

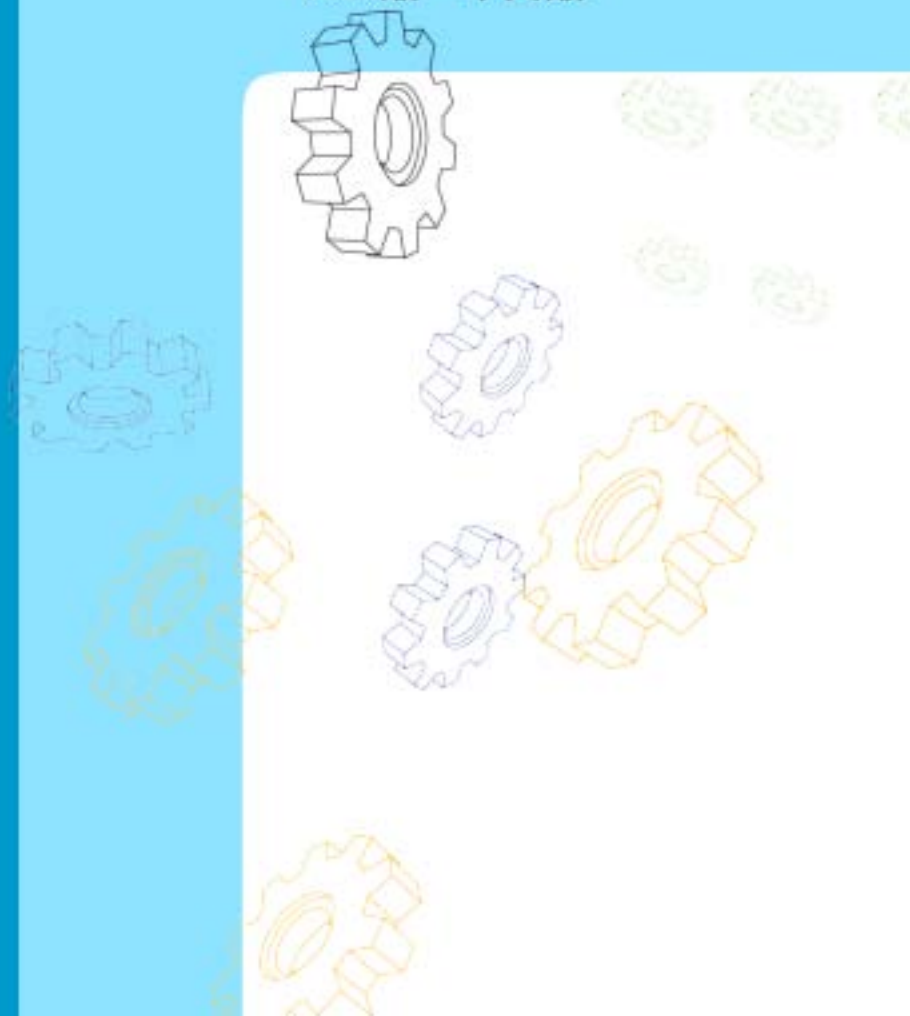
How to Think Like a Computer Scientist: Java Version

---

---

# 如何像计算机科学家一样思考

Java版 • 中文版



# 如何像计算机科学家一样思考

**Java** 版/中文版

作 者: Allen B. Downey

翻 译: 朱 珠 (Aaron)

E-Mail: [aaron\\_go@sogou.com](mailto:aaron_go@sogou.com)

文档版本: CHS\_0.1

2004 年 10 月—2005 年元月

## 版权申明

本文档由 Aaron 翻译制作。

文档基于 GNU 自由文档执照（GNU Free Documentation License）。

您可以自由地复制和发送这份文档，前提是必须保留本申明。

版权所有©2005 Aaron

# 目录

第 0 章 前言 .....	1
为什么要写这本书.....	1
背后的哲学观.....	1
面向对象编程（Object-oriented programming/OOP） .....	2
数据结构.....	3
计算机科学高级职业考试.....	3
Free Books！ .....	3
噢，标题！ .....	4
第 1 章 编程方法论 .....	5
1.1 什么是编程语言？ .....	5
1.2 什么是程序？ .....	7
1.3 什么是程序调试？ .....	8
1.4 形式语言和自然语言 .....	10
1.5 第一个程序.....	11
1.6 术语表.....	12
第 2 章 变量和型别 .....	14
2.1 关于打印的更多内容.....	14
2.2 变量.....	15
2.3 赋值.....	15
2.4 打印变量.....	16
2.5 Java 关键字 .....	17
2.6 运算符.....	18
2.7 运算符优先级.....	19
2.8 字符串运算符.....	19
2.9 复合.....	19
2.10 术语表.....	20
第 3 章 方法 .....	21
3.1 浮点数.....	21
3.2 浮点数转换为整数.....	22
3.3 数学方法.....	22
3.4 复合.....	23
3.5 新增方法.....	24
3.6 类和方法.....	26

---

3.7 多方法程序.....	26
3.8 形参和实参.....	27
3.9 堆栈图.....	28
3.10 多参数方法.....	29
3.11 带返回结果的方法.....	30
3.12 术语表.....	30
第4章 条件语句和递归 .....	31
4.1 取模运算符.....	31
4.2 条件执行.....	31
4.3 选择执行.....	32
4.4 链式条件.....	33
4.5 嵌套条件.....	33
4.6 返回语句.....	34
4.7 型别转换.....	34
4.8 递归.....	35
4.9 递归方法的堆栈图.....	36
4.10 约俗和法则.....	37
4.11 术语表.....	37
第5章 带返回值的方法 .....	38
5.1 返回值.....	38
5.2 程序开发.....	39
5.3 复合.....	42
5.4 重载.....	42
5.5 布尔表达式.....	43
5.6 逻辑运算符.....	44
5.7 布尔方法.....	45
5.8 再谈递归.....	45
5.9 “掩耳盗铃” .....	48
5.10 最后一例.....	48
5.11 术语表.....	49
第6章 迭代 .....	50
6.1 多次赋值.....	50
6.2 迭代.....	50
6.3 while 语句.....	51
6.4 表.....	52
6.5 二维表.....	54

---

6.6 封装和通用化.....	55
6.7 方法的好处.....	56
6.8 再谈封装.....	56
6.9 本地变量.....	57
6.10 再谈通用化.....	57
6.11 术语表.....	59
第 7 章 字符串及其它 .....	60
7.1 向对象调用方法.....	60
7.2 length 方法.....	61
7.3 遍历.....	61
7.4 运行时错误.....	62
7.5 阅读文档.....	62
7.6 indexOf 方法 .....	62
7.7 循环和计数.....	63
7.8 自增 1 和自减 1 运算符.....	64
7.9 字符算术运算.....	65
7.10 刚性的字符串.....	66
7.11 不可比的字符串.....	66
7.12 术语表.....	67
第 8 章 有趣的对象 .....	68
8.1 有趣在哪里.....	68
8.2 包.....	68
8.3 Point 对象 .....	68
8.4 实例变量.....	69
8.5 对象做参数.....	70
8.6 矩形.....	70
8.7 作为返回值的对象.....	71
8.8 刚性的对象.....	71
8.9 别名.....	72
8.10 null .....	73
8.11 垃圾收集.....	73
8.12 对象和基本类型.....	74
8.13 术语表.....	75
第 9 章 创建对象 .....	76
9.1 类定义与对象的型别.....	76
9.2 Time 类 .....	76

---

9.3 构造器.....	77
9.4 再谈构造器.....	78
9.5 创建新对象.....	78
9.6 打印对象.....	79
9.7 对象的操作.....	80
9.8 纯粹函数.....	81
9.9 修正方法.....	82
9.10 填充方法.....	83
9.11 哪种方法最好.....	84
9.12 递增开发 vs.规划.....	84
9.13 通用化.....	85
9.14 算法.....	86
9.15 术语表.....	86
第 10 章 数组.....	88
10.1 访问数组元素.....	88
10.2 拷贝数组.....	89
10.3 for 循环语句.....	90
10.4 数组和对象.....	91
10.5 数组长度.....	91
10.6 随机数.....	91
10.7 随机数数组.....	92
10.8 计数.....	93
10.9 矩形图.....	94
10.11 术语表.....	95
第 11 章 对象的数组.....	96
11.1 复合.....	96
11.2 纸牌对象——Card.....	96
11.3 printCard 方法.....	97
11.4 sameCard 方法.....	99
11.5 compareCard 方法.....	100
11.6 牌的数组.....	101
11.7 printDeck 方法.....	102
11.8 查找.....	102
11.9 Deck 和 subdeck.....	105
11.10 术语表.....	106
第 12 章 数组的对象.....	107

---

12.1 Deck 类 .....	107
12.2 洗牌.....	108
12.3 排序.....	109
12.4 subdeck 方法 .....	109
12.5 洗牌和发牌.....	110
12.6 合并排序.....	111
12.7 术语表.....	113
第 13 章 面向对象编程 .....	114
13.1 编程语言和风格.....	114
13.2 对象和类的方法.....	114
13.3 当前对象.....	114
13.4 复数.....	115
13.5 复数函数.....	116
13.6 另一个复数函数.....	116
13.7 修饰方法.....	117
13.8 toString 方法.....	117
13.9 equals 方法 .....	118
13.10 调用对象方法.....	119
13.11 奇怪的错误.....	119
13.12 继承.....	120
13.13 可拖动的矩形.....	120
13.14 类的层次.....	121
13.15 面向对象设计.....	121
13.16 术语表.....	122
第 14 章 链表 .....	123
14.1 对象内的引用.....	123
14.2 Node 类.....	123
14.3 作为容器的链表.....	124
14.4 链表与递归.....	125
14.5 无穷链表.....	126
14.6 基本歧义定理.....	127
14.7 节点的方法.....	128
14.8 修改链表.....	128
14.9 包装方法和助手方法.....	129
14.10 IntList 类 .....	129
14.11 恒量.....	131



---

14.12 术语表.....	131
第 15 章 堆栈 .....	132
15.1 抽象数据型别.....	132
15.2 堆栈 ADT .....	132
15.3 Java 中的 Stack 对象.....	133
15.4 包装类.....	134
15.5 创建包装对象.....	134
15.6 再谈创建包装对象.....	135
15.7 取值.....	135
15.8 包装类中的有用方法.....	135
15.9 后缀表达式.....	136
15.10 解析.....	136
15.11 实现 ADT .....	138
15.12 堆栈 ADT 的数组实现 .....	138
15.13 缩放数组.....	139
15.14 术语表.....	140
第 16 章 队列和优先队列 .....	142
16.1 队列 ADT .....	142
16.2 蒙版.....	143
16.3 链队列.....	145
16.4 循环缓冲.....	146
16.5 优先队列.....	149
16.6 元类.....	150
16.7 优先队列的数组实现.....	150
16.8 优先队列的委托者 (client) .....	151
16.9 Golfer (高尔夫球手) 类 .....	152
16.10 术语表.....	154
第 17 章 树 .....	156
17.1 树节点.....	156
17.2 建立树.....	157
17.3 遍历树.....	157
17.4 表达式的树.....	158
17.5 遍历.....	158
17.6 封装.....	159
17.7 定义元类.....	160
17.8 实现元类.....	160

---

17.9 Vector 类 .....	161
17.10 Iterator 类.....	163
17.11 Glossary .....	163
第 18 章 堆 .....	165
18.1 树的数组实现.....	165
18.2 性能分析.....	167
18.3 合并排序的分析.....	169
18.4 额外开销.....	171
18.5 优先队列的实现.....	171
18.6 堆的定义.....	173
18.7 堆的 remove 方法.....	173
18.8 add 方法.....	175
18.9 堆的性能.....	175
18.10 堆排序.....	176
18.11 术语表.....	177
第 19 章 映射 .....	178
19.1 数组、向量和映射.....	178
19.2 Map ADT .....	178
19.3 内建的 HashMap .....	178
19.4 一个向量实现.....	181
19.5 List 元类 .....	183
19.6 HashMap 实现 .....	183
19.7 哈希函数.....	184
19.8 缩放哈希映射.....	185
19.9 哈希表调整操作的性能.....	185
19.10 术语表.....	186
第 20 章 霍夫曼编码 .....	187
20.1 长短编码.....	187
20.2 频率表.....	188
20.3 霍夫曼树.....	188
20.4 super 方法 .....	190
20.5 解码 (Decoding) .....	191
20.6 编码 Encoding .....	192
20.7 术语表.....	192
附录 A 程序开发计划.....	193
附录 B 调试.....	198

B.1 运行时错误 .....	198
B.2 运行时错误 .....	200
B.3 语义错误 .....	203
附录 C Java 中的输入/输出 .....	208
附录 D Graphics 类 .....	210
D.1 Slate 和 Graphics 对象 .....	210
D.2 对一个 Graphics 对象调用方法 .....	210
D.3 坐标 .....	211
D.4 未完成的米老鼠 .....	211
D.5 其它绘图命令 .....	213
D.6 Slate 类 .....	214

## 第 0 章 前言

“当我们享受他人的发明带来的巨大便利时，也应该很乐意提供一个用自己的发明来为别人服务的机会，而且我们应该慷慨地免费地这样做。”

——本杰明·富兰克林

引自 埃德蒙德·S·摩根 著《本杰明·富兰克林传》

### 为什么要写这本书

这是我在 1999 年于 Colby 大学任教时所著一书的第四版。其时我任教一门用 Java 语言讲解的计算机科学课程，但是苦于找不到自己喜欢的教科书。首先，那些教材太大部头了！即使我想让学生读这些动辄 800 多页密密麻麻的技术手册，他们也办不到。而且我也不想让他们那么做。这些资料大多太过专注于 Java 语言的特定细节和一些到了期末就会被学生忘得干干净净的类库，这些东西把我真正想找的材料搞的扑朔迷离。

另外一个问题是，对于面向对象程序设计的介绍太草率，使得很多在其它部分学得很好的学生一讲到“对象”就碰壁，甚至不管在开学、中间还是期末同样如此。

于是我开始了本书的编写，十三天时间里每天写一章，第十四天进行编辑校订，然后把书送去复印装订。在开课第一天把书发到学生手里的时候，我告诉他们希望他们一个星期读一章，换句话说，他们的阅读速度应该是我写作速度的 1/7。

### 背后的哲学观

下面是一些指导本书编写并使之得以别具一格的主要观念：

- 词汇表很重要。学生必须能够谈论“程序”并且明白是在说什么。我力图介绍尽量少的术语；第一次遇到一个术语就进行精心定义；每一章后面都专门列出新词表。在教学中，随堂测验和考试都有词汇测试题，要求学生用适当的词进行简要回答。

- 为了编写程序，学生要懂得算法 (*algorithm*)、了解编程语言，而且还要有调试程序的能力才行。太多的书遗漏了调试的知识。而这本书有一个关于调试的附录和一个关于程序开发（帮助避免进行不必要调试）的附录。我建议尽早阅读并且经常回顾这些材料。

- 一些概念需要时间来了解。本书中一些较难以理解的概念比如递归，反

复出现了数次。通过反复提到这些概念，让学生有机会复习巩固，或者是如果第一次漏过了也还有机会可以补上来。

●力图用最少的 Java 语句实现最强大的编程能力。本书的目的是传授编程和计算机科学的一些引导性概念，而不是讲授 Java 语言本身。我有意不讲一些语言特性，例如并非必要的 *switch* 语句，还有大多数类库，特别是像 AWT 这样保持快速更新而且极可能被取代的类库。

采取风格简约的写作方法有很多优点。不包括练习的话每章大约十页。上课讨论之前要求学生先通读一遍该章——他们乐意这样做并且对课本理解的很充分。这些准备让课堂上有充分的时间讨论较难的材料，包括课堂作业，还有不在课本中的补充话题。

当然，简约并非十全十美。在这里没有多少本质上有趣的东西，我所用的大多数例子主要是为了演示一种语言结构的最基本用法，因此大多数练习潜心于字符串操作和数学概念。我认为这些东西有一部分还是很有趣，但是更多能让学生对计算机科学产生浓烈兴趣的东西诸如图形处理、声音处理和网络应用等等，却被一笔带过了。

原因在于，很多更加有趣的内容却包含了大量的细节而不是更多的概念。从教育学的角度来看，这意味着不值得投入更多的注意力。所以，在讨学生喜欢的材料和智力含量更高的材料之间采取了折衷的选择。我希望使用这本书的时候师可以自己找到对他所教班级学生来说最好的平衡点。为了帮助教学，这本书有一个涵盖了图形、键盘输入输出的附录。

## 面向对象编程（Object-oriented programming/OOP）

有些书一上来就介绍“对象”；另一些则是一步一步慢慢来，逐渐地建立面向对象的风格。这本书大概可以划入极端“对象靠后”的路线。

Java 中很多面向对象的特性是得到以往的语言启发，这些结构是有历史可溯的。如果学生不知道这些特性是为了解决什么问题而设计的，很多内容就难以向他们解释清楚。

当然，阻碍面向对象编程绝非目的；恰好相反，为了能尽快的引入 OOP，我尽量做到只介绍一次概念，第一次就解释清晰，所以在增加新概念之前让学生对每个已引入的概念都专门练习。这样在第 13 个概念上终于引入了面向对象编程。

## 数据结构

在 2000 年秋季，我执教了第二门引导课程——数据结构，于是本书新增了如下几章：线性表、堆栈、树和散列表。

这些新增的每一章给出了一种数据结构的定义、至少一种用到了该数据结构的算法，以及至少一种实现版本。多数情况下当然也有一个用标准的 *java.utils* 包实现的版本。这样一来，教师就能选择是否讨论实现版本或者让学生做一个练习来实现。大多数时候我都同时用了一个 *java.utils* 包的实现版本来描述一种数据结构和对应的接口。

## 计算机科学高级职业考试

在 2001 年夏季，我和缅因州立数理学院的老师们合作编写一本帮助学生准备计算机科学高级职业考试（The Computer Science Advanced Placement Exam /CSAPE）的 Java 版本教材，而当时的教材用的是 C++。翻译工作的进展非常快，因为一旦开始，我就发现我写这本书所选用的材料跟考试大纲简直太吻合了。

自然而然地，当校方宣布考试改为采用 Java 语言，我也制定了计划要更新这本书中相关的 Java 的内容。因为看了 CSAPE 的考纲，我发现它对 Java 的介绍很全而且跟我的选择范围非常相似。

2003 年 1 月，我完成了本书第四版，做了如下修改：

- 新增了一章涵盖霍夫曼编码。
- 重写了我发现有疑问的几个小节，包括向 OOP 的过渡和关于堆栈的讨论等。
- 增强了调试和程序开发的附录。
- 新增了一些小节以提高对 CSAPE 考纲的覆盖率。
- 收集了我在曾经执教的班级用过的练习、检测和考试问题，放在相应章节的后面；同时设计了一些意在帮助准备 CSAPE 的问题。

## Free Books!

从一开始，这本书和其后续版本都遵照 GNU 自由文档执照的形式存在。读者可以免费的下载这本书的各种格式文档，并打印出来或者直接在屏幕上阅

读；教师可以自由的用打印机把本书打印出来，并且根据需要复印任意数量。而最重要的是，任何人都可以根据需要自由的修改本书。你可以下载 **LaTeX** 源代码，然后添加、删除、编辑或者重排其中的材料，以达到使本书最大程度为你或你执教的班级学生所用的目的。

有人已经把本书翻译成了其它计算机语言（包括 **Python** 和 **Eiffel**）和其它自然语言（包括西班牙语和法语），很多这样的衍生版本都遵循 **GNU** 自由文档执照而存在。

这种发行办法有很多优点，但是也有一个不利的地方：所有这些书从来没有得到正式的编排和校对。在开源软件的影响下，我已经习惯于尽早的发布然后经常更新本书。我尽全力把错误减少，但是我更依赖读者帮我找出问题。

一直以来反响都非常好。我几乎每天都收到读过并且非常喜欢本书的人们不辞繁琐的给我发送“**bug** 报告”。一般说来我都能立即改正错误并且发布更新后的版本。每次我有时间进行修订，或者当读者不惜时间给我发回反馈的时候，我感到这本书是一个不断进步的作品。

## 噢，标题！

我有一大堆关于本书标题的伤心往事，不是每个人都明白——基本上——那是一个笑话。阅读本书也许不会真的让你学会像计算机科学家那样思考。那需要时间、经验，也许在加上一些课程。

但是标题中有一个核心事实：这本书不是专门讲 **Java** 的，对具体语言的使用只是编程工作的一部分。如果这本书是成功的，它就是写思考方法的。计算机科学家们自有一套**问题求解**（*problem-solving*）的方法，一种精巧解决问题的路子，那是独一无二的，而又是通用的、强大的法子。我希望这本书能给你关于上述方法到底是什么样子的一个认识，最后使你从某种程度上发现自己在像一个计算机科学家那样思考。

Allen Downey

于 马萨诸塞州，波士顿

2003 年 3 月 6 日

# 第 1 章 编程方法论

这本书和这门课（计算机科学导引）的目的是教会你像一个计算机科学家那样思考。我喜欢计算机科学家思考的方式，因为这种方式整合了数学、工程和自然科学的最佳特点。计算机科学家像数学家那样使用形式语言来表示概念（特别地，叫做**计算**—*computation*）；像工程师那样设计事物，进行权衡折衷然后把部件集成成为一个系统；像自然科学家那样通过假言和预设来观测复杂的系统。

对于一个计算机科学家来说，唯一最重要的事情就是问题求解。所谓“**问题求解**（*problem-solving*）”的能力就是，能够抽象问题并创造性的思考解决办法，然后清晰而且准确无误的描述出解决方案。久经验证的结果表明，学习编程的过程是一个练习问题求解技巧的绝佳机会。这就是为什么这一章要叫做“编程方法论”。

一方面，你将要学习编程——一种有用的技巧；在另一层面上讲，你会用编程做为达到目的的手段，在学习的过程中，目标将慢慢变得清楚起来。

## 1.1 什么是编程语言？

这里要学习的是一门相当新派名唤 Java 的编程语言（第一版是由 Sun 公司在 1995 年发布）。Java 是一例**高阶语言**（*high-level language*），你可能还听说过其它诸如 Pascal、C 或者 C++ 和 FORTRAN 这样的高阶语言。

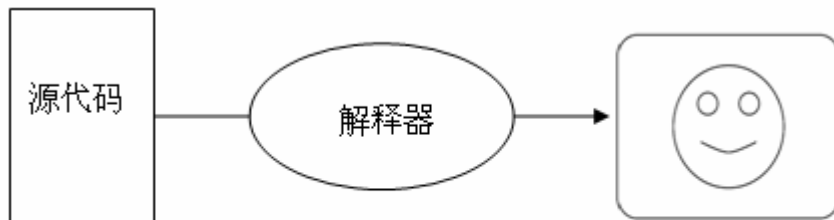
正如你可以从字面上推断出来的那样，有“高阶”语言就有“低阶”语言。**低阶语言**（*low-level language*）通常指**机器语言**（*machine language*）或者**汇编语言**（*assembly language*）。计算机只能执行用低阶语言编写的程序，因此，用高阶语言编写的程序必须经过翻译才能运行。翻译过程需要消耗时间——这是高阶语言的一个小小缺点。

而优点则是不可胜数。首先，用高阶语言编程更容易，这意味着写程序需要的时间更少，写出来的东西也更短小易读，也更容易改正。其次，高阶语言是**可移植**（*portable*）的，这意味着写出来的程序只需要稍加修改甚至不用改就可以直接在不同的计算机上运行。低阶程序只能在一种特定的机器上**运行**（*run*），要换地方跑（*run*）只有另写。

正是由于上述优点，几乎所有程序都是用高阶语言编写的，低阶语言只用来写少数几种特殊的应用程序。



程序的翻译有两种方式：**解释**（*interpreting*）和**编译**（*compiling*）。**解释器**（*interpreter*）是用来读取高阶程序代码然后照着做的程序。实际上，它一行一行的翻译程序代码，然后依次执行命令。



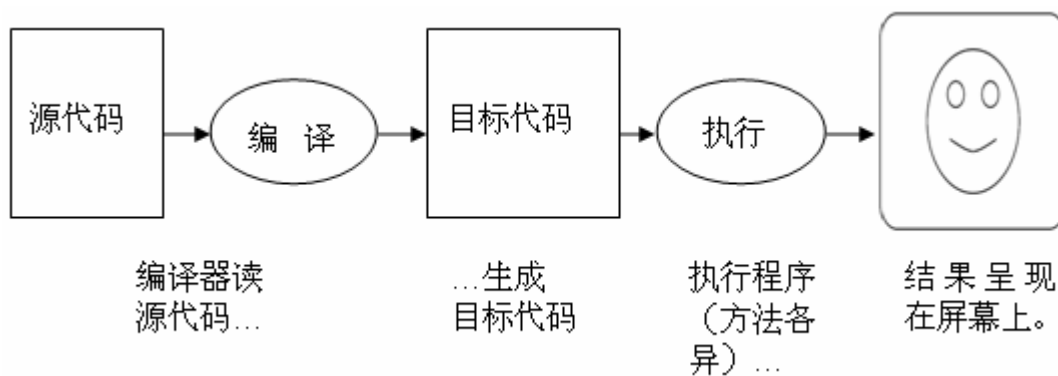
解释器读取源代码...

