州的名称。写声明

```
private String[][] stateMap = new String[26][26];
```

可以做到这一点，它创建了一个 26×26 的数组，其中每个值都初始化为 null。

然后可以使用下面的方法插入各州的代码：

```
private void putStateName(String code, String name) {
    char c1 = code.charAt(0);
    char c2 = code.charAt(1);
    stateMap[c1 - 'A'][c2 - 'A'] = name;
}
```

该方法将州名指派到二字母代码中第一个字母表示的行和第二个字母表示的列。如果采用这种方法，可以用对 putStateName 的一连串调用来初始化 stateMap 数组，代码如下：

```
putStateName("AK", "Alaska");
putStateName("AL", "Alabama");
```

该设计中，getStateName 方法如下所示：

```
private String getStateName(String code) {
    char c1 = code.charAt(0);
    char c2 = code.charAt(1);
    return stateMap[c1 - 'A'][c2 - 'A'];
}
```

方法所做的就是将二字母代码分解为字符，然后查找二维 stateMap 数组中对应位置的元素。如果数组元素对应州代码，它必须由某个 putStateName 调用初始化。如果没有，该元素必须包含初始化时给它指定的 null 值。在这两种情况下，getStateName 返回的值正是所需的值。选择数组元素占用的时间是 $O(1)$ 。

13.2.4 散列法思想

虽然 HashMap 类没有完全使用 13.1 节介绍的方法，但也有某些类似之处。使用二维数组表示法来表示二字母代码与州名之间的映射，可以实现 $O(1)$ 性能，因为 getStateName 知道从哪里查找。如果二字母州代码在 HashMap 类中，它一定位于代码中两个字符指定的行和列。知道查找的位置就省去了搜索，而这正是过程变慢的原因。

要在映射更常见的语境中也能知道查找的位置，为了理解如何实现这一目标，有必要考察一下字典。如果要在字典中查找单词，可以从第一个条目开始，接着是第二个，第三个，直到找到单词为止。当然，这种策略是线性搜索算法，它需要 $O(N)$ 时间。相反，可以使用二进位算法，从中间打开字典。将搜索的单词与页面上的第一个条目进行比较，就可以确定查找的单词是在前一半还是在后一半。将这种算法重复应用到字典越来越小的部分，可以找到想要的单词，这样比使用线性搜索方法更快。然而，真正查字典时可能不会应用这两种策略。大多数字典旁边都有翻阅标记，它表示每个字母的条目出现的位置。可以在 A 部分查找以 A 开头的单词，在 B 部分查找以 B 开头的单词，等等。

13.2.5 散列码

HashMap 类使用的策略在细节方面不同，但在概念上相似。HashMap 实现方式也使用键的值来计算查找的位置。它首先将键转换为称为散列码的整数。每个 Java 对象都有散列码，

只是用户通常不知道而已。例如，州缩写列开头的字符串“AK”的散列码是 2090，列结尾“WY”的散列码是 1786。知道这些字符串为什么有散列码对编写程序没有任何帮助。唯一重要的是记住每个 Java 对象都有散列码。

HashMap 类使用键的散列码来确定在哪里查找那个值。例如字符串“AK”的散列码是 2090，HashMap 实现方式就在保存散列码 2090 的位置查找“AK”。查找键“WY”时，就在保存散列码 2786 的位置查找。查找这些键的一种可能策略本章稍后介绍，但基本思想在理解所有细节之前就应该很清楚。因为散列码告诉 HashMap 实现到哪里查找特定键，所以它不会经历只会让速度变慢的搜索过程，就可以找到指定的键。

大多数情况下是这样，但不总是这样。有时，HashMap 实现确实必须搜索键。这种搜索没有完全背离 $O(1)$ 性能目标，其原因是这种实现不必搜索很远。而且，从本章稍后对这种实现方式更详细的介绍中可以知道，如果 HashMap 开始变得过多，HashMap 可以动态调整其内部表示法以减少搜索。

13.2.6 复制散列码

尽管每个 Java 对象都有散列码，但不同的对象有时会有相同的散列码，认识到这一点很重要。毕竟，对象可能比表示其散列码的 32 位整数要多。例如，包含 7 个小写字母的字符串数是 26^7 ，结果是 8 031 810 176。然而，可能的散列码数是 2^{32} ，它的值只有 4 294 976 296。因为 7 个字母的组合数比这些整数要多，所以至少有些 7 个字母的字符串有相同的散列码。

在数学中，要将较大的组映射到一个较小的组上时，必须使用复制，这种原理称为文件架原理(pigeonhole principle)，名称来源于将信放入桌上的文件架里这一思想。如果信比文件架多，就必须在有些文件架里放置多封信。

假设可能的散列码数量足够大，遇到两个共享相同散列码的对象这种情况不会经常出现。例如，如果查看包含 10 个或更少字母的英语单词，Java 的 hashCode 方法只产生了两对共享相同散列码的单词：*buzzards* 与 *rights*, *hierarch* 与 *crinolines*。而找到匹配散列码的可能性很小，文件架原理只说明存在这种可能。HashMap 的代码必须考虑到这种可能性，当两个对象有相同的散列码时，代码应该找到一种方法继续查找。

计算机科学的研究文献充满了解决这个难题的策略。最简单的策略是保留一列对象，这些对象的散列码在 HashMap 中共享相同的空间；13.2.7 小节会介绍使用这种策略的一种简单实现方式。只要重复很少，这些列通常只有一项。因此，在 HashMap 中查找项通常只需要查看已有的一项。然而，有时这一列有多项，但找到值所需的平均时间仍然是 $O(1)$ 。

13.2.7 映射类的简单实现

虽然 Java HashMap 类的完整实现超出了本书的范围，但如果查看一个简化版本——如图 13-3 中 SimpleStringMap 的代码——就很容易理解最近几节提出的问题。

图 13-3 中的 SimpleStringMap 类只实现了 get 和 put 操作，忽略了更复杂的方法。这些方法是 Java 集合架构中 Map 接口的一部分。而且，SimpleStringMap 类总是使用键和值的字符串，这说明它简化了使用参数化类型所涉及的复杂性。

```

/** This class implements a simplified version of a Map class for strings. */
public class SimpleStringMap {

    /** Creates a new SimpleStringMap with no key/value pairs */
    public SimpleStringMap() {
        bucketArray = new HashEntry[N_BUCKETS];
    }

    /**
     * Sets the value associated with key. Any previous value for key is lost.
     * @param key The key used to refer to this value
     * @param value The new value to be associated with key
     */
    public void put(String key, String value) {
        int bucket = Math.abs(key.hashCode()) % N_BUCKETS;
        HashEntry entry = findEntry(bucketArray[bucket], key);
        if (entry == null) {
            entry = new HashEntry(key, value);
            entry.setLink(bucketArray[bucket]);
            bucketArray[bucket] = entry;
        } else {
            entry.setValue(value);
        }
    }

    /**
     * Retrieves the value associated with key, or null if no such value exists.
     * @param key The key used to look up the value
     * @return The value associated with key, or null if no such value exists
     */
    public String get(String key) {
        int bucket = Math.abs(key.hashCode()) % N_BUCKETS;
        HashEntry entry = findEntry(bucketArray[bucket], key);
        if (entry == null) {
            return null;
        } else {
            return entry.getValue();
        }
    }

    /*
     * Scans the entry chain looking for an entry that matches the specified key.
     * If no such entry exists, findEntry returns null.
     */
    private HashEntry findEntry(HashEntry entry, String key) {
        while (entry != null) {
            if (entry.getKey().equals(key)) return entry;
            entry = entry.getLink();
        }
        return null;
    }
}

```

图 13-3 SimpleStringMap 类的实现

```

/* Private constants */
private static final int N_BUCKETS = 7;

/* Private instance variables */
private HashEntry[] bucketArray;
}

/* Package-private class: HashEntry */
/*
 * This class represents a pair of a key and a value, along with a reference
 * to the next HashEntry in the chain. The methods exported by the class
 * consist only of getters and setters.
 */
class HashEntry {

    /* Creates a new HashEntry for the specified key/value pair */
    public HashEntry(String key, String value) {
        entryKey = key;
        entryValue = value;
    }

    /* Returns the key component of a HashEntry */
    public String getKey() {
        return entryKey;
    }

    /* Returns the value component of a HashEntry */
    public String getValue() {
        return entryValue;
    }

    /* Sets the value component of a HashEntry to a new value */
    public void setValue(String value) {
        entryValue = value;
    }

    /* Returns the next link in the entry chain */
    public HashEntry getLink() {
        return entryLink;
    }

    /* Sets the link to the next entry in the chain */
    public void setLink(HashEntry nextEntry) {
        entryLink = nextEntry;
    }

    /* Private instance variables */
    private String entryKey;      /* The key component for this HashEntry */
    private String entryValue;    /* The value component for this HashEntry */
    private HashEntry entryLink; /* A reference to the next entry in the chain */
}

```

图 13-3 (续)

SimpleStringMap 实现使用的基本策略是，将键/值对存储到键散列码确定的列表。这些列表传统上称为存储区。存储区的数量由常量 N_BUCKETS 确定，它在该实现中定义为值 7，虽然这个值在实际中有点小。然而，使用较小的 N_BUCKETS 值，更容易图解数据的内部结构。

调用 get 或 put 时，方法首先要确定哪个桶对应指定的键。虽然桶的索引由散列码确定，但可能的散列码数量比存储区的数量要多。要将 hashCode 方法的结果转换为存储区数，最简单的方法是用散列码除以 N_BUCKETS，然后取余数。从 get 和 put 方法的第一行可以看出，需要使用散列码的绝对值，确保%运算符不会用于负数。

通俗地说，如果键的散列码转换返回桶的索引，计算机科学家称为键散列到存储区。两个或更多键散列到同一个存储区的情况——这各情况比找到两个有相同散列码的键常见——称为冲突。

实例变量 bucketArray 是 HashEntry 对象的数组。每个 HashEntry 对象都包含键和值，以及对下一个散列到该存储区的记录项的引用。每次用新键调用 put 时，实现方式会给链添加一个新的 HashEntry。图 13-4 显示了添加键“AK”、“AL”、“AR”、“AZ”和“CA”（这些键都散列到存储区）之后存储区链的状态。

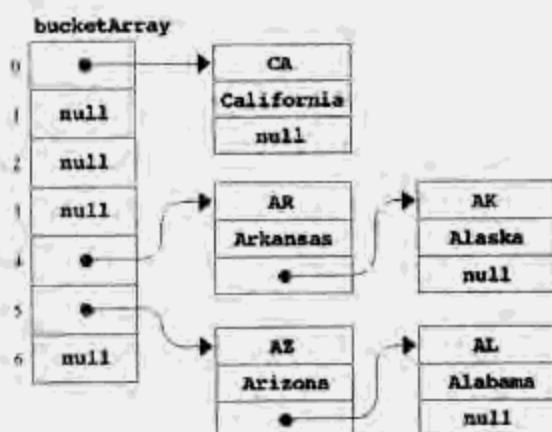


图 13-4 将键添加到存储区

图中的存储区链用各 HashEntry 对象中的链接连接而成，方法与第 7.4 节链接 Beacons of Gondor 相同。每条链都是称为链表的重要编程结构的示例，13.3 节会继续讨论链表。

13.2.8 实现 hashCode 方法

大多数情况下，不需要深入了解类实现其 hashCode 方法的细节。从整体上来看，隐藏细节是好事，如果 hashCode 方法注重隐藏的话。例如，Java String 类——少于 16 个字符的字符串——的 hashCode 方法在功能上与下面的方法相同：

```

public int hashCode() {
    int hash = 0;
    for (int i = 0; i < length(); i++) {
        hash = 31 * hash + charAt(i);
    }
    return hash;
}
    
```

虽然可以完成方法中的语句、理解每条语句的功能，但理解代码这么写的原因更具有挑战性。

在许多方面，编写有效的 `hashCode` 方法的任务与第 6 章介绍的编写有效的随机数生成器问题相似。创建随机数生成器的所需的数学知识超出了本书的范围。写好的 `hashCode` 方法也依赖于同样的数学原则。从整体上看，应该将编写 `hashCode` 方法的任务留给专家，就像自己产生随机数生成器是不明智的一样。

遗憾的是，即使通常可以避免写随机数生成器，但有时不得不写 `hashCode` 方法。如果实现一个类，有人可能很想将它作为键使用，那么就要实现那个类的 `hashCode` 方法。实现的时候，记住下面有效 `hashCode` 方法的标准很重要。

- 如果在相同对象上调用 `hashCode` 方法，必须返回相同的代码。这个要求是散列策略的本质特征之一。`HashMap` 代码知道寻找特定键的位置，因为它的散列码决定它要查找的位置。如果代码可以改变，`HashMap` 实现方式会将它放在一个地方，然后在另一个地方查找它。
- `hashCode` 的实现必须与 `equals` 方法的实现相一致，因为在内部，散列机制使用 `equals` 方法来比较键。本条件比第一条更有力。第一条说特定对象的散列码不能随意改变，而本条件中的新要求强化了第一条。它强调，任意两个由 `equals` 方法确认相等的对象必须有相同的散列码。
- `hashCode` 方法应该避免返回导致冲突的散列码。例如，对于 `String` 类而言，使用第一个字符的内部代码作为其散列码就不合适。如果这样做，每个以相同字符开头的字符串都会在 `HashMap` 里冲突，这样会极大地降低性能。
- `hashCode` 方法应该相对容易计算。如果写的 `hashCode` 方法需要花很长时间计算，就埋没了散列的主要优点，即基本算法运行速度非常快。

如果确实要实现 `hashCode` 方法，可以采取一种简单策略来达到这些标准。大多数情况下，定义的类都包括指定对象值的实例变量。要计算对象的散列码，可以计算其确定组件的散列码，然后以一种不增加冲突机率的方法将各部分联合起来。在这种情况下，第 11 章介绍的位运算符`^`特别有用。与`&`和`|`运算符不一样，`^`运算符没有减少其操作数的信息内容。因此，如果取对象组件的散列码，然后用位运算符(`^`)将它们结合起来，通常可以写出能够运行良好的 `hashCode` 方法。

常见错误

决不要重写 `equal` 方法的定义。除非同时重写 `hashCode` 方法，重写一个而不重写另一个方法总是会违背 `hashCode` 的设计标准，因此不可能使用这个类作为 `HashMap` 键。