...结果在屏幕上显示。

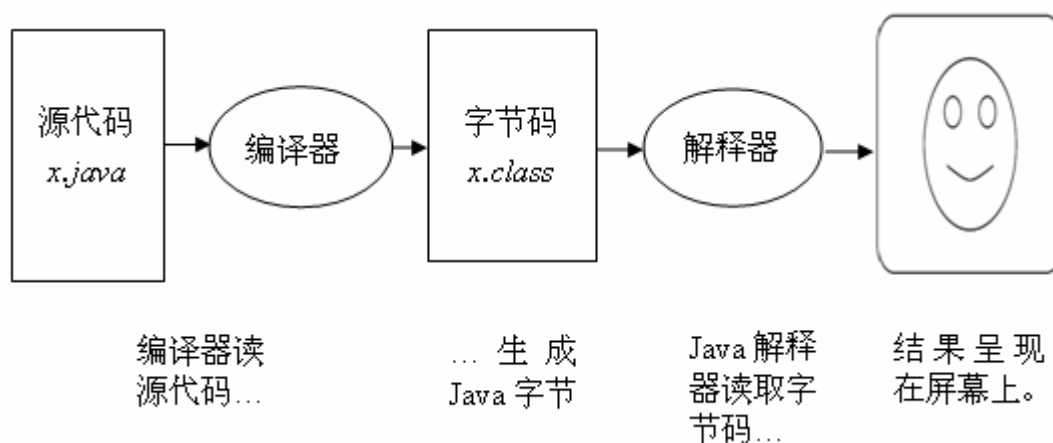
**编译器**（*compiler*）则是这样一种程序：它读取高阶程序代码但并不执行而把它们一次全部翻译好。一般说来编译程序是一个单独的步骤，经过编译的代码可供稍候执行使用。在这种情况下，高阶程序叫做**源代码**（*source code*），而翻译好的程序叫做**目标代码**（*object code*）或者**可执行代码**（*executable*）。

举个例，假如你要用 C 语言写一个程序。你需要一个文本编辑器（一个很简单的文字处理程序）用来写你的程序，程序写好之后，把它保存在一个叫做 *program.c* 的文件里，“*program*”是一个随意取的名字，而后缀“.c”是一个约定，表明该文件包含用 C 语言编写的源代码。

然后，视你的编程环境而定，忘记文本编辑器，运行编译器吧。编译器读取源代码，翻译、执行，生成一个新的文件 *program.o*——包含目标代码，或者 *program.exe*——包含可执行代码。



Java 语言有点不同寻常，因为它同时既是编译型的又是解释型的，Java 编译器并不把 Java 程序翻译成机器语言，而是生成 Java **字节码**（*byte code*）。字节码很容易解释（速度也很快），这有点像机器语言；而且字节码也是可移植的，这像高阶语言。因此，在一台机器上编译 Java 程序然后通过网络传送到另一台机器上解释是可行的——这种能力正是 Java 比其它很多高阶语言强的地方。



虽然这一过程看起来可能有点复杂，不过在大多数编程环境（有时叫做开发环境）里面，这些步骤都是自动完成的。通常要做的只是写好程序然后点一个按钮或者输入一条命令来运行它就是了。话说回来，知道后台到底发生了什么步骤是很有用的，这样即使出了什么错也能找到问题所在。

## 1.2 什么是程序？

**程序** (*program*) 是一个精确说明如何进行计算的指令序列。这里的计算可以是一些数学上的计算，比如解方程或者求多项式的根，也可以是符号运算，比如查找和替换一个文档中的字或者（如果程序足够奇妙的話）编译一个程序。

**指令** (*instruction*) —— 以后将叫做**语句** (*statement*) —— 在不同的编程语言中看上去并不一样，但有为数不多的几个基本操作是大多数语言都能执行的：

**输入** (*input*)：从键盘、文件或者其它说明设备获取数据。

**输出** (*output*)：把数据显示到屏幕，或者存入一个文件，或者发送给其它设备。

**数学运算** (*math*)：执行基本的数学运算操作，比如加法和乘法。

**测试** (*testing*)：检测特定的情况并执行相应的语句。

**重复** (*repetition*)：执行一些不断重复的动作，具体情况通常取决于一些变量。

对于程序来说，上面这些指令足够了。每个你曾用过的程序，不管有多么复杂，都是由执行上面这些操作的语句组成的。所以，编程可以说就是这样一个过程：把大问题和复杂任务层层分解，直到最后获得的子任务简单得可以用这些基本操作之一来完成。

## 1.3 什么是程序调试？

编程是一个复杂的过程，因为是人做的事情，所以难免经常出错。出于一个稀奇古怪的原因，程序错误专门被叫做**臭虫** (*bug*)，而找到这些 *bugs* 并且加以纠正的过程就叫做**调试** (*debugging*)。

程序中可能出现的错误不过数种，区别加以对待能让调试更加迅速。

### 1.3.1 编译时错误

编译器只能翻译语法正确的程序，否则将导致编译失败无法运行程序。**语法** (*syntax*) 是指程序的结构和关于这结构的规则。

例如，在英语中语句必须以大写字母开头以句点结尾。所以下面两个英文语句都是语法错误的：

this sentence contains a syntax error. So does this one

对于大多数读者来说，一点语法错误不是很严重的问题，因为我们可以读自由诗而不会误解意思。

可惜编译器就没有这么宽容了。只要有哪怕一个语法错误，编译器将输出一条错误提示信息然后就罢工，这样你根本没有办法运行你的程序。

更糟糕的是 **Java** 的语法规则比英语的语法规则还要多，而且编译器给出的错误提示信息往往帮助不大，在开始编程职业生涯的前几个星期，你可能会将大把大把的时间花在纠正语法错误上。等到有了些经验，还是会犯这样的错误，不过会少的多，而且你能更快地发现这种错误。

### 1.3.2 运行时错误

第二种错误类型是**运行时错误** (*run-time error*)，这类错误因为只在运行时而不是编译时发生而得名。在 **Java** 中，当解释器运行字节码出错时就是产生了一个运行时错误。

好消息是，**Java** 更接近一种安全的语言所以很少发生运行时错误，尤其是像我们接下来几个星期将编写的这类简单程序更不易出这样的错。

不过到了这学期后面，你可能会遇到更多运行时错误，尤其是当我们开始谈到对象和引用（第 8 章）的时候。

在 **Java** 中，运行时错误被叫做**异常** (*exception*)。大多数环境下，这些异常以窗口或者对话框的形式弹出来，告诉你发生了什么，程序正在做什么，这些信息对调试很有帮助。

### 1.3.3 逻辑错误和语义错误

第三类错误是**逻辑错误** (*logical error*) 和**语义错误** (*semantic error*)。如果程序里有逻辑错误, 编译和运行都会很顺利, 看上去也不产生任何错误信息, 但是程序没有干它该干的事情, 而是干了些别的什么。在指定情形下, 程序只会按你的要求做。

问题出在你写的程序不是你想要的, 这意味着程序的意思 (即语义) 是错的。确定逻辑错误在哪里需要十分清醒的头脑, 因为要通过观察程序的输出而回过头来判断它到底在做什么。

### 1.3.4 富有试验性的调试

通过本课程你将掌握最重要的技巧之一就是调试。可能会让你感受一些沮丧, 但是调试也是编程中最具智力含量、挑战和乐趣的部分。

某种角度看调试就像在搞侦察, 根据掌握的线索来推断是什么过程和事件导致了你所看到的结果。

调试也像是一门试验科学, 每次想到哪里可能有错, 就修改程序然后再试一次。如果假设是对的, 就能预测出修正的结果, 然后一步一步的靠近正确的程序; 如果假设错误, 对不起只好请另外找思路。“当你把不可能的东西全部剔除, 剩下的——即使看起来不可能——就一定是事实。” (引自 A. Conan Doyle 著 *The Sign of Four*)。

对有的人来说, 编程和调试是一回事, 编程就是逐步调试直到获得期望结果为止的过程。这种观念表明, 应该总是从一个能正确运行的小规模程序开始, 一步步做细微的改动一边进行调试, 这样就总是能得到一个正确的程序。

举个例, Linux 操作系统包含了成千上万行代码, 可开始时它仅仅是 Linus Torvalds 拿来琢磨 Intel 的 80386 芯片用的。据 Larry Greenfield 说, “Linus 的早期工程之一是编写一个切换打印 AAAA 和 BBBB 的程序, 这玩意儿后来进化成了 Linux”。(引自 Linux 用户指南 Beta1 版)

在本章的后面部分会给出更多关于调试和编程实践的建议。

## 1.4 形式语言和自然语言

**自然语言** (*natural language*) 就是人类讲的语言, 比如英语、西班牙语和法语。这类语言不是人为设计 (虽然有人试图强加一些规则) 而是自然进化的。

**形式语言** (*formal language*) 是为了特定应用而人为设计的语言。举个例, 数学家们用的符号就是一种形式语言, 用来标识数字和字符特别有效。再如, 化学家用形式语言来表示分子的化学结构。而最重要的是请记住:

**编程语言是专门设计用来表达计算的形式语言。**

**Programming languages are formal languages that have been designed to express computations.**

正如前面提到的, 形式语言有严格的语法规则。举个例,  $3 + 3 = 6$  是一个语法正确的数学命题, 而  $3 = +6\$$  则不是。同样,  $H_2O$  是一个语法无误的分子式, 而  $_2Zz$  则不是。

语法规则有两道调料: **符号** (*token*) 和 **结构** (*structure*)。符号是语言的基本元素, 比如词、数和化学元素之于各门语言。 $3 = +6\$$  的问题在于  $\$$  不是一个合法的数学术语 (迄今为止据我所知); 类似地,  $_2Zz$  不是一个合法的化学式, 因为没有一种元素的缩写是  $Zz$ 。

语法规则的第二个范畴是语句结构, 也就是符号的排列方式。 $3 = +6\$$  语法上是错误的, 因为不能在等号之后紧跟加号。类似地, 分子式中必须把下标放在化学元素名称之后而不是前面。

当阅读一个英文句子或者一种形式语言的语句时, 你不得不设法搞清楚句子的结构是什么样 (在自然语言中你只是没有意识到, 但确实这样做了)。这个分析句子结构的过程称之为**解析** (*parsing*)。

举个例, 当你听到 “*The other shoe fell.*” 这个句子, 你理解 “*the other shoe*” 是主语而 “*fell*” 是动词。一旦解析完成, 你就搞懂了句子的意思, 或者说搞懂了它的语义。如果知道 *shoe* 是什么东西, *fall* 意味着什么, 你还能理解这个句子主要暗示的内容。

虽然形式语言和自然语言有很多共同之处, 包括符号、结构、语法和语义,

但是也有很多不一样的地方。

#### 多义性-ambiguity

自然语言充满了多义性，人们通过上下文的线索和其它一些信息来解决这个问题。形式语言的设计要求是清晰、毫不含糊的，这意味着每一个语句必须有确切的含义而不管上下文如何。

#### 赘词-redundancy

为了消除多义性减少误解，自然语言引入了相当多的赘词。结果是自然语言经常变得罗哩罗嗦，而形式语言则绝少赘词更加简洁。

#### 文意一致性-literalness

自然语言充斥着成语和隐喻，我说 “*The other shoe fell*”，实际可能没有鞋，也没有什么东西倒了。而形式语言中字面上和实际意思是一样的。

说自然语言长大的人（实际上没有人例外），往往有一个适应形式语言的困难过程。某种意义上，形式语言和自然语言之间的不同正像诗歌和散文的区别，当然，更多：

#### 诗歌 poetry

词语的发音和意思一样重要，全诗做为整体创造出一种效果或者表达出一些感情。多义性不仅是常见的而且是刻意使用的。

#### 散文 prose

词语的字面意思显得更重要，而且结构能传达出更多的意思。散文比诗歌更经得起分析，但仍然充满多义性。

#### 程序 program

计算机程序是毫不含糊而文意一致的，能够完全通过符号和结构的分析加以理解。

我有一些关于阅读程序（包括其它形式语言）的建议。首先，请记住形式语言远比自然语言紧凑，所以要多花点时间来读。其次，结构很重要，从上到下从左到右地读往往不是一个好办法，而应该学会在大脑里解析（*parse*）：识别符号，分解结构。最后，请记住细节的影响，诸如错误的拼写和标点这样在自然语言中可以忽略的小毛病会把形式语言搞得面目全非。

## 1.5 第一个程序

习惯上人们用新语言写的第一个程序被叫做 *Hello, World*。因为它所做的事情就是打印 “*Hello, World.*”。

用 Java 写这个程序是下面的样子：

```
class Hello {  
    // main: generate some simple output  
    public static void main (String[] args) {  
        System.out.println ("Hello, world.");  
    }  
}
```

有人用 *Hello, World.* 程序的简单程度来判断一门编程语言的优劣，用这个标准看，Java 实在干得不够漂亮。即使最简单的程序也富含许多对编程新手来说难于解释的内容，现在忽略大多数这些内容，只解释其余一小部分。

所有 Java 程序都由类组成，类的**声明**（*definition*）具有如下格式：

```
class CLASSNAME {  
    public static void main (String[] args) {  
        STATEMENTS  
    }  
}
```

类名（CLASSNAME）可以任意指定，在第一个例子中是 `Hello`。

第二行中的“`public static void`”现在可以忽略，但是要注意“`main`”。`main` 是专门指出程序在哪里开始执行的方法名称，程序运行的时候从 `main` 方法包含的第一条语句开始依次执行，直到完成最后一条语句而退出。

`main` 方法中的语句条数没有限制，而第一例中只有一条，即打印语句，向屏幕上输出一条信息。有点让人困惑的是，“打印”有时表示在屏幕上显示，有时表示向打印机发送。本书中不会谈及向打印机发送，我们将在显示器屏幕上进行“打印”。

在屏幕上打印的语句是 `System.out.println`，括号内是要打印的内容。语句最后是一个分号（`;`），每个语句都要有。

语法上还有一些需要注意的地方：首先，Java 用把各组内容放在花括号（`{` 和 `}`）内。最外层的括号（第 1 和第 8 行）中是类的定义，里面的括号中包含 `main` 方法的定义。

其次，第 3 行以 `//` 开头，表示该行包含一条**注释**（*comment*），是一些可以放在程序中间，通常用来解释程序做什么的文字。当编译器遇到 `//`，就忽略此后直到一行结束的内容。

## 1.6 术语表

**问题求解**-*problem-solving*：把问题形式化并找出解决办法，然后清晰而且准确

无误的描述出解决方案的过程。

**高阶语言-high-level language:** 像 Java 这样为了便于人读写而设计的编程语言。

**低阶语言-low-level language:** 为了便于计算机执行而设计的编程语言。也叫做机器语言 (machine language) 或是汇编语言 (assembly language)。

**形式语言-formal language:** 为表达数学概念或是计算机编程这样的专门目的而设计的语言, 所有编程语言都是形式语言。

**自然语言-natural language:** 人类所讲的任何自然进化得来的语言。

**可移植性-portability:** 程序能在多种计算机上运行的特性。

**解释-interpret:** 一次一行地执行高阶语言编写的程序。

**编译-compile:** 一次把一个高阶语言编写的程序全部翻译成低阶语言以备稍后执行使用。

**源代码-source code:** 未经编译的用**高阶语言**编写的程序。

**目标代码-object code:** 编译器将程序翻译得到的结果。

**可执行代码-executable:** 目标代码的另一种叫法。

**字节码-byte code:** Java 程序使用的一种特殊目标代码。字节码类似低阶语言, 但是又像高阶语言可移植。

**语句-statement:** 程序的组成部分, 指明程序运行时执行的一个动作。打印 (*print*) 语句在屏幕上显示一个输出结果。

**注释-comment:** 包含程序的有关信息但对程序运行没有影响的组成部分。

**算法-algorithm:** 解决某一类问题的一般步骤。

**臭虫-bug:** 程序的错误。

**语法-syntax:** 程序的结构。

**语义-semantics:** 程序的意思。

**解析-parse:** 检查程序, 分析语法结构。

**语法错误-syntax error:** 造成程序不可解析 (因而也不能编译) 的错误。

**异常-exception:** 造成在程序运行期间失败的错误, 也叫**运行时错误** (*run-time error*)。

**逻辑错误-logical error:** 造成程序所做和预期不一致的错误。

**调试-debugging:** 找出并除去三种错误中任何一种的过程。



## 第 2 章 变量和型别

### 2.1 关于打印的更多内容

变量上一章提到在 `main` 方法内可以放任意多条语句。下面用一个进行多行打印的程序为例说明：

```
class Hello {  
    // main: generate some simple output  
    public static void main (String[] args) {  
        System.out.println ("Hello, world."); // print one line  
        System.out.println ("How are you?"); // print another  
    }  
}
```

如你所见，把注释放在一行语句的后面和单独放在一行都是合法的。放在双引号内的字符序列是一个整体，叫做**字符串**（*string*）。组成字符串的可以是字母、数字、标点符号或者其它任何特殊字符。

`println` 是“**print line**”的简写。它在每一行之后加上了一个特殊字符——换行符，使得在屏幕上光标被移到下一行，再次调用 `println` 的时候新的字符被显示在这一行上。

要把多个语句打印的内容显示在输出端的同一行上，可以用 `print` 命令：

```
class Hello {  
    // main: generate some simple output  
    public static void main (String[] args) {  
        System.out.print ("Goodbye, ");  
        System.out.println ("cruel world!");  
    }  
}
```

这个程序将只产生一行输出：“Goodbye, cruel world!”。注意：“Goodbye”和第二个引号之间有一个空格，它出现在输出结果中，表明它对程序的行为产生了影响。

而引号外的空格则不会影响程序的动作。看这个例子：

```
class Hello {public static void main (String[] args) {  
    System.out.print ("Goodbye, ");  
    System.out.println ("cruel world!"); } }
```

这个版本和上一个比较，无论编译还是运行结果都一样。

同样，换行也不影响程序的行为，所以可以写成：

```
class Hello { public static void main (String[] args) {  
    System.out.print ("Goodbye, "); System.out.println  
    ("cruel world!");}}
```

这样也可以正常工作。但是你可能已经注意到了，程序变得越来越难读。

恰当的换行和空格有助于把程序排列得美观便于阅读和定位语法错误。

## 2.2 变量

操纵变量的能力是编程语言最强大的功能之一。**变量** (*variable*) 是一块被命名用来存放**值** (*value*) 的内存空间。值可以打印、存贮或者进行其它操作。我们打印的字符串 "Hello, World."、"Goodbye, " 等等都是值。

要存贮值就要创建变量。例如要存贮字符串，就这样声明一个新的字符串型变量：

```
String fred;
```

这个语句声明了一个名为 `fred` 的变量，它的型别是 *String*。每个变量都有一个型别，用来指明它能存贮什么型别的值。例如 `int` 型别的变量能存贮整数，`String` 型别的能存贮字符串。

细心的话就会发现，有的型别以大写字母开头，有的以小写字母开头。稍后我们就会知道这种区别的重要性，现在只要保证别写错就行了。Java 没有什么 `Int` 或者 `string` 型别，如果你试图造一个出来，编译器会生气的。

创建整数型别变量的语法像这样：

```
int bob;
```

——其中 `bob` 是任意取的变量名。通常一个好的变量名一看就明白是用来做什么的。例如这几个变量：

```
String firstName;  
String lastName;  
int hour, minute;
```

很容易理解它们存贮的值。此例还演示了声明多个同型别变量的语法，`hour` 和 `minute` 都是整数 (*int* 型别)。

## 2.3 赋值

创建了变量，接下来要存贮值。**赋值语句** (*assignment statement*) 像这个

样子：

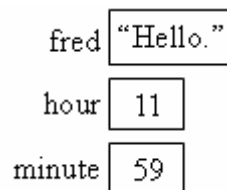
```
fred = "Hello.";    // 把值"Hello."赋给 fred
hour = 11;          // 给 hour 赋值 11
minute = 59;        // 将 minute 赋为 56
```

注释看起来大相径庭——也许还有更花哨的说法——但实际是谈论同一件事情：

- 声明变量，创建一块已命名的内存空间
- 用赋值语句让变量存贮一个值

形象地表示变量的常用办法是画一个框，里面写值的内容，框外写变量名。

下图示范了上面三条赋值语句的效用：



规则：声明变量要选择合理型别，赋值要和变量的型别匹配。把 hour（小时）声明为 *String* 型别有点莫名其妙，而把 "Hello" 赋值给 *int* 型别变量 minute 则完全是错误的。

我们常常说，第一条规则就是：任何规则都有例外。真是迷人的规则。有很多办法可以在型别之间转换，而且 Java 经常很“体贴”地自动帮你转换型别。现在只管记住规则，稍后我们会专门讨论这个问题。

还有一些容易混淆的东西：有的字符串看上去像整数，但它不是。例如 fred 的值可能是 1、2、3 组成的字符串 "123"，但这跟 123 不是一回事。

```
fred = "123";    // 合法
fred = 123;      // 非法
```

## 2.4 打印变量

前面用来打印字符串的命令也可以打印出变量的值。

```
class Hello {
    public static void main (String[] args) {
        String firstLine;
        firstLine = "Hello, again!";
        System.out.println (firstLine);
    }
}
```

这个程序声明了一个名为 `firstLine` 的变量并为它赋值 `"Hello, again!"`，然后打印该值。实际上“打印变量”意思是打印变量的值。要打印变量名，只有求助于双引号。这样：

```
System.out.println ("firstLine");
```

稍微思考一下这个程序片断：

```
String firstLine;  
firstLine = "Hello, again!";  
System.out.print ("The value of firstLine is ");  
System.out.println (firstLine);
```

没错，程序输出 `The value of firstLine is Hello, again!`

无论打印什么型别的变量语法都是一样的：

```
int hour, minute;  
hour = 11;  
minute = 59;  
System.out.print ("The current time is ");  
System.out.print (hour);  
System.out.print (":");  
System.out.print (minute);  
System.out.println (".");
```

这段程序将输出 `The current time is 11:59`。

注意：为了在一行上打印多个值而使用多个 `print` 命令最后一定要有一个 `println`。在很多环境中，`print` 命令的输出一直要等到调用 `println` 之后才被显示，如果你忽略了 `println`，程序根本就不会把存贮的输出显示出来。

## 2.5 Java 关键字

在前面几个小节里说可以为变量任意取名，实际不是那么绝对。有一组特定的词是 Java 的**保留字** (*reserved*)，用来供编译器解析程序结构。如果你用这些词来做变量名一定会自找麻烦。这些棘手的家伙有时也叫做**关键字** (*keywords*)，包括 *public*、*class*、*void*、*int*，等等。

完整的保留字清单可以在这个网址找到：

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/keywords.html>

这是 Java 的发明者 Sun 公司发布的网站，本书多处参考了它提供的 Java 文档。但与其去记忆这个清单，不如善用具体开发环境提供的语法高亮等功能来掌握这些关键字。在许多专门的编辑器或者开发工具内部的编辑器中，输入

关键字会被显示为彩色，不同的程序组成部分也会用各种不同的颜色显示。例如字符串可能是红色的，而一般说来关键字是蓝色的，注释则被显示成绿色。所以如果你输入的变量名显示为蓝色，注意了：它可能会导致编译器做一些莫名其妙的事情。

## 2.6 运算符

**运算符** (*operator*) 是专门用来表示像加法和乘法这类简单计算的符号。大多数运算符都很容易掌握，就是普通的数学符号。例如，两个整数相加的符号就是+。

下面是几个含义很直观的合法表达式：

```
1+1  hour-1  hour*60 + minute  minute/60
```

**表达式** (*expression*) 可以包含变量名和数字。在任何情况下执行计算前变量名会被值替换。

加减法和乘法一如平常，但是除法有些不一样。看下面的程序：

```
int hour, minute;
hour = 11;
minute = 59;

System.out.print ("Number of minutes since midnight: ");
System.out.println (hour*60 + minute);

System.out.print ("Fraction of the hour that has passed: ");
System.out.println (minute/60);
```

产生输出：

Number of minutes since midnight: 719

Fraction of the hour that has passed: 0

第一行输出正如所料，可是第二行有点不对劲：变量 `minute` 的值 59 除以 60 应该是 0.98333 而不是 0。产生误差的原因是 Java 执行了**整数除法** (*integer division*)。

如果两个**操作数** (*operand*) 都是整数型别，结果也必定是整数，而且整数除法是只舍不进。

这种情况下有一个替代的办法——计算百分比：

```
System.out.print ("Percentage of the hour that has passed: ");
System.out.println (minute*100/60);
```

结果是：Percentage of the hour that has passed: 98

依然被舍去了一部分，但至少很接近正确答案。为了得到更精确的答案，

需要另一个能够贮存小数的型别叫做**浮点数** (*floating-point*)，下一章就会谈到。

## 2.7 运算符优先级

一个表达式里出现多个运算符的时候，计算次序取决于运算符的**优先级** (*precedence*) 规则。完完整整地解释优先级规则会大费唇舌，现在看看最基本的几条：

- 乘除法优先级高于加减法。所以  $2*3-1$  等于 5 而不是 4， $2/3-1$  等于 -1 而不是 1（记住整数除法  $2/3$  等于 0）。

- 如果运算符优先级相同则从左到右进行计算。所以表达式 `minute*100/60` 中先做乘法，得到 `5900/60`，然后做除法得到 98。

- 如果你想摆脱上面两条的限制（或者记不清楚规则）时，可以使用括号。括号内的表达式被首先执行，所以  $2 * (3-1)$  等于 4。还可以用括号使表达式更加易读，例如可以写 `(minute * 100) / 60`，这不会改变结果。

## 2.8 字符串运算符

一般说来不能对字符串做数学运算，即使看起来像数字的字符串也不行。下面的表达式是非法的（`fred` 是 *String* 型别）：

```
fred - 1      "Hello"/123      fred * "Hello"
```

顺便说说，你能从这些表达式里判断出 `fred` 的型别是整数还是字符串吗？不能。判断一个变量型别的办法只能是看声明它的语句。

有趣的是，虽然和你的期望不完全一致，但 `+` 运算符确实可以进行字符串运算。`+` 运算符用于字符串表示**连接** (*concatenation*)，即把两个操作数（这里是字符串）首位相连。所以 `"Hello, " + "world."` 产生新的字符串 `"Hello, world."`，`fred + "ism"` 把后缀 `"ism"` 添加到 `fred` 的后面，无论 `fred` 的值是什么。

## 2.9 复合

目前为止我们已经分别讨论了变量、表达式和语句，但是还没有谈到怎么把它们联合起来使用。

编程语言最有用的功能之一就是小块元素组合起来成为复合语句。例如，有了乘法和打印语句，自然而然就可以一次做两件事情：

```
System.out.println (17 * 3);
```

准确地讲不能说是“一次”，因为实际上是先算乘法再打印。要点在于，包含数字、字符串和变量的表达式，可以在一个打印语句里面使用。前面已经有一例：

```
System.out.println (hour*60 + minute);
```

同时，在赋值语句右边可以放任意长度的表达式：

```
int percentage;  
percentage = (minute * 100) / 60;
```

目前看来可能体会不到这种能力的好处，但后面有一些例子将表明正是因为复合语句而使得优雅而简洁的表达计算成为可能。

注意：不能滥用表达式。赋值语句的左边只能是一个变量名而不能是表达式，因为左边指向内存中用来存贮结果的一片区域。表达式不能表示存贮区域，只能代表值。所以这样写是非法的：`minute+1 = hour;`

## 2.10 术语表

**变量-variable**：一块被命名用来贮存值的内存区域。每个变量都有一个型别，创建变量时就要声明相应的型别。

**值-value**：可以贮存的诸如一个数字或字符串（或其它稍后会讲到）的内容。每个值属于一类型别。

**型别-type**：值的集合。变量的型别决定了它能存贮何种值。目前为止我们见过的型别是整数型别（Java 中的 `int`）和字符串型别（Java 中的 `String`）。

**关键字-keyword**：被保留下来供编译器解析程序用的单词。不能用 `public`, `class` 和 `void` 等等关键字做变量名。

**语句-statement**：一行表示一个命令或者动作的代码。目前见过的是声明语句、赋值语句和打印语句。

**声明-declaration**：创建一个新变量并指明其型别的语句。

**赋值语句-assignment**：把值赋予变量的语句。

**表达式-expression**：变量、运算符和值的组合，用来表示一个运算结果。表达式的型别取决于运算符和操作数。

**运算符-operator**：用来表示加减法和字符串连接这类简单计算的符号。

**操作数-operand**：运算符操作的值。

**优先级-precedence**：执行运算的次序。

**连接-concatenate**：把两个操作数首尾相连。

**复合-composition**：把简单表达式和语句组合成为一个复合语句/表达式，获得简洁表示一些复杂计算的能力。

## 第3章 方法

### 3.1 浮点数

上一章处理非整数的时候遇到了一点麻烦，我们用计算百分比代替计算分数的方法近似地处理了这个问题。而更常用的解决方案是采用既可以表示整数也可以表示小数的**浮点数** (*floating-point*)。在 Java 中，浮点数叫做**双精度型别** (*double*) 数。

创建浮点数变量并赋值的语法和其它型别一样。看例子：

```
double pi;  
pi = 3.14159;
```

声明变量的同时就赋值也是合法的：

```
int x = 1;  
String empty = "";  
double pi = 3.14159;
```

实际上这种声明/赋值复合语句的语法更常见，并且有时也叫做**初始化语句** (*initialization*)。

虽然浮点数很有用，但它常常导致困惑，因为整数和浮点数似乎有很多重叠的部分。例如值 1，是整数还是浮点数？或者二者皆是？

严格地讲，Java 把值 1 视为整数而 1.0 才是浮点数——即使看起来像是同一个数。它们属于不同的型别，而在不同型别之间赋值是非法的。例如，下面的语句是非法的：

```
int x = 1.1;
```

因为左边的变量是 *int* 型别而右边的值是双精度数。

特别是因为 Java 在很多地方会自动转换型别，所以很容易忘记这个规则。例如：

```
double y = 1;
```

技术上讲语法是错误的，Java 允许这样写但是会自动把 *int* 型别转换为双精度型别。这种宽松带来方便，但是也能导致问题发生：

```
double y = 1 / 3;
```

你可能期望 y 被赋值为合法的双精度数 0.333333，可实际上它会被赋值为 0.0。原因在于，等号右边是两个整数型别数相除，于是 Java 做了整数除法得到 0，转换为双精度数，结果就是 0.0。



一旦找到问题所在，解决办法就是把右边改为浮点数的表达式：

```
double y = y = 1.0 / 3.0;
```

这次如愿以偿，y 被赋值为 0.333333。

目前为止谈到的运算——加、减、乘、除法——都能用于浮点数值。不过有趣的是背后的机制却完全不同，事实上大多数 **CUP** 有专门执行浮点运算的硬件。

## 3.2 浮点数转换为整数

前面已经提到，如有必要 Java 会把整数自动转换为双精度数，因为转换时不会丢失信息。反过来把浮点数转换为整数却需要舍入。为了让程序员（阁下）意识到丢失了小数部分，Java 不会自动执行转换。

最简单的办法是用**型别浇铸符**（*typecast*）。这么叫是因为它允许你把一个型别的值“倒进（cast into）”另一个型别中就好像浇铸一样，而不是当垃圾倒掉。

太不幸了，型别浇铸的语法很丑陋：你得把型别放在括号里做为一个运算符。像这样：

```
int x = (int) Math.PI;
```

运算符 (int) 把它的跟班转换为整数型别，所以 x 就得到了值 3。

型别浇铸的优先级高于数学运算，所以下一例中，PI 先被转换为整数，结果 x 得到值 60 而不是 62。

```
int x = (int) Math.PI * 20.0;
```

转换为整数型别也是只舍不进，即使小数部分是 0.99999999。

这两条性质（优先级和舍入——实际上只舍不入）把型别浇铸搞的笨拙不堪。

## 3.3 数学方法

在数学课上应该见过 sin 和 log 这样的函数，也学过计算  $\sin(\pi/2)$  和  $\log(1/x)$  的方法。第一步要算括号内的表达式的值——函数的自变量或者叫参数。例如： $\pi/2$  约等于 1.57， $1/x$  等于 0.1（假设 x 的值是 10）。

然后，不管通过查表还是各种计算方法，这才算函数值，1.571 的正弦是 1，0.1 的对数是 -1（假设 log 代表底为 10 的对数函数）。

这个步骤可以不断重复从而计算像  $\log(1/\sin(\pi/2))$  这样更复杂的表达式。先算内函数的参数，然后算内函数的值……

Java 提供了一个内建函数的集合，其中包括了大多数你能想到的数学运算。这些函数叫做方法，大部分数学方法都是对双精度数执行运算。

调用数学方法的语法和打印命令差不多：

```
double root = Math.sqrt (17.0);  
double angle = 1.5;  
double height = Math.sin (angle);
```

第一句为 `root` 赋值 17 的平方根；第二、三句查找 1.5 的正弦值并赋予 `angle`。Java 假定正弦函数和其它三角函数计算的是弧度的函数值。要把角度转换为弧度可以先除以 360 再乘以  $2\pi$ 。为了方便 Java 把  $\pi$  做为一个内建值。

```
double degrees = 90;  
double angle = degrees * 2 * Math.PI / 360.0;
```

注意 `PI` 要全部大写。也许你更喜欢 `Pi`、`pi` 或者 `pie`，可惜 Java 概不认帐。`Math` 类中另外一个有用的数学方法是 `round`，能够把一个浮点数四舍五入为最近似的整数并返回一个 `int` 型别的值：

```
int x = Math.round (Math.PI * 20.0);
```

本例中调用方法前先做乘法，最后结果是 63（由 62.8319 进位舍入得到）。

### 3.4 复合

像数学函数一样，Java 方法也可以进行组合。这表示可以把一个表达式做为另一个的组成部分。例如可以把任何表达式做为一个方法的参数：

```
double x = Math.cos (angle + Math.PI/2);
```

这句首先取得 `Math.PI`，除以 2，然后把结果和变量 `angle` 的值相加，和被做为参数传递给 `cos` 方法（注意 `PI` 是变量名不是方法，所以没有参数，连空参数()也没有）。

也可以取得一个方法的结果再把这结果当做参数传递给另一个方法：

```
double x = Math.exp (Math.log (10.0));
```

Java 中对数函数总是用 `e` 做为底，所以这条语句先算出以 `e` 为底 10 的对数值，然后再求这个值以 `e` 做为指数的幂，最后把结果赋给 `x`——希望你明白我说的是什么。

### 3.5 新增方法

到目前为止我们一直用的是 Java 内建的方法，但要创建新方法也是可以的。事实上我们见过一个方法的定义：`main`。`main` 方法比较特殊，它指明了程序执行入口，不过 `main` 方法的语法和其它方法定义是一样的：

```
public static void NAME ( LIST OF PARAMETERS ) {  
    STATEMENTS  
}
```

自定义的方法名可以任意取——除了 `main` 和其它关键字。参数列表（如果有的话）表明要调用方法必须提供的信息。

`main` 方法唯一的参数就是 `String[] args`，它指明调用 `main` 方法必须提供一个字符串数组（第 10 章会讨论数组）。下面写两个没有参数的方法，语法像这样：

```
public static void newLine () {  
    System.out.println ("");  
}
```

方法名叫 `newLine`，空空如也的括号表明没有参数。它只有一条语句，参数 `""` 表明打印一个空字符串。打印一个不包含任何字符的字符串看起来可能有点滑稽，但是请记住用 `println` 输出完后下一次输出将位于新的一行，所以这条语句的作用就是换行。

和调用内建 Java 命令的语法一样，可以在 `main` 方法内调用这个新方法：

```
public static void main (String[] args) {  
    System.out.println ("First line.");  
    newLine ();  
    System.out.println ("Second line.");  
}
```

程序输出：

First line.

Second line.

注意两行中的空行。如果想使中间有更多的空行怎么办？那就重复调用同一个方法：

```
public static void main (String[] args) {  
    System.out.println ("First line.");
```

```
newLine ();  
newLine ();  
newLine ();  
System.out.println ("Second line.");  
}
```

或者也可以写一个新的方法 `threeLine` 打印三个空行：

```
public static void threeLine () {  
    newLine (); newLine (); newLine ();  
}  
public static void main (String[] args) {  
    System.out.println ("First line.");  
    threeLine ();  
    System.out.println ("Second line.");  
}
```

有几点是这个程序值得注意的地方：

- 可以重复调用同一个过程。这在实际运用中很常见有效。
- 一个方法可以调用另一个方法。本例中 `main` 调用 `threeLine`, `threeLine` 调用 `newLine`。同样地，这也很常见并且一样有用。
- 在 `threeLine` 方法中三个语句写在一行上，语法是正确的（记得吗，空格和换行不改变程序的意义）。回过头来说，把每个语句单独放一行对于方便阅读程序更好些。为了节约空间有时本书会打破这个规则。

现在为什么要不厌其烦创建这些新方法可能还不是很清楚。真正的原因有很多，但这个例子只能说明两点：

**1.**创建新方法使你有机会给一组语句起一个名字。方法通过一个英语单词就隐藏了许多复杂的计算和咒语似的命令，简化了程序。比较 `newLine` 和 `System.out.println("")`，哪个更直观？

**2.**创建新方法剔除了唠叨的代码使程序更短小。例如要打印九行连续的空行要怎么做？只需要调用三次 `threeLine` 方法。

## 3.6 类和方法

把前面几小节的零散代码整合起来，一个完整的类定义就应运而生了：

```
class NewLine {  
    public static void newLine () {  
        System.out.println ("");  
    }  
    public static void threeLine () {  
        newLine (); newLine (); newLine ();  
    }  
    public static void main (String[] args) {  
        System.out.println ("First line.");  
        threeLine ();  
        System.out.println ("Second line.");  
    }  
}
```

第一行指明这是一个名叫 `NewLine` 的新类的声明。**类** (*class*) 是方法的集合。这一例中 `NewLine` 包含三个方法 `newLine`、`threeLine` 和 `main`。

前面还见过 `Math` 类，它有 `sqrt`、`sin` 以及其它许多方法。调用数学函数的时候，必须指明类名 (`Math`) 和函数名，这也是调用内建方法和自定义方法在语法上的细微区别：

```
Math.pow (2.0, 10.0);  
newLine ();
```

第一条语句调用 `Math` 类中的 `pow` 方法（把第一个参数做底第二个参数做指数进行幂运算）。第二条语句调用 `newLine` 方法，**Java** 假定它在 `NewLine` 类中。

如果试图从错误的类中调用方法，编译器会出错。例如输入：

```
pow (2.0, 10.0);
```

编译器会像这样说，“在类 `NewLine` 中找不到 `pow` 方法”。你可能见过这条信息并且奇怪它为什么要到你的类里找 `pow` 方法，现在知道为什么了吧。

## 3.7 多方法程序

看见包含多个方法的程序就试图从头到尾读一遍，估计要把脑袋搞晕。因为这本来就不是程序执行的顺序。

执行动作总是从 `main` 方法的第一条语句开始，而不管它在程序的任何位置

（这里我故意放在程序的尾巴上）。语句被一一逐行执行，直到调用了方法。方法的调用好比在执行流程中绕道而行，不去执行下一条语句，而是跳到所调用方法的第一条语句，执行完毕其中的所有语句，再返回到离开的地方继续下去。

听起来很简单，然而你必须记住：方法可以调用方法。于是，在 `main` 方法中间可能不得不跳过去执行 `threeLine` 中的语句；但是执行 `threeLine` 方法的时候，又要中断三次转为执行 `newLine` 方法。

在这个的部分，`newLine` 方法调用了内建的 `println` 方法，又导致另一次绕道而行。幸运的是 `Java` 相当擅长于跟踪流程，于是当 `println` 方法完成后，又回到了离开 `newLine` 方法的地方，然后又回到离开 `threeLine` 方法的地方，最后回到 `main` 方法以使程序得以中止。

从技术上准确地讲，程序并不在 `main` 方法最后中止。`Java` 解释器在调用 `main` 方法结束后处理剩余一切，包括删除窗口和例行清扫等等事务。

那么这个差劲的故事寓意到底何在？那就是：不要从头到尾像看童话一样读一个程序，要按照执行流程来读。

### 3.8 形参和实参

前面用到的一些内建方法带有形参。**形参** (*parameter*) 是提供给方法以使其进行工作的值。例如要求一个数的正弦首先要知道该数是多少，所以 `sin` 方法用一个双精度数做为形参；要打印一个字符串就要提供字符串，于是 `println` 方法用 *String* 型别的值做形参。

有的方法不止一个形参，例如 `pow` 方法有两个双精度型别的形参，一个做底一个做指数。

注意，在这些情况下不仅要声明有几个形参同时还要说明分别都是什么型别。所以写类声明时自然而然要在形参列表中分别说明每个形参的型别，就像这样：

```
public static void printTwice (String phil) {  
    System.out.println (phil);  
    System.out.println (phil);  
}
```

这个方法只有一个形参，名唤 `phil` 型别是 *String*。无论形参的值是什么（这里无法看出来）都被打印两次。这里的 `phil` 可以是别的什么名字，不过通常最好取一个比这更直观的形参名。

要调用这个方法就要提供一个 *String* 型别的值。例如有一个 `main` 方法如下：

```
public static void main (String[] args) {  
    printTwice ("Don't make me say this twice!");  
}
```

这里提供的字符串叫做实参。专业的说法就是把实参“传递 (*pass to*)”给方法。本例创建了一个值为 "Don't make me say this twice!" 的字符串，并把它做为一个实参传递给方法 `printTwice`，违背其本意照样打印两次。

还有一个办法，如果创建了一个 `String` 型别的变量，也可以用来做实参：

```
public static void main (String[] args) {  
    String argument = "Never say never.";  
    printTwice (argument);  
}
```

特别提请重视：此处做为实参传递的变量名 (`argument`，并非关键字) 和方法定义中的形参名 (`phil`) 没有任何相干。

再重复一次：

**形参名和做为实参传递的变量名没有任何联系。**

换句话说，它们可以一样也可以不一样，关键是要搞清楚二者并非一个事物。除非碰巧拥有一样的值（在这段程序中都是字符串 "Never say never."）。

做为实参提供的值必须和所调用方法的形参型别一致。这条规则非常关键，可是在 Java 中有两个原因使之显得很复杂：

- 有的方法可以接收多种型别的实参。例如传递任意型别的值给 `print` 和 `println`，都能正确打印出来。然而这是少数例外。

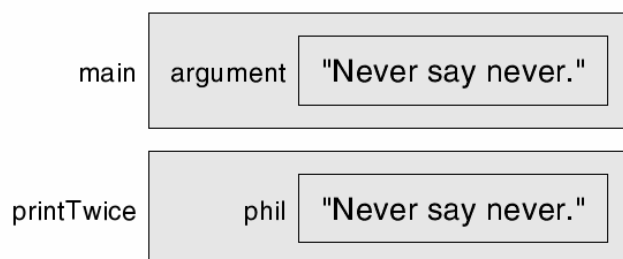
- 一旦你对该规则不恭，编译器将产生一条古怪的信息。它不会这样告诉你：“对方法传递的值参是错误的型别”，而极有可能说结果是找不到能接收该型别值参的方法。多看到几次这样的提示，你就能解释是怎么回事了。

### 3.9 堆栈图

形参和其它变量只能在其自己的方法内存在。在 `main` 方法的有效范围内，没有 `phil`，如果非要用，编译器会抱怨的。类似的，`printTwice` 方法内部也没有 `argument` 这东西。

跟踪每个变量定义点的办法之一是画堆栈图 (*stack diagram*)。

上一例的堆栈图是这个样子：



每个方法有一个灰色的盒子叫做框 (*frame*), 里面包含方法的形参和本地变量, 方法名在框外面, 通常又把每个值写在单独的盒子里, 相应的变量名放在旁边。

译注: 本章开始讨论并频繁提及“参数”(parameter), 而另外一个常常也被翻译为“参数”的英文单词是 argument。实际上二者的定义有细微但关键的区别, 但即使在许多英文资料中有时也被混用。除了在本节(2.8)中因为专门讨论的缘故把它们区别出来分别译作常见的“形参”(parameter)和“实参”(argument), 在以前和后面的章节中除非特别需要, 将统一译作“参数”以便于行文和阅读。重要的是看到“参数”的时候心里明白到底说的是哪种参数就行了。如果有疑问或者觉得译文不便理解, 可以对照原文阅读。由于译者功力低微时间有限, 做出如此 ugly 的解释, 还望见谅。

### 3.10 多参数方法

声明和调用带有多个参数的方法常常是错误产生的根源。首先要记得为每个参数声明。看这个例子:

```
public static void printTime (int hour, int minute) {
    System.out.print (hour);
    System.out.print (":");
    System.out.println (minute);
}
```

你可能想写成 `int hour, minute`, 可惜这种格式对声明变量合法, 对参数则不是。

另一个造成困扰的根源是原本不必为实参声明的型别。下面就是这种错误的例子:

```
int hour = 11;
int minute = 59;
```



```
printTime (int hour, int minute);    // 错误!
```

这里 Java 自会从声明中判断 `hour` 和 `minute` 的型别。做为参数传递的时候又去声明型别不仅是多此一举而且是非法的。正确的语法是：

```
printTime (hour, minute).
```

### 3.11 带返回结果的方法

你可能注意到了，有的方法返回结果，例如 `Math` 类的方法，有的方法例如 `println` 和 `newLine` 只是执行一些动作而不返回结果。这就引发了几个问题：

- 如果调用一个方法但是不对结果做任何处理（就是说不把结果赋值给变量，也不做为一个更大的表达式的一部分）会怎么样？

- 如果把一个 `print` 方法做为一个表达式的一部分会怎么样？例如像这样：

```
System.out.println ("boo!") + 7;
```

- 可以自己写能产生结果的方法吗？还是只能写一些像 `newLine` 和 `printTwice` 这么滑稽的玩意儿？

第三个问题的答案是：“可以，自定义的方法也可以返回值。”而且后面会有很多章就做这件事情。至于另外两个问题，留给你去试一试然后自己做出回答。实际上对 Java 中什么是合法还是非法有疑问时，最好的答案就在编译器那里。

### 3.12 术语表



**浮点数-floating-point:** 变量（或者值）的型别，既能包含小数也能包含整数。Java 中叫做双精度数。

**类-class:** 具名的方法集合。目前为止拥过 `Math` 和 `System` 类，还有自己写的 `Hello` 和 `NewLine` 类。

**方法-method:** 有一定功能，具名的一个语句序列。方法不一定有参数也不一定生成结果。

**形参-parameter:** 为调用方法而提供的一系列信息。从具备值和型别的角度看形参像是变量。

**实参-argument:** 调用方法时提供的值。这个值必须和相应的形参匹配型别。

**调用-invoke:** 使方法得以执行。

**型别浇铸符-typecast:** 把型别由此转彼的运算符。在 Java 中把型别名放在括号里构成。例如 `(int)`。

## 第 4 章 条件语句和递归

### 4.1 取模运算符

取模运算符对整数（或整型表达式）进行操作，得到第一个操作数除以第二个操作数的余数。Java 的取模运算符是百分号%。语法和其它运算符很相似：

```
int quotient = 7 / 3;  
int remainder = 7 % 3;
```

第一个运算符是整数除法，所以得到 2，7 除以 3 等于 2 余数为 1，所以第二个运算符的作用结果是 1。

实践中取模运算符的用处非常大。例如可以用来检验一个数能否被另一个数整除：如果  $x \% y$  等于 0，则  $x$  可以被  $y$  除尽。

还有就是可以利用取模运算符来提取一个数最右边的一个或多个数字。例如  $x \% 10$  结果得到  $x$  最右边的那个数字（基于十进制），类似的  $x \% 100$  取得  $x$  最右边的两位数字。

### 4.2 条件执行

为了编写有用的程序，常常需要检测特定条件然后相应的改变程序行为。**条件语句**（*conditional statement*）给出了这种能力。最简单的是 `if` 语句：

```
if (x > 0) {  
    System.out.println ("x is positive");  
}
```

括号内的表达式叫做条件。如果条件为真，则语句花括弧内的语句得以执行，如果条件为不为真就什么也不发生。

条件可以包含任何比较运算符，有时也叫**关系运算符**：

```
x == y    // x 等于 y  
x != y    // x 不等于 y  
x > y     // x 大于 y  
x < y     // x 小于 y  
x >= y    // x 大于等于 y  
x <= y    // x 小于等于 y
```

虽然这些运算符都是众所周知的，但在 Java 的语法中和数学符号 123 还是不一样。最常见的错误就是把 `=` 当成 `==` 来使用。请记住 `=` 是赋值运算符 `==` 才是比

较运算符，而且 Java 中也没有 `=<` 和 `=>` 这两种运算符。

条件运算符两边必须是一样的型别。只能是整形和整形比较、双精度型和双精度型比较。不幸的是，字符串无法在这里进行比较。比较字符串的办法要等几章后才会谈到。

## 4.3 选择执行

条件执行的第二种形式是选择执行，当有两种可能的时候，条件决定执行哪一个。语法像这样：

```
if (x%2 == 0) {  
    System.out.println ("x is even");  
} else {  
    System.out.println ("x is odd");  
}
```

如果 `x` 除以 2 的余数是 0，表明 `x` 是偶数，代码中的 `even` 即英语单词“偶数”，所以将输出信息“`x` 是偶数”。如果条件不为真，执行第二条语句，打印“`x` 是奇数”。由于条件非真即假，所以二者之一必有一个被执行。

稍微跑一下题，实践中常常用到奇偶分类检验，可以像下面这样把上面的代码“打包”进一个方法：

```
public static void printParity (int x) {  
    if (x%2 == 0) {  
        System.out.println ("x is even");  
    } else {  
        System.out.println ("x is odd");  
    }  
}
```

现在有了一个名叫 `printParity` 的方法，能根据你乐意提供的任何整数打印出相应的信息。在 `main` 方法中可以这样调用该方法：

```
printParity (17);
```

请总是记住，调用方法时不要声明参数的型别，Java 自会去搞清楚。要惯于抵抗写这种玩意儿的诱惑：

```
int number = 17;  
printParity (int number); // 错误的诱惑！
```

## 4.4 链式条件

有时需要检查一系列相关条件以选择执行一至多个动作。达到目的的办法之一是把一连串 `if` 和 `else` 链接 (*chain*) 起来:

```
if (x > 0) {
    System.out.println ("x is positive");
} else if (x < 0) {
    System.out.println ("x is negative");
} else {
    System.out.println ("x is zero");
}
```

这个链条可以任意伸长,当然,失去控制了也很难以阅读。要便于阅读最好像上例演示这样采用标准的锯齿状排列。把所有语句和花括弧对齐可以减少犯语法错误的可能而且即使出错也能很快找到。

## 4.5 嵌套条件

为了对链式条件进行补充,还可以把一个条件嵌入另一个。上面的例子也可以这么写:

```
if (x == 0) {
    System.out.println ("x is zero");
} else {
    if (x > 0) {
        System.out.println ("x is positive");
    } else {
        System.out.println ("x is negative");
    }
}
```

最外层的条件有两种可能。第一个分支况包含一个简单的打印语句,而第二个分支包含另一个条件语句,该条件自身又有两个分支,不过这次简单了,两个分支都是打印语句。

再提醒注意,锯齿状的排列虽然能让结构显得清晰,但嵌套很快就会让程序难以阅读。一般说来要尽量避免使用嵌套语句。

话说回来,这种结构很常见,所以最好还是慢慢习惯它吧。

## 4.6 返回语句

**返回语句** (*return statement*) 允许在程序流探底之前终止执行。假如要侦测一个错误条件就可以用到:

```
public static void printLogarithm (double x) {  
    if (x <= 0.0) {  
        System.out.println ("Positive numbers only, please.");  
        return;  
    }  
    double result = Math.log (x);  
    System.out.println ("The log of x is " + result);  
}
```

这段程序定义了一个 `printLogarithm` 方法, 带有一个 *double* 型别的参数 `x`。首先它检查 `x` 是否不大于零, 符合条件就打印一条出错提示, 然后用 `return` 语句退出方法, 随即执行流程返回到调用者, 剩余的部分不被执行。

因为左边的变量是双精度型别, 所以右边用一个浮点数作为值。

## 4.7 型别转换

那么要怎么避免写出像 `"The log of x is " + result` 这样的一个操作数是 *String* 型别而另一个是 *double* 型别的表达式呢? 放心, 这回 Java 聪明地帮我们解决了麻烦: 在进行字符串连接运算之前, 把 *double* 型别的值自动转换为 *String* 型别。

这种结构的取舍是设计一种编程语言的时候常常遇到的问题: 形式化原则和易用性的矛盾。一边是形式化原则——要求形式语言具有简单的规则而且尽量少特例; 一边是易用性——要求编程语言便于实际应用。

更多的时候易用性占了上风, 通常这对那些讨厌刻板僵硬的形式规则的专家级程序员来说是个喜讯, 而对热衷于林林总总的规则和特例的初哥程序员则是一个坏消息。本书强调通行法则忽略特例以图简化问题。

虽然如此, 但知道这些特例是大有裨益的, 这样在把两个表达式“相加”的时候, 如果其中一个是 *String* 型别, 你就知道 Java 将把另一个转换成 *String* 型别再执行字符串连接运算。现在想想, 把一个整数和一个浮点数“相加”的时候会怎么样?