例如，图 13-5 中的代码将两个有用的方法添加到第 6 章介绍过的 `Rational` 类。`equals` 方法可以用来确定两个 `Rational` 数是否相等，`hashCode` 方法允许使用 `Rational` 对象作为 `HashMap` 中的键。`hashCode` 的实现方式采用前一段里提到的策略。它首先计算实例变量 `num` 和 `den` 的散列码，然后使用位运算符(`^`)将这些值结合起来。实际中最好是用其中的某个值乘以一个小的素数，如图 13-5 所示。

图 13-5 中的代码也强调了 Java 中类设计的一个重点。`equals` 方法非常有用，以至于希望它是第 6 章 `Rational` 类的初始实现的一部分。根据用户现在的知识，`equals` 方法本身不难实现。问题是重写 `equals` 方法而没有重写 `hashCode` 方法，这样的编程实践非常糟糕。如果在定义类

的过程中出现了这种错误，就不能使用该类作为 hashMap 中的键。Rational 类的原始定义不包括 equals，因为当时没有准备编写相关的 hashCode 方法。

```

/*
 * Returns true if this rational number is equal to the specified
 * object. Because the implementation for the Rational class always
 * reduces fractions to lowest terms, all this method needs to do is
 * make sure that both the numerators and denominators match.
 * @param obj The value to which this object is being compared
 * @return A boolean value indicating whether this number is equal to obj
 */
public boolean equals(Object obj) {
    if (obj instanceof Rational) {
        Rational r = (Rational) obj;
        return this.num == r.num && this.den == r.den;
    } else {
        return false;
    }
}

/*
 * Returns a hash code for this Rational object. That hash code is
 * derived from the hash codes for its two components. This design
 * decision ensures that two Rational objects that are equal will have
 * the same hash code.
 * @return An integer hash code for this object
 */
public int hashCode() {
    return new Integer(num).hashCode() ^ (37 * new Integer(den).hashCode());
}

```

图 13-5 Rational 类的 equals 和 hashCode 的实现

13.2.9 调整存储区的数量

虽然 hashCode 方法的特征是一个因素，但很明显，HashMap 中发生冲突的可能性也取决于存储区的数量。如果存储区的数量偏小，冲突自然会发生得更频繁。冲突降低了 HashMap 的效率，因为它们强迫 get 和 put 方法搜索更长的链。随着 HashMap 中的存储区不断被填满，冲突的次数也在增加，HashMap 的性能不再是 $O(1)$ 。

保持 HashMap 高性能的最好策略是，现有存储区填满以后，动态增加存储区的数量。例如，可以设计一种实现方式，当键的数量与存储区的数量之间的比率超过某个极限时，就分配更大的存储区数组。遗憾的是，如果增加存储区的数量，所有存储区的号码也会改变，这就意味着扩充存储区数组的代码必须从原来的 HashMap 中重新将所有键输入到新的 HashMap。这个过程称为重新散列。虽然重新散列要消耗时间，但它很少发生，因此对应用程序的平均运行时间影响不大。本章练习 4 中将有机会实现 SimpleStringMap 类的重新散列算法。

13.3 Java 集合架构

虽然关于 HashMap 类还有很多内容需要介绍，但在 Java 集合架构的语境中作为整体描述其他特征更有意义。其中的介绍会紧密结合其他类中的类似特征。虽然 ArrayList 和 HashMap 类本身很重要，但 Java 集合架构的真正力量是它允许更抽象地考察集合。

13.3.1 Java 集合架构的结构

构成 Java 集合架构的主要类和接口如图 13-6 所示。图中的方框对应 Java 中可用的 3 种不同结构。名称是 Java 代码使用的标准字体的方框表示类定义。名称是斜体字的方框表示抽象类，它们不能自己构造，但可以通过继承将数据结构和方法传递给它们的子类。图 13-6 既包括具体类也包括抽象类，所以用户对图的某些部分应该很熟悉。图 13-6 中新的类是表示接口的方框，它们标记为“interface”。用来标记类与其超类之间关系的直线是实线，而类与接口之间的直线是虚线。

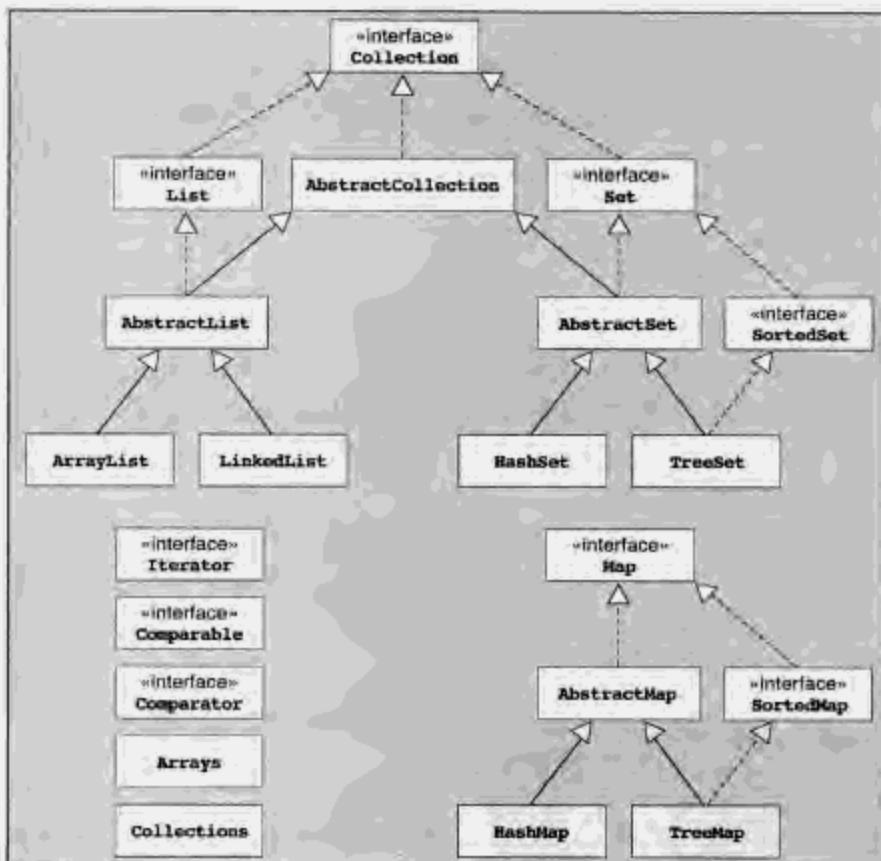


图 13-6 Java 集合架构中的类和接口

接口关系与类层次结构没有联系。虽然每个类只有一个超类——类继承超类的行为，但单个类可以实现任何数量的接口。然而，这些接口形成了自己的层次结构，如图 13-6 所示。如果一个接口扩充了现有接口，新的接口就是原来接口的子接口，就像子类扩充超类一样。所有实现子接口的类也自动实现原始接口。

为了说明这些关系在 Java 集合架构中如何结束，有必要图解某个具体类(例如图 13-6 左边的 `ArrayList` 类)的继承关系。按照类层次结构，`ArrayList` 类是 `AbstractList` 类的子类，`AbstractList` 类是 `AbstractCollection` 的子类。这种继承层次结构中的实线表示每个 `ArrayList` 对象都是 `AbstractList` 和 `AbstractCollection`。而且，`ArrayList` 类继承了其超类里定义的方法，而不需要自己实现这些方法。

如果看看表示接口结构的虚线就会知道, `ArrayList` 实现 `List` 和 `Collection` 接口。它实现 `List` 接口是因为它的超类实现 `List` 接口。可以确定 `ArrayList` 以两种方法实现 `Collection`。一方面, 实现 `List` 接口的所有类也实现 `Collection` 接口, 因为 `List` 是 `Collection` 的子接口。另一方面, `AbstractList` 类继承了其超类 `AbstractCollection` 的结构和行为, 而 `AbstractCollection` 直接实现 `Collection` 接口。

虽然本文没有特别关注类图, 但也要注意, 这些图使用的符号与统一建模语言(Unified Modeling Language, 通常简写为 UML)使用的符号是一致的。UML 图已经通过国际标准组织认可, 这说明这样的图已经被全球广泛接受。然而, UML 图比本书中使用的有限示例更有表现力, 它可以表示超出简单继承之外的许多其他关系。

图 13-6 中的另一个有趣特征是, Java 集合架构中的类属于 3 个不同的群集。最上面是一个层次结构, 其中包含实现 `Collection` 接口的所有类, 例如现在很熟悉的 `ArrayList` 类。右下角是一组类, 它们实现 `Map` 接口, 如 `HashMap` 类。左下角是一组类和接口, 它们超出了该层次结构。接下来几节将详细介绍该架构的不同部分。

13.3.2 Collection 层次结构

Java 集合架构中最大的一组类都是从图 13-6 上面的 `Collection` 接口传下来的。在 Java 5.0 中, 通过指定尖括号内类的名称, 可以指定对象的类型, 如 `ArrayList<Integer>`。相同的模式可以应用于 `Collection` 层次结构中的所有类, 因此, 写 `LinkedList<String>` 就可以指定字符串的链表的类型。如第 11 章所述, 在方法的一般描述中, 值的类型设计为 `<T>`。Java 的较早版本不允许使用类型参数, 只能使用 `Object`。

由于 `ArrayList` 实现 `Collection` 接口, 因此它必须实现该接口中的所有方法。`Collection` 接口中最重要的方法如图 13-7 所示。通过第 11 章对 `ArrayList` 类的介绍, 图中大多数方法都比较熟悉了。唯一的新方法是 `iterator`, 它也应用于 `ArrayList` 类, 尽管前面没有介绍它。`iterator` 方法和它依赖的 `Iterator` 接口将在 13.3.4 小节介绍。

<code>boolean add(<T> value)</code>	将指定的值添加到集合。如果集合改变, 方法返回 true
<code>boolean remove(<T> element)</code>	删除指定元素的第一个实例。如果有的话: 如果找到匹配的项, 值是 true
<code>void clear()</code>	从集合中删除所有值
<code>int size()</code>	返回集合中值的个数
<code>boolean contains(<T> value)</code>	如果集合包含指定的值, 返回 true
<code>boolean isEmpty()</code>	如果集合中没有元素, 返回 true
<code>Iterator iterator()</code>	返回允许客户查看集合中值的迭代器

图 13-7 Collection 接口指定的有用方法

然而, 图 13-7 中的 `Collection` 接口遗漏了 `ArrayList` 类中的一些方法, 如 `get` 和 `set` 方法, 以及允许在特定位置添加值的 `add` 方法。这些方法不是 `Collection` 接口的一部分, 因为它们在这个抽象级别没有意义。能够在特定索引位置添加新值, 首先取决于有索引位置的集合。

Collection 接口没有提供这种保证。索引位置的思想出现在 List 接口级别，该接口描述对有序集合的操作。更通用的 Collection 接口必须满足无序集合的需要，它们在 Java 架构中称为集合(set)。

同样，在学习数学的过程中，也遇到过这种集合。通常，集合是对象的无序收集，其中每个对象最多出现一次。Collection 接口中方法符合集合的要求，就像符合列表一样，尽管这些操作的意义可能随着域改变而改变。如果将 add 方法应用于列表，值会插入到最后。因为集合是无序的，对于集合而言词组“最后”没有什么意义，但还是可以添加新元素。如果连续两次将相同的值添加到列表，两个副本都会添加到列最后。在集合中，用已经在集合中出现过的值调用 add 无效，因为每个值只能出现一次。这种行为上的不同可以解释 add 方法为什么返回值。在列表中，add 方法总是返回 true，这导致结果在某种程度上没有用。在集合中，可以使用 add 方法的结果来确定集合是否包含该值。

13.3.3 区分行为和表示法

Collection 层次结构中的第一条分界线在列表和集合之间，如 13.3.2 小节所述。这种划分主要是行为上的划分。列表和集合的行为不同。列表中的元素是有序的，而集合中的元素是无序的。因此，在列表中说索引位置有意义，而在集合中这样说就没有意义。

然而，层次结构中下一级别的划分是表示法。在 Java 集合架构中，`AbstractList` 类和 `AbstractSet` 类被划分为两个具体子类。`ArrayList` 和 `LinkedList` 类为 `List` 接口提供两种不同实现，而 `HashSet` 和 `TreeSet` 类为 `Set` 接口提供不同实现。从这些类实现的方法以及这些方法的效果来看，`ArrayList` 和 `LinkedList` 类在功能上是一样的。虽然有一些行为上的差异——最明显的就是 `TreeMap` 实现 `SortedSet` 接口，而 `HashMap` 不实现 `SortedSet` 接口——`Map` 接口的两种实现方式在功能上也非常相似。

这些成对的类之间的不同点主要在于它们的实现方式，更具体地说，在于每个类用来存储集合的值的表示策略。例如，`ArrayList` 实现存储内部数组中的元素。相反，`LinkedList` 实现使用图 13-3 中 `SimpleStringMap` 实现使用的风格，利用引用链接将对象连接在一起。顾名思义，`HashSet` 实现使用散列码来确定集合中的成员。`TreeSet` 实现使用一种称为二进制树形网络的数据结构来存储元素。二进制树形网络超出了本书讨论的范围，如果继续学习计算机科学，就会学到它们。

这些成对的类在实现方式上各有不同，这就提出了一个有趣的问题。如果客户不那么关心实现的细节，那么提供相同基本模型的多种实现方式就显得不合适。为什么不选择一种列表实现，命名为 `List`，然后让客户依赖它呢？这样就会消除 `ArrayList` 类和 `LinkedList` 类之间的不同，降低集合架构的复杂性。

允许客户选择一种特定实现方式的原因是，`ArrayList` 和 `LinkedList` 有不同的性能特征，尽管它们共享相同的抽象行为。在 `ArrayList` 前面添加新值需要 $O(N)$ 时间。在链表前面添加元素可以在 $O(1)$ 时间内完成。相反，选择索引位置 k 上的元素会降低这些性能衡量标准。在 `ArrayList` 中，选择元素总是常量时间操作。然而，`LinkedList` 实现必须跟踪前面元素里的链接来找到位置 k 上的元素，这就意味着这个操作在最坏情况下的性能是 $O(N)$ 。根据哪种类型的操作在应用程序中占支配地位，可以选择一种或另一种列表表示法使程序的行为最优化。

Java 集合架构的好处之一是, 它不强迫选择特定表示法, 除了在对构造函数的调用中之外。例如, 如果要写 `alphabetize` 方法, 它用一列字符串作为参数, 最好这样写方法头:

```
public void alphabetize(List<String>)
```

而不要将参数指定为 `ArrayList` 或 `LinkedList`。因为这两个类实现 `List` 接口, 将参数声明为 `List`, 意思就是它可以包含这些具体的列表类。