## 4.8 递归

上一章提到在方法中调用方法是合法的，前面已经举了几个例子，但我故意没有讲到一个方法其实也可以调用自身。现在谈论这么做的好处似乎还不明显，但这已被证明是程序能做的最不可思议、最有趣的事情之一。

例如下面的方法：

```
public static void countdown (int n) {  
    if (n == 0) {  
        System.out.println ("Blastoff!");  
    } else {  
        System.out.println (n);  
        countdown (n-1);  
    }  
}
```

方法名是 `countdown`，取一个整数做参数。如果参数值是零，打印 **Blastoff**，否则就调用一个名叫 `countdown` 的方法——它自己——并把原参数减 1 后作为传递的参数。

如果在 `main` 方法内像这样：`countdown (3)`；调用这个方法会发生什么呢？

```
countdown 方法从 n=3 执行，n 不为零，所以打印其值 3；然后调用它自身...  
    countdown 方法从 n=2 执行，n 不为零，所以打印其值 2；然后调用它自身...  
        countdown 方法从 n=1 执行，n 不为零，所以打印其值 3；然后调用它自身...  
            countdown 方法从 n=0 执行，n 为零，所以打印其值 Blastoff；然后返回。  
        以 n=1 调用的 countdown 方法返回。  
    以 n=2 调用的 countdown 方法返回。  
以 n=3 调用的 countdown 方法返回。  
最后回到 main 方法。
```

——多么漫长曲折的旅途啊！最后整个输出是：

3

2

1

**Blastoff!**

作为讨论第二个例子的出发点，先看看前面写的 `newLine` 和 `threeLine` 方法。

```
public static void newLine () {
```

```
System.out.println ("");  
}  
public static void threeLine () {  
    newLine (); newLine (); newLine ();  
}
```

这里虽然也用到了方法对方法的调用，但如果想打印 2 个空行或者 106 个空行，用这两个方法就太笨拙了。一个更好的替代方法是：

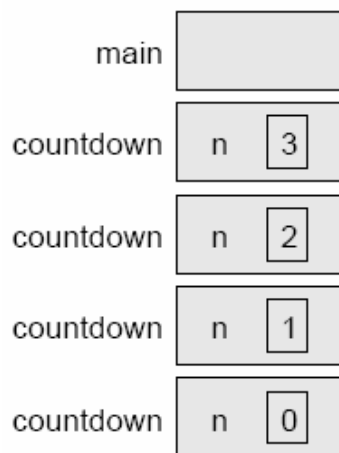
```
public static void nLines (int n) {  
    if (n > 0) {  
        System.out.println ("");  
        nLines (n-1);  
    }  
}
```

这个程序和第一个例子非常像，当  $n$  大于 0 的时候就打印一个空行，然后调用自身打印第  $n-1$  个空行，这样反复，最后打印的空行数就是  $1 + (n-1)$ ，即打印了  $n$  个空行。

一个方法对自身进行调用的过程就叫**递归** (*recursion*)，这样的方法称为**递归方法** (*recursive method*)。

## 4.9 递归方法的堆栈图

上一章讲过可以用堆栈图来表示进行方法调用时程序的状态。可以用这种图来更容易的理解递归方法。请记住，每当一个方法被调用时都会创建一个该方法的新实例，该实例包含自己的变量和参数。下图是用  $n = 3$  做参数调用 `countdown` 方法时的堆栈图：



图示了一个 `main` 方法和四个 `countdown` 的实例，每个实例都有不同的值做

参数。堆栈底部以 `n=0` 做参数的 `countdown` 方法是最基本的情况，没有用到递归调用，所有后面再没有 `countdown` 的实例。

`main` 方法的框是空的因为这个 `main` 方法没有任何参数或自己的变量。

## 4.10 约俗和法则

前面的章节中时有“一般说来”或者“通常”这样的字眼，这是因为在两个选择中没有特别具有说服力的理由要选择一个而舍弃另一个，于是就按照约俗来取舍。既然大家都这么做，为了自己好，还是要熟悉约俗并且照章办事，这样写出来的程序才便于他人理解。同时，区分至少三种规则就显得很重要：

**法则 (Divine law)**：这个词用来指代那些基于逻辑或数学原理的事实。这些法则对任何编程语言（或者其它形式化系统）都适用。例如，对于给定的四边形至少要有四个已知条件才能确定它的位置和大小。又比如两个整数相加可以交换位置，这是加法定义的一部分，和 `Java` 无关。

**Java 的规则 (Rules of Java)**：不可违反的 `Java` 语法和语义学规则，否则将导致程序不能编译或者运行。有些部分是可以任选的，比如符号“+”既可以表示加法也可以表示字符串连接；另外一些则反映了潜在的编译或者执行过程中的限制，比如必须声明形参的型别，而实参则相反。

**风格和约俗 (Style and convention)**：有很多规则不是编译器强加的，但对写出正确的程序——可调试、可修改并且可被别人解读——却是必要的。例如花括弧的锯齿状排列，还有变量、方法和类的命名约定。

学习的过程中，我将力图指明不同的情况适用的是什么类型的规则，但是学生也应该随时开动脑筋。读到这里，你可能已经搞清楚类名总是以大写字母开头；变量名和方法名以小写字母开头，如果一个名字包含多个词；通常把每个词的第一个字母大写——现在看看 `newLine` 和 `printParity` 两个名称，适用的是哪些规则？

## 4.11 术语表

**取模运算符-modulus**：对整数进行操作的运算符，结果是两数相除的余数。`Java` 中用百分号 (%) 来表示。

**条件语句-conditional**：根据情况而选择执行与否和如何执行的语句块。

**链接-chaining**：把几个条件语句首尾相连的一种办法。

**嵌套-nesting**：把一个条件句放在另一个条件句的一个或两个分支里。

**递归-recursion**：调用和当前执行方法相同的方法的过程。



## 第 5 章 带返回值的方法

### 5.1 返回值

前面用到的一些诸如 `Math` 函数这样的内建方法产生了结果，就是说，调用方法的效果就是生成了一个新的值，这个值被赋给一个变量或作为表达式的一部分。例如：

```
double e = Math.exp (1.0);  
double height = radius * Math.sin (angle);
```

不过迄今为止我们自己写的方法却都是 `void` 方法，不返回值。调用 `void` 方法时，不必传递参数：

```
nLines (3);  
g.drawOval (0, 0, width, height);
```

本章要写的方法将会返回一些东西，用“多产”来形容这种方法真是再贴切不过了。第一个例子叫做 `area`，取一个双精度数做参数，返回一个半径已知的圆的面积：

```
public static double area (double radius) {  
    double area = Math.PI * radius * radius;  
    return area;  
}
```

第一条应该注意方法的声明不再是以表示 `void` 方法的 `public static void` 开头，而是以表示将返回一个双精度数的 `public static double` 开头。那么 `public static` 是什么意思呢？别急，后面会讲。

其次要注意的就是最后一行是一个 `return` 语句，这句的含义是“立即从方法中返回，并把 `return` 后的表达式作为返回值”。被返回的expressions 的复杂度是任意的，所以这个方法可以写得更简洁：

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

话说回来，使用像 `area` 这样的临时变量可以让调试更容易。但无论如何，返回语句中表达式的值必须和方法返回值的型别一样。换句话说，如果声明的是 `double` 型别参数，就得保证方法最终生成一个双精度数。如果你想什么也不返回或者返回了错误类型的表达式——毫无疑问——编译器会找你的麻烦<sup>^^</sup>。

有时把多个返回语句分别放在几个条件分支中是很有用的：

```
public static double absoluteValue (double x) {  
    if (x < 0) {  
        return -x;  
    } else {  
        return x;  
    }  
}
```

显然这里的两个返回语句只有一个会被执行。虽然在一个方法中有多个返回语句是合法的，但是应该谨记，只要有一条返回语句执行，那么方法将不会继续执行后续的语句而是立即终止。

出现在返回语句之后（或别的什么地方）永远也不会被执行的代码叫做**无用代码**（*dead code*）。有的编译器会在出现无用代码时提出警告。

如果把返回语句放在条件语句中，就得保证程序执行中的每一个可能路径都能抵达返回语句。例如：

```
public static double absoluteValue (double x) {  
    if (x < 0) {  
        return -x;  
    } else if (x > 0) {  
        return x;  
    } // 错啦!!  
}
```



错在哪里？好吧：如果  $x$  碰巧是 0，则两种条件都不被满足，方法不能抵达返回语句就结束了。典型的编译器出错信息会是“absoluteValue 方法中需要 return 语句”，这条信息不是很古怪吗？明明有两条 return 语句啊。

## 5.2 程序开发

到了这里想必你已经对 Java 方法是什么和能干什么有一个认识了——但可能还不清楚如何才能胜任方法的编写工作。下面就推荐一种我称之为**递增开发**（*incremental development*）的技术。

举个例子，已知两点坐标  $(x_1; y_1)$  和  $(x_2; y_2)$  要找出两点之间的距离，根据定义——

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

第一步要考虑的是 `distance` 方法在 Java 中应该是什么样子，换句话说——输入（参数）和输出（返回值）是什么。

本例中，两点自然就是参数，用四个双精度数来表示它们也很容易想到，不过稍后会看到 Java 有一个 `Point` 对象可用；返回值不用说当然是距离，双精度数。

可以先描绘出一个轮廓来：

```
public static double distance
    (double x1, double y1, double x2, double y2) {
    return 0.0;
}
```

要想程序能编译通过，语句 `return 0.0;` 就是必须的占位符。显而易见，在这个水准上的程序什么有用的事情也干不了，但是在把程序弄得更复杂之前，很值得试着编译一下以确定是否存在任何语法错误。

为了测试这个新方法，必须给一组示例值以调用方法。在 `main` 方法内某处加上：

```
double dist = distance (1.0, 2.0, 4.0, 6.0);
```

选这一组值正好满足水平距离为 3 垂直距离为 4——计算结果应该是 5（勾三股四弦五）。测试方法的时候，知道正确结果是非常必要的。

检查完方法定义的语法错误，就可以开始一行一行增加代码了。每次递增一些改动，就重新编译运行一次。这样一来，任何时刻都能准确判断错误在哪里——当然是最后添加的一行。

下一步计算是求  $x_2 - x_1$  和  $y_2 - y_1$  的差，并把两个值分别存贮在临时变量 `dx` 和 `dy` 中。

```
public static double distance
    (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    System.out.println ("dx is " + dx);
    System.out.println ("dy is " + dy);
    return 0.0;
}
```

打印语句的作用是在继续下一步之前让我们有机会检查中间值，这里应该

分别是 3.0 和 4.0。

写完方法后可以（也应该）移除打印语句。这样的代码叫做“脚手架（*scaffolding*）”，因为它的作用是帮助搭建程序，但却不是最后产品的一部分。有个常用的好办法是把这些“脚手架”留下来，只不过将它注释掉，这样既不影响程序的动作，也便于以后万一用得到。

开发工作的下一步是计算  $dx$  和  $dy$  的平方。也许应该用 `Math.pow` 方法，但更简单快速的办法是直接把两个数各自相乘。

```
public static double distance
    (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    System.out.println ("dsquared is " + dsquared);
    return 0.0;
}
```

再一次，编译/运行程序、检查中间值（应该是 25.0）。

最后用 `Math.sqrt` 方法进行计算并返回结果。

```
public static double distance
    (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    double result = Math.sqrt (dsquared);
    return result;
}
```

之后在 `main` 方法中就可以打印检验结果的值了。

等积累了更多的编程经验后你会发现自己经常一次写很多行代码同时还得进行调试，尽管采取递增开发也不能避免上述情况发生，但却能节省很多调试时间。

这个过程的关键在于：

- 从正常工作的程序开始，做细微的、递增的改变。任何时候如果出错都能找到错在哪里。
- 用临时变量存贮中间值以便打印出来检查。
- 程序正常工作后应该考虑移除“脚手架”或者把多个语句组合成一个复合表达式，但前提是不能让程序变得难读。

## 5.3 复合

如你所料，定义了新方法后就可以把它作为一个表达式的一部分，不仅如此，还可以用已有方法来构建新方法。例如：已知两点，圆心和圆周上一点，如何求圆面积？

设圆心的坐标存贮在变量 `xc` 和 `yc` 中，圆周上点的坐标在 `xp` 和 `yp` 中。第一步要求圆的半径，也即两点间的距离。太走运了——正好有一个 `distance` 方法专门做这件事情。

```
double radius = distance (xc, yc, xp, yp);
```

第二步计算圆面积并返回。

```
double area = area (radius);
```

```
return area;
```

全部打包进一个方法，得到：

```
public static double fred
    (double xc, double yc, double xp, double yp) {
    double radius = distance (xc, yc, xp, yp);
    double area = area (radius);
    return area;
}
```

方法名居然是 `fred`，怪怪的——下一节再说吧。

临时变量 `radius` 和 `area` 对于开发和调试很有用处，不过一等程序正常工作就该念念**紧箍咒**让它变简洁点了：

```
public static double fred
    (double xc, double yc, double xp, double yp) {
    return area (distance (xc, yc, xp, yp));
}
```

## 5.4 重载

上一节中的 `fred` 和 `area` 完成相似的功能——求圆面积——但带的参数却不一样。`area` 要半径，`fred` 则要两点。

如果两个方法做的是一件事，取一样的名字比较自然，如果把前面的 `fred` 命名为 `area` 也更能为大家理解。

不止一个方法具有同样的名字叫做**重载** (*overloading*)，在 Java 中只要各个版本的参数不一样就是合法的。所以前面的 `fred` 可以正名了：

```
public static double area
    (double x1, double y1, double x2, double y2) {
    return area (distance (xc, yc, xp, yp));
}
```

当调用一个重载方法时，Java 根据提供的参数确定调用的是哪个版本。如果写：

```
double x = area (3.0);
```

Java 就会寻找以一个双精度数做参数名为 `area` 的方法，所以这里用的是第一个版本，把参数当做半径来解释。如果写：

```
double x = area (1.0, 2.0, 4.0, 6.0);
```

Java 就会用 `area` 的第二个版本。最绝的是，第二个版本的 `area` 调用了第一个版本的 `area`。

许多 Java 内建的命令都被重载了，这就意味着有众多版本的方法可以接受不同数目和型别的参数。例如不同版本的 `print` 和 `println` 能接受一个各种型别的参数。在 `Math` 类中，`abs` 方法有一个接受双精度数的版本和一个接受整数的版本。

尽管重载特性很有用，但应该慎用，否则——你调试的版本并非被调用的版本——可能把你自己搞到抓狂。

事实上给了我们一个进行调试的训诫：务必搞清楚眼睛盯着的程序和运行的程序是一个版本！有时发现修改不迭然而程序运行效果一成不变——这正是搞错了调试目标程序的警告信号。验证的办法就是贴上一条打印语句（随便打印什么都行）并确认程序行为是有否相应的变化。

## 5.5 布尔表达式

前面见过的大多数运算生成的结果和操作数的型别是一样的。例如+把两个整数相加得到一个整数，或者两个双精度数相加得到一个双精度数，不一而足。

唯一的例外是关系运算符，比较整数和浮点数然后返回 `true` 或者 `false`。`true` 和 `false` 是 Java 的特殊值，合起来就是一个叫做**布尔值**（*boolean*）的型别。还记得吗，定义一个型别的时候我说它是值的集合。这么说的话整数集合、双精度数和字符串集合都是大家伙，相比之下布尔值集合一点也不大。

使用布尔表达式/变量和使用其它型别的表达式方法其实差不多：

```
boolean fred;
fred = true;
boolean testResult = false;
```

第一句是简单的变量声明语句，第二句是赋值语句，第三句是初始化语句。值 *true* 和 *false* 是 Java 关键字，所以不同的开发环境会做相应的彩色显示。

前面说过，条件运算符的结果是一个布尔值，因此可以把比较结果存贮在一个变量中：

```
boolean evenFlag = (n%2 == 0); // true if n is even
boolean positiveFlag = (x > 0); // true if x is positive
```

把这些变量留给后面的条件语句使用：

```
if (evenFlag) {
    System.out.println ("n was even when I checked it");
}
```

做这种用途的变量常常被叫做**标帜** (*flag*) ——因为它像旗帜一样标识出某些条件的满足与否。

## 5.6 逻辑运算符

Java 有三个逻辑运算符：与、或、非，分别用符号 `&&`、`||` 和 `!` 代表。这三个运算符的语义（意思）和字面意思是一样的。例如 `x > 0 && x < 10` 当且仅当 `x` 大于 0 且小于 10 时为真。

对 `evenFlag || n%3 == 0`，当两个条件中的任意一个为真时为真。也即，当 `evenFlag` 为真或 `n` 是奇数时表达式 `evenFlag || n%3 == 0` 的值为真。

最后一个，非运算符作用是否定或者对一个布尔表达式取反。因此如果 `evenFlag` 值为 `true` 则 `!evenFlag` 值为 `false`。

逻辑运算符常常可以提供一个简化嵌套循环的办法，例如，如何把下面的代码改写成一个单分支条件句？

```
if (x > 0) {
    if (x < 10) {
        System.out.println ("x is a positive single digit.");
    }
}
```

## 5.7 布尔方法

方法可以返回布尔值，这对隐藏复杂的条件检测很方便。例如：

```
public static boolean isSingleDigit (int x) {  
    if (x >= 0 && x < 10) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

注意方法名：`isSingleDigit`。布尔方法的命名通常听起来就像一个肯定/否定的命题；返回值是布尔值，换句话说每个返回语句都要提供一个布尔表达式。

这段代码很“直率”但未免有点罗嗦。记住表达式 `x >= 0 && x < 10` 的类型是 `boolean`，所以直接返回它没有任何错误，而且就完全避免了 `if` 语句：

```
public static boolean isSingleDigit (int x) {  
    return (x >= 0 && x < 10);  
}
```

在 `main` 方法中可以照常调用这个方法：

```
boolean bigFlag = !isSingleDigit (17);  
System.out.println (isSingleDigit (2));
```

第一行为 `bigFlag` 赋值 `true`，由于 17 并非一位数；第二行打印 `true` 因为 2 是一位数。没错，`println` 也被重载为可以处理布尔值了。

布尔方法最通常用的地方还是在条件句中：

```
if (isSingleDigit (x)) {  
    System.out.println ("x is little");  
} else {  
    System.out.println ("x is big");  
}
```

## 5.8 再谈递归

现在有了返回值的方法，该把兴趣转移到如何拥有完整的——能表达任何计算的编程语言上了。现有的众多程序都可以利用本书前面讲过的语言特性重写，事实上，只要很少一点命令就能操纵诸如键盘、鼠标、磁盘等等设备，足



够了。

这一非凡构想的缔造者是阿兰·图灵（Alan Turing）——最早的计算机科学家之一（也许有人要争论说他是一位数学家，但要知道早期的很多计算机科学家开始都是数学家）。这个猜想就是众所周知的**图灵理论**（*Turing thesis*）。如果修过**计算理论课程**（*the Theory of Computation*），就有机会看到它的证明。

为了获得对迄今为止学过的工具到底能做什么的概念，先来看几个应用了递归定义的数学函数。从定义表述包含了被定义对象这个角度来说，递归定义和循环定义很相似。真正的循环定义基本上用处不大：

**佶屈聱牙：（成语）形容佶屈聱牙的样子。**

假如在词典里看到这样的解释定义，我打赌你会气得大骂编辑。回过头来，如果去查数学函数中阶乘的定义，大约会找到这么一个式子：

$$0! = 1$$

$$n! = n \cdot (n-1)!$$

（别把表示阶乘的数学符号!和 Java 的逻辑非运算符!搞混淆了。）这个定义说 0 的阶乘是 1，其它任何数  $n$  的阶乘是  $n$  和  $n-1$  的乘积。于是 3 的阶乘是 3 乘以 2!，而 2 的阶乘是 2 乘以 1!，接下来 1 的阶乘是 1 乘以 0!。

复合起来，得到 3! 等于 3 乘以 2 乘以 1 乘以 1，结果是 6。

如果一个概念是递归的，就能写一个 Java 程序来实现它的定义。第一步要确定该函数的参数和返回值是什么——经过思考判断出阶乘取一个整数为参数并且返回一个整数：

```
public static int factorial (int n) {  
    }  
}
```

如果参数碰巧是零就直接返回 1：

```
public static int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    }  
}
```

否则——有趣的部分来了——就要递归调用 factorial 方法以求  $n-1$  的阶乘然后与  $n$  相乘。

```
public static int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```

```
int recurse = factorial (n-1);  
int result = n * recurse;  
return result;  
}  
}
```

这个程序的执行流程和上一章的 `nLines` 有点类似。如果以值 3 调用 `factorial` 方法：

3 不为零所以执行第二个分支语句，计算  $n - 1$  的阶乘...

2 不为零所以执行第二个分支语句，计算  $n - 1$  的阶乘...

1 不为零所以执行第二个分支语句，计算  $n - 1$  的阶乘...

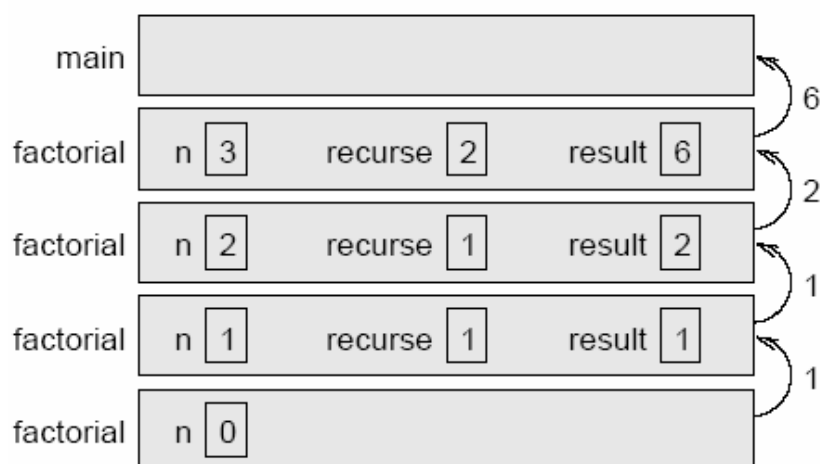
0 就是零，执行第一个分支语句，停止递归立即返回值 1。

返回的值 1 和值为 1 的  $n$  相乘，返回结果。

返回的值 1 和值为 2 的  $n$  相乘，返回结果。

返回的值 2 和值为 3 的  $n$  相乘，积为 3，这个结果被返回给 `main` 方法或是 `factorial (3)` 的调用者。

下面是上述函数调用序列的堆栈图：



图中示意了返回值被往堆栈上方回传。

注意 `factorial` 的最后一个实体没有本地变量 `recurse` 和结果的值，因为当  $n=0$  的时候二者所在分支没有被执行。

## 5.9 “掩耳盗铃”

顺着程序的执行流程看下去，于是在像上一章一样，很快你就大汗淋漓。换一个办法，“掩耳盗铃”——遇到方法调用的时候，别跟着执行流程走，只要假定方法能正确工作并返回相应的值就行了。

事实上在使用内建方法的时候就在实践“掩耳盗铃”：调用 `Math.cos` 或者 `drawOval` 方法的，无须检验方法的实现，只要假定它们能工作的很好，因为编写这些内建类的都是真正专业的程序员好手。

当然，调用自定义方法的时候照请不误。例如在 5.7 节中写来检测一个数是不是在 0 到 9 之间的 `isSingleDigit` 方法。一旦确信自己的方法正确无误（通过测试和检验代码），调用的时候就不用再去看方法的代码。

对递归程序仍然如此。遇到了递归调用不要跟着执行流程走，只要假定递归调用能干活（得到正确结果），然后问问自己，“假如能算出  $n-1$  的阶乘，能不能算出  $n$  的阶乘呢”，显然能，只要把  $n-1$  的阶乘和  $n$  相乘就是了。

当然，如果方法都没写好就假定它能正确工作，岂不是很怪异。不错，把这种干法叫做“掩耳盗铃”就是这个原因了。

## 5.10 最后一例

在上一个例子中用了临时变量来使步骤更加清晰同时也便于代码的调试，但实际可以省去几行：

```
public static int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial (n-1);  
    }  
}
```

从现在起我将尽量使用简洁的版本讲解，不过推荐你在开发代码时用更详细的版本。如果灵感丰富，等正常程序正常运转后再精简不迟。

除了阶乘，第二个数学函数中递归定义的经典例子就是斐波纳契数列 (*Fibonacci*)，定义如下：

$$\text{fibonacci}(0) = 1$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

翻译成 Java 程序就是：

```
public static int fibonacci (int n) {  
    if (n == 0 || n == 1) {return 1;} else {  
        return fibonacci (n-1) + fibonacci (n-2);  
    }  
}
```

即使  $n$  很小，但如果试图跟着执行流程走，脑袋也会爆炸。不过根据“掩耳盗铃”的精神，先假定两个递归调用（没错，两个也行）都能正确工作，然后显而易见把两个加起来就得到了正确结果。

## 5.11 术语表

**返回型别-*return type***：方法声明中指出方法返回值型别的部分。

**返回值-*return value***：作为方法调用结果的值。

**无用代码-*dead code***：常常因为处在返回语句之后而永远不能被执行的程序段。

**脚手架-*scaffolding***：程序开发中使用但不是最终版本一部分的代码。

***void***：用来标识一种特殊的 `void` 方法，该方法不返回值。

**重载-*overloading***：方法同名不同参。调用重载方法时 Java 根据提供的参数自动匹配方法的版本。

**布尔变量-*boolean***：只有 `true` 和 `false` 两种值的变量。

**标帜-*flag***：用来记录条件或者状态信息的变量（通常是布尔变量）。

**条件运算符-*conditional operator***：比较两个操作数得到一个布尔值来指明二者关系的运算符。

**逻辑运算符-*logical operator***：把布尔变量组合起来并得出新布尔值的运算符。

**初始化-*initialization***：声明变量的同时进行赋值的语句。

## 第 6 章 迭代

### 6.1 多次赋值

前面没有讲到，但在 Java 中确实可以给一个变量赋一个以上的值。第二次赋值的作用是用新值替换变量的旧值。

```
int fred = 5;
System.out.print (fred);
fred = 7;
System.out.println (fred);
```

这段程序输出 57，因为第一次打印 fred 的值是 5，第二次是 7。

这种多次赋值就是我把变量说成是一个值的容器的原因。为一个变量赋值就改变了容器装的内容，如图所示：

<code>int fred = 5;</code>	fred	5
<code>fred = 7;</code>	fred	<del>5</del> 7

当为一个变量多次赋值时，区分赋值语句和等式尤其重要。Java 用符号 = 表示赋值，很容易把类似  $a = b$  的语句理解成等式——可惜它不是。

首先，等式的左右可以交换，赋值语句则不能。例如在数学里  $a = 7$  那么  $7 = a$ ，而 Java 中  $a = 7$ ; 是合法的  $7 = a$ ; 则不是。