将 `ArrayList` 和 `LinkedList` 类推广到公共架构的能力完全取决于 Java 中定义接口的能力。虽然 `ArrayList` 和 `LinkedList` 类的确共享一些它们从 `AbstractList` 类继承来的方法, 但这两个类的基本表示法完全不同, 很难在标准的继承模型下统一它们。然而, 很容易使用公有方法的通用名称, 并坚持每个类实现相同的概念模型这一观点。

13.3.4 迭代器

考察 `HashSet` 和 `TreeSet` 类之间的差别之前, 必须理解 Java 集合架构如何解释迭代器的概念。从第 4 章开始就在以 `while` 和 `for` 循环的形式使用迭代器。迭代集合中的元素, 每次一个元素, 是一种类似的操作。

在类似 `ArrayList` 的有序集合中, 可以使用 `for` 循环来循环元素。例如, 如果将变量 `nameList` 声明为 `List<String>`, 那么就可以使用下面的代码打印元素(每行一个元素):

```
for (int i = 0; i < nameList.size(); i++) {
    String name = nameList.get(i);
    println(name);
}
```

使用 `ArrayList` 的时候, 可能会使用 `for` 循环, 因为元素有索引数。如果使用 `Set` 对象, 这种策略就无效, 因为集合中找不到第一个元素。

为了迭代所有集合——有序的或者无序的集合——中的值, Java 集合架构包含 `Iterator` 接口, 它指定两种基本方法:

- `hasNext` 断言方法, 如果集合有迭代器没有提供的值, 它返回 `true`。
- `next` 方法, 它返回集合中的下一个元素。

和 `Collection` 层次结构中的类和接口一样, `Iterator` 类属于 Java 5.0 中的参数化类型, 这就意味着可以在尖括号内包含元素类型。因此, 如果有包含字符串的集合, 可以使用 `Iterator<String>` 来迭代该集合的元素。

下面的代码示例用来说明如何使用 `Iterator` 工具复制本节前面介绍的 `for` 循环的结果:

```
Iterator<String> iterator = nameList.iterator();
while (iterator.hasNext()) {
    println(iterator.next());
}
```

这个循环与前面的循环有相同的结果, 但它不再依赖索引数, 因此可用于集合。

for 语句的扩充语法

```
for (type variable : collection) {
    statements
```

其中：

type 是集合元素的类型 *variable* 是呈现每个值的变量；
collection 是实现迭代的集合对象；
statements 是重复的语句。

Java 中的迭代器非常有用，因此 Java 5.0 包含新的语言特征来支持它们。对于所有实现 iterator 方法的集合类，Java 5.0 都可以使用新的 for 语句(如上边语法框所示)来表示集合中的迭代。例如，for 语句的新语法允许简化输出 nameList 中各元素的代码，代码如下：

```
for (String name : nameList) {
    println(name);
}
```

以这种形式书写代码时，迭代器不会显式出现，即使在实现中需要用到它。

因为所有列表元素都是有序的，所以 ArrayList 的迭代器按它们在列表中出现的顺序返回元素。在无序列表中，没有非常明确的理由要在另一种顺序基础之上选择一种顺序。迭代器可能必须在内部选择某种顺序，以避免跳过元素或将某元素返回两次，但客户不必理解这种顺序。

为了说明集合类在确定迭代顺序方面的灵活性，可以再考察一下本章前面转换二字母州代码的示例。在基于 HashMap 的实现中，变量 stateMap 被声明为 Map<String, String>，然后使用一长串对 put 方法的调用来初始化，所有调用都只替代二字母代码和州名即可，代码如下：

```
stateMap.put("AK", "Alaska");
```

虽然可以使用迭代器工具来显示两个字母代码和州名称的表格，只有当表格的行按字母顺序排列以方便找到特定项时，这种表格才有用。编写代码创建这样的表格之前，需要知道迭代 HashMap 中的键是否会产生按相应顺序排列的键。

可以写简单的测试程序来研究这个问题。虽然 Map 类本身没有迭代器，但是映射中包含的键形成了一个集合。在 HashMap 中，每个键最多出现一次，键和键之间没有明显的顺序关系。Map 接口指定 keySet 方法，它返回键的集合。然后可以使用这个集合来创建迭代器。

下面的方法显示了指定映射中的键，并将一些州代码压缩到了同一个输出行：

```
private void listKeys(Map<String, String> map, int nPerLine) {
    String className = map.getClass().getName();
    int lastDot = className.lastIndexOf(".");
    String shortName = className.substring(lastDot + 1);
    println("Using " + shortName + ", the keys are:");
    Iterator<String> iterator = map.keySet().iterator();
    for (int i = 1; iterator.hasNext(); i++) {
        print(" " + iterator.next());
        if (i % nPerLine == 0) println();
    }
}
```

例如，如果将 stateMap 初始化为包含图 13-2 中二字母州代码的 HashMap，调用

```
listKeys(stateMap, 13);
```

会产生，如图 13-8 所示输出。

如果这些键的顺序有逻辑，对于查看运行示例的人而言，他们也看不见这个逻辑。因为集合是无序的，实现可以自由使用任何有利于实现方式的策略。在 `HashMap` 中，循环键最简单的方法是进入每个存储区，然后循环存储区里面的键，每个键各一次。那么键的顺序就取决于将它们分配到存储区的方法，也就是依次取决于它们的散列码。

如果用 TreeMap 代替 HashMap，情况就不一样。改变之后再运行相同的方法，输出如图 13-9 所示。



图 13-8 使用 HashMap 的输出



图 13-9 使用 TreeMap 的输出

本示例中，键是按字母顺序排列的，这一定会让代码表更可用。造成这种差别的原因是 TreeMap 类实现 SortedMap 接口，该接口确保迭代会按自然顺序进行。

剩下的唯一问题就是用什么来定义顺序。对于许多类型而言，答案很明显。各种数字类型可以按数字顺序排序。字符串和字符可以按字母顺序排序。但像 GObject 这样的类该按什么排序呢？有没有方法可以定义自然顺序，让它确定红色矩形应该出现在绿色椭圆之前还是之后？当然没有。因此，SortedMap 和 SortedSet 接口就是唯一与自然顺序类似的类。

Java 中，支持排序概念的类通过实现 Comparable 接口来体现。实现 Comparable 的类必须定义方法

```
public int compareTo(Object obj)
```

它将当前对象与 `obj` 进行比较，返回一个表示其相对顺序的整数。如果当前对象在 `obj` 之前，`compareTo` 方法会返回负整数。如果当前对象在 `obj` 之后，`compareTo` 返回正整数。如果两个对象相等，`compareTo` 返回 0。`String` 类实现这种方法，与包装器类用于所有数字类型一样。`compareTo` 方法定义对象间的有序关系，该顺序称为该类的默认顺序。

13.3.5 Arrays 和 Collections 方法库

虽然它们似乎是集合类层次结构中重要的链接，但名为 `Arrays` 和 `Collections`——不要与名为 `Collection` 的接口混淆——在 Java 集合架构中担当着不同角色。这两个类只导出作为通用工具的静态方法。`Arrays` 类导出的方法适用于数组类型；`Collections` 类导出的方法适用于所有集合。本节介绍 `Collections` 类中最有用的方法；用户自己完全可以掌握 `Arrays` 类中类似的方法。

图 13-10 显示了 Collections 类导出的一些最重要静态方法。从图中可以看出，列表开头是第 12 章介绍的两种算法方法——搜索和排序。Collections 类导出这些操作作为适用于所有列表的工具方法，这一点说明不必亲自写这些方法。例如，如果要给名为 examScores 的 ArrayList<Integer> 排序，只要调用

```
Collections.sort(examScores);
```

就可以了。使用程序包的好处就是，有人已经完成了实现和测试算法的艰巨工作；需要的时候，使用这些方法非常方便。

`sort` 和 `binarySearch` 方法通常使用默认顺序，该顺序由适用于其元素类型的 `compareTo` 方法确定。然而，`Collections` 类以第二种形式提供这些方法，在这种形式中，用户可以自己提供比较方法。要这样做，需要提供一个实现 `Comparator` 接口的对象。`Comparator` 接口与 `Comparable` 接口尽管有些类似，但两者并不完全相同。`Comparator` 接口指定方法

```
public int compare(<T> v1, <T> v2)
```

其中，`<T>` 表示 `Comparator` 接口使用的参数类型。和 `Comparable` 接口中的 `compareTo` 方法一样，`compare` 方法也会返回值，这个值的符号表示 `v1` 和 `v2` 之间的顺序关系。这两个接口之间的主要不同是，`Comparable` 描述的顺序用于本类中的元素，而 `Comparator` 描述的顺序可以应用于其他类的对象。

<code>static void binarySearch(List list, Object key)</code>	使用折半搜索在按默认顺序排序的列中寻找对象
<code>static void sort(List list)</code>	使用其默认顺序将列按升序排列
<code>static void binarySearch(List list, Object key, Comparator cmp)</code>	使用折半搜索有序列中的对象。该列使用 <code>cmp</code> 作为比较器函数
<code>static void sort(List list, Comparator cmp)</code>	使用 <code>cmp</code> 作为比较器函数将列按升序排列
<code>static Object min(List list)</code>	将列里最小的元素作为对象返回
<code>static Object max(List list)</code>	将列里最大的元素作为对象返回
<code>static void reverse(List list)</code>	倒转列中元素的顺序
<code>static void shuffle(List list)</code>	随机打乱列里的元素
<code>static void swap(List list, int i, int j)</code>	交换索引位置 <code>i</code> 和 <code>j</code> 上的两个元素
<code>static void rotate(List list, int n)</code>	将最后 <code>n</code> 个元素移动到列的开头(如果 <code>n</code> 是负数，就将前 <code>n</code> 个元素移动到结尾)
<code>static boolean replaceAll(List list, Object oldValue, Object newValue)</code>	用 <code>newValue</code> 取代 <code>oldValue</code> 的每个实例。如果有改变，返回 <code>true</code>

图 13-10 Collections 类导出的有用方法

可以使用 `binarySearch` 和 `sort` 的扩充形式来改变排序规则，改变由元素类型确定的默认顺序。图 13-11 中的 `SortIgnoringCase` 程序排序输入文件的行，并不区分大小写字母。字符串的默认顺序认为大小写非常重要，例如电影名 `THX 1138` 就在 `Tarzan` 之前，因为按照词典顺序，大写字母在小写字母之前。

```

import acm.program.*;
import acm.util.*;
import java.io.*;
import java.util.*;

/** This program sorts the lines of a file ignoring the case of letters */
public class SortIgnoringCase extends ConsoleProgram implements Comparator<String> {

    public void run() {
        println("This program sorts a file without regard to case.");
        BufferedReader rd = openFileReader("Enter input file: ");
        List<String> lines = readLineList(rd);
        Collections.sort(lines, this);
        Iterator<String> iterator = lines.iterator();
        while (iterator.hasNext()) {
            println(iterator.next());
        }
    }

    /*
     * Implements a string comparison method that ignores case.
     * This method implements the Comparator<String> interface.
     */
    public int compare(String s1, String s2) {
        return s1.toUpperCase().compareTo(s2.toUpperCase());
    }

    /*
     * Reads all available lines from the specified reader and returns a List<String>
     * containing those lines. This method closes the reader at the end of the file.
     */
    private List<String> readLineList(BufferedReader rd) {
        List<String> lineList = new ArrayList<String>();
        try {
            while (true) {
                String line = rd.readLine();
                if (line == null) break;
                lineList.add(line);
            }
            rd.close();
        } catch (IOException ex) {
            throw newErrorException(ex);
        }
        return lineList;
    }

    /* Include the code for openFileReader from Chapter 12. */
}

```

图 13-11 不区分大小写字母排序文件的程序

13.4 面向对象设计的原则

虽然设计良好的面向对象程序包提供的工具本身很有意思，但学习这样的程序包非常重要，原因之一是这样做可以提供自己设计所需的模型。在各个层次——实现特定功能的方法、提供一组方法的类或者包含一组类的程序包——都要对设计进行仔细思考。如果编程完全是为了

了自己消遣，那设计问题就可能不那么重要。然而，计算行业中这种情况极少。程序都是合作开发的。在所有大项目中，从事实现不同部分的编程人员必须同意共享一套公共的风格约定，同意将他们各自的部分合并到一起。如果没有合作协议，编程过程通常就会变得混乱不堪。因此，重要的是，一个人该如何设计方法、类和程序包，以便于其他编程人员使用。恰当设计的原则是什么呢？

要深刻理解这些原则，就要从不同角度理解客户和实现者如何看待方法、类或程序包提供的工具。客户只想知道什么操作可用，而不关心实现的细节。对于实现者而言，这些操作如何运行的细节都是基本问题。

然而，对于实现者而言，记住客户的需要这一点至关重要。如果要为其他人设计有效资源，需要平衡几个竞争因素。下面的标准是在程序包层次上讲的，但这些因素也适用于更低层次的类和方法。

- 统一。每个程序包应该定义一致的抽象化风格，并且有明确统一的主题。如果类不在那个主题内，它就不应该是程序包的一部分。
- 简单。程序包设计应该尽量简单以方便客户。在某种程序上基本实现方式本身很复杂，程序包必须尽量隐藏这种复杂性。
- 充分。对于要使用程序包的客户而言，程序包必须提供满足他们需要的类和方法。如果程序包遗漏了某个重要操作，客户可能会放弃使用，进而开发自己的工具。和简单一样重要，设计师必须避免将程序包简化到无用的地步。
- 灵活。设计良好的程序包应该很通用，能够满足不同客户的需要。为客户提供有限定义操作的程序包，远远不如可以在许多不同情况下使用的程序包有用。
- 稳定。导出为程序包一部分的类里定义的方法应该继续保持相同结构和效果，甚至是随着程序包一起发展。改变类的行为会强迫客户改变他们的程序，这样就降低了其效用。

下面几节将详细讨论这些标准。

13.4.1 统一主题的重要性

Unity gives strength.

—Aesop, The Bundle of Sticks, 6th century BCE

设计良好的程序包或类的主要特征是提供统一、一致的抽象化风格。在某种程度上，这条标准暗示应该选择程序包里的类及类里的方法，让它们反映一致的主题。例如，`acm.program` 程序包提供一组方便构造程序的类，所有这些类都是 `Program` 类的子类，`Program` 类形成该程序包的基础。同样，`acm.graphics` 程序包中的类以相同方式结合在一起，支持在画布上排列图形对象。这些类导出的每个方法在整体上都符合程序包的模型。

统一主题的原则也影响了类中方法的设计。类里的方法应该尽量以一致的方式运行。方法操作方面的差异会让客户使用类变得更加困难。例如，`acm.graphics` 程序包里的所有方法使用以像素指定的坐标和以度指定的角。如果某个类的实现者决定改变类里的某个方法，这种方法需要不同的度量单位，因为失去了一致性，所以客户不得不记住在各种情况下该使用什么单位。

13.4.2 简单和信息隐藏的原则

Embrace simplicity.