还有，数学里的等式总是成立的，如果  $a = b$  成立，它就总是成立；在 Java 中，赋值语句可以让两个变量相等，但不代表一直都相等。

```
int a = 5;
int b = a; // a and b are now equal
a = 3; // a and b are no longer equal
```

第三行改变了 a 的值而没有改变 b 的值所以二者不再相等。为了避免混淆，许多程序语言采用别的符号来进行赋值，比如 <- 或者 :=。

多次赋值虽然常用但也要慎用。如果在程序中的不同部分不断修改变量值，代码会非常难读更不易调试。

### 6.2 迭代

计算机的拿手好戏之一就是自动执行重复的任务。反复做同样的或相似的工作而不出错——计算机可以，人不行。

前面已经写过利用递归来执行反复任务的程序，例如 `nLines` 和 `countdown`。这种重复叫做迭代。Java 提供了一些语言特性来使得编写交互式程序更加轻松，下面就来看看其中两个：`while` 语句和 `for` 语句。

## 6.3 while 语句

用 `while` 语句可以重写 `countdown` 方法：

```
public static void countdown (int n) {  
    while (n > 0) {  
        System.out.println (n);  
        n = n-1;  
    }  
    System.out.println ("Blastoff!");  
}
```

`while` 语句读起来几乎和英文一样。这段代码的意思就是“当 `n` 大于 0 时，继续打印 `n` 的值然后把 `n` 减 1，当 `n` 减为 0 时，打印 `Blastoff!`”。

比较正规的来说，`while` 语句的执行流程如下：

1. 检验括号的条件，或真或假。
2. 条件不为真则退出 `while` 语句继续执行下一条语句。
3. 条件为真则将花括弧内的语句顺序执行一遍，然后返回到第 1 步。

这种流程叫做**循环** (*loop*)，因为第三步又重新回到了顶部。注意，如果第一次到达该循环时条件不为真，则其内部的语句将不被执行。通常循环内部的所有语句一起被叫做**循环体**。

循环体应该改变一个或多个变量的值，以使最后条件变为 `false` 而后循环终止。否则循环将成为永远反复的**无穷循环** (*infinite loop*)。这则洗发香波的使用说明总是给计算机科学家带来无穷的乐子：先用本香波轻揉秀发，然后用清水冲洗，如此反复——正是一个无限循环！

忘记那些泡沫吧，回头再看看 `countdown` 方法。可以证明该循环是有限的，因为 `n` 的值是有限的，于是 `n` 的值随着循环（每次迭代）减小，最后变为零。然而有时就不大容易辨别了：

```
public static void sequence (int n) {  
    while (n != 1) {  
        System.out.println (n);  
        if (n%2 == 0) { // n为偶数  
            n = n / 2;  
        } else { // n为奇数  
            n = n*3 + 1;  
        }  
    }  
}
```

循环的条件是  $n \neq 1$ ，所以循环将直到  $n$  为 1 使得条件为 `false` 才停止。

每步迭代时程序先打印  $n$  的值然后检验其奇偶性。如果是偶数就把  $n$  值除以 2，如果是奇数就用  $3n + 1$  来替代  $n$  的值。例如，初始值（传给 `sequence` 的值）为 3，结果将是数列 3, 10, 5, 16, 8, 4, 2, 1。

既然  $n$  的值时增时减，就不能很明显的证明  $n$  总是能达到 1，以使程序退出。对一些特殊值可以确保程序能终止。例如以 2 的正整数幂开始，就总是能得到最后得  $n$  为 1。上一个例子正是以 16 ( $2^4$ ) 开始的数列结尾的。撇开特例，到目前为止还不能证明或证伪这个程序是否对任何自然数  $n$  均能到达 1。

## 6.4 表

从循环中受益最大的事情是生成和打印表格数据。例如在计算机普及之前，人们只有手工计算对数、正弦和余弦等等数学函数。

为了简便，有专门的书籍收录了能查到各种函数值的长度吓人的表。制作这样的表是一项既缓慢又沉闷的工作，而且结果总是错漏百出。

计算机刚开始崭露头角的时候，一个误区就是，人们尖叫“太棒了！可以用计算机生成这些该死的表，而且不会出错”！这话是没错，但却短视的可怜——很快计算机（还有计算器）的普及就让那些“该死的表”失业了。

表的用途并不是就此消失。实际情况是在有的运算中，计算机利用表来查找近似值，然后再通过计算来提高精度。不过难免有时用到的潜在的表是错的，最有名的错误出在 Intel 的 Pentium 芯片用来做浮点数除法的表中。

虽然那种长度吓人的表已经不如过去吃香了,但却是演示迭代的最好例子。下面的程序在左边打印一系列值,右边是相应的对数值:

```
double x = 1.0;
while (x < 10.0) {
    System.out.println (x + " " + Math.log(x));
    x = x + 1.0;
}
```

程序输出:

```
1.0  0.0
2.0  0.6931471805599453
3.0  1.0986122886681098
4.0  1.3862943611198906
5.0  1.6094379124341003
6.0  1.791759469228055
7.0  1.9459101490553132
8.0  2.0794415416798357
9.0  2.1972245773362196
```

仅仅看这些值,能判断出来  $\log$  函数的默认底是多少吗?

2 的幂对计算机科学至关重要,常常要求以 2 为底的对数,这时要用如下公式:

$$\log_2 x = \log_e x / \log_e 2$$

改一改打印语句

```
System.out.println (x + " " + Math.log(x) / Math.log(2.0));
```

得到:

```
1.0  0.0
2.0  1.0
3.0  1.5849625007211563
4.0  2.0
5.0  2.321928094887362
6.0  2.584962500721156
7.0  2.807354922057604
8.0  3.0
9.0  3.1699250014423126
```

因为 1、2、4、8 以 2 做底的对数都是整数,所以它们都是 2 的整数幂。要



求其它 2 的整数幂的对数，可以把程序改成下面的样子：

```
double x = 1.0;
while (x < 100.0) {
    System.out.println (x + " " + Math.log(x) / Math.log(2.0));
    x = x * 2.0;
}
```

这次不是在每个循环中把 x 加一个数得到一个等差数列，而是乘以一个数得到一个等比数列。结果是：

```
1.0  0.0
2.0  1.0
4.0  2.0
8.0  3.0
16.0 4.0
32.0 5.0
64.0 6.0
```

对数表大概已经没有什么用处了，但对计算机科学家来说 2 的幂表却很有用。在你空闲的时候不妨记一下小于 65536（2 的 16 次幂）的 2 次幂表。

## 6.5 二维表

二维表分为行和列，行列的交叉点是值。乘法表是最好的例子。下面以 1 到 6 的乘法表为例说明。

从 2 下手是个不错的主意，把所有值放在一行中。

```
int i = 1;
while (i <= 6) {
    System.out.print (2*i + " ");
    i = i + 1;
}
System.out.println ("");
```

第一行初始化变量 i，作为计数器或者叫**循环变量**（*loop variable*）使用。循环过程中 i 的值由 1 步增为 6，到达 7 循环终止。每次循环打印 2-i 的值并紧跟三个空格，并且因为用的是 print 语句所以输出都在一行上。

提请注意：在 2.4 讲过，一个或多个 print 的最后一定要有一个 println。程序输出：

```
2 4 6 8 10 12
```

到现在一切都很顺利。下一步就要开始封装和通用化了。

## 6.6 封装和通用化

**封装** (*encapsulation*) 通常意味着写一段代码然后打包进一个方法，这样就能充分发挥方法的所有优点。4.3 节的 `printParity` 和 5.7 的 `isSingleDigit` 方法就是这样的例子。

**通用化** (*generalization*) 意味着取一个特——比如打印 2 的积——将它推广为更一般的情况，比如打印一个整数 `n` 的积。

下面的方法封装了上一节的循环并通用化为打印 `n` 的积。

```
public static void printMultiples (int n) {  
    int i = 1;  
    while (i <= 6) {  
        System.out.print (n*i + " ");  
        i = i + 1;  
    }  
    System.out.println ("");  
}
```

这次封装仅仅添加了第一行，声明参数和返回型别；通用化则只是把值 2 换为参数 `n`。

如果以参数 2 调用该方法会得到和前面一样的结果。参数为 3 则输出：

3 6 9 12 15 18

参数 4 输出：

4 8 12 16 20 24

现在猜到怎么打印乘法表了吧：用不同的参数反复调用 `printMultiples` 方法。更进一步说，在另一个循环中一行一行地反复调用这个方法。

```
int i = 1;  
while (i <= 6) {  
    printMultiples (i);  
    i = i + 1;  
}
```

注意这个循环和 `printMultiples` 内的循环何其相似。所做的仅仅是把 `print` 语句换成了一个方法调用语句。

程序输出：

```
1 2 3 4 5 6
2 4 6 8 10 12
3 6 9 12 15 18
4 8 12 16 20 24
5 10 15 20 25 30
6 12 18 24 30 36
```

一个不怎么规矩的乘法表。如果你讨厌这么凌乱的排列，Java 提供了方法能更好的控制输出格式，不过这里就懒得管了。

## 6.7 方法的好处

前面提到过“方法的所有优点”，你可能会想“到底是什么优点”。下面就是一些方法之所以有用的理由：

- 为一序列语句取一个名字使程序便于阅读/调试。
- 把一个冗长的程序化分为一些方法就可以“分而治之、合而统之”：分别独立进行调试，然后再复合组成一个整体。
- 方法的使用让递归和迭代变得更简单。
- 设计良好的方法可以为众多程序使用。写好一个方法调试妥当就可以重用。

## 6.8 再谈封装

为了再次说明封装，把上一节的代码打包进一个方法：

```
public static void printMultTable () {
    int i = 1;
    while (i <= 6) {
        printMultiples (i);
        i = i + 1;
    }
}
```

这个过程演示了一个很平常的开发方案：逐步的在 `main` 方法或其它什么地方增加一些代码，当能够工作的以后，就把这些代码提取出来打包进一个方法——这就是写程序的过程。

这种办法之所以有效是因为，有时即使已经开始写代码了还是难以把程序化分成几个方法，这种法子可以让程序员边写边构思设计。

## 6.9 本地变量

你可能在想是如何做到让 `printMultiples` 和 `printMultTable` 中都有 `i` 的。不是说过一个变量只能声明一次吗？如果其中一个方法修改了变量的值不会出问题吗？

两个问题的答案都是：“非也。”因为 `printMultiples` 和 `printMultTable` 中的 `i` 不是同一个变量。二者虽然名称一样，但不是指向同一块内存区域的，改变一个对另一个不会产生影响。

在一个方法内部声明的变量叫做本地变量，因为它仅仅对它所在的方法内部有效。在本地变量的“老巢”之外无法访问该变量，所以不同方法的变量可以尽情使用相同的名字。

一般来说为了避免混淆不同方法的变量也最好不一样，但总有很好的理由需要重用名字。例如 `i`、`j` 和 `k` 常用做循环变量，如果刻意为了避免重复而不用它们，将适得其反把程序弄的很难读懂。

## 6.10 再谈通用化

再举一个通用化的例子。假定想打印一个任意大小的乘法表，而不仅仅是一个  $6 \times 6$  的表，可以在 `printMultTable` 方法中加上参数：

```
public static void printMultTable (int high) {  
    int i = 1;  
    while (i <= high) {  
        printMultiples (i);  
        i = i + 1;  
    }  
}
```

值 `6` 被换成了参数 `high`。如果用参数 `7` 调用 `printMultTable` 就得到：

```
1 2 3 4 5 6  
2 4 6 8 10 12  
3 6 9 12 15 18  
4 8 12 16 20 24  
5 10 15 20 25 30
```

6 12 18 24 30 36

7 14 21 28 35 42

一切正常。只是如果想打印一个方阵，就意味着另外还得为 `printMultiples` 方法加上参数以指明表的列数。

为了看看有多烦，故意把参数也叫做 `high`，顺便演示不同方法的参数可以有一个名字（和本地变量相似）：

```
public static void printMultiples (int n, int high) {
    int i = 1;
    while (i <= high) {
        System.out.print (n*i + " ");
        i = i + 1;
    }
    newLine ();
}

public static void printMultTable (int high) {
    int i = 1;
    while (i <= high) {
        printMultiples (i, high);
        i = i + 1;
    }
}
```

注意加上参数后必须修改方法的第一行（称为接口或者原型），同时也要修改调用 `printMultTable` 的地方。然后——如愿以偿，程序生成了一个 7x7 的表：

1 2 3 4 5 6 7

2 4 6 8 10 12 14

3 6 9 12 15 18 21

4 8 12 16 20 24 28

5 10 15 20 25 30 35

6 12 18 24 30 36 42

7 14 21 28 35 42 49

如果一个方法写的恰当，常常会发现得到的程序仍然有一些意料之外的潜力可以改进。例如，上面的乘法表是对称的，因为  $ab = ba$ ，所以表中的每一项都出现了两次。这样一来只打印不重复的一半就可以节约不少打印机的油墨啦。要做到这点只需要改动 `printMultTable` 方法中的一行就行了。把

```
printMultiples (i, high);
```

改成

```
printMultiples (i, i);
```

得到:

1

2 4

3 6 9

4 8 12 16

5 10 15 20 25

6 12 18 24 30 36

7 14 21 28 35 42 49

原因这里不说了，留做思考。

## 6.11 术语表

**循环-loop:** 当满足一定条件才开始或终止的反复执行的语句。

**死循环-infinite loop:** 条件永远为真的循环。

**循环体-body:** 循环内的语句。

**迭代-iteration:** 包括检测条件在内的循环体的每一次执行。

**封装-encapsulate:** 把一个大的程序化分为一些部件（如方法），并（通过使用本地变量等方法）使各个部件相对保持独立，以降低整体的复杂度。

**本地变量-local variable:** 在方法内部声明并且只能存在于方法内部的变量。本地变量不能在声明它的方法外部访问，并且也不和其它方法发生冲突。

**通用化-generalize:** 用相对一般的元素（比如变量和参数）代替非必要的特例（如固定值）。通用化使代码的功能更具普适性、更可能被重用，甚至有时更易编写。

**开发方案-development plan:** 开发程序的规划。本章演示了这样一种开发风格：首先编写简单的、完成特定任务的代码，然后进行封装和通用化。在 5.21 节还演示过递增开发，后面的章节会给出更多的开发风格。

## 第 7 章 字符串及其它

### 7.1 向对象调用方法

对 Java 和其它面向对象的语言来说，对象是数据和相关方法的容器。这些方法操作对象、执行计算，有时会修改对象的数据。

迄今为止讨论过的型别中只有字符串是对象。那么根据对象的定义，一个 *String* 型别的对象里面装的是什么呢？应该调用什么方法以对 *String* 对象进行操作呢？*String* 对象包含的是组成字符串的字元。能对字符串进行操作的方法相当多，本书只涉及很少一些，关于其余部分的文档可以在这里找到：

<http://java.sun.com/j2se/1.4.1/docs/api/java/lang/String.html>

我们要考察的第一个方法是 `charAt`，它可以从一个字符串中提取字元。为了保存得到的结果，要一个能贮存单个字元（而不是字符串）的变量型别。单个字元叫做**字符**（*character*），贮存字符值的变量型别是 `char` 型别。

使用 `char` 型别的方式和前面讲过的型别一样：

```
char fred = 'c';
if (fred == 'c') {
    System.out.println (fred);
}
```

和字符串放在双引号中不一样，字符被放在单引号中。字符的值只能是单个字母或者符号。

下面是 `charAt` 方法的用法：

```
String fruit = "banana";
char letter = fruit.charAt(1);
System.out.println (letter);
```

形如 `fruit.charAt` 的语法指明对名叫 `fruit` 的对象调用了 `charAt` 方法。`1` 是传递给方法的参数，因为我想取得字符串的第一个字母。结果是一个字符，贮存在一个字符变量 `letter` 中。打印结果却给了我一个“惊喜”：

a

——a 是“banana”的第一个字母？除非你是一个计算机科学家——不知道是什么理由让他们总是喜欢从零开始数数：“banana”的“第 0 个”字母是 b，第 1 个字母是 a，第 2 个是 n。

如果想得到一个字符串的“第 0 个”字母，只好把零作为参数传递：

```
char letter = fruit.charAt(0);
```

## 7.2 length 方法

接下来是第二个操作字符串的方法 `length`，它返回字符串中字符的个数。例如：

```
int length = fruit.length();
```

空括号 `()` 表明 `length` 没有参数，返回的值是一个整数。注意：变量和方法同名是合法的（不过会给人的阅读带来麻烦）。

试试像这样找字符串的最后一个字母：

```
int length = fruit.length();
char last = fruit.charAt (length); // 错啦!!
```

这样不行。因为从 0 开始计数，6 个字母就是第 0 到第 5，所以 "banana" 没有“第 6 个”字母，要取得最后一个字符就得把长度减一作为参数：

```
int length = fruit.length();
char last = fruit.charAt (length-1);
```

## 7.3 遍历

对字符串最常见的操作就是，从头开始，对每一个字符进行一定处理，直到最后一个。这种方式叫做**遍历** (*traversal*)。最常用来编码实现遍历的是 `while` 语句：

```
int index = 0;
while (index < fruit.length()) {
    char letter = fruit.charAt (index);
    System.out.println (letter);
    index = index + 1;
}
```

这个循环语句块遍历打印字符串的每一个字母并单独输出在一行上。注意条件是 `index < fruit.length()`，表示当 `index` 等于字符串长度时条件为假，然后停止执行循环体。最后访问的字符是第 `fruit.length()-1` 个。

循环变量的名称是 `index`（索引）。`index` 是在一个有序集中（本例中即字符串中的字符的集合）用来指出某个成员位置的变量或值。有序集表明每个字母都有一个唯一的索引号。

作为练习，试试写一个方法：取一个字符串为参数，把它的字母反向输出在一行上。



## 7.4 运行时错误

回到 1.3.2 节,还记得讲过运行时错误吗——在程序开始运行后才出现的错误,Java 中的运行时错误叫做异常。

我们到目前为止还没有考察过什么运行时错误,因为前面的任务还没有麻烦到引起运行时错误的程度。现在遇到了。使用 `charAt` 命令的时候如果提供的 `index` 值是负数或者大于 `length-1`,就会得到一个异常:特别地这里是 `StringIndexOutOfBoundsException`。用手头的编译器试试看到底是什么样子。

如果程序导致了异常,就会打印一条出错并指明异常的类型和位于程序的哪部分,然后程序就终止了。

## 7.5 阅读文档

访问下面的链接

<http://java.sun.com/j2se/1.4/docs/api/java/lang/String.html>

查看 `charAt` 的说明,就会得到下面(或者类似)的文档:

```
public char charAt(int index)

Returns the character at the specified index.

An index ranges from 0 to length() - 1.

Parameters: index - the index of the character.

Returns: the character at the specified index of this string.

The first character is at index 0.

Throws: StringIndexOutOfBoundsException if the index is out of
range.
```

第一行是方法的原型,说明了方法的名称、参数型别和返回值的型别。

下一行说明了方法的用处。再下来解释参数和返回值。这里的解释有点画蛇添足,这不过是为了复合文档的标准格式罢了。最后一行解释该方法可能导致的异常。

## 7.6 indexOf 方法

某种程度上,`indexOf` 和 `charAt` 刚好相反。`charAt` 根据 `index` 返回在该索引号上的字符,`indexOf` 的作用则是找一个字符的索引号。

如果索引号超出范围则 `charAt` 方法将失败并导致一个异常。如果要想其索引号的字符没有在字符串中出现则 `indexOf` 失败,返回一个值-1。

```
String fruit = "banana";  
int index = fruit.indexOf('a');
```

这段程序找字母'a'在字符串中的索引号。这里的情况是字母出现了三次，那么 `indexOf` 的任务到底是找哪个呢。根据文档的说明，它会返回第一次出现的索引号。

寻找后续出现的索引号，要用 `indexOf` 方法的一个候选版本（参考 5.4 节关于重载的内容）。该版本的第二个参数指明要从第几个字符开始找。像这样调用

```
int index = fruit.indexOf('a', 2);
```

从第 2 个字符（即 n）开始，找到了索引号为 3 的第 2 个 a。如果要找的字符刚好就是开始的字符就直接返回该索引号，所以

```
int index = fruit.indexOf('a', 5);
```

返回 5。开始查找的索引号有可能超过 `index` 的范围，文档的说明中给出的判断方法有点诡异：

**indexOf** returns the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to `fromIndex`, or -1 if the character does not occur.

要搞清楚这段话的意思，最好的办法就是在不同情形下试一试。下面是试验结果：

- 当起始索引号大于等于 `length()`，结果是 -1，说明从起始索引号之后没有要查找的字母。
- 如果起始索引号是负数，结果是 1，说明要找的字母索引号大于 起始索引号。

再回过头看看文档的说明，发现 `indexOf` 的行为和定义完全吻合，虽然一开始没有看明白，但现在已经搞清楚它的工作方式，可以用在我们写的程序里了。

## 7.7 循环和计数

下面的程序记录'a'在字符串中出现的次数：

```
String fruit = "banana";  
int length = fruit.length();  
int count = 0;
```

```
int index = 0;
while (index < length) {
    if (fruit.charAt(index) == 'a') {
        count = count + 1;
    }
    index = index + 1;
}
System.out.println (count);
```

程序演示了一个常见的用语：**计数器** (*counter*)。变量 `count` 被初始化为 0，每找到一个 'a' 就自增 1。退出循环的时候 `count` 得到的结果就是 a 的总数。

练习一：把这段代码封装进方法 `countLetters` 并通用化以接受一个字符串和要找的字母作为参数。

练习二：利用 `indexOf` 重写该方法以直接由 a 的索引号确定的总数，而不是一个一个字符的检查。

## 7.8 自增 1 和自减 1 运算符

由于广为使用，Java 专门为自增 1 和自减 1 分配了运算符。`++` 把当前的整数或字符 `+1`，`--` 则是 `-1`。两个运算符都不能对双精度数、布尔值或者字符串进行操作。从语法上讲把一个自变量自增 1 的同时作为一个表达式的一部分是可以的。例如

```
System.out.println (i++);
```

看看，自增是在打印之前生效呢还是打印之后？这样的表达式很容易搞混淆，所以不推荐使用。这里也不回答前面的问题，要知道答案可以试试。

利用自增 1 运算符可以重写计数器：

```
int index = 0;
while (index < length) {
    if (fruit.charAt(index) == 'a') {
        count++;
    }
    index++;
}
```

一个常见的错误是写这样的代码：

```
index = index++; // 错误!!
```

不幸的是，这么写在语法上是合法的，所以编译器不会提出警告。执行这

条语句的结果是 `index` 的值没有改变。这样的 bug 很难跟踪。

请注意：你可以写 `index = index + 1`；也可以写 `index++`；但是不要把二者混合在一起。

## 7.9 字符算术运算

可能有点古怪，但字符确实可以进行算术运算。表达式 `'a' + 1` 得到值 `'b'`；类似的，如果有一个名为 `letter` 的字符变量，那么做 `'a'` 运算就能得到该字母在字母表中的位置了（心里应该清楚 `'a'` 是第 0 个字母，`'z'` 是第 25 个）。

这种运算对于将包含数字的字符如 `'0'`、`'1'` 和 `'2'` 转换为相应的数字十分有用。二者并非一回事，例如试试这样做：

```
char letter = '3';
int x = (int) letter;
System.out.println (x);
```

本来期望的值是 3，但视环境不同，可能得到的是 51——用来表示字符 `'3'` 的 ASCII 码，也可能是别的结果，但不大可能是 3。要把 `'3'` 转换为相应的整数值，要用到减 `'0'` 运算：

```
int x = (int) (letter - '0');
```

从技术角度讲，两个例子中的型别浇铸运算符都并非必须，因为 Java 会自动把字符转换为整数。这里主要是为了强调二者之间并非一回事，应该尽量搞清楚这个问题。

如果不喜欢这种略显丑陋的办法，还有一个选择——`Character` 类的 `digit` 方法。例如：

```
int x = Character.digit (letter, 10);
```

第一个参数字符变量 `letter` 的值将被转换为相应的数字，第二个参数 10 表明采用十进制。

字符算术运算的另一用途是循环遍历字母表。例如在 Robert McCloskey 所著的书 *Make Way for Ducklings* 中小鸭子的名字按字母顺序排列：`Jack`、`Kack`、`Lack`、`Mack`、`Nack`、`Ouack`、`Pack` 和 `Quack`。下面就按顺序循环打印这些名字：

```
char letter = 'J';
while (letter <= 'Q') {
    System.out.println (letter + "ack");
    letter++;
}
```

注意，作为对算术运算符的补充，条件运算符也可以操作字符。程序输出

是：

Jack

Kack

Lack

Mack

Nack

Oack

Pack

Qack

当然，不尽完善之处是没有打印出 **Ouack** 和 **Quack** 的大名。

练习：修改程序以纠正漏掉了两只小鸭子名字的错误。

## 7.10 刚性的字符串

查看官方文档中 *String* 的有关方法时，你也许注意到了 `toUpperCase` 和 `toLowerCase` 方法。这两者常常迷惑我们，因为听起来它们似乎有改变一个字符的效用。然而事实是，没有任何方法可以改变一个字符串，因为字符串是**刚性的**（*immutable*）。

当调用 `toUpperCase` 方法操作一个 *String* 型别值的时候，得到一个新的字符串作为返回值。例如：

```
String name = "Alan Turing";  
String upperName = name.toUpperCase ();
```

第二行代码执行后 `upperName` 拥有了值 `"ALAN TURING"`，而 `name` 的值也仍然是 `"Alan Turing"`。

## 7.11 不可比的字符串

比较字符串的异同或首写字母的先后往往非常必要。如果能使用诸如 `==` 和 `>` 这样比较运算符那实在是太好了——然而不行……

要比较字符串，只有用 `equals` 和 `compareTo` 方法，例如：

```
String name1 = "Alan Turing";  
String name2 = "Ada Lovelace";  
if (name1.equals (name2)) {  
    System.out.println ("The names are the same.");  
}
```

```
int flag = name1.compareTo (name2);
if (flag == 0) {
    System.out.println ("The names are the same.");
} else if (flag < 0) {
    System.out.println ("name1 comes before name2.");
} else if (flag > 0) {
    System.out.println ("name2 comes before name1.");
```