—Lao-tzu, The Way of Lao-tzu, c. 550 BCE

因为面向对象设计的主要目的是减少编程过程中的复杂性，所以简单是必需的设计标准。一般来说，程序包和类应该易于使用。基本的实现方式可以实现极其复杂的操作，但客户应该能够以一种简单、抽象的方法考察这些操作。

如第6章中有关 javadoc 这一节所述，使用 Java 程序包和类所需的文档通常在 Web 上。例如，如果想知道如何使用 Math 类，可以访问包含其 javadoc 描述的 Web 页面。如果该文档设计良好，它只会提供客户所需要知道的信息，决不会提供更多信息。对于客户而言，获得过多信息和获得过少信息一样糟糕，因为附加信息会让类变得更难以理解。通常，设计的真谛不在于它显示的信息，而在于它隐藏的信息。

设计类(甚至是单个方法)时，应该让客户尽可能少地看到实现的细节。在这方面，也许最好不要将类看作客户与实现之间的通道，而是要看作将它们分开的墙，如图 13-12 所示。

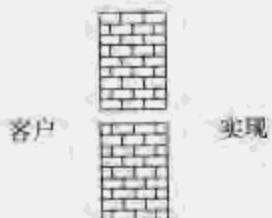


图 13-12 类在客户与实现之间

就像希腊神话中分开 Pyramus 和 Thisbe 这一对情人的墙一样，这面分开客户和实现的墙上的一个小裂缝，它让墙两边能够互相通信。然而，墙的主要目的是将两边分开。

因为图 13-11 所示的墙在两侧概念抽象化之间形成了边界，所以这面墙通常称为抽象化边界。最理想的情况是，类实现方式所包含的所有复杂性全部在墙面的实现这一侧。如果让复杂性不在客户那一侧，设计就是成功的。如第 5 章所述，保持限制到实现方式域的细节是信息隐藏过程的一部分。

对于类设计而言，信息隐藏的原则有重要的实践意义。写类时，应该确保没有在成为 javadoc 描述一部分的注释中显示实现的细节。特别是如果着重关注实现方式，很可能忍不住在公共注释中显露用来写这种实现方式的高明思想。要拒绝这种诱惑。文档的这个位置应该写其他编程人员阅读的内部注释。公有文档要考虑到客户的利益，应该只包含客户所需的信息。

同样，应该尽可能简单地设计类里的方法。如果可以减少参数的数量或找到排除容易混淆的特殊情况的方法，客户就更容易理解应该如何使用方法。而且，限制类导出方法的总数是一个好的实践，这样客户就不会被选择所淹没，从而丧失对整体的把握。

13.4.3 满足客户的需要

Everything should be as simple as possible, but no simpler.

—attributed to Albert Einstein

然而，简单只是问题的一部分。将类难以理解或者复杂的部分全部扔掉，就可以让它变得简单。但这样做也会让这个类变得没有用。有时客户需要实现包含内部复杂性的任务。没有提供客户们所需的工具，只是将类变得很简单，这不是一种有效的策略。类必须提供足够的功能来满足客户需要。在类设计中如何保持简单和完整性之间合适的平衡是编程面临的基本挑战之一。

大多数情况下，类的客户不仅关心特定方法是否可用，还关心基本实现方式的效率。例如，如果编程人员要为空中交通管制开发一套系统，需要调用类提供的方法，这些方法必须快速返回正确答案，因为迟到的答案和错误的答案一样都可能导致灾难的发生。

在很大程度上，对于实现方式而言效率是关注的重点，而不是抽象设计。即使这样，设计类时，考察实现方式策略也颇有价值。例如，假设有两个设计选择。如果确定一种设计可以快速有效地实现——如果没有强制性原因——就可以选择这种设计。

13.4.4 灵活性的好处

Give us the tools and we will finish the job.

—Winston Churchill, radio address, 1941

特别适合某个特定客户需要的类对其他人可能没有用。好的类抽象能够满足许多不同客户的需要。要这样做，它必须非常一般化，以解决广泛的问题，而不是仅限于某个特定目的。通过给客户提供最大的灵活性，可以为自己创建的类扩展潜在客户。

希望让类保持通用具有重要的实践意义。写程序时，通常会发现需要特别的工具。如果确定这个工具非常重要，需要放到公共可用的程序包中，那么就需要改变思考模式。设计类的公用版时，必须忘记应用程序，它会导致用户首先想到工具，而不是为最普遍的潜在客户设计这样的工具。

在 Java 中，接口提供了最好的策略之一。这些策略让设计变得灵活。如本章所述，定义 List、Set 和 Map 的接口就可以指定这些抽象概念的行为，而无需限制实际的表示法。

13.4.5 稳定的意义

People change and forget to tell each other. Too bad—causes so many mistakes.

—Lillian Hellman, Toys in the Attic, 1959

受客户欢迎的程序包和类通常有另一种属性，这种属性让它们对于客户而言特别有用：它们经过很长一段时期还能保持稳定。稳定的类——特别是稳定的接口——通过建立明确的责任边界，简化了维护大型编程系统的问题。只要客户使用不变的类的基本形式，实现者和客户都可以在各自抽象化边界的那一侧进行改变。

例如，假设您是 Math 类的实现者。在工作过程中，发现有一种更高明可用于计算 sqrt 方法的新算法。这种算法可以将计算平方根所需的时间减少一半。如果您对客户说，sqrt 有一种新的实现方式，能和以前一样运行，但是运行速度更快，客户可能很高兴。但是，如果说方法的名称已经改变了，或者它在使用过程有某些新的限制，客户肯定会很生气。为了使用平方根“改进后的”实现方式，他们不得不改变自己的程序。改变程序很消耗时间，也很容易出错。为了不改变程序，许多客户会放弃额外的效率。保持稳定性——至少不要强迫现有客户修改代

码——会让每个人的生活更轻松。

13.5 小结

本章重点介绍了两个相关主题。第一，介绍了 Java 集合架构及其包含的类，最有名的是 `ArrayList` 和 `HashMap` 类。第二，它将该架构用作面向对象设计的例证示例。本章介绍的重点是：

- Java 内嵌数组模型上的 `ArrayList` 类，它最重要的好处之一是如果需要，就可以使用动态分配来扩充其内部数组。
- 分配更多的数组元素，记录数组的容量和有效大小，可以实现 `ArrayList` 类中使用的动态分配策略。容量是已分配的元素数量，有效大小是实际使用的元素数量。当有效大小超过容量时，必须分配更大数组来增加容量。
- Java 的灵活性主要通过接口实现，它可以指定类的行为，而无需限制其根本表示法。在 Java 集合架构中，每个具体的类都实现更一般的接口，这些接口可以更方便地关注类的抽象行为。
- `HashMap` 类是映射的一般概念使用最广泛的实现方式。映射指定键和值之间的联系。`HashMap` 类的基本操作是 `put` 和 `get`，它们允许客户将键与值关联起来或检索与指定键相关的值。
- `HashMap` 类使用称为散列的算法来加快寻找特定键的过程。每个 Java 类都包括 `hashCode` 方法，它返回一个整数，这个整数表示 `HashMap` 实现方式寻找特定键的位置，这种算法依赖这一点得以实现。当 `HashMap` 类的空间用完时，只要允许 `HashMap` 类增加其内部数据结构的大小，那么 `put` 和 `get` 方法就都会在 $O(1)$ 时间内运行。
- 如果要实现自己类里的 `hashCode` 方法，一种有用的策略是，先计算对象组件的散列码，用一个代码乘以一个小素数，然后使用为运算符(\wedge)将这些值结合起来，以计算对象的散列码。
- 如果重写某个类里的 `equals` 方法，就必须重写其 `hashCode` 方法。
- Java 集合架构为接口指定的各个抽象行为描述提供了多种具体的表示法。例如。`ArrayList` 和 `LinkedList` 实现 `List` 接口；`HashMap` 和 `TreeMap` 实现 `Map` 接口。客户可以选择最支持特定应用程序性能需要的表示法。
- Java 集合架构包括 `Iterator` 接口，它以每次一个的方式查看集合的元素。这种工具非常有用，因此 Java 5.0 给 `for` 语句引入了一种新语法来简化基于 `Iterator` 循环的格式。
- `Arrays` 和 `Collections` 类包含一组静态效用方法，分别用来操作数组和集合。这些方法包括 `sort` 和 `binarySearch`。
- 有效设计在编程过程的各个层次都非常重要。设计程序包和类时，必须让它们统一、简单、充分、灵活和稳定。

13.6 复习题

1. 概述本章用来实现 `ArrayList`(只使用数组)的动态行为的策略。

2. 容量和有效大小之间有何不同?
3. 什么让 Java 接口比抽象类更灵活?
4. 定义映射的两种基本操作是什么?
5. 判断题: 相同的值可以在映射里多次出现。
6. hashCode 方法的结果类型是什么?
7. 简述文件架原理。
8. 术语“冲突”在 HashMap 语境中是什么意思?
9. SimpleStringMap 使用什么策略来解决冲突问题?
10. acm.graphics 库中的 GPoint 类使用两个私有实例变量来存储点的坐标, 将它们声明如下:

```
private double x;
private double y;
```

- 编写适用于该类的 hashCode 方法。
11. 本章在讨论 hashCode 方法时强调, 在 Object 类中有一种方法, 在没有重写 hashMap 时, 也决不能重写它。这种方法是什么?
 12. 图 13-6 是如何说明 TreeSet 实现 Collection 接口的?
 13. 什么是迭代器?
 14. 描述 Java 5.0 用来指定集合中迭代的特殊语法。
 15. 如果没有 Java 5.0, 在使用 Java 集合架构时需要做哪些改变?
 16. 接口 Comparable 和 Comparator 之间有什么不同?
 17. 简述 Collection 接口和 Collections 类在功能上的不同点。
 18. 如何写 compare 方法, 让数组 ArrayList<String> 中的字符串按长度升序排列?
 19. 关于设计良好的程序包、类和方法, 本章提出了哪 5 条标准?

13.7 编程练习

1. 重新实现图 13-3 中的 SimpleStringMap 类, 让它使用两个并行数组来存储键和对应的值。和在图 13-1 中的 readFile 方法中一样, 需要的时候, 实现方式应该能够扩展这些数组。

2. 修改练习 1 的解决方案, 使 put 方法总是让键在内部数组中按排序顺序排列。保持有序数组可以使用折半搜索及 put 和 get 方法。而且, 使用这种策略让实现方式满足了 SortedMap 接口的约束, 尽管那个接口的完全实现方式超出了本练习的范围。

3. 扩充 SimpleStringMap 类的实现方式, 让它也导出方法

```
public void delete(String key);
```

该方法可以从映射中删除包含指定键的记录项。对比图 13-3 中使用的实现方式, 在某种程度上说, 实现这种方法更需要技巧。

4. 扩充 SimpleStringMap 类的实现方式, 让存储区里的数组可以动态扩展。实现方式应该记录映射中的项数, 当项数多于存储区数的一半时, 实现重新散列操作。

5. 虽然本文使用的存储区——链方法在实际中非常有效, 但也有许多其他策略可以解决

HashMap 中的冲突问题。在计算早期——当时内存很小，所以引用其他指针的成本也被计算在内——散列映射通常使用一种称为散列地址策略，它更节省内存。在这种策略中，键/值对直接存储在数组里，如图 13-13 所示。

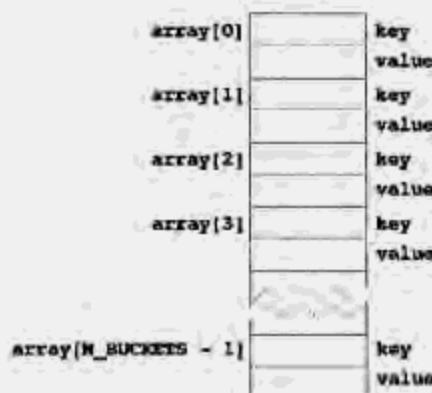


图 13-13 键值/对

例如，如果键散列到#3 存储区，散列表格的散列地址实现方式就会将该键及其值直接放入 array[3] 这一项中。

这种方法的问题是，array[3] 可能分配给另一个散列到同一个存储区里的键。处理这种冲突最简单的方法是将每个新的键存储在它所需散列位置或该位置之后的第一个空闲单元。因此，如果键散列到#3 存储区，get 和 put 方法首先在 array[3] 找到或插入该键。然而，如果这一项已经有其他键，这些方法就会找到 array[4]，继续这一过程，直到它们找到空记录项或者有匹配键的记录项为止。如果索引到了数组结尾，它应该返回开头。这种策略称为线性探测。

重新实现使用散列地址和线性探测的 SimpleStringMap 类。如果客户要在已经填满的表格中输入新的键，方法就会生成错误。另外，扩充该实现方式，让它支持练习 3 中的 delete 方法，或者像练习 4 中那样动态扩展。

6. 假设有一个名为 EnglishWords.txt 的数据文件，它包含所有正确的英语单词，每个单词一行。如果要在应用程序中使用这一列单词，最好有一些方法能够使用单词的完整列表。单词列表除了不包含定义之外，其他方面和传统字典一样。在计算机科学中，没有定义的单词字典有时称为专业词汇。

定义 Lexicon 类，它导出图 13-14 中所示的构造函数和方法。实现该类之后，可以将它用于下面的 run 方法，生成一列合法的二字母的单词(这对于 Scrabble 游戏者而言非常重要)

```
public void run() {
    Lexicon legalWords = new Lexicon("EnglishWords.txt");
    Iterator<String> iterator = legalWords.iterator();
    while (iterator.hasNext()) {
        String word = iterator.next();
        if (word.length() == 2) {
            println(word);
        }
    }
}
```

写该程序的过程中，使用 `HashMap` 类作为实现方式的一部分绝对合适。从脚本实现相同的行为是不好的编程实践。

<code>new Lexicon(String filename)</code>	创建新的 Lexicon 对象，并初始化为包含指定文件中的所有单词
<code>boolean isWord(String word)</code>	如果专业词汇中有指定的单词，返回 true
<code>int getWordCount()</code>	返回专业词汇中的单词数
<code>String getRandomWord()</code>	返回从专业词汇中随机选择的单词
<code>Iterator<String> iterator()</code>	返回迭代器，它查看专业词汇中的单词以不可预知的顺序

图 13-14 `lexicon` 类导出的方法

7. 第 8 章有一个编程示例是 PigLatin 转换器，其完整代码如图 8-14 所示。如第 8 章所述（事实证明有地区方言），形成英语单词的 Pig Latin 对等语的规则是：

- 如果单词以辅音字母开始，将词首的辅音字母字符串（第一个元音字母前的所有字母）从单词开头移动到末尾，然后加上后缀 `ay`，这样就形成了它的 Pig Latin 对等语。
- 如果单词以元音字母开始，只需要添加后缀 `way` 就行了。

有一小部分英语单词，将它们转换为 Pig Latin 后，结果还是合法的英语单词。例如 `trash`，变成 Pig Latin 之后是 `ashray`，如图 13-15 所示。



图 13-15 对单词进行转换

同样，英语单词 `plunder` 变成了单词 `underplay`。以元音开头的单词在某种程度上不是很有趣，如单词 `any` 变成了 `anyway`。

使用练习 6 中的 `Lexicon` 类来实现显示所有这类单词的 `ConsoleProgram`。

8. 文本顺便提到，在 10 个字母以内的英语单词中，只有两对单词的散列码相同：`buzzards` 和 `righto` 是一对，`crinolines` 和 `hierarch` 是另一对。编写 Java 程序确认这一说法。

9. 在讨论 `Comparator` 接口时，用来解释排序与大小写无关的示例是两部电影名称《THX 1138》和《Tarzan》。在此特定示例中，字母的大小写非常重要，而排序名称更重要的问题是，在排序时，如果冠词 `a`，`an` 和 `the` 出现在标题开头，忽略它们。

写交互式程序，它要求用户输入包含一列电影名称的文件，每个名称一行。程序应该将整个文件读入一列，使用 `Collections.sort` 排序该列，然后将排列后的行写回文件。提供给 `Collections.sort` 的 `Comparator` 应该忽略出现在名称开头的冠词以及所有大小写之间的区别。

10. 散列的重要应用程序之一是表示二维数字数组，其中相对较少的项是非零项。这样的数组称为稀疏矩阵。使用 `HashMap` 代替数组来存储元素的好处是，实现方式实际上不必将数组空间指派给零元素。如果元素不在 `HashMap` 中，则假设它为 0。

设计并实现称为 `SparseMatrix` 的类，让它使用 `HashMap` 来模仿一个 `double` 类型的二维数组。类应该导出构造函数以及 `get` 和 `put` 方法，分别用来检索元素的值和将元素设置为新值。问题的重点是设计这些方法的参数结构。

第 14 章

展望

The best way to predict the future is to invent it.

—Alan Kay, Xerox Palo Alto Research Center, 1971



Anita Borg (1949—2003)

正常来说，“展望”这一章应该描绘一幅个人图画，这个人将创造计算机科学的未来。未来属于那些目前正在学习计算机科学的人——那幅图画就是他们未来的写照。但未来也是那些为下一代铺平道路的人创造的。在 Anita Borg 短暂的一生中，她孜孜不倦地工作，不仅成为了计算机科学的研究科学家，而且还(特别是在妇女当中)大力推广计算学科。她是为计算机科学女性服务的 Systers 联机社区的创始人，也是 Grace Hopper Celebration of Women in Computing 和 Institute for Women in Computing 的创始人，在她 2003 年因癌症去世后，Institute for Women in Computing 更名为 Anita Borg Institute。然而，Anita 的精神永远活在那些克服困难走进她所深爱的这个充满活力而又令人兴奋的领域并在其中取得成功的人们心中。

通过本文前 13 章的学习，您已经掌握了 Java 编程的基础以及计算机科学的许多重要概念。然而，计算机科学是一个广泛而且不断扩展的领域，需要学习的仍然还很多。本章介绍另外 4 个主题，如果继续学习，肯定会遇到它们：

- 递归
- 并发
- 网络
- 编程模式

因为这些主题对于现代计算机科学非常重要，所以许多学校在初级编程课程中介绍它们。为了确保本文能够满足广大院校的需要，本章简要介绍一下各主题。即使不需要将它作为课程学习，但如果要继续研究计算机科学，通读本章也大有裨益。以后在更高级的课程中碰到这些思想时，就不会感到太陌生，从而能够迅速理解它们。

14.1 递归

用来解决编程问题的许多算法策略在计算领域之外都有对应的策略。重复执行任务时，使用的是迭代。决策时，练习的是条件控制。因为这些操作很常见，所以大多数人使用控制语句 `for`、`while` 和 `if` 时不会有太大问题。

然而，在解决许多复杂编程问题之前，必须掌握一种强大的问题解决策略，它在现实中几乎没有直接对应的策略。这种策略就是递归，它是一种解决方案方法，它将大问题简化为相同形式的小问题加以解决。“相同形式”在定义中至关重要，否则定义描述的就是第 5 章介绍的逐步细化策略。这两种策略都包括分解。递归之所以特殊是因为递归解决方案中的子问题与原问题具有相同形式。

如果是新的编程人员，第一次听到将问题分解为相同形式的子问题这种思想，可能不太理解。和重复或者条件测试不一样，递归不是日常生活中的概念。因为它不常见，所以学习如何使用它有点难度。必须对递归形成直观认识，将它看作与其他控制结构一样普通。对于大多数学习编程的用户而言，达到这种理解水平需要大量时间和实践。即使这样，学习使用递归所付出的努力也是值得的。作为问题解决工具，递归非常强大，以至于有点近乎神奇。另外，使用递归通常能以简单而又非常优美的方式编写复杂的程序。

14.1.1 递归的简单示例

为了更好理解递归的概念，假设您是一个大型慈善组织的基金协调员，这家慈善组织与其他许多类似的组织一样，志愿者众多，但资金短缺。您的任务是募集 1 000 000 美元满足组织开支。

如果您认识某个人，他愿意给您签一张 1 000 000 美元的支票，您的任务就完成了。但是，您可能没有朋友是慷慨的百万富翁。在这种情况下，必须以较小数额募集 1 000 000 美元。如果给组织平均募集 100 美元，那么就可以选择不同策略：给 100 000 个朋友打电话，向每人要 100 美元。但是，又可能没有 100 000 个朋友。那么该怎么办呢？

面对超过自身能力的任务时，解决方法通常就是将部分工作委托给其他人。您的组织有大量志愿者。如果在全国不同地方找到 10 名专业支持者，任命他们为地区协调员，这 10 个人每人负责募集 100 000 美元。

募集 100 000 美元比募集 1 000 000 美元相对简单，但也不容易。那么这些地区协调员又应该怎么办呢？如果他们采取同样的策略，他们会依次委托任务。如果他们招募到 10 名募款志愿者，这些人每人只需要募集 10 000 美元就行了。这种委托过程可以一直继续，直到志愿者能够一次募集到所需的钱为止。因为平均捐献是 100 美元，最后一个等级的募款志愿者很可能找到一个人愿意捐这么多钱，这样就不需要再继续委托。

如果用伪代码表示这种募集策略，其结构如下所示：

```

private void collectContributions(int n) {
    if (n <= 100) {
        Collect the money from a single donor.
    } else {
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
    }
}

```

该伪代码转换最值得注意的是行

```
Get each volunteer to collect n/10 dollars.
```

只是原始问题在更小范围内的复制。任务的基本特征(募集 n 美元)保持不变；唯一的不同是 n 取较小的值。而且，因为问题相同，所以可以调用原始方法解决它。因此，前面伪代码的行可以最终被下面的行取代：

```
collectContributions(n / 10);
```

如果捐献数额大于 100 美元，`collectContributions` 方法会始终调用自身，注意到这一点很重要。在编程过程中，让方法调用自身是递归的定义特征。

14.1.2 Factorial 函数

`collectContributions` 示例很有用，因为它以一种简单明了的方式介绍了递归的思想。但它对在实践中如何使用递归没有过多讨论，主要是因为它使用的原始操作(找到 10 名志愿者然后募集资金)在 Java 程序中不方便表示。要对递归形成实际认识，有必要考察一个更适合于编程领域的示例。

最好在简单数学函数的语境中说明递归，例如第 5 章介绍的阶乘方法。整数 n 的阶乘(数学中表示为 $n!$)是 1 与 n 之间所有整数的乘积。如第 5 章所述，可以使用 `for` 循环实现该阶乘，如下面的实现方式所示，它摘自图 5-6。

```

private int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

```

然而，这种实现方式没有利用阶乘的重要属性。每个阶乘都以下面的形式与下一个较小数的阶乘相关：

$$n! = n \times (n - 1)!$$

因此， $4!$ 等于 $4 \times 3!$ ， $3!$ 等于 $3 \times 2!$ ，依次类推。为了确保该过程在某个点停止下来，数学家定义 $0!$ 等于 1。因此，阶乘方法的传统数学定义是：

$$n! = \begin{cases} 1 & \text{如果 } n = 0 \\ n \times (n-1)! & \text{否则} \end{cases}$$

这个定义是递归的，因为它依据 $n-1$ 的阶乘定义 n 的阶乘。新的问题是找到 $n-1$ 的阶乘。它与原始问题的形式相同，这就是递归的定义特征。因此，可以使用相同的过程，依据 $n-2$ 的阶乘定义 $n-1$ 的阶乘。而且，可以逐步实现该过程，直到找到 0 的阶乘(数学家定义为 1)为止。到那时，递归终止，然后可以回到所有级别，依次用下一个数乘以每个结果。

这种方法最令人兴奋的地方是可以直接将它作为解决方案使用。因为 Java 允许方法递归地调用自身，所以可以用下面的方法实现 factorial 方法，它是数学定义的直接转换：

```
private int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

如果使用数学定义，写 factorial 的递归实现方式就很简单。另一方面，第一次学习递归时，这种实现方式省略了一些东西。即使它明确反映了数学定义，递归实现也很难确定实际计算步骤在哪里发生。调用 factorial 时，就想让计算机给出答案，在递归实现中，看到的是将对 factorial 的一个调用转换为另一个调用的公式。因为计算中的步骤不明确，所以计算机得出正确答案有时似乎很神奇。

然而，如果遵循计算机赋值其他方法调用的逻辑，它其实并不神秘。计算机给递归 factorial 方法调用赋值时，过程与赋值其他方法调用的过程一样。为了说明该过程，假设执行语句

```
int fact = factorial(4);
```

作为 run 方法的一部分。run 方法调用 factorial 时，Java 运行时系统创建一个新的方框，并将参数值复制给形式参数 n 。factorial 方框临时取代 run 方框，如图 14-1 所示。

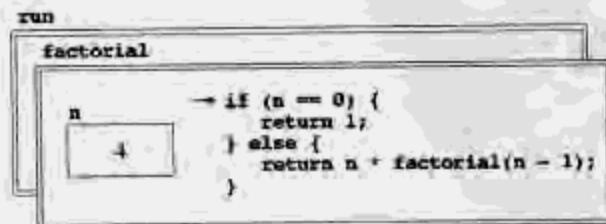


图 14-1 factorial 方框临时取代 run 方框

图中，factorial 主体的代码显示在方框里，以方便记录程序中的当前位置(如箭头所示)。图中的箭头出现在代码开头，因为所有方法调用从方法体的第一条语句开始。

计算机继续给方法体赋值，从 if 语句开始。因为 n 不等于 0，所以控制继续执行 else 从句。此时程序必须求出并返回表达式

```
n * factorial(n - 1)
```

的值。赋值该表达式需要计算 `factorial(n - 1)` 的值，需要递归调用。调用返回时，程序就用结果乘以 `n`。此时计算的状态如图 14-2 所示。

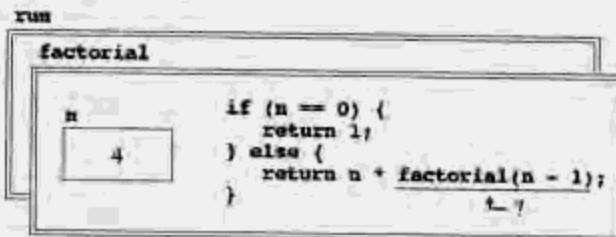


图 14-2 调用返回时的计算

对 `factorial(n - 1)` 的调用返回后，结果就被下划线表达式取代，该表达式让计算继续。

下一步计算给对 `factorial(n - 1)` 的调用赋值，它首先给参数表达式赋值。因为 `n` 的当前值是 4，所以参数表达式 `n - 1` 的值是 3。然后计算机给 `factorial` 创建一个新方框，方框中形参被初始化为该值。因此，下一个方框如图 14-3 所示。

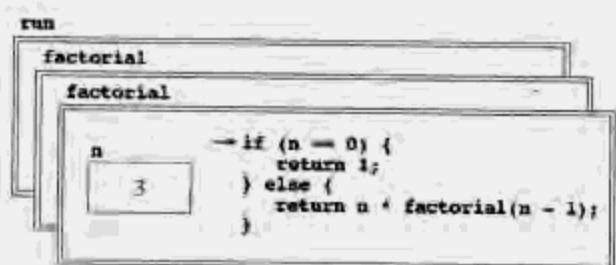


图 14-3 初始化型号

现在有两个方框标记为 `factorial`。在最新的方框中，计算机开始计算 `factorial(3)`。前面的方框(它们被新创建的方框所遮蔽)中，`factorial` 方法正在等待对 `factorial(n - 1)` 调用的结果。

然而，当前计算在最上面的方框中进行。因为 `n` 不等于 0，所以控制传递到 `if` 语句的 `else` 从句，在那里计算机必须给 `factorial(n - 1)` 赋值。然而，在这个方框中，`n` 等于 3，因此所需的结果要通过调用 `factorial(2)` 计算。和前面一样，过程需要创建一个新堆栈方框，如图 14-4 所示。

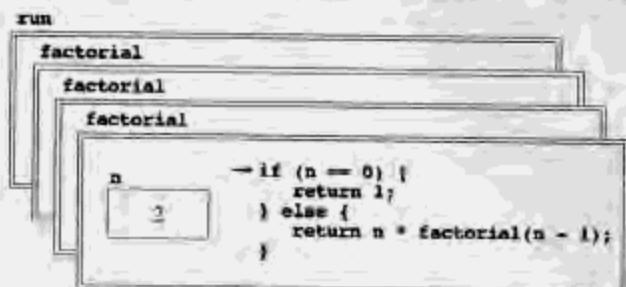


图 14-4 调用 `factorial(2)`

按照相同的逻辑，此时程序必须调用 `factorial(1)`，它再调用 `factorial(0)`，这样就创建了两个新的堆栈方框，结果堆栈配置如图 14-5 所示。

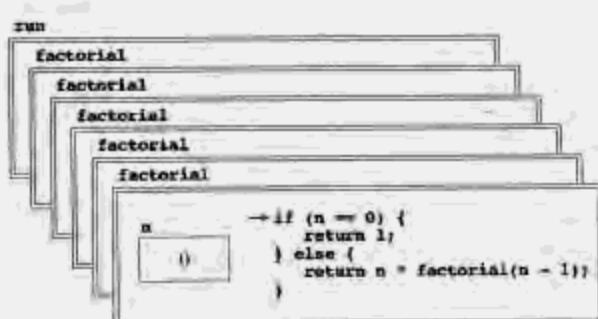


图 14-5 堆栈配置结果

此时情况有所改变。因为 n 等于 0，所以方法将值 1 返回给其调用者。当前调用的方框消失了，前面的方框出现在堆栈最上面，如图 14-6 所示。

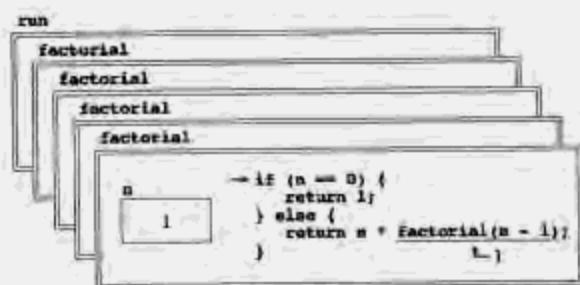


图 14-6 factorial(1)

从这里开始，计算就回到每个递归调用，用 factorial 在各个层面返回的值计算下一个层面的结果。例如，在此方框中，对 $\text{factorial}(n-1)$ 的调用被值 1 取代，因此这个层面的结果可以表示为：