这种语法的怪异之处就在于：要对两个字符串进行比较就得把其中之一作为参数传递给另一个。

返回值倒是十分直观，如果两者所含字符一致就返回 `true`，否则返回 `false`。

`compareTo` 的返回值有点奇怪。它反映的是两个字符串中相异的第一对字符在字母表中索引号的差。如果值为 0，表示二者包含的字符是一样的；如果第一个字符串（作为方法调用者的那个）中相异的字符在字母表中排前面，则差为负数，否则差为正数。这个例子中返回值是正 8，因为作为参数的 "Ada" 的第二个字母比 "Alan" 的第二个字母索引号靠前 8 个位置。

使用 `compareTo` 方法常常很考脑筋，因为不查文档就难以记住到底哪个在先哪个在后。不过好消息是比较其它型别或对象的接口都很标准，所以只要搞清楚一个其它的也自然掌握了。为了彻底澄清，我要承认用 `==` 运算符直接比较字符串是合法的，虽然几乎没有用对的时候。况且这种办法没有什么实际意义，所以还是别用为妙。

## 7.12 术语表

**对象-object**：数据和相关方法的容器。迄今为止用过的对象是内置的 `String` 型别对象（字符串对象）。

**索引-index**：在一个有序集中用来指出某个成员（例如字符串中的字符）位置的变量或值。

**遍历-traverse**：从头开始，逐一对集合中的所有元素进行类似处理。

**计数器-counter**：用来进行计数的变量，通常初始化为 0 然后进行自增 1 运算。。

**自增 1 运算-increment**：把变量的值加一。Java 的自增 1 运算符是 `++`。

**自减 1 运算-decrement**：把变量的值减一。Java 中自减 1 运算符是 `--`。

**异常-exception**：一种运行时错误。异常可以导致程序执行终止。

## 第 8 章 有趣的对象

### 8.1 有趣在哪里

虽然字符串也是变量，可惜它并非有趣的对象，理由是：

- 它是刚性的
- 没有实例变量
- 不必用 `new` 命令来创建

在这一章要用到 Java 语言定义两种对象——`Point` 和 `Rectangle`。一开始要说明的是，这些点和矩形并非我们在屏幕上看到的点和矩形，而是一些包含数据的变量，正如整数型别和双精度型别，和其它型别的变量一样，是用来执行计算的。

`Point` 和 `Rectangle` 类被定义在 `java.awt` 包中，所以先要进行导入。

### 8.2 包

内建的 Java 类被分成了许多个包，包括 `java.lang`——包含了迄今为止我们用过的大多数类；`java.awt`——包含了附属于 Java 抽象窗口工具集（*Abstract Window Toolkit/AWT*）的类，包括窗口、按钮、图形类的定义。

要使用包中的类就要先进行导入。在附录 D 中的程序以 `import java.awt.*` 开头，星号\*指明要导入 AWT 包中的所有类。如果乐意也可以写出要导入的类名，不过这样做没有什么必要。`java.lang` 包是自动导入的，所以前面写过的那许多程序才不需要导入语句。

所有导入语句出现在程序的最开始，在类定义之前。

### 8.3 Point 对象

在对底层的级别上，一个点就是被当做单一对象的两个数字（坐标）。在数学符号系中点的坐标通常写在一对括号里，中间用逗号隔开。例如，`(0; 0)` 代表原点，`(x; y)` 代表横坐标为 `x` 纵坐标为 `y` 的点。

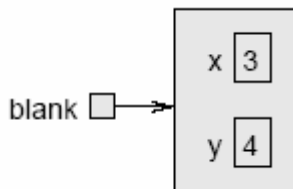
在 Java 中，点用 `Point` 对象表示。要创建一个新的点就要用 `new` 命令：

```
Point blank;  
blank = new Point (3, 4);
```

第一行照例是变量声明：`blank` 的型别是 `Point`。第二行看上去很滑稽，调

用了 `new` 命令，指明了新对象的型别，同时还提供参数。很显然，参数就是新点的坐标 (3; 4)。

实际上 `new` 命令是对新 `point` 对象的引用 (*reference*)，关于引用的具体含义以后会详细讨论。现在最重要的是变量 `blank` 包含了对新创建的对象的一个引用。有标准的图用以表示这个赋值语句，如下所示：



按照常规，变量名 `blank` 放在盒子外面值放在里面。此处的值是一个引用，图中用一个小方框带箭头表示。箭头指向被引用的对象。

大盒子表示新创建的对象，包含两个变量。`x` 和 `y` 是实例变量的名称。

整个合起来，程序中的所有变量、值和对象叫做状态。像这样表示程序状态的图叫做**状态图** (*state diagram*)。程序改变，状态也随之改变，因此可以把状态图想像成程序执行过程中排下来的一幅快照。

## 8.4 实例变量

组成对象的数据片断有时叫做**组件** (*component*)，或者**记录** (*record*)，又或者**字段** (*field*)。在 Java 中叫做**实例变量** (*instance variable*)，因为每个对象作为其类的一个**实例** (*instance*)，都拥有一份自己的实例变量的**拷贝** (*copy*)。

拿汽车驾驶台上的工具箱来说明。每辆汽车都是汽车类的一个实例，各自都有一个工具箱。如果要我从你的工具箱里拿点东西，首先得告诉我你的汽车是哪一辆。

类似地，要想读取一个实例变量的值首先就得指明是哪个对象的。在 Java 中用点号来做这件事。

```
int x = blank.x;
```

表达式 `blank.x` 的意思是“找到对象 `blank` 的引用，取的 `x` 的值”。这条语句把取得的值赋给了名为 `x` 的本地变量。注意，本地变量 `x` 和实例变量 `x` 并无瓜葛。点号的目的是明白无误的指出引用的是哪个变量。

点号可以出现在任何 Java 表达式中，所以下面的语句都是合法的。

```
System.out.println (blank.x + ", " + blank.y);
```

```
int distance = blank.x * blank.x + blank.y * blank.y;
```

第一行打印 3,4，第二行算出距离的值为 25。



## 8.5 对象做参数

可以照常把对象作为参数传递。例如

```
public static void printPoint (Point p) {  
    System.out.println("(" + p.x + ", " + p.y + ")");  
}
```

这个方法以一个点作为参数然后按照表示点的标准格式（即坐标）打印出来。如果像这样调用方法——`printPoint (blank)`，将打印(3,4)。事实上 Java 内建了打印 Point 对象的方法。如果调用方法

```
System.out.println(blank);
```

就会得到

```
java.awt.Point[x=3,y=4]
```

这是 Java 打印对象的标准格式：类名称后紧跟对象内容，包括实例变量名和值。

再看一个例子，重写 5.2 节的 `distance` 方法，不用四个双精度数而取两个点对象作为参数。

```
public static double distance (Point p1, Point p2) {  
    double dx = (double) (p2.x - p1.x);  
    double dy = (double) (p2.y - p1.y);  
    return Math.sqrt (dx*dx + dy*dy);  
}
```

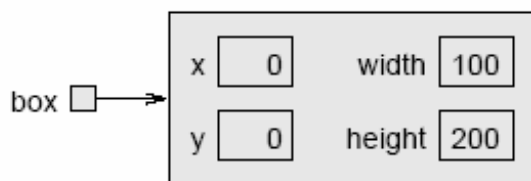
型别浇铸在此并非必不可少，只是想提醒注意 Point 对象的实例变量其实是整数。

## 8.6 矩形

矩形和点有些相仿，只是矩形有四个实例变量，名为 `x`、`y`、`width` 和 `height`。除了这个，其它部分几乎完全一样。

```
Rectangle box = new Rectangle (0, 0, 100, 200);
```

此句创建了一个新的 Rectangle 对象并使 box 引用它。下图展示了这条赋值语句的作用：



打印 box，得到

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

可见，Java 内建的方法同样知道如何打印一个 Rectangle 对象。

## 8.7 作为返回值的对象

可以写返回对象的方法。例如，findCenter 方法取一个 Rectangle 对象做参数然后返回一个包含矩形中心坐标的 Point 对象：

```
public static Point findCenter (Rectangle box) {  
    int x = box.x + box.width/2;  
    int y = box.y + box.height/2;  
    return new Point (x, y);  
}
```

注意，这么做是可以的：用 new 创建一个新对象然后立即将其作为返回值。

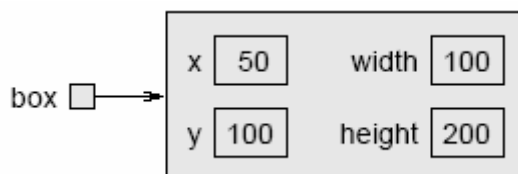
## 8.8 刚性的对象

可以通过为实例变量赋值改变对象的内容。例如为了“移动”一个矩形且不改变它的大小可以修改 x 和 y 的值：

```
box.x = box.x + 50;
```

```
box.y = box.y + 100;
```

结果如图所示：



可以把这段代码封装进一个方法并通用化为根据需要任意移动：

```
public static void moveRect (Rectangle box, int dx, int dy) {  
    box.x = box.x + dx;  
    box.y = box.y + dy;  
}
```

```
}
```

变量 `dx` 和 `dy` 指出在各个方向上移动矩形的距离。

调用该方法的作用是移动作为参数传递的 `Rectangle` 对象。

```
Rectangle box = new Rectangle (0, 0, 100, 200);  
moveRect (box, 50, 100);  
System.out.println (box);
```

运行这段代码会打印出

```
java.awt.Rectangle[x=50,y=100,width=100,height=200].
```

把对象作为参数传递以进行改动十分有用，但也可能给调试带来困难，因为不能总是清楚地知道方法调用是否修改了参数。稍后会讨论这种编程风格的优劣所在。

同时，我们应该充分利用 Java 内建方法的豪华功能，里面就包括了一个 `translate` 方法专门用来做和 `moveRect` 方法一样的事情——不过调用的语法小有区别。不必把 `Rectangle` 对象作为参数，对 `Rectangle` 对象调用 `translate` 只需要传递 `dx` 和 `dy` 两个参数。

```
box.translate (50, 100);
```

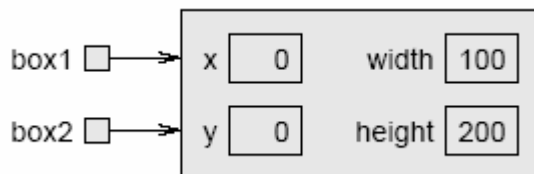
效果完全一样。

## 8.9 别名

记住当为一个对象变量赋值时实际是赋予对一个对象的引用，因此可能有多个变量指向同一个对象。例如这两行代码：

```
Rectangle box1 = new Rectangle (0, 0, 100, 200);  
Rectangle box2 = box1;
```

产生一个这样的状态图：



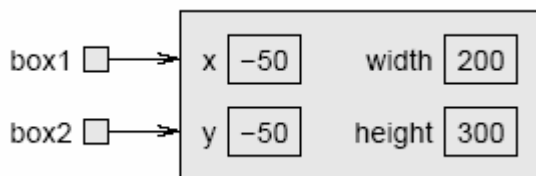
`box1` 和 `box2` 都指向同一个对象。换句话说，这个对象有两个名字，`box1` 和 `box2`。一个人有两个名字就叫做**别名**（*aliasing*），对象同样如此。

两个变量成为别名的时候，任何改变对二者同样产生影响。例如：

```
System.out.println (box2.width);
```

```
box1.grow (50, 50);  
System.out.println (box2.width);
```

第一行打印 100，即 box2 所引用 Rectangle 对象的宽。第二行对 box1 调用 grow 方法，将 Rectangle 对象在每个方向上增加了 50 像素（参考官方文档以了解细节）。效果如图所示：



正如图中清晰的所示，对 box1 的改动对 box2 同样有效。于是第三行打印的值为 200，即放大后的矩形宽度。（顺便说一句，Rectangle 对象的坐标为负是完全合法的。）

如你所料，即使对本例如此简单的例子，调用别名的代码也很容易把人弄糊涂，并且很难调试。一般说来最好避免使用别名，如果用就要小心仔细些。

## 8.10 null

创建一个对象变量等于创建了对一个对象的引用。在使变量指向具体的对象之前，变量的值是 null。null 是一个特殊的值（同时也是 Java 关键字），意思就是“没有对象”。

仅仅声明一个 Point 对象变量 blank 等于如下初始化语句：

```
Point blank = null;
```

其状态图如下：

blank □

用不带箭头的小方块代表值 null。

如果试图使用 null 对象，不管是访问实例变量还是调用方法，都会得到一个 NullPointerException 异常。系统会打印一条出错信息然后程序终止。

```
Point blank = null;  
int x = blank.x;           // NullPointerException  
blank.translate (50, 50); // NullPointerException
```

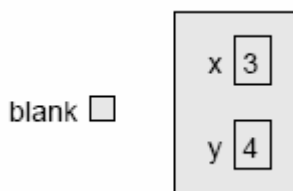
话说回来，将 null 对象作为参数或返回值都是合法的。事实上这么做很常见，例如要表达空集或者指示一种错误情况。

## 8.11 垃圾收集

8.9 节谈论多个变量引用同一个对象的情形,那么没有变量引用对象的时候会怎么样呢? 例如:

```
Point blank = new Point (3, 4);  
blank = null;
```

第一行创建了一个 `Point` 对象并让 `blank` 引用它。第二行改变了 `blank` 使之不再引用新创建的对象,也不引用别的什么(指向 `null` 对象的意思就是这个)。



如果一个对象没有被引用,则无法读写其值或对其调用方法。实际上这个对象等于不存在了。虽然可以将它继续留在内存中,却只不过是浪费空间罢了。因此在程序运行时 `Java` 系统会周期性地查找无用对象并进行回收——这个过程就叫**垃圾收集** (*garbage collection*)。然后,原本被无用对象占用的空间将又可以为新的对象使用。 ?

不必为垃圾收集做任何事,通常也不用担心会出什么乱子。

## 8.12 对象和基本类型

`Java` 有两大类型别,基本型别和对象型别。基本型别:诸如 `int` 型别和 `boolean` 型别,都以小写字母开头;对象型别则以大写字母开头。这种区别有用之处在于能提醒人注意它们的许多不同:

- 声明基本变量的时候就得到了一块存放基本型别值的存储空间。但要获得一个对象的储存空间就得使用 `new` 命令。

- 基本型别初始化的时候会得到一个相应的默认值,例如整数的默认值是 `0`、布尔值的默认值是 `true`。对象型别的默认值是 `null`,即没有对象。

- 基本型别独立性很好,因为一个方法不能做任何事情来影响另一个方法的变量。对象型别则不是完全独立的所以用起来可能比较费神。如果把引用当做参数传递给一个对象,所调用的方法可能会修改对象,效果是可以观察到的。调用一个对象的方法可能发生同样的事情。这一点用的好就可以是很好的特性,但需要格外小心。

基本型别和对象型别之间还有另一个区别。我们无法向 Java 语言中增加新的基本型别（除非你是标准委员会的成员^\_^），但是可以创建新的对象型别。下一章将看到是怎么回事。

## 8.13 术语表

**包-package:** 类的容器。内建的 Java 类用包来进行组织。

**AWT:** 抽象窗口工具集。最大和最常用的 Java 包。

**实例-instance:** 一种事物的一个实体。我的猫是猫科动物的一个实例。每个对象都是某种类的实体。

**实体变量-instance variable:** 构成对象的有名称的数据项。每个对象（实例）都有所属类的实例变量的一份拷贝。

**引用-reference:** 指明一个对象的值。在状态图中，引用看上去像一个箭头。

**别名-aliasing:** 两个以上变量引用同一个对象的情形。

**垃圾收集-garbage collection:** 找到没有引用的对象将其占用空间回收的过程。

**状态-state:** 在程序执行过程的某一点上，对所有变量和对象及其值的完整描述。

**状态图-state diagram:** 表示程序状态的快照图。

## 第 9 章 创建对象

### 9.1 类定义与对象的型别

写一个类定义就在创建一个新的对象型别，类名就是型别名称。翻回 1.5 节可以看到，我们在那里定义了一个名叫 `Hello` 的类，也就是创建了一个名为 `Hello` 的型别。不必用 `Hello` 声明变量，也不必用 `new` 命令创建就有 `Hello` 对象。

这个例子没有什么实际意义，也不能让大家心情更愉快。本章将创建一些有用的新对象型别的类定义。

下面是本章最重要的观点：

- 定义一个新类的同时也就创建了一个同名的对象型别。
- 类定义好比是对象的模板：指明了对象具有的实例变量和能对其执行操作的方法。
- 每个对象属于某种对象型别，即是某种类的实例。
- 调用 `new` 命令创建对象的时候 `Java` 会调用一个名为构造器的特殊方法以初始化实例变量。要提供一个或多个构造器作为类定义的一部分。
- 典型的做法是，对某型别的所有操作被放在类的定义中。

下面是关于类定义语法的一些注意事项：

- 类名（即对象型别）总是以大写字母开头，便于和基本型别或变量名区分开来。
- 通常一个类单独放在一个文件中，文件名和类名相同，后缀名是 `.java`。例如 `Time` 类被定义在 `Time.java` 文件中。
- 在任何程序中有一个类被指定为启动类。启动类中必须有一个 `main` 方法，作为程序开始执行的入口。其它方法允许有 `main` 方法，但不会被执行。

按照这些关键的注意事项，下面来看看一个自定义型别：`Time`。

### 9.2 `Time` 类

创建一个新 `Object` 型别的动机常常是想把相关的数据片断封装进一个对象以达到作为一个独立单元进行操纵（例如作为参数传递）的目的。前面已经讲过了两个内建型别，`Point` 和 `Rectangle` 类。

作为举例，我们将亲自实现一个类：`Time`，用来记录一天的时间。

构成时间的信息片断包括小时（*hour*）、分钟（*minute*）和秒（*second*）。每

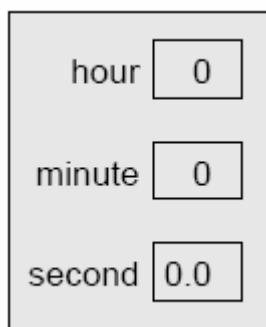
个时间对象都包含这些数据，所以要创建实例变量用来存放这三个数据。

第一步要确定每个变量的型别。一眼就能确定 `hour` 和 `minute` 应该是整数；为了给后面留点事做先把 `second` 定为双精度数，这样可以记录秒的小数部分。

实例变量在类定义的开始部分声明，不能写在任何方法定义内，象这样：

```
class Time {  
    int hour, minute;  
    double second;  
}
```

单独看这一小段代码是合法的类定义，根据定义一个 `Time` 对象的状态图将是这个样子：



声明了实例变量下一步通常要定义类的构造器。

## 9.3 构造器

**构造器**（*constructor*）的任务通常是初始化实例变量。它的语法和其它类很像，但是有三点不同：

- 构造器的名称和类名一样。
- 构造器没有返回型别和返回值。
- 关键字 `static` 被省略了。

下面用 `Time` 类的构造器为例说明：

```
public Time () {  
    this.hour = 0;  
    this.minute = 0;  
    this.second = 0.0;  
}
```

注意，在 `public` 和 `Time` 之间可以有一个返回型别名称，但构造器没有。凭这一点我们（还有编译器）才能判断这是一个构造器。



空括号 () 标明这个构造器没有任何参数。构造器的每一行语句为实例变量初始化一个任意值（这时候是午夜）。`this` 是一个特殊的关键字，用来指代当前创建的对象。`this` 的用法和别的对象名是一样的。例如可以读写 `this` 的实例变量，也可以把它当做参数传递给其它方法。

`this` 无须声明也不用 `new` 命令创建。实际上还不能赋值，`this` 是系统创建的，你能做的只是往它的实例变量中存放值。

写构造器时常犯的错误就是把 `return` 语句放在最后——要抵制这种诱惑。

## 9.4 再谈构造器

构造器和方法一样可以重载，这就是说可以写带有不同参数的多个构造器。根据 `new` 命令的参数，Java 能调用匹配的构造器。

常见的情况是：一个构造器没有参数（如前面），另一个构造器的参数表和实例变量表完全一样。例如：

```
public Time (int hour, int minute, double second) {  
    this.hour = hour;  
    this.minute = minute;  
    this.second = second;  
}
```

参数的型别和名称与实例变量的完全一致。构造器的全部工作就是把参数的信息复制给实例变量。

如果回过头去看 `Point` 类和 `Rectangle` 类的文档，就会发现两个类都提供像这样的双份构造器。重载构造器使得可以灵活的创建一个对象然后再填充信息或者在创建对象之前就把所有信息准备齐全。

目前为止似乎还看不出有趣在哪里，事实上也不怎么有趣。写构造器是一个沉闷、机械的过程。写上两个以后你就会发现，一边梦游一边也可以干这件事情，只要照抄实例参数表就万事大吉。

## 9.5 创建新对象

虽然构造器看上去很像方法，但不能直接调用，而是在用 `new` 命令是由系统为新对象分配空间，然后调用构造器初始化实例变量。

下面的程序演示了创建和初始化 `Time` 对象的两种办法：

```
class Time {  
    int hour, minute;
```

```
double second;
public Time () {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
}
public Time (int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
}
public static void main (String[] args) {
    // 创建并初始化 Time 对象的途径之一
    Time t1 = new Time ();
    t1.hour = 11;
    t1.minute = 8;
    t1.second = 3.14159;
    System.out.println (t1);
    // 做同样事情的另一种办法
    Time t2 = new Time (11, 8, 3.14159);
    System.out.println (t2);
}
}
```

练习一下，试试弄清楚程序的执行流程。

在 main 方法中，第一次调用 new 命令是时候没有提供参数，于是 Java 调用第一个构造器，接下来的几行为每一个实例变量赋值。

第二次调用 new 命令提供了符合第二个构造器的参数。这样对实例变量的初始化控制更精确（也较为有效率），不过比较难读，到底哪个值被赋予哪个实例变量并非一目了然。

## 9.6 打印对象

这个程序的输出是

Time@80cc7c0

Time@80cc807

Java 打印自定义对象的时候，会打印出型别名和每个对象独占的 16 进制码。这个代码本身没有什么意义，事实上可能随机器不同而改变，甚至在每次运行时也不一样。但如果调试时需要跟踪一个对象，它就很有价值了。

为了让打印的对象更易被用户（而不是程序员）看懂，常常要写像 `printTime` 这样的打印方法：

```
public static void printTime (Time t) {  
    System.out.println (t.hour + ":" +  
                        t.minute + ":" + t.second);  
}
```

比较这个版本和 3.10 节中的 `printTime` 方法。

如果把 `t1` 和 `t2` 作为参数传递给该方法，则输出 `11:8:3.14159`，虽然是按时间的格式组织，但不完全是标准格式。例如当分钟和秒小于 10 的时候，习惯用 0 在十位占位，还有秒的小数部分也一般也会被舍弃。换句话说，应该是这个样子：`11:08:03`。

很多语言都有简单的法子来控制数字的输出格式，而 Java 中却没有这么简单。

Java 提供了非常强大的工具，既可以控制像时间和日期这类具有格式数据的输出，也能很好的解释格式化的输入。不行的是，这些工具的使用方法并不那么简单，所以本书将不讨论这个问题。如果想，可以看看官方文档中 `java.util` 包里关于 `Data` 类的说明。

## 9.7 对象的操作

虽然没有把时间按照理想的格式打印出来，但一样可以写出能操作 `Time` 对象的方法。接下来的数节将演示几种操作对象的方法的可能接口。对许多操作而言都有几种可供选择的接口，现在请考虑如下几种方案各自的利弊：

**纯粹函数-pure function:** 取对象和/或基本数据作为参数但不修改对象。返回值可以是基本数据也可以是用方法创建的一个新对象。

**修正方法-modifier:** 取对象为参数并进行部分或全体修改，一般返回 `void`。

**填充方法-fill-in method:** 参数是一个将被方法填充的“空”对象。从技术上讲，这也是一种修正方法。

## 9.8 纯粹函数

如果一个方法的结果仅依赖于参数，并且不会有修改参数或打印动作之类的附加作用，这个方法就是一个纯粹函数。调用纯粹函数的唯一结果是返回一个值。

接下来的例子中有两个 `Time` 对象，比较二者然后返回一个布尔值来标识第一个时间是否滞后于第二个：

```
public static boolean after (Time time1, Time time2) {  
    if (time1.hour > time2.hour) return true;  
    if (time1.hour < time2.hour) return false;  
    if (time1.minute > time2.minute) return true;  
    if (time1.minute < time2.minute) return false;  
    if (time1.second > time2.second) return true;  
    return false;  
}
```

如果两个时间一致又会返回什么呢？这样的结果得体吗？如果你在为这个方法写文档，你会不会专门说明这种情况呢？

第二个例子是 `addTime` 类，计算两个时间的和。例如现在时间是 9:14:30，面包要在烤箱里呆 3 小时 35 分钟，你要用 `addTime` 来计算面包的出炉时间。

下面是这个方法的草稿，还不是很完善：

```
public static Time addTime (Time t1, Time t2) {  
    Time sum = new Time ();  
    sum.hour = t1.hour + t2.hour;  
    sum.minute = t1.minute + t2.minute;  
    sum.second = t1.second + t2.second;  
    return sum;  
}
```

这个方法虽然返回了一个 `Time` 对象却不是一个构造器。如果有点混淆最好回过头去看看构造器的语法。

接下来是如何使用这个方法的例子，假定 `currentTime` 包含现在所处的时间，`breadTime` 包含烤面包要用的时间。那么就可以用 `addTime` 计算面包出炉的时间了。

```
Time currentTime = new Time (9, 14, 30.0);  
Time breadTime = new Time (3, 35, 0.0);
```

```
Time doneTime = addTime (currentTime, breadTime);  
printTime (doneTime);
```

程序输出 12:49:30.0——正确。话说回来，也有得不到正确结果的时候。你能想出来吗？

——问题在于这个方法不能处理秒或分钟相加结果大于 60 的情况，这种情况下要把“多余”的秒数或分钟数进位到分钟或者小时位。

下面是第二个——正确的版本。

```
public static Time addTime (Time t1, Time t2) {  
    Time sum = new Time ();  
    sum.hour = t1.hour + t2.hour;  
    sum.minute = t1.minute + t2.minute;  
    sum.second = t1.second + t2.second;  
    if (sum.second >= 60.0) {  
        sum.second -= 60.0;  
        sum.minute += 1;  
    }  
    if (sum.minute >= 60) {  
        sum.minute -= 60;  
        sum.hour += 1;  
    }  
    return sum;  
}
```