```
return n * [1];
```

在这个堆栈方框中，因为 n 的值等于 1，所以该调用的结果就是 1。这个结果被传回给其调用者，和前面示例中一样。

返回几个层面后，计算就到了值为 6 的原始阶乘方框，如图 14-7 所示。

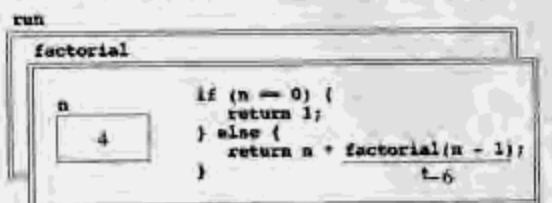


图 14-7 计算原始阶乘方框

此时因为 n 等于 4，赋值 return 语句将值 24 返回给 run 方法。

14.1.3 信任的递归式跳跃

14.1.2 小节 factorial(4) 示例的目的是显示计算机像对待其他方法一样对待递归方法。面对递归方法，可以（至少在理论上）模仿计算机的操作，并弄清楚它将会做什么。通过画方框、记

录所有变量，可以复制计算机的操作，得出答案。然而，如果这样做，通常会发现过程非常复杂，这让问题变得更难理解。

考察递归程序时，必须撇开这些基本细节，着重关注各操作层面。在单个层面可以假设，只要该调用的参数在某个方面比原始参数简单，递归调用就会自动获得正确答案。例如，要计算 n 等于 4 时的 $\text{factorial}(n)$ ，递归实现方式必须计算表达式

```
n * factorial(n - 1)
```

的值。通过将当前 n 的值替换进表达式，可以得到结果是：

```
4 * factorial(3)
```

就在这里停下来。已经完成了！计算 $\text{factorial}(3)$ 比计算 $\text{factorial}(4)$ 这个原始任务要简单。因为它更简单，所以可以假设它有用。我们知道 $\text{factorial}(3)$ 等于 $3 \times 2 \times 1$ ，即 6。所以调用 $\text{factorial}(4)$ 的结果是 4×6 ，即 24，这就是正确答案。

假设所有更简单的递归调用有效，这是一种基本的编程策略，称为信任的递归式跳跃。除非有使用递归方法的丰富经验，否则形成信任的递归式跳跃不容易。毕竟，写程序时，即使是经验丰富的编程人员，程序第一次不能正常运行的可能性也很大。暂时抛开这种不信任，假设它能够正常运行，这违背了自己对程序正确性的怀疑态度。即使这样，掌握递归的概念还是需要克服这种心理障碍。分解递归方法的多个层面不可避免地会让问题变得更难解决。

14.1.4 递归的范例

大多数递归方法与 14.1.2 节的 factorial 方法有相同的基本结构。典型递归方法的主体有下面这种范例形式：

```
if (test for a simple case) {
    Compute and return the simple solution without using recursion.
} else {
    Divide the problem into one or more subproblems that have the same form.
    Solve each of the problems by calling this method recursively.
    Return the solution from the results of the various subproblems.
}
```

递归的 factorial 方法符合这种范例，下面的方法求整数 n 的 k 次幂，它也符合这种范例：

```
private int raiseIntToPower(int n, int k) {
    if (k == 0) {
        return 1;
    } else {
        return n * raiseIntToPower(n, k - 1);
    }
}
```

raiseIntToPower 的实现方式依赖下面的数学属性：

$$n^k = \begin{cases} 1 & k = 0 \\ n \times n^{k-1} & \text{否则} \end{cases}$$

计算阶乘或求整数的幂都有自然的递归解决方案，因为问题满足如下条件：

- 可以确定简单情况，这种简单情况下的答案容易确定。
- 可以应用递归分解，将问题更复杂的实例分解为相同类型的简单问题，这些简单问题也可以应用相同的解决方案策略进行解决。

直接根据数学定义得出递归分解时(就像 factorial 和 raiseIntToPower 方法这种情况)应用递归并不特别困难。然而，当问题本身变得更复杂，需要仔细考虑递归分解方法时，情况就变了。

14.1.5 小节将解决更复杂的问题，这些问题的递归分解方法不是很明显。

14.1.5 图形递归

在 20 世纪 70 年代后期，IBM 一位名叫 Benoit Mandelbrot 的研究员很兴奋地出版了一本有关不规则碎片形的书。不规则碎片形是一种几何结构，这种结构以各种不同标准重复相同的模式。虽然数学家很早就知道不规则碎片形，但直到 20 世纪 80 年代这一科目才引起人们的兴趣，因为计算机的发展可以利用不规则碎片形完成比以前更多的任务。

不规则碎片形最早的示例之一是 Koch 不规则碎片形，以它的发明者 Helge von Koch 的名字命名。因为随着图形变得越来越精细，它会显示出美丽的、对称的六边形，所以它通常被称为雪花不规则碎片形，这种不规则碎片形的最简单形式(称为顺序 0 的雪花不规则碎片形)是等边三角形，如图 14-8 所示。

要获得下一个更高级别顺序的雪花不规则碎片形，只要用一组直线取代当前雪花的所有线段就行了。这些直线上除了中间 $1/3$ 是从图形向外的三角形凸起之外，看起来很像原始直线。因此，第一步是用一组 4 条较短线段(像下面这样排列)取代顺序 0 雪花中的所有线段，如图 14-9 所示。



图 14-8 等边三角形



图 14-9 4 条较短线段

将这种转换应用到原始三角形的三条边，生成顺序 1 的雪花，如图 14-10 所示。

如果用新的包含三角楔形的直线再次取代所有线段，就会创建顺序 2 的雪花，如图 14-11 所示。

这样再次取代所有这些线段，就会产生如图 14-12 所示的顺序 3 不规则碎片形，它已经有点像雪花。



图 14-10 顺序 1 的雪花



图 14-11 顺序 2 的雪花



图 14-12 顺序 3 的雪花

如果用 acm.graphics 程序包中的工具，特别是如果从面向对象的角度而不是从程序的角度来看待这个问题，写创建雪花不规则碎片形的程序很简单。但本示例的目的是创建不规则碎片

形对象，它们都是多边形。因此，定义 SnowflakeFractal 类作为 GPolygon 的子类就很自然。

例如，假设要创建顺序 0 的三角形不规则碎片形作为图形对象。该策略与图 9-27 中用来创建 GStar 类的策略类似。下面 SnowflakeFractal0 的类定义定义了 GPolygon 子类，它的边长作为参数传递给构造函数，参考点是三角形中心：

```
public class SnowflakeFractal0 extends GPolygon {
    public SnowflakeFractal0(double edge) {
        addVertex(-edge / 2, edge / (2 * Math.sqrt(3)));
        addPolarEdge(edge, 0);
        addPolarEdge(edge, 120);
        addPolarEdge(edge, 240);
    }
}
```

和使用 GStar 类一样，这种实现方式唯一复杂的地方是找出起始点(三角形的左下角)的坐标。对 addPolarEdge 的 3 个调用添加了三角形的边。第一条边与底部平行，下两条边相对于前一条边按逆时针方向旋转 120° 。

如果要画顺序 1 的雪花，要做的就是用添加三角楔形折线的方法调用，取代构造函数中对 addPolarEdge 的所有调用。类的新版本(此时称为 SnowflakeFractal1)如下：

```
public class SnowflakeFractal1 extends GPolygon {
    public SnowflakeFractal1(double edge) {
        addVertex(-edge / 2, edge / (2 * Math.sqrt(3)));
        addFractalLine(edge, 0);
        addFractalLine(edge, 120);
        addFractalLine(edge, 240);
    }

    private void addFractalLine(double r, int theta) {
        addPolarEdge(r / 3, theta);
        addPolarEdge(r / 3, theta + 60);
        addPolarEdge(r / 3, theta - 60);
        addPolarEdge(r / 3, theta);
    }
}
```

到目前为止一切顺利。已经为每个不规则碎片形顺序创建了新类，但该过程没有提供问题的一般解决方案。最好是 SnowflakeFractal 构造函数采用不规则碎片形的顺序以及边长。然后它将该顺序传递给 addFractalLine 方法，画所需顺序的 3 条折线。将 order 参数添加到 addFractalLine 非常符合递归解决方案。order 是 0 时，出现最简单的情况，因为顺序 0 的折线就是直线。递归理解是，顺序 k 的折线由顺序 $k-1$ 的 4 条折线组成。可以使用这一点写包含 order 参数的 SnowflakeFractal 方法更通用的实现方式。为此，必须修改 addFractalLine 方法，让它递归地调用它本身，除了在 order 参数是 0 这种最简单的情况下之外(在那种情况下边界是直线)。图 14-13 显示了 SnowflakeFractal 类的完整实现。

```

import acm.graphics.*;

/**
 * Defines a new GObject class that appears as a snowflake fractal. Because
 * the result is a GPolygon, you can fill the snowflake or move it as a unit.
 */
public class SnowflakeFractal extends GPolygon {

    /**
     * Creates a new SnowflakeFractal centered at the origin with the
     * specified edge length and fractal order.
     * @param edge The length of an edge in the order-0 snowflake
     * @param order The order of this fractal
     */
    public SnowflakeFractal(double edge, int order) {
        addVertex(-edge / 2, -edge / (2 * Math.sqrt(3)));
        addFractalLine(edge, 0, order);
        addFractalLine(edge, -120, order);
        addFractalLine(edge, +120, order);
    }

    /**
     * Adds a fractal line to the polygon with the specified radial
     * length, starting angle, and fractal order.
     * @param r The length of the line
     * @param theta The direction in which to draw the line (in degrees)
     * @param order The order of this fractal
     */
    private void addFractalLine(double r, int theta, int order) {
        if (order == 0) {
            addPolarEdge(r, theta);
        } else {
            addFractalLine(r / 3, theta, order - 1);
            addFractalLine(r / 3, theta + 60, order - 1);
            addFractalLine(r / 3, theta - 60, order - 1);
            addFractalLine(r / 3, theta, order - 1);
        }
    }
}

```

图 14-13 显示雪花不规则碎片形的 Gpolygon 的子类

14.1.6 递归式思考

和编程的其他方面不一样，学习使用递归需要全面考察编程过程。如果采用简化方法记录复杂递归分解中的所有步骤，就必须管理大量细节——而这最好留给计算机。记录本章前面 factorial(4) 的计算步骤需要好几个页面。相反，从整体上看，

```
factorial(4) = 4 * factorial(3)
```

只需要一行。这两种递归方法概念上的复杂性差别很大。采用整体的观点，可以将大多数递归程序的复杂性减少到可以理解解决方案的程度。另一方面，如果因怀疑解决方案的正确性而采用简化方法，大量细节肯定会让用户无法理解解决方案的高级结构。

用正确方法考察递归问题不容易。学习有效使用递归需要实践再实践。对于大多数用户而言，掌握概念就要花几年时间。但是，事实证明递归是编程指令表中最强大的方法之一，所以花这些时间也是值得的。

14.2 并发

本书开头,为了帮助用户理解 Java 开发的基本原理,图 1-4 列出了一些基本原则,Java 设计师认为这些原则可以将 Java 与其他编程语言区分开。其中一条原则是 Java 应该“是多线程的,在实现多并发活动所需的应用程序中体现高性能。”然而,到目前为止,本书的程序还没有特别用到 Java 原则所述的实现“多并发活动”这种能力。本节将简要介绍并发及 Java 用来支持它的特征。

14.2.1 进程与线程

虽然硬件结构不断增加,并且逐步形成了包含多处理器的计算机,但大多数计算机只有一个 CPU,它每次执行一个程序。从 20 世纪 60 年代早期开始,计算机使用了一种称为多遍编程的策略,它让计算机暂时(通常以数十或数百毫秒计算)运行一个程序,然后停下来,接着以同样长时间运行一个完全独立的程序,通过这样来模仿并发执行。因为计算机在这些不同的程序间来回转换,所以它们似乎都在运行。

尽管计算机每次只执行一个程序,但从一个程序转换到另一个程序时,它必须保存暂停程序的执行状态,恢复启动程序的状态。包含程序及其状态的概念实体称为进程。

多线程的思想扩充这种思想,从而反映出单个程序也可以从这种平行操作中受益。作为完成任务共同努力的一部分,程序应该启动几个似乎同时运行的独立活动(这些活动称为线程,有时称为控制线程),就像在多遍编程环境中一样。线程与进程之间的差别并不大。进程通常描述整个程序的执行状态,而线程描述相同程序内独立活动的执行状态。和进程不一样,多线程可以在同一个程序内共存,并共享访问程序资源。

到目前为止本书中的所有程序都只有一个主要线程。程序初始化时就创建了线程,它的唯一任务就是执行程序的 run 方法。然而,其他线程在后台很积极。特别是 Java 创建了事件线程,它响应用户的动作,例如单击按钮或拖动鼠标。就是线程在调用事件侦听器并引起程序的响应。然而,也可以创建自己的线程,这将在下一节介绍。

14.2.2 并发的简单示例

第 4 章介绍了用来使程序变成动画的简单模型,其伪代码形式如下:

```
public void run() {  
    initialize the graphical objects in the display;  
    while (you want to keep running the animation) {  
        Update the properties of the objects that you wish to animate.  
        pause(delay time);  
    }  
}
```

因为动画在 run 方法的实现方式中完成,所以这个模型使用的主要程序线程专门为执行 run 方法这一目的而创建。因为该模型只包含一个控制线程,所以时步间更新对象的所有代码必须在此方法内发生。许多情况下,使用独立控制线程让每个对象变成动画更方便。

图 14-14 中 AnimatedSquare 类的代码是这种方法的简单示例。本示例中的代码定义 GRect 子类,其大小由传递给构造函数的参数设置。该类的不同之处是,它实现 Runnable 接口,这意味着对象可以作为新的控制线程的语境。当线程启动(启动方式与在 Program 类中完全一样)时, run 方

法指定发生的情况。这里，run方法使用标准的动画循环移动方块，每50个时步改变一次位置。

```

import acm.graphics.*;
import acm.util.*;
import java.awt.*;

/**
 * This class creates an animated square that has its own thread of control.
 * Once started, the square moves in a random direction every time step.
 * After N_STEPS time steps, the square picks a new random direction.
 */
public class AnimatedSquare extends GRect implements Runnable {

    /* Creates a new AnimatedSquare of the specified size */
    public AnimatedSquare(double size) {
        super(size, size);
    }

    /* Runs when this object is started to animate the square */
    public void run() {
        for (int t = 0; true; t++) {
            if (t % CHANGE_TIME == 0) {
                direction = rgen.nextDouble(0, 360);
            }
            movePolar(DELTA, direction);
            pause(PAUSE_TIME);
        }
    }

    /* Private constants */
    private static final double DELTA = 2;      /* Pixels to move each cycle */
    private static final int PAUSE_TIME = 20;    /* Length of time step */
    private static final int CHANGE_TIME = 50;   /* Steps before changing direction */

    /* Private instance variables */
    private RandomGenerator rgen = RandomGenerator.getInstance();
    private double direction;
}

```

图 14-14 AnimatedSquare 类

实现 Runnable 接口的对象称为可运行对象。要激活程序其他部分的可运行对象，首先要创建附属于它的线程。例如，执行下面的代码，可以创建 AnimatedSquare 及其相关线程：

```

AnimatedSquare square = new AnimatedSquare(75);
Thread squareThread = new Thread(square);

```

如果将方块添加到 GraphicsProgram 画布，调用其线程上的 start，可以移动它，如下所示：

```

squareThread.start();

```

让线程风格动画有趣的是，它可以让许多不同的对象变成动画，每个对象都使用自己的线程。例如，图 14-15 中的代码创建两个 AnimatedSquare 对象，填充它们，将一个对象的颜色设置为红色，另一个设置为绿色，然后将它们添加到画布，让它们相距 1/3 边界距离。用户单击时，程序启动这两个线程。

```

import acm.program.*;
import java.awt.*;

/**
 * This program tests the AnimatedSquare class by putting two squares
 * on the screen and having them move independently.
 */
public class TestAnimatedSquare extends GraphicsProgram {
    public void run() {
        double x1 = getWidth() / 3 - SQUARE_SIZE / 2;
        double x2 = 2 * getWidth() / 3 - SQUARE_SIZE / 2;
        double y = (getHeight() - SQUARE_SIZE) / 2;
        AnimatedSquare redSquare = new AnimatedSquare(SQUARE_SIZE);
        redSquare.setFilled(true);
        redSquare.setColor(Color.RED);
        add(redSquare, x1, y);
        AnimatedSquare greenSquare = new AnimatedSquare(SQUARE_SIZE);
        greenSquare.setFilled(true);
        greenSquare.setColor(Color.GREEN);
        add(greenSquare, x2, y);
        Thread redSquareThread = new Thread(redSquare);
        Thread greenSquareThread = new Thread(greenSquare);
        waitForClick();
        redSquareThread.start();
        greenSquareThread.start();
    }
    /* Private constants */
    private static final double SQUARE_SIZE = 75;
}

```

图 14-15 测试 AnimatedSquare 类的程序

如果要做的只是让图形显示变成动画，那么就没有必要优先使用可运行对象的策略，而不使用更熟悉的在程序自身的 run 方法中实现动画的策略。它比我们一直使用的风格更具优势，以前的风格，最明显的就是难以保持可运行对象同步。因为每个可运行对象都有单独的线程，所以这些线程以不同速率运行。让相同代码更新动画包含的所有对象可以确保所有更新同时发生。

如果真的决定使用单独线程让图形程序中的单个对象变成动画，需要讨论一些库程序包，这些程序包很容易控制动画风格。

特别是，acm.util 程序包包括称为 Animator 的类，它扩充基本的 Thread 类来支持更灵活的交互式控制策略。例如，Animator 类可以方便地创建使用按钮启动、停止的控件栏以及方便地单步执行动画。Animator 类的细节超出了本书的范围，但可以查看其 javadoc 页面获取更多信息。

常见错误

侦听器方法用于短时间运行，以免占用 Java 的事件线程。所以，应该确保侦听器方法能够快速完成任务，如决不要调用 pause 方法。

然而，需要记住重要一点。有时可以考虑在某个侦听器方法里放置一个动画循环。例如，如果要让一个对象在单击它时，它就开始移动，可以试着在 mouseClicked 方法的定义内放置一个循环来实现。虽然这种策略开始可能有效，但它会产生严重的问题，因为它占用了 Java 的事件线程，表现为它不能响应其他事件。作为通用规则，应该确保所有侦听器方法能够迅速完成。

并发这个主题比本节概述的要复杂得多。特别是，确保两个线程在运行时不相互影响非常重要。如果一个线程改变了另一个线程所依赖的变量的值，那么这两个线程就很容易相互影响。

确保只有一个线程能够使用代码关键部分的特定数据项，这是互斥问题。Java语言及其库包含许多可以管理并发的工具，但这些工具超出了本书介绍的范围。

14.3 使用网络

影响Java开发的另一个重要因素是Java早期Internet和World-Wide Web(WWW)的迅速增长。因为applet机制使在Web页面下运行Java程序成为可能，所以Java迅速成为了基于Web应用程序的主导语言之一。

Java对基于Web编程的支持远不止applet。标准Java库程序包包括一种称为java.net的程序包，它允许程序延伸到网络，访问网络上的大量信息。

虽然java.net程序包中的大多数工具超出了介绍性文本的范围，但也有一些可以实现的简单操作。例如，图14-16中的openNetworkReader方法，打开网络URL上的BufferedReader，方法与第10章中openFileReader方法打开输入文件一样。在程序中可以使用这种方法阅读WWW任何位置URL的内容。这种方法可以成为用户自己基于Web应用程序的核心。谁知道呢？也许您能创建下一个Google或YouTube。

```
/*
 * This program prompts the user for a URL and returns a BufferedReader
 * that allows clients to read data from that URL. If the URL does not
 * exist or cannot be read, the user is given another chance to enter
 * a valid URL.
 * @param prompt The string used to prompt the user for a URL
 * @return A BufferedReader opened so that it reads data from the specified URL
 */
public BufferedReader openNetworkReader(String prompt) {
    BufferedReader rd = null;
    while (rd == null) {
        try {
            URL url = new URL(readLine(prompt));
            InputStream in = url.openStream();
            rd = new BufferedReader(new InputStreamReader(in));
        } catch (IOException ex) {
            println("Can't open that URL.");
        }
    }
    return rd;
}
```

图14-16 openNetworkReader的代码

14.4 编程模式

本书最后一个主题是一个概念的扩展，这个概念在前几章发挥了重要作用，它就是编程模式。在本书的不同地方已经出现过一些编程模式的示例，从简单术语(例如表示x值递增的x++表达式)一直到更大的伪代码模板(例如第4章首先介绍且在第14.2节再次回顾的动画模板)。随着编程语言和系统变得空前复杂，人们使用的编程模式也变得强大，甚至是术语现在也可以包含解决特殊问题的解决方案策略。

从极具影响的《设计模式：可复用面向对象软件的基础》(Addison-Wesley,1995)一书出版以来，设计模式的使用就变得非常普及。这本书由 Erich Gamma, Richard Helm, Ralph Johnson 和 John Vlissides 组成的 4 人小组完成，描述了大量解决各种不同编程问题的模式。这些模式的范围从在单个方法内可以实现的简单模式一直延伸到更大的需要类和接口层次结构的示例。虽然许多设计模式着重讨论了编程问题，这些问题超出了初级内容的范围，但如果要继续学习计算机科学和编程，最好记住这本书。

14.4.1 模型/视图/控制器模式

要理解模式的运行原理，有必要至少详细考察一种模式。例如，模型/视图/控制器的经典模式，它广泛用于用户—界面设计。这种模式是 Smalltalk 80 系统的一部分，因此它在《设计模式》一书的前面。

这种模式的基本思想是，将必须使用用户界面的程序与表示应用程序整体状态的基本结构区分开来。设计的模型部分保持应用程序的状态，但与用户的所有交互完全分开。用户与模型的交互通过给控制器发送命令实现，用户可以通过视图查看所有计算的结果。图 14-17 说明了这些组件之间的关系。



图 14-17 模型/视图/控制器模式示意图

模型/视图/控制器模式采用一种简洁的结构风格，它主要有 3 个优点。

- 模型/视图/控制器模式中的单个类只对总任务的一个部分负责，这样就很方便单独理解它们。可以阅读方法的代码，而不必拘泥于图形对象和交互器的细节。
- 模型/视图/控制器模式可以让应用程序拥有多个相同模式的视图。例如，大多数现代字处理器允许在同一个文件上打开多个编辑窗口。一个窗口里的改变自动反映到共享相同模型的其他窗口。
- 学习过面向对象设计现代原理的编程人员应该很熟悉模型/视图/控制器模式，可以利用对这种思想的理解，简化编码过程。

14.4.2 说明性示例：用图表示电子数据表的数据

电子数据表是支持多视图应用程序的有效示例。假设有一组数，它们对应于电子数据表数据的单行。该数组可能包含下面的值。

312	352	420	472
9	1	2	3

这些元素的值刚好是 *Harry Potter and the Sorcerer's Stone*(北美之外的英语读者称之为 *Harry Potter and the Philosopher's Stone*)最后一章 Hogwarts 的房屋在 Dumbledore 教授最后调整

之前积累的点数。电子数据表程序会以一种直观的方法表示这些值。例如，图 14-18 显示了这些数据的直方图和饼图。



图 14-18 直方图和饼图

如果使用电子数据表程序，可以用一种方法将这两幅图添加到电子数据表中，并且在更新电子数据表时会自动更新这两幅图。在这种情况下，数据表相当于模型，两个图表示视图。

不必实现数据表的各个部分，很容易将这些图形嵌入第 10 章介绍的交互式应用程序中。

应用程序的控件栏包括 JTextField，用户可以在里面输入 4 个整数，这 4 个整数表示 4 个 Hogwarts 房屋的点数，这 4 个 Hogwarts 房屋以字母顺序排列：Gryffindor、Hufflepuff、Ravenclaw 和 Slytherin。控件栏也包含 JButton 来更新图形。因此程序的初始状态如图 14-19 所示。

单击 Graph 按钮时，HousePoints 读取 JTextField 中的数，然后更新这两幅图。这样，当 Dumbledore 教授决定他要“拿出最后一些点数”时，Gryffindor 就又增加了 170 个点。如果改变 JTextField 中的第一个数，然后单击 Graph 按钮，显示就变成如图 14-20 所示。

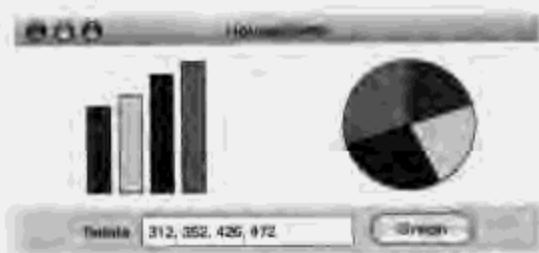


图 14-19 初始状态的直方图和饼图

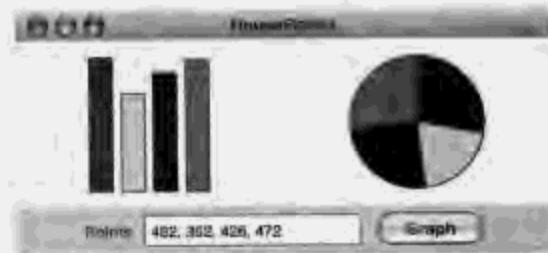


图 14-20 更新后的直方图和饼图

如果要使用模型/视图/控制器模式实现这个程序，需要定义一些类，这些超出了 HousePoints 代码本身。首先，需要 HousePointsModel 类来记录每个房屋的点数。所有房屋的信息都是一个包含 4 个整数的数组。HousePointsModel 类应该包含一些方法，例如

```
public void setHousePoints(int[] points)
```

它给指定数组设置内部数据；或者

```
public int[] getHousePoints()
```

它检索存储的值，以便视图确定要显示的值。另外，当用户改变数据时，模型必须记录需要通知的视图。因此，HousePointsModel 类应该导出方法

```
public void addView(HousePointsView view)
```

它将新值添加到视图的内部列表。控制器调用 setHousePoints 时，应该将改变通知这些视图。

HousePointsView 类是抽象类，它有两个分别用于两种图形类型的具体子类：BarGraphView 和 PieGraphView。这两个类支持 update 方法——update 方法按照模型的当前数据重新构造图片。引发更新的事件序列如下：

- (1) 用户在窗口底部的 JTextField 中输入新值，单击 Graph 按钮。这些交互器表示模型的控制器部分。
- (2) 控制器调用模型上的 setHousePoints，作为参数传递新值的数组。
- (3) 模型更新其内部存储，然后调用每个视图上的 update，作为参数传递模型。
- (4) 各视图调用模型上的 getHousePoints，检索新值。
- (5) 各视图更新显示，以对应新值。

图 14-21 显示了 HousePointsModel 类、HousePointsView 类和 LineGraphView 类可能的实现方式。主程序及 PieChartView 类的实现方式留在练习中进行。

```
/*
 * This class keeps track of the data in the model but is not responsible
 * for the actual display, which is managed by the registered views.
 */
public class HousePointsModel {

    /** Creates a new HousePointsModel with no views */
    public HousePointsModel() {
        housePoints = new int[0];
        views = new ArrayList();
    }

    /** Adds a view to the list of views for this model */
    public void addView(HousePointsView view) {
        views.add(view);
    }