虽然正确，但开始有点长了。后面会给出一个比较短小的候选办法。

上面的代码演示了两个新鲜的运算符：加赋值+=和减赋值-=。这两个运算符提供了更简洁的途径来表示自增和自减赋值运算。它们有点像++和--，但有如下两个特点：**(1)**对双精度数和整数都可以操作；**(2)**增加的不必是 1。

sum.second -= 60.0;的效果等同于 sum.second = sum.second - 60;

## 9.9 修正方法

下面将 increment 方法作为修正方法的例子，它把一个给定的秒数加给 Time 对象。还是先写一个草稿：

```
public static void increment (Time time, double secs) {  
    time.second += secs;  
    if (time.second >= 60.0) {
```

```
        time.second -= 60.0;
        time.minute += 1;
    }
    if (time.minute >= 60) {
        time.minute -= 60;
        time.hour += 1;
    }
}
```

第一行运算的余数和前面的处理方法一样。

这个方法正确吗？如果参数 `secs` 远大于 60 会怎么样呢？那样的话减一次 60 就不够了，所以要一直减到 `second` 小于 60 为止。要达到目的只要用 `while` 语句替换 `if` 语句就行了：

```
public static void increment (Time time, double secs) {
    time.second += secs;
    while (time.second >= 60.0) {
        time.second -= 60.0;
        time.minute += 1;
    }
    while (time.minute >= 60) {
        time.minute -= 60;
        time.hour += 1;
    }
}
```

目的达到了，但效率并非很高。有没有别的办法可以不用迭代呢？

## 9.10 填充方法

偶尔我们能看见 `addTime` 这样的类用了不同的接口（即不同的参数和返回值）来编写，这样就用不着每次调用 `addTime` 方法的时候都创建一个新对象，只要调用者提供一个“空对象”来让 `addTime` 存放结果就好了。比较下面的版本和前面的版本：

```
public static void addTimeFill (Time t1, Time t2, Time sum) {
    sum.hour = t1.hour + t2.hour;
    sum.minute = t1.minute + t2.minute;
```

```
sum.second = t1.second + t2.second;
if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
}
if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
}
}
```

这种办法的有点之一是调用者可以选择利用同一个对象来完成一系列加法。这种做法效率比较高，虽然可能导致一些让人迷惑的微妙错误。对于大多数庞大的程序来说，消耗一点运行时间以避免很多调试时间是绝对值得的。

## 9.11 哪种方法最好

修正方法和填充方法能做的纯粹函数也可以做到。事实上有的编程语言仅允许使用纯粹函数，叫做**函数编程语言**（*functional programming language*）。有的程序员相信用纯粹函数比修正方法可以更快地开发程序并且不易出错。然而，有时修正方法却比较简便，有的地方纯粹函数程序效率也不如后者。

总的来说，我推荐在合适的时候总是写纯粹函数，除非修正方法有压倒性的优点才转为使用它。这种方法可以称为函数编程风格。

## 9.12 递增开发 vs.规划

本章中演示了一种叫做**迭代改进快速原型**（*rapid prototyping with iterative improvement*）的编程方法。每个例子中我都先写一个执行基本计算的草稿（或者叫原型），然后在不同的情况下测试，找到不足的地方就加以修正。

虽然这种办法可以很有效，但也有可能导致很严重的问题：写出的代码复杂程度可能超过必须——因为要应付许多特殊情况；程序不可靠——因为难以保证所有错误都被发现。

替代的选择是**高度规划**（*high-level planning*）——这种方法只要具备一些洞察力就可以让编程更加轻松。这个例子需要洞察的就是：一个 `Time` 对象是一个三位的 60 进制数！秒是“个位”，分钟是“60 位”，小时是“3600 位”。

我们写好的 `addTime` 和 `increment` 方法有效的执行了 60 进制加法, 因为其中执行了“进位”。

接下来首先把 `Time` 对象转换为双精度数:

```
public static double convertToSeconds (Time t) {  
    int minutes = t.hour * 60 + t.minute;  
    double seconds = minutes * 60 + t.second;  
    return seconds;  
}
```

剩下的事情是把双精度数再转换成 `Time` 对象。可以专门写一个方法, 但是写成第三个构造器更有意思:

```
public Time (double secs) {  
    this.hour = (int) (secs / 3600.0);  
    secs -= this.hour * 3600.0;  
    this.minute = (int) (secs / 60.0);  
    secs -= this.minute * 60;  
    this.second = secs;  
}
```

这个构造器有点特殊, 在给实例变量赋值的时候执行了一些计算。

你可能需要说服自己相信这里用来转换进制的技术是正确的。假定你被自己说服了, 就可以用这两个方法重写 `addTime` 方法:

```
public static Time addTime (Time t1, Time t2) {  
    double seconds = convertToSeconds (t1) + convertToSeconds (t2);  
    return new Time (seconds);  
}
```

这比原来的版本简练了很多, 也很容易验证它的正确性 (前提是假定调用的方法无错)。

练习: 用同样的办法重写 `increment` 方法。

## 9.13 通用化

实际上在 60 进制和 10 进制数之间转换比处理时间的难度大。比起时间处理, 进制转换更加抽象, 直觉基本派不上用场。



不过一旦有了把时间当做 60 进制数的观念,并且把精力放在写进制转换方法(`convertToSeconds` 和第三个构造器)上,就得到了一个更简短易读、便于调试且可信度高的程序。

甚至以后还能添加更多的特性。例如,要把两个时间相减以求延迟,老实巴交的办法就会完全用“借位”来实现减法运算。而用进制转换方法实现会更加容易。

讽刺的是,有时把问题弄难就是弄简单。(“弄难”的意思实际是说更一般化、抽象化,特殊情形少则出错的机会也就少。)

## 9.14 算法

当为一类问题写解法而不是写特殊问题的解法时,就在写一个**算法**(*algorithm*)。第一章就提到了这个词,但当时没有仔细的定义。这个概念很不好定义,所以下面多尝试几个办法。

首先排除掉不是算法的部分。例如在学习一位数乘法的时候,可能背过乘法表,花了半天时间记住了其中 100 个特定的解——然而这样你学到的知识并不是算法。

但你可能是个“懒”学生,喜欢搞点“骗术”来逃避背诵的苦役。例如要算  $n$  和 9 相乘的积,可以先写下  $n - 1$  做十位数,再用  $10 - n$  做个位数——这个“骗术”是做一位数乘法的通解。这就是一个算法!

类似的,进位做加法、借位做减法还有长除法都是减法。算法的特征之一就是实现时不需要智力。算法只是机械地根据一组简单规则依次执行的过程。

在我看来,人类用大量的时间呆在学校里只是为了学习如何执行一些——毫不夸张地说——不需要智力的算法,这真是一种羞辱!

话说回来,设计算法的过程却十分有趣而且充满了智力挑战,正是编程的神髓所在。

许多事情人做的自然而然,既不困难也不用有意思的去想,而这些却很难表达成算法。理解自然语言就是一个极好的例子。人人都懂,但是迄今为止无人能解释我们是怎么听懂的,至少无法用算法的形式描述出来。

本书后面将给你机会设计多种问题的一些简单算法。

## 9.15 术语表

**类-class:** 以前把类定义为相关方法的容器,这一章了解到类定义也是一种对象模板。

**实例-*instance***: 类的一个成员。每个对象都是某种类的实例。

**构造器-*constructor***: 用来初始化新创建对象的实例变量的方法。

**启动类-*startup class***: 包含 `main` 方法作为程序开始执行入口点的类。

**函数-*function***: 结果只依赖于参数，并且不会有修改参数或打印动作之类的附加作用，仅仅返回一个值的方法。

**函数编程风格-*functional programming style***: 程序设计中大部分主要方法都时函数的编程风格。

**修正方法-*modifier***: 取对象为参数并进行部分或全体修改的方法，一般返回 `void`。

**填充方法-*fill-in method***: 这种方法的参数是一个将被方法填充的“空”对象，不产生返回值。这种方法通常不是最好的选择。

**算法-*algorithm***: 为解决一类问题而机械地执行的一组指令。

## 第 10 章 数组

**数组** (*array*) 是一组有索引号的值的集合。整数、双精度数或者其它任何型别的值都可以构成数组，但一个数组中的所有值都必须是同样的型别。

表示数组型别的语法是在其它 Java 型别后面加上一对方括号[]。例如 `int[]` 表示整数数组型别，`double[]` 表示双精度数数组型别。

声明数组型别变量的方法和声明其它型别变量的方法一样：

```
int[] count;
```

```
double[] values;
```

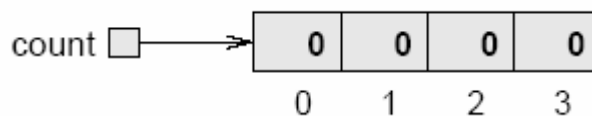
如果不进行初始化，变量的值就是 `null`。要创建一个数组用 `new` 命令。

```
count = new int[4];
```

```
values = new double[size];
```

第一个赋值语句使 `count` 指向 4 个整数的数组；第二句使 `values` 指向一个双精度数组，数组元素的数目取决于 `size` 的值。可以把任意整数型别表达式作为数组的长度。

下图示意了在状态图中数组的表示方法：



盒子内的粗体数字是数组的元素，外面的细体数字是每个盒子的索引号。分配一个数组的时候所有元素被初始化为 0。

### 10.1 访问数组元素

向数组中存放值使用[]运算符。例如 `count[0]` 指向第 0 个数组元素，`count[1]` 指向第 1 个数组元素。

[] 运算符可以用在表达式的任何地方：

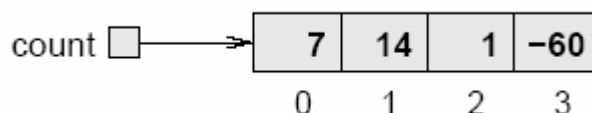
```
count[0] = 7;
```

```
count[1] = count[0] * 2;
```

```
count[2]++;
```

```
count[3] -= 60;
```

所有语句都是合法的。这段代码的输出是：



注意数组元素的索引号是从 0 开始的，这和 `String` 的索引号一样。超越数组边界是很常见的错误，这会导致一个 `ArrayOutOfBoundsException` 异常。和所有异常一样，给出一条出错信息后程序就退出。

索引号的表达式只要是 `int` 型别就行。对数组进行索引的一个普遍方法是利用循环变量。例如：

```
int i = 0;
while (i < 4) {
    System.out.println (count[i]);
    i++;
}
```

这个标准的 `while` 循环从 0 计数增加到 4，当循环变量 `i` 的值为 4 时，条件为假循环终止。也就是说循环体只有在 `i` 为 0、1、2 和 3 的时候执行。

每次对循环的遍历都用 `i` 对数组进行索引并打印第 `i` 个元素。这样的数组遍历很常用，数组和循环在程序中往往形影不离。

## 10.2 拷贝数组

拷贝一个数组变量实际是在拷贝对数组的引用，例如：

```
double[] a = new double [3];
double[] b = a;
```

这段代码创建了有三个双精度数的数组并将两个变量都指向它。这也是一种形式的别名。



任何改变都会同时影响两个变量，这并非我们乐意看到的行为。所以需要 对数组进行拷贝，通过分配一个新的数组并把每个元素都复制进去就能做到。

```
double[] b = new double [3];
int i = 0;
while (i < 4) {
    b[i] = a[i];
}
```

```
        i++;  
    }
```

## 10.3 for 循环语句

以往讨论的循环语句通常有如下要素：第一步都是初始化变量；用这个变量进行测试或者检测条件；循环体内对变量执行一定操作，比如自增运算。

这种循环太没有特色，所以有了一个更加简洁的候选方案：for 循环语句。通用的语法如下：

```
for (INITIALIZER; CONDITION; INCREMENTOR) {  
    BODY  
}
```

这种语句和下面的 while 语句完全等效：

```
INITIALIZER;  
while (CONDITION) {  
    BODY  
    INCREMENTOR  
}
```

当然，for 语句更简洁，因为它把所有循环相关的语句放在了一起，这样也便于阅读。例如：

```
for (int i = 0; i < 4; i++) {  
    System.out.println (count[i]);  
}
```

等价于：

```
int i = 0;  
while (i < 4) {  
    System.out.println (count[i]);  
    i++;  
}
```

练习：写一个 for 语句来复制数组的所有元素。

## 10.4 数组和对象

数组在很多方面的行为都类似于对象：

- 声明一个数组变量的同时就有了对数组的引用。
- 必须用 `new` 命令创建一个数组本身。
- 把数组作为参数传递等于传递了引用，这意味着被调用的方法可以改变数组内容。

前面见过的一些对象比如 `Rectangle` 和数组很类似，二者都是值容器。这就产生了一个问题，“四个整数的数组和一个 `Rectangle` 对象有什么区别？”

回头看看本章开头对“数组”的定义就能发现不同之处：数组元素用索引号标识，而对象的元素（实例变量）有自己的名称（譬如 `x`、`width` 等等）。

另一个不同点就是数组的元素必须是相同型别。虽然 `Rectangle` 对象也是如此，但其它对象却未必都只有一种型别的实例变量（例如 `Time`）。

## 10.5 数组长度

事实上数组也有一个带名称的实例变量：`length`。望文生义可知 `length` 包含了数组的长度（元素的个数）。比起用常量来，把这个值作为循环的上界是再好不过了。这样，如果数组的长度改变了也不必到程序中去修改循环语句的内容，无论数组大小都能正确的工作。

```
for (int i = 0; i < a.length; i++) {  
    b[i] = a[i];  
}
```

循环体最后一次执行时 `i` 的值是 `a.length - 1`，也即最后一个元素的索引。当 `i` 值等于 `a.length` 时，条件为循环体终止执行，避免了引起异常——太好了。这段代码假定了数组 `b` 的长度至少和数组 `a` 一样。

练习：以一个整数数组为参数写一个 `cloneArray` 方法，创建一个同样大小的数组，然后把第一个数组的每个元素复制到新数组中，最后再返回新数组的引用。

## 10.6 随机数

大多数计算机程序每次执行的时候总是做同样的事情，这被称为具有**确定的**（*deterministic*）性质。通常来讲确定性是个好东西，因为我们期望同样的计算得到同样的结果。但在有的应用中，我们希望计算机有一定的不可预测性。

游戏就是一个最好的例子，当然还有更多。

让程序真正的达到**非定性的**（*nondeterministic*）水平在实践中被证明是极难的，不过使之看起来接近非定性的却有不少途径。方法之一就是生成一个随机数用来决定程序的结果。Java 提供了生成伪随机数的内建方法，从数学角度讲生成的不是真正的随机数，但对要达到的目标来说可以认为就是了。

请查看 `Math` 类中 `random` 方法的文档。返回值是一个介于 0.0 和 1.0 之间的双精度数，准确地说，是一个大于等于 0.0 小于 1.0 的数。每次调用 `random` 方法就得到随机序列上的下一个数。请运行下面的代码作为例子：

```
for (int i = 0; i < 10; i++) {  
    double x = Math.random ();  
    System.out.println (x);  
}
```

要生成一个介于 0.0 和上界如 `high` 之间的双精度数，可以把 `x` 和 `high` 相乘。那么又要如何生成介于 `low` 和 `high` 之间的随机数呢？如何生成一个随机整数呢？

练习 10.1 写一个 `randomDouble` 方法，取两个双精度数 `low` 和 `high` 为参数，返回一个双精度随机数 `x` 使得  $low \bullet x < high$ 。

练习 10.2 写一个 `randomInt` 方法取两个参数 `low` 和 `high`，返回一个介于（包括等于）两者之间的随机整数。

## 10.7 随机数数组

如果正确实现了 `randomInt`，则介于 `low` 和 `high` 之间的每个值出现的几率是一样的。如果生成许多数，每个值出现的次数应该基本相同。

检验这个方法是否正确的途径之一就是：生成大量的随机值，存入一个数组，对每个值出现的次数进行统计。

下面的方法取数组长度做参数，分配一个新的整数数组，用随机值填充，然后返回对新数组的引用。

```
public static int[] randomArray (int n) {  
    int[] a = new int[n];  
    for (int i = 0; i < a.length; i++) {  
        a[i] = randomInt (0, 100);  
    }  
    return a;  
}
```

返回型别是 `int[]`，意味着方法返回的是一个整数数组。要测试这个方法最简便的途径就是写一个方法来打印数组的内容。

```
public static void printArray (int[] a) {  
    for (int i = 0; i<a.length; i++) {  
        System.out.println (a[i]);  
    }  
}
```

下面的代码生成了一个数组并将其打印出来：

```
int numValues = 8;  
int[] array = randomArray (numValues);  
printArray (array);
```

在我的机器上输出是

27

6

54

62

54

2

44

81

看上去相当随机，你的结果可能不一样。

如果这是一次考试的分数就不太妙了，老师可能会把结果绘制成矩形图，以便分析每个分段的分布情况。

对于考试分数，可能需要 10 个计数器来分别保存 90 分以上的人数、80 分以上的人数……接下来的几节就要开发代码来生成这个统计图。

## 10.8 计数

一个解决问题的好途径应该是先写一个较容易的简单实用的方法，然后把这个方法集成进一个完整的方案里。当然，没有办法可以预先得知那种方法是最实用的，不过随着经验的积累自然能想到更好的主意。

同时什么才是容易写的部分也并非一目了然，而一种好的思路就是想想能不能把问题分解以适用以前遇到过的求解模式。

回到 7.7 节，看看循环遍历一个字符串以统计一个给定字母出现次数的思



路。可以把那个程序作为“遍历计数”模式的一个例子。这个模式的要素是：

- 一个可以遍历的集合或容器，比如数组或字符串。
- 对容器中的每个元素都适用的检测。
- 记录有多少元素通过了检测的计数器。

在这里容器是一个整数数组；检测一个分数值属于哪个给定区间。

一个名为 `inRange` 的方法用来记录分数在各个区间的分布数量。参数是数组和分别标识区间上界和下界的两个整数。

```
public static int inRange (int[] a, int low, int high) {  
    int count = 0;  
    for (int i=0; i<a.length; i++) {  
        if (a[i] >= low && a[i] < high) count++;  
    }  
    return count;  
}
```

在行文的时候我没有细致的说明临界点是否包括在区间内，但是看看代码就会明白下界被包括在内而上界则没有。这能保证没有元素被重复统计。

现在可以用写好的方法将分数塞进对应的区间了：

```
int[] scores = randomArray (30);  
int a = inRange (scores, 90, 100);  
int b = inRange (scores, 80, 90);  
int c = inRange (scores, 70, 80);  
int d = inRange (scores, 60, 70);  
int f = inRange (scores, 0, 60);
```

## 10.9 矩形图

前面写出来代码有点罗嗦，然而对仅有的几个区间还可以忍受。不过，假设要统计每个分数出现的次数，就有 100 个可能值。你想不想写：

```
int count0 = inRange (scores, 0, 1);  
int count1 = inRange (scores, 1, 2);  
int count2 = inRange (scores, 2, 3);  
...  
int count3 = inRange (scores, 99, 100);
```

我可不干。这里实际要做的事情是存放 100 个整数，更好的办法是利用索

引号来访问，所以应该立即想到数组。

不论是单独用一个计数器还是一个计数器的数组，计数模式是一样的。针对这个问题，首先在循环外初始化数组，然后在循环体中调用 `inRange` 来存放结果：

```
int[] counts = new int [100];
for (int i = 0; i<100; i++) {
    counts[i] = inRange (scores, i, i+1);
}
```

这里有点炫耀技巧的东西，循环变量 `i` 担任了两个角色：在数组中是索引号，在 `inRange` 中是参数。

### 10.10 单遍历方案

虽然前面得到的代码能达到目的，但并没有达到应该的效率。每次调用 `inRange` 方法，都要遍历整个数组。随着区间数目的增加，遍历次数也更多。

最好是只遍历数组一次，一次计算完各个值所属的区间，同时自增相应的计数器。在这个例子中，这点计算微不足道，因为可以用分数作为计数器数组的索引号。

最后的代码遍历了分数的数组并且一次就生成了矩形图。

```
int[] counts = new int [100];
for (int i = 0; i < scores.length; i++) {
    int index = scores[i];
    counts[index]++;
}
```

## 10.11 术语表

**数组-array**：一组有索引号的同型别值的集合。

**容器-collection**：任何包含了一组项目或元素的数据结构。

**元素-element**：数组中的一个值。`[]`运算符用来选择一个元素。

**索引号-index**：用来标识数组元素的一个整数变量或值。

**确定性-deterministic**：程序每次被调用都做同一件事情的性质。

**伪随机数-pseudorandom**：一序列看上去貌似随机的数，实际上是具有确定性的计算机产生的。

**矩形图-histogram**：统计学概念。本书指用来统计指定区间中的值的出现次数的整数数组。

## 第 11 章 对象的数组

### 11.1 复合

到目前为止我们见过了多种复合的例子。所谓复合就是把语言的各种成分进行有机组合。我们见到的第一个例子就是用方法调用作为一个表达式的一部分。另一个例子是那些锯齿状排列的语句：在 `if` 语句中放 `while` 循环或者另一个 `if` 语句，等等。

见识了这些方式，也学习了数组和对象，对接下来要学习对象的数组应该不会感到吃惊了。事实上，也有包含数组的对象（作为实例变量），还有数组的数组，还有包含对象的对象，等等不一而足。

接下来的两章将以纸牌对象（*Card*）为例来看看对象的数组。

### 11.2 纸牌对象——Card

如果你没有玩过纸牌的话，乘现在的大好时光赶紧去买一副吧，不然这一章就没有什么意思啦。一副牌 52 张，每张属于四种花色之一，四种花色各有 13 个品相。花色依次是：黑桃 *Spade*、红心 *Heart*、方块 *Diamond* 和梅花 *Club*（按桥牌中的等级降序排列）。品相是 *Ace*、2、3、4、5、6、7、8、9、10、*Jack*、*Queen* 和 *King*。根据玩的是什么游戏而定，*Ace* 的品相可能比 *King* 高但也可能比 2 还要低。

如果要定义一个新对象来表示纸牌，很显然实例变量就是品相 `rank` 和花色 `suit`。不过实例变量的型别就不大明显了。可能是字符串，其中包含了例如 "*Spade*" 花色或者 "*Queen*" 品相。这种实现的问题在于难以比较两张牌的 `rank` 高低或者 `suit` 异同。

候选方案之一是用整数来对 `rank` 和 `suit` 进行**编码**（*encode*）。“编码”在这里的意思不是有人爱说的加密（转换为密码），计算机科学家说的 *encode* 意思是“定义一个从想表示的事物到一个数字序列的映射”。例如

Spades	↦	3
Hearts	↦	2
Diamonds	↦	1
Clubs	↦	0

这个图的显著特点就是从 `suit` 到 `integer` 值的映射是有序的，这样就可以

通过比较整数来比较花色了。表示 rank 的映射也相当直观，每个数字品相就对应同样的整数，而人头牌的对应关系如下：

Jack	↦	11
Queen	↦	12
King	↦	13

用数学符号来表示映射的原因是映射并非 Java 程序的一部分，这属于程序设计的范畴，但不会显式地出现在代码中。Card 对象型别的类定义可以是这样：

```
class Card
{
    int suit, rank;
    public Card () {
        this.suit = 0; this.rank = 0;
    }
    public Card (int suit, int rank) {
        this.suit = suit; this.rank = rank;
    }
}
```

像以往一样，提供了两个构造器，一个为每个实例变量取得一个参数，另一个没有参数。要创建一个表示梅花 3 的对象，可以用 new 命令：

```
Card threeOfClubs = new Card (0, 3);
```

第一个参数 0 表示花色是梅花 Club。

## 11.3 printCard 方法

创建一个新类的时候，通常第一步是声明实例变量并写好构造器。第二步则是写每个对象都有的标准方法，包括一个用来打印对象的方法，还有一至两个用来比较对象的方法。下面先写用来打印的 printCard 方法。

为了按便于人类阅读的方式把 Card 对象打印出来，就得把整数映射到词汇上。通常用一个字符串数组来完成这件事。创建字符串数组的方法和创建其它基本型别数组的方法一样：

```
String[] suits = new String [4];
```

接下来为数组的元素赋值：

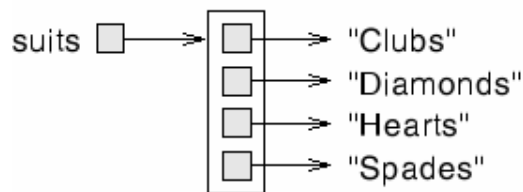
```
suits[0] = "Clubs";
```

```
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

创建并初始化数组是非常普遍的操作，所以 Java 提供了一种特殊的语法格式来完成这个任务：

```
String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
```

这个语句的效果和单独声明、分配然后再赋值完全一样。这个数组的堆栈图应该是这样：



数组元素是对字符串的引用，而不是字符串本身。这一点对所有对象组成的数组都成立，稍后会讨论更多细节。现在还需要另一个字符串数组来对 rank 进行编码：

```
String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
                  "7", "8", "9", "10", "Jack", "Queen", "King" };
```

这里的"narf"作用是数组第 0 个元素位置的占位符，但在程序中不会用到。有效的品相只是 1-13。本来不必浪费掉第一个元素的，也可以像通常那样从 0 开始，不过把 2 编码为 2、3 编码为 3 等等这样比较自然。

可以把 suit 和 rank 作为这两个数组相应的索引号。在方法 printCard 中：

```
public static void printCard (Card c) {
    String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
    String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
                      "7", "8", "9", "10", "Jack", "Queen", "King" };
    System.out.println (ranks[c.rank] + " of " + suits[c.suit]);
}
```

表达式[c.suit]的目的是“把 c 对象的实例变量 suit 作为 suits 数组的索引，以选取相应的字符串”。

```
Card card = new Card (1, 11);
printCard (card);
```

这段代码输出是：

Jack of Diamonds.

## 11.4 sameCard 方法

same（一样的）在自然语言中出现的时候，其含义要根据你的思路才能确定，这样一来具体情况就多了。

例如，我说 Chris and I have the same car，实际是说他的车和我的车是同样的型号；我说 Chris and I have the same mother，我的意思是说我们的母亲是同一个人。所以 same 的意思根据上下文是不一样的。

谈论对象的时候，同样具有这种歧义存在。例如说，两个 Card 对象是“一样的”：这是说二者含有相同的数据（rank 和 suit）呢还是二者是同一个 Card 对象？

验证两个引用是否指向同一个对象要使用 == 运算符。例如：

```
Card card1 = new Card (1, 11);
Card card2 = card1;
if (card1 == card2) {
    System.out.println ("card1 and card2 are the same object.");
}
```

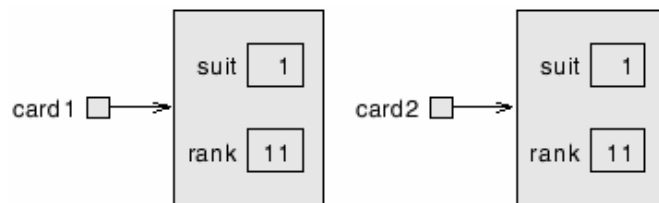
这种等式叫做**浅比较**（*shallow equality*），因为它只比较引用而不是对象的内容。要比较对象内容——**深比较**（*deep equality*），通常就写一个名为 sameCard 形式的方法。

```
public static boolean sameCard (Card c1, Card c2) {
    return (c1.suit == c2.suit && c1.rank == c2.rank);
}
```

如果创建了两个含有相同数据的不同对象就可以用 sameCard 来比较它们是否相同了：

```
Card card1 = new Card (1, 11);
Card card2 = new Card (1, 11);
if (sameCard (card1, card2)) {
    System.out.println ("card1 and card2 are the same card.");
}
```

因为这里的 card1 和 card2 是含有相同数据的两个对象，所以条件为 true。



如果 `card1 == card2` 为 `true` 的状态图会是什么样呢？

在 7.11 节中说不要对字符串用 `==` 运算符，因为当时没有明确其含义。现在知道，比较的不是字符串的内容（深比较）而是检查二者是否是同一个对象（浅比较）。

## 11.5 compareCard 方法

对基本型别来说，条件运算符可以比较值并判断大小。不过这类运算符（`<` 和 `>` 还有其它一些）不能用于对象型别。`String` 类有内建的 `compareTo` 方法，`Card` 类则要现写，我们将其定名为 `compareCard`。稍后将用这个方法对一副牌排序。

有的集合是完全有序的，于是可以比较任意两个元素并判断大小。例如整数和浮点数都是有序的。有的集合则是无序的，就是说没有一种讲得通的办法可以比较元素的大小。例如水果的种类就是无序的，所以不能空泛地说苹果比橙子大。`Java` 中 `boolean` 型别是无序的，不能说 `true` 比 `false` 大。

纸牌是部分有序的，就是说有时可以比较两张牌有时则不能。例如梅花 3 比梅花 2 大，方块 3 比梅花 3 大，那么梅花 3 和方块 2 哪个更大呢？一个的品相高一个的花色大，这就取决于哪个条件优先，品相还是花色。

要让牌变得可比，就要确定品相和花色哪个条件更重要。坦白说选择是完全随意的。为了方便比较，把花色作为更重要的依据。如果你买了一副新牌就会发现它把所有的梅花放在一起，然后全部是方块等等，如此排列。

确定好规则就可以写 `compareCard` 方法了。取两个 `Card` 对象做参数，返回值 1 表示第一张牌大，-1 表示第二张大，0 表示两张牌花色品相都一样（深比较）。直接给出比较的返回值有点容易混淆，但这是作比较的标准方法。

首先比较花色：

```
if (c1.suit > c2.suit) return 1;
```

```
if (c1.suit < c2.suit) return -1;
```

如果两句都不为真，`suit` 肯定一样，再比较品相：

```
if (c1.rank > c2.rank) return 1;
```

```
if (c1.rank < c2.rank) return -1;
```

如果都不为真品相一定一样，这个比较法，`Ace` 就比 2 点小。

练习：修改代码让 Ace 的品相高于 King，然后封装进一个方法。

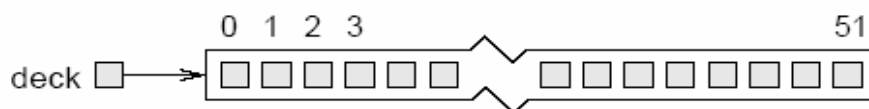
## 11.6 牌的数组

这一章选择牌作为对象的原因是：牌的数组有一个显而易见的用处——牌套。这儿的代码创建了一副 52 张的新牌放在牌套（*deck*）里：

deck of 52 cards:

```
Card[] deck = new Card [52];
```

下面是对象的状态图：



最重要的是看到数组仅仅包含了对对象的引用，而没有包含任何 **Card** 对象。这个数组的所有元素被初始化为 `null`。通常访问数组元素的办法如下：

```
if (deck[3] == null) {
    System.out.println ("No cards yet!");
}
```

但是如果试图访问一个不存在在 `Card` 对象的实例变量将会引起一个 `NullPointerException` 异常。

```
deck[2].rank; // NullPointerException
```

虽然发生了异常，但访问第 2 张牌（实际是第三张——因为从 0 开始计数）的 `rank` 的语法是正确的。这实际上又是一例复合，是访问数组元素和对象的实例变量的语法复合。

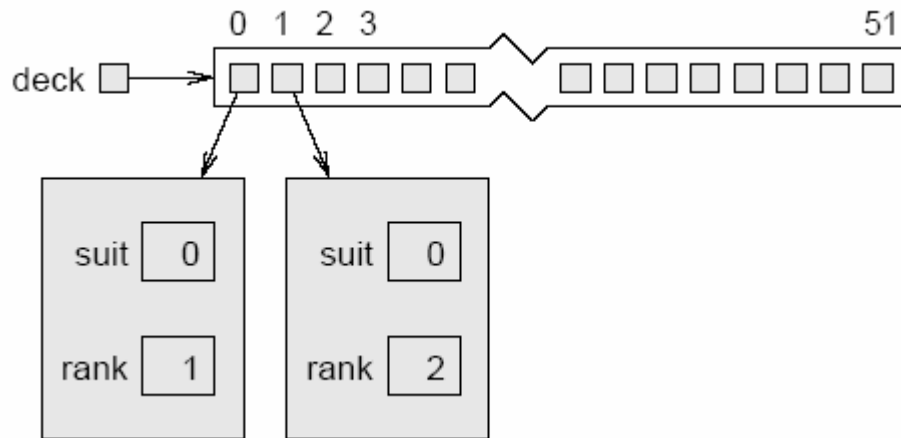
把一个牌套里的牌重新排列可以用嵌套循环：

```
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
    for (int rank = 1; rank <= 13; rank++) {
        deck[index] = new Card (suit, rank);
        index++;
    }
}
```

外循环从 0 到 3 枚举花色；对每种花色内循环从 1 到 13 枚举品相。由于外循环迭代 4 次内循环迭代 13 次，所以循环体总共被执行了 52 次（ $13 \times 4$ ）。

变量 `index` 用来记录下一张牌在 `deck` 中的位置。下面的状态图示意了头两张牌分配好位置之后 `deck` 的情况：





## 11.7 printDeck 方法

使用数组的时候有一个方法打印数组内容是很方便的。前面已经数次见过遍历数组的模式，所以对下面的方法不应该感到陌生。

```
public static void printDeck (Card[] deck) {  
    for (int i=0; i<deck.length; i++) {  
        printCard (deck[i]);  
    }  
}
```

deck 的型别是 Card[]，而每个 deck 的元素也有自己的型别，所以 deck[i] 是 printCard 的合法参数。

## 11.8 查找

下一个要写的方法是 findCard，用来搜索整个纸牌数组以确定是否有特定的牌。这个方法的用处不是很明显，但可以借机演示查找的两种方式，顺序查找和二分查找。顺序查找较为简单，主要思路就是遍历 deck 把要找的牌和牌套里的每张牌比较直到找到为止。

找到了就返回纸牌出现的索引号，否则就返回-1。

```
public static int findCard (Card[] deck, Card card) {  
    for (int i = 0; i< deck.length; i++) {  
        if (sameCard (deck[i], card)) return i;  
    }  
    return -1;  
}
```

`findCard` 的参数名为 `card` 和 `deck`，参数名和其型别名一样看起来似乎很古怪（变量 `card` 的型别是 `Card`）。这不仅合法也很常见，然而有时也会使代码比较难读。在这里这么做是可以的。

找到牌之后方法立即返回，这就是说找到了牌就不必继续遍历整副牌。如果循环终止也找不到牌，就返回-1，说明要找的牌不在牌套中。

如果牌套中的牌是完全无序叠放的，那么就没有比这样查找更快的办法了。只有检查每一张牌，否则没有别的办法能确定要找的牌在哪里。

而在字典中找一个单词的时候并没有搜索每一个单词，原因是单词按照字母顺序进行了排列。这种情况就可以采用类似二分法的查找算法：

1. 从中间某处开始。
2. 将当前页中的词和要找的词相比较。
3. 如果找到了就停止。
4. 如果要找的词在当前页所有词的后面，就跳到词典中当前页的后面部分再开始第 2 步。
5. 如果要找的词在当前页所有词的前面，就跳到词典中当前也之前的部分再开始第 5 步。

如果最后发现要找的单词位于某页上的两个紧挨着的单词之间，就可以推断词典中没有要找的单词。还有一种可能是要找的单词被排错了，不过这就和假定单词是按照字母表排版的前提矛盾了。

回过来看牌套中的情况，如果知道牌是按顺序放的，就可以写出 `findCard` 方法的一个更快版本。最好的法子是用递归方法来写一个二分搜索，因为二分法天生就是递归的。

写一个 `findBisect` 方法的技巧是取两个索引号 `low` 和 `high` 作为参数，用来标识要搜索的数组片断的区间（包括 `low` 和 `high`）。

1. 开始搜索。选一个介于 `low` 和 `high` 中间的索引号（命名为 `mid`），然后将要找的牌和该索引号的牌比较。
2. 如果找到了就停止。
3. 如果索引为 `mid` 的牌比要找的牌品高，就继续在 `low` 到 `mid-1` 之间查找。
4. 如果索引为 `mid` 的牌比要找的牌品低，就继续在 `high` 到 `mid+1` 之间查找。

第 3 步和第 4 步让人怀疑就是递归调用。下面把这个过程翻译成 Java 代码：

```
public static int findBisect (Card[] deck, Card card,
                             int low, int high) {
    int mid = (high + low) / 2;
    int comp = compareCard (deck[mid], card);
```

```
    if (comp == 0) {
        return mid;
    } else if (comp > 0) {
        return findBisect (deck, card, low, mid-1);
    } else {
        return findBisect (deck, card, mid+1, high);
    }
}
```

注意 `compareCard` 仅仅被调用了一次并把结果存放起来。

虽然这段代码运用了二分搜索的核心思路，但却遗漏了一条。像这么写的话如果要找的牌不在牌套中，程序将一直做递归。需要一种办法来检测这种情况并加以处理（返回-1）。

要判断目标牌是否在牌套里最容易的办法就是看 `high` 是不是比 `low` 小。慢，牌当然可能在牌套里，我的意思是说，在 `low` 到 `high` 区间上牌套里的分段中没有要找的牌。（译注：意思是说，上界 `high` 比下界 `low` 还要小，这个区间实际上不存在。）

在上面的代码中加入一行就能解决这个问题：

```
public static int findBisect (Card[] deck, Card card,
                             int low, int high) {
    System.out.println (low + ", " + high);
    if (high < low) return -1;
    int mid = (high + low) / 2;
    int comp = deck[mid].compareCard (card);
    if (comp == 0) {
        return mid;
    } else if (comp > 0) {
        return findBisect (deck, card, low, mid-1);
    } else {
        return findBisect (deck, card, mid+1, high);
    }
}
```

在开始加入了一条打印语句输出递归调用的序列以判断递归确实全部完成。

用这段代码进行试验：

```
Card card1 = new Card (1, 11);
```

```
System.out.println (findBisect (deck, card1, 0, 51));
```

得到下面的输出

0, 51

0, 24

13, 24

19, 24

22, 24

23

然后“捏造”一张不存在的牌（方块 15），试着查找这张“假牌”，得到的结果是

0, 51

0, 24

13, 24

13, 17

13, 14

13, 12

-1

然而这些测试不能证明程序是正确的。事实上，再多的测试也不能证明一个程序的正确性。不过话说回来，通过较少的情况来检验代码，就可以说服自己程序是正确的了。

这个问题用二分查找的递归次数并不多，典型情况是六七次，等于把 `compareCard` 的调用从顺序查找的 52 次减少到了一位数字。一般说来二分查找比顺序查找快许多，而且数组越大差距就越明显。

递归程序中容易犯两个错误，一是忘记在递归中包含基本情形，一是递归调用永远也不能抵达基本情形。两种情况都会导致无穷递归，最后 Java 将抛出一个 `StackOverflowException` 异常。

## 11.9 Deck 和 subdeck

看看 `findBisect` 方法的接口

```
public static int findBisect (Card[] deck, Card card,  
                             int low, int high)
```

把三个参数 `deck`、`low` 和 `high` 考虑成一个参数并记作 `subdeck`，也许更好理解。这种思路很常用，有时我把它叫做抽象参数。“抽象”的意思是说并非程序代码的一部分，而是用来在较高层次描述程序的函数。

例如，当调用一个方法并传递一个数组和边界值 `low` 与 `high` 的时候，没有办法保证不让被调用的方法访问越过边界的数组部分。所以字面上看并没有传递 `deck` 的一个子集 (*subset of the deck* <译注：即一个 *subdeck*>)，实际传递的是整个 `deck` (译注：即代码中方法的第一个参数 `Card[] deck`)，但考虑到接收参数的方法是按相应的规则工作的，所以把所有这些参数看成一个抽象参数 `subdeck` 就是可行的，而且这么做就达到了在较高层次描述问题的目的。

在 9.7 节你可能也发现了这种抽象，那里提到“空的”数据结构。(译注：为了准确理解这段字数不算少的论述，这里抄录 9.7 节的英文原文：*fill-in method: One of the arguments is an "empty" object that gets filled in by the method. Technically, this is a type of modifier.*) 把“空的 (*empty*)”放在双引号里就是为了表明这种提法并不完全准确。所有变量在任何时候都是有值的，创建的时候就被赋予了一个默认值。所以，并没有什么空对象这样的东西。

然而如果程序能确保变量的当前值在被改写之前不会被读取，那么当前值到底是什么也就无关紧要了。抽象地讲，把这样的变量想成是“空的”就更容易理解其含义。

程序的含义比具体的编码方式更重要，这种思想是计算机科学家思维方式的重要部分。“抽象”这个词有时被滥用，以致于它的意思反而被忘记了。不过抽象的确是计算机科学的一个核心概念（在其它很多领域也是如此）。

“抽象”的一个较为综合的定义是：通过简化性的描述以回避不必要的细节并抓住关键行为从而为复杂系统建模的过程。

## 11.10 术语表

**编码-encode:** 通过在二者间建立映射，用一个集合的值来表示另一个集合的值。

**浅比较-shallow equality:** 引用的比较。比较结果为 `true` 就表示两个引用指向同一个对象。

**深比较-deep equality:** 值的比较。比较结果为 `true` 就表示两个引用指向的对象具有同样的值。

**抽象参数-abstract parameter:** 一组参数合起来作为一个参数。

**抽象-abstraction:** 对程序（或别的事物）进行高于具体代码（或事物的具体形式）层次解释的过程。

## 第 12 章 数组的对象

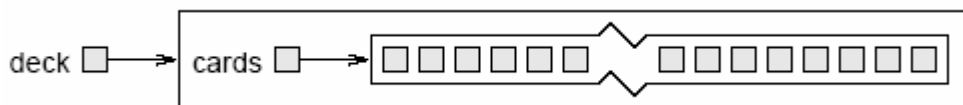
### 12.1 Deck 类

上一章用到了对象的数组，还提到对象可以包含数组作为实例变量。本章将创建一个新的对象 Deck，其中包含一个 Card 对象的数组作为实例变量。

类定义如下：

```
class Deck {  
    Card[] cards;  
    public Deck (int n) {  
        cards = new Card[n];  
    }  
}
```

实例变量名为 cards 以把 Deck 对象和它自身包含的 Card 数组区别开来。下面的状态图示意了一个 Deck 对象没有被分配 Card 对象时的情形：



和以往一样，构造器初始化实例变量，但这次要用 new 命令来创建纸牌的数组，不过并没有创建任何纸牌对象。所以还要写一个构造器以创建一个标准的牌套并把 52 张牌装满：

```
public Deck () {  
    cards = new Card[52];  
    int index = 0;  
    for (int suit = 0; suit <= 3; suit++) {  
        for (int rank = 1; rank <= 13; rank++) {  
            cards[index] = new Card (suit, rank);  
            index++;  
        }  
    }  
}
```

注意这个方法和 buildDeck 方法有多像，只不过是把语法改动了一下就成了构造器。使用 new 命令来调用：

```
Deck deck = new Deck ();
```

既然有了 Deck 类，把所有属于 Deck 的方法放进 Deck 类的定义更便于理解

代码的含义。看看以前写过的方法，其中 `printDeck`（11.7 节）还用的上。下面是为了用于 `Deck` 对象而改写的代码：

```
public static void printDeck (Deck deck) {  
    for (int i=0; i<deck.cards.length; i++) {  
        Card.printCard (deck.cards[i]);  
    }  
}
```

最明显的改变是参数型别，由 `Card[]` 改为 `Deck`。第二个改变是不再使用 `deck.length` 来获取数组的长度，因为这里的 `deck` 是一个 `Deck` 对象而不再是数组。`deck` 包含了一个数组，但它本身不是数组。因此只有写 `deck.cards.length` 来提取这个 `Deck` 对象的数组并取得其长度。

同样的道理，只有用 `deck.cards[i]` 而不是 `deck[i]` 来对数组的元素进行访问。最后一个改变是调用 `printCard` 方法时必须明确定义它的类 `Card`。

至于其它方法，是否应该包括在 `Card` 类还是 `Deck` 类中一时还不明显。例如 `findCard` 类取一个 `Card` 对象和 `Deck` 对象做参数，似乎放在两个类里都有道理，。

练习：把 `findCard` 方法放进 `Deck` 类，稍加改写以使第一个参数是一个 `Deck` 对象而不是一个 `Card` 对象的数组。

## 12.2 洗牌

绝大多数纸牌游戏都要洗牌，就是说把牌按任意顺序放。在 10.6 节讨论过如何生成随机数，不过要怎么利用随机数来洗牌并不明显。

一种办法就是对人洗牌的过程进行建模，即把一副牌分成两组，轮换从各组里取牌来混和在一起。人并不是完美的洗牌机，大约 7 次迭代能使牌序基本完全打乱。而计算机程序就更恼火了，每次洗牌都并非真的很随机。事实上，经过 8 次彻底的洗牌，会发现牌序又回到了开始之前的情况——对这个断言的讨论，请参考<http://www.wiskit.com/marilyn/craig.html>或者在网上搜索关键词“*perfect shuffle*”。

一个较好的洗牌算法是对牌套中的纸牌进行遍历，每次迭代挑两张纸牌交换位置。下面是这个算法的大体描述，为了给程序画一个“草图”，我把 Java 语句和英语混合使用，通常叫做伪代码：

```
for (int i=0; i<deck.length; i++) {  
    // choose a random number between i and deck.cards.length  
    // swap the ith card and the randomly-chosen card
```

```
}
```

使用伪代码的好处在于它常常能帮助我们弄清楚需要什么方法。这个例子中需要的是一个 `randomInt` 这样的方法，用来选择一个介于参数 `low` 和 `high` 之间的随机整数；还需要一个 `swapCards` 方法交换两张牌的索引号。看看 10.6 节就应该知道如何写 `randomInt` 方法，不过要小心生成超出边界值索引号的可能性。

相信你也能自己写出 `swapCards` 方法。唯一的技巧就是如何决定交换对纸牌对象的引用还是纸牌对象的内容。选哪种办法有什么关系吗？哪个更快？

剩余的实现留作练习。

## 12.3 排序

现在整副牌已经被洗的乱七八糟，要想办法把它们重新按顺序放好。具有讽刺意味的是，排序的算法和洗牌的算法非常相似。这种算法有时也叫做选择排序，反复遍历数组每次选出剩余最小的牌。

第一次迭代找到最小的牌并和第 0 位的牌交换，在第 *i* 次，找到第 *i* 位右边最小的牌和第 *i* 张牌交换。

下面是选择排序的伪代码：

```
for (int i=0; i<deck.length; i++) {  
    // find the lowest card at or to the right of i  
    // swap the ith card and the lowest card  
}
```

伪代码有助于设计**助手方法** (*helper method*)。 `swapCards` 方法前面已经实现，所以需要的仅仅是写一个新的方法 `findLowestCard`，这个方法取一个纸牌数组和开始查找的索引号为参数。

这里还是把实现留给读者作为练习。

## 12.4 subdeck 方法

如何表示一手牌或者其它一组牌呢？一种选择是创建一个名为 `Hand` 的新类，以扩展 `Deck` 类；另一种办法是下面要演示的，用一个碰巧不到 52 张纸牌的 `Deck` 对象来表示一手牌。

这里要用到一个方法 `subdeck`，取 `deck` 的一个子集和一个索引号区间做参数，返回包含指定那组牌的新的 `Deck` 型别对象：

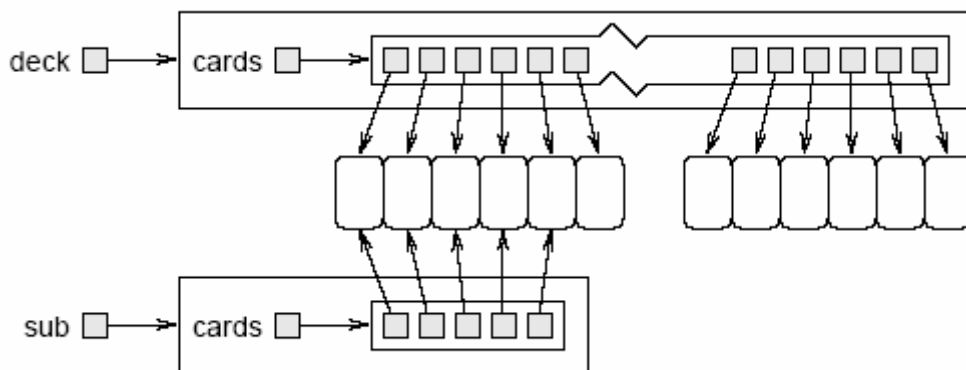


```
public static Deck subdeck (Deck deck, int low, int high) {
    Deck sub = new Deck (high-low+1);
    for (int i = 0; i<sub.cards.length; i++) {
        sub.cards[i] = deck.cards[low+i];
    }
    return sub;
}
```

deck 的子集长度是  $high-low+1$ ，因为边界的两张牌也包括在内。这类计算很容易混淆并引起“偏一 (*off-by-one*)”错误，画一张图常常能使我们避免犯这种错误。

因为向 `new` 命令提供了一个参数，所以调用的将是第一个构造器，仅仅分配数组而不分配任何纸牌。在 `for` 循环内，`subdeck` 获得从 `deck` 复制而来的引用。

下面是用参数 `low=3` 和 `high=7` 创建的 `subdeck` 对象的状态图。结果是一手和原来的 `deck` 共享的 5 张牌，这些牌也有了别名。



别名不是一件好事，因为一个 `subdeck` 的改变会影响到另一个 `subdeck`，真正打牌的时候不会有这样的事情。不过如果这些问题对象是刚性的，那么别名的危害就小了。这意味着没有任何理由可以改变一张牌的 `rank` 和 `suit`，创建的纸牌对象被当做刚性对象。于是 `Card` 对象有别名才是合理的选择。

## 12.5 洗牌和发牌

在 12.2 节写了洗牌算法的伪代码，假设有一个方法名为 `shuffleDeck` 取一个 `deck` 对象作为参数，于是可以这样来创建一个 `deck` 对象并洗牌：

```
Deck deck = new Deck ();
```

```
shuffleDeck (deck);
```

然后用 `subdeck` 方法来发几手牌：

```
Deck hand1 = subdeck (deck, 0, 4);  
Deck hand2 = subdeck (deck, 5, 9);  
Deck pack = subdeck (deck, 10, 51);
```

这段代码把前 5 张牌作为第一手发出，接下来的 5 张作为第二手，剩余的留在牌套里。

你有没有想过实际的纸牌游戏中是怎么发牌的，是不是轮转给每个玩家一次发一张牌呢？我想过了，但是我意识到对计算机程序来说没有必要这样做。轮转发牌的约定主要是为了降低洗牌不彻底的影响并且让发牌的人不容易作弊。两件事情都与计算机不相干。

这个例子提醒我们利用隐喻进行设计时容易遇到的陷阱：有时我们把不必要的限制强加给计算机，或者期望计算机具备它根本就没有的能力，都是因为我们不假思索的把隐喻扩展用在了不恰当的点上。要小心这种误导性的类比。

## 12.6 合并排序

12.3 节中考察了一个效率不是很高的简单排序算法，要对  $n$  个项目进行排序就要遍历数组  $n$  次，一次遍历的时间和  $n$  成正比，于是整个时间就和  $n^2$  成正比。

这一节中要粗略的描述一个更高效的算法叫**合并排序** (*mergesort*)。合并排序对  $n$  个项目进行排序所花的是的时间和  $n \log n$  成正比。这个公式看起来不显眼，但随着  $n$  增大， $n^2$  和  $n \log n$  之间的差别会达到惊人的地步。取几个  $n$  值试试就知道了。

合并排序的基本思路是：先把两个 subdeck 分别排好，再把二者合并为一个有序的 deck，就会十分容易（也很快）。

下面拿一副牌来试试：

1. 分成两组各约 10 张的牌墩，把两墩牌都按顺序排好放在面前，注意要最小的牌在上面且牌的正面向上。
2. 比较两墩牌的最上面一张并抽出较小的，把抽出的牌翻过来放在牌套里。
3. 重复第二步直到有一墩牌被拿完，然后把剩下一墩一起翻过来放到牌套里。

最后得到的就是一副按顺序排列的牌。下面是伪代码：

```
public static Deck merge (Deck d1, Deck d2) {  
    // create a new deck big enough for all the cards  
    Deck result = new Deck (d1.cards.length + d2.cards.length);
```

```
// use the index i to keep track of where we are in
// the