    /** Sets the house points data to the contents of the integer array */
    public void setHousePoints(int[] points) {
        housePoints = new int[points.length];
        for (int i = 0; i < points.length; i++) {
            housePoints[i] = points[i];
        }
        notifyViews();
    }

    /** Returns a copy of the internal house points data */
    public int[] getHousePoints() {
        int[] points = new int[housePoints.length];
        for (int i = 0; i < points.length; i++) {
            points[i] = housePoints[i];
        }
        return points;
    }

    /* Calls update(this) on every view to reconstruct its display */
    private void notifyViews() {
        for (int i = 0; i < views.size(); i++) {
            HousePointsView view = (HousePointsView) views.get(i);
            view.update(this);
        }
    }

    /* Private instance variables */
    private int[] housePoints;
    private ArrayList views;
}
```

图 14-21 用图表示房子点数的程序

```
import acm.graphics.*;
import java.awt.*;

/**
 * This abstract class defines the operations that any specific view class
 * must support. Each HousePointsView is a GCompound that responds to update
 * messages from the model.
 */
public abstract class HousePointsView extends GCompound {

    /**
     * Creates a new HousePointsView with a given model and size
     */
    public HousePointsView(double width, double height) {
        background = new GRect(width, height);
        background.setFilled(true);
        background.setColor(Color.WHITE);
    }

    /**
     * Each subclass must define a method to create the graph
     */
    public abstract void createGraph(int[] data);

    /**
     * Updates the display image from the model
     */
    public void update(HousePointsModel model) {
        removeAll();
        add(background);
        createGraph(model.getHousePoints());
    }

    /**
     * Returns a color to use for the kth data value
     */
    public Color getColorForIndex(int k) {
        return COLORS[k % COLORS.length];
    }

    /* Private constants */
    private static final Color[] COLORS = {
        Color.RED, Color.YELLOW, Color.BLUE, Color.GREEN,
        Color.PINK, Color.CYAN, Color.MAGENTA, Color.ORANGE
    };

    /* Private instance variables */
    private GRect background;
}
```

图 14-21 (续)

```
import acm.graphics.*;  
  
/**  
 * This class represents a concrete implementation of the  
 * GraphView class that builds a bar chart. The chart is  
 * scaled so that the maximum value fills the vertical space.  
 */  
public class BarGraphView extends HousePointsView {  
  
    /** Creates a new BarGraphView */  
    public BarGraphView(double width, double height) {  
        super(width, height);  
    }  
  
    /** Arranges the data as a set of bars */  
    public void createGraph(int[] data) {  
        int n = data.length;  
        double max = maxIntArray(data);  
        if (max == 0) return;  
        double sep = (getWidth() - n * BAR_WIDTH) / (n + 1);  
        for (int i = 0; i < n; i++) {  
            double height = data[i] / max * getHeight();  
            double x = i * (BAR_WIDTH + sep);  
            double y = getHeight() - height;  
            GRect bar = new GRect(x, y, BAR_WIDTH, height);  
            bar.setFilled(true);  
            bar.setFillColor(getColorForIndex(i));  
            add(bar);  
        }  
    }  
  
    /* Returns the maximum value of an integer array (or 0 if empty) */  
    private int maxIntArray(int[] array) {  
        if (array.length == 0) return 0;  
        int largest = array[0];  
        for (int i = 1; i < array.length; i++) {  
            largest = Math.max(largest, array[i]);  
        }  
        return largest;  
    }  
  
    /* Private constants */  
    private static final double BAR_WIDTH = 20;  
}
```

图 14-21 (续)

14.5 小结

本章简要介绍了 4 个有趣的话题(递归、并发、网络和设计模式)如果继续学习计算机科学，肯定会遇到这些问题。本章介绍的重点是：

- 递归是解决问题的过程，它将问题分解为具有相同形式的子问题。许多实际问题都可以使用递归解决方案，事实证明在这些领域中，它是一种非常强大的工具。
- 开发递归解决方案的前几步是：①确定简单情况，要很容易确定这些情况下的答案；②找到递归分解，将程序分解为更简单的问题，这些问题可以应用相同的方式解决。
- 开始采用递归技术时，重要的是学会“递归式思考”。特别是当自己的方法应用于更简单的子问题时，要相信它们可以正确运行。这种信任称为信任的递归式跳跃。
- 在现代计算应用程序中，通常有许多不同活动同时进行。在计算机科学中，这种类型的同步过程称为并发。Java 广泛支持并发，特别是在线程层面上。线程是同一个程序内发生的并发进程。
- 因为开发 Java 时，人们对 WWW 的兴趣不断增长，所以该语言也广泛支持基于网络的编程。这些工具大多数都包含在 java.net 程序包里。
- 作为创建应用程序的工具，编程模式变得越来越重要，特别是在 1995 年出版《设计模式》之后。如果继续学习计算机科学，肯定有机会详细探讨这一领域。
- 一种特别重要的编程模式是模型/视图/控制器模式，它为设计用户界面提供了有用的结构。

14.6 复习题

1. 递归过程的基本特征是什么？
2. 下面的方法计算哪个数学函数：

```
private int mystery(int x) {
    if (x == 0)
        return 1;
    else
        return 2 * mystery(x - 1);
}
```

3. 设计递归解决方案时，需要确定问题的哪两个方面？
4. “信任的递归式跳跃”是什么意思？作为编程人员，这种思想为什么很重要？
5. 线程和更传统的进程概念有何不同？
6. 哪个程序包包含大多数用于 WWW 的 Java 库工具？
7. 对现代面向对象编程发挥重要作用的“四人帮”指哪 4 个人？
8. 在模型/视图/控制器模式中，模型和视图之间的任务如何划分？

14.7 编程练习

1. 第 4 章练习 9 介绍了一种 Fibonacci 序列，其中前两项是 0 和 1，后面每一项都是它前两项之和。序列开始是

$$\begin{aligned}
 F_0 &= 0 \\
 F_1 &= 1 \\
 F_2 &= 1 \quad (F_0 + F_1) \\
 F_3 &= 2 \quad (F_1 + F_2) \\
 F_4 &= 3 \quad (F_2 + F_3) \\
 F_5 &= 5 \quad (F_3 + F_4) \\
 F_6 &= 8 \quad (F_4 + F_5)
 \end{aligned}$$

后面所有项都是这种形式。写方法 Fib(n) 的递归实现方式，让它返回第 n 个 Fibonacci 数。该实现方式必须只依赖于序列中各项之间的关系，不能使用任何迭代结构，如 for 和 while。

2. 时间：1777 年；地点：位于殖民地某个地方的华盛顿将军的阵营。给您分配了一项危险的侦察任务：计算英军可用于大型加农炮的军火数量，这些加农炮正在炮轰革命军。幸运的是，英军(整洁而有序)地将炮弹堆成了一个金字塔形，最上面是 1 颗炮弹，它的下面有 4 颗炮弹，这 4 颗炮弹下面有 9 颗炮弹，依次类推。然而，遗憾的是，英国士兵也很警惕，在您计算出金字塔的层数之后，可能会被发现。幸运的是，您设法返回了自己的军队。

虽然 150 年前还没有发明计算机，但您的任务是写递归函数 Cannonball，它将金字塔的层数作为参数，返回其中炮弹的数量。该函数必须递归式运行，不能使用任何迭代结构，如 while 或者 for。

3. 第 9 章练习 10 显示了如何使用 Pascal 三角形来形象化组合函数 $C(n, k)$ (该组合函数请参考第 5 章)。从前面的 Pascal 三角形中可以看出，表中的每一项要么沿着某一条对角边排列，在这种情况下它的值是 1；要么在三角形内部，在这种情况下，它的值是上面左右对角两项之和。

利用对 Pascal 三角形这种结构的理解，写 combinations 方法的递归实现方式，该方法不能使用循环或乘法，也不能调用 factorial。

第 8 章练习 6 将回文定义为一种字符串，它的来回读法完全相同，如 “level” 或 “noon”。在那个练习中，要求写断言方法 isPalindrome(str) 来测试 str 是不是回文。虽然在那个时候，可以使用迭代策略来解决这个问题，但现在可以用递归策略写 isPalindrome 方法。递归的理解就是，较长的回文内部一定包含较短的回文。例如，字串 “level” 由回文 “eve” 和两端的单字符串 “l” 组成。使用这种理解写 isPalindrome 的递归实现方式。

几乎所有讨论递归的计算机科学课程中，都会介绍 19 世纪的一个难题，它是原型的递归问题。这个难题的名字叫 Hanoi 塔，它由 3 座塔组成，其中一座塔上有一些圆盘(在难题的商业版本中通常是 8 个)，从塔底到塔顶按大小降序排列，如图 14-22 所示。

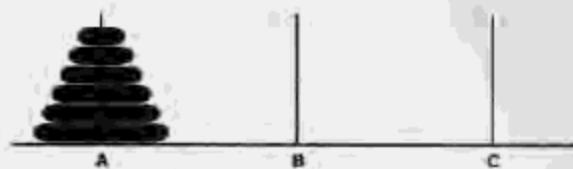


图 14-22 Hanoi 塔

难题的目的是，按下列规则，将这组圆盘从塔 A 移动到塔 B：

- (1) 一次只能移动一个圆盘。
- (2) 不能将大圆盘放在小圆盘上面。

写程序显示将塔上 N 个圆盘从塔 A 转移到塔 B 所需的全部步骤。例如，当 N 等于 3 时，

程序应该生成如图 14-23 所示的输出。

```
This program solves the Tower of Hanoi problem
Enter the number of disks: 3
A -> B
A -> C
B -> C
A -> B
C -> B
C -> B
A -> B
```

图 14-23 Hanoi 塔程序的示例输出

解决这个问题的关键是找到分解问题的方法，这种分解方法允许将原始的 Hanoi 塔问题转换为相同形式的简单问题。

6. 人们对不规则碎片形如此感兴趣的原因之一是，事实证明它们在有些实践语境中非常有用。例如，画山的计算机图像及其他一些地形特征最成功的方法都要使用不规则几何学。

假设要求用不规则碎片形连接点 A 和点 B，让它看起来像地图上的海岸线。最简单的可能策略是在两点之间画一条直线，如图 14-24 所示。



图 14-24 直线

这就是顺序 0 时的海岸线，它表示递归的基本情况。

当然，真正的海岸线在某些地方会有一些小的半岛或海港，所以真实的海岸线图形应该是有的地方凸出去、有的地方凹进来。第一种近似是，用相同的折线(14.1.5 小节的程序使用该折线来创建雪花图形)取代直线，如图 14-25 所示。



图 14-25 折线

该过程产生顺序 1 的海岸线。然而，为了有传统海岸线的感觉，直线中的三角楔形应该以相等概率时而向上、时而向下。

如果用随机方向的不规则碎片形取代顺序 1 不规则碎片形中的每条线段，就会产生顺序 2 海岸线，如图 14-26 所示。



图 14-26 顺序 2 海岸线

继续该过程，最后就可以画出很逼真的海岸线，如图 14-27 所示。

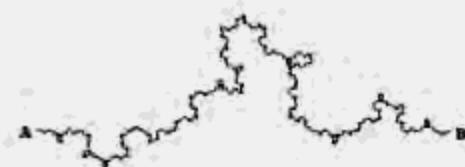


图 14-27 最终的海岸线

虽然这条海岸线在许多方面与文中的雪花不规则碎片形很相似，但它没有封闭，这一点导致不可能使用 GPolygon 作为图形的子类。

修改图 14-13 中的 AnimatedSquare 类，让红色方块继续以原始示例中的随机方式移动，但绿色方块总是以相同速度直接移动到红色方块的当前位置。要这样做，需要加强 AnimatedSquare 的定义，让两个方块可以通信。

通过实现主程序和 PieChartView 类，完成图 14-21 中 HousePoints 应用程序的实现方式。

去图书馆或书店，查看 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 所著的《设计模式》。仔细查看各种模式，找出一种可能有用的模式。写简单的应用程序，说明这种模式的用法。

- 面向对象分析与设计（UML 2.0版）
- 数据结构与算法——C++版（第3版）
- 数据挖掘原理与应用——SQL Server 2005 数据库
- 嵌入式系统——体系结构、编程与设计
- 信息安全原理（第2版）
- C++大学教程（第3版）
- 计算机网络教程（第4版）
- MATLAB科学计算
- Linux系统管理完全手册
- 嵌入式C语言编程与Microchip PIC
- 数据库设计、应用开发和管理
- Linux系统管理完全手册
- Java大学教程（第2版）
- 卓有成效的软件项目管理
- 关系数据库和SQL编程
- 多媒体技术及应用（第7版）
- 软件测试的有效方法（第3版）
- Visual Basic.NET程序设计（第6版）
- Unix原理与应用（原书第4版）
- 编程语言：原理与范型（第2版）
- Linux管理基础教程（第4版）
- 数据仓库工具箱
- 网格计算核心技术
- 无线网络原理与应用

自1995年首次发布以来，Java编程语言作为一种教学语言变得日益重要，现在已经成为初级计算课程的标准语言。Java语言可以让学生编写高度交互式程序，这充分激发了他们的学习兴趣。但Java语言很复杂，老师和学生们在理解Java语言的结构时，复杂性成为了最大的障碍。

在本书中，斯坦福大学教授、著名的计算机科学教育领导者Eric S. Roberts着重强调了更适合于初学者的友好讲解方式，使用ACM Java库简化编程。本书简练清晰地介绍了传统CS1课程的内容，同时也包含了最近的Computing Curriculum 2001报告计算机科学卷中指定为CS101O或CS111O课程的全部主题。

本书特色

- ◆ 采用现代面向对象方法，从零开始介绍最有用的类层次结构
- ◆ 全文使用图形和交互式程序，充分激发学生的学习兴趣
- ◆ 使用传记简介、引用以及哲学片段来突出计算的历史和理性背景
- ◆ 着重强调算法和问题解决，而今天的初级教科书通常忽略了这一点

作者简介



Eric S. Roberts，美国斯坦福大学计算机科学系教授，并担任主管教学事务的系主任。同时还由于教学改革所取得的成就被评为Charles Simonyi荣誉教授。他于1980年获得哈佛大学应用数学博士学位，并曾在加州Palo Alto的DEC公司的系统研究中心工作了5年。作为一位成功的教育工作者，Roberts还获得了1993年的Bing Award奖。

信息网站：<http://www.tup.com.cn>
<http://www.tupwk.com.cn>
读者信箱：wkservice@vip.163.com
投稿信箱：bookservice@263.net



ISBN 978-7-302-18441-6



9 787302 184416 >

定价：59.80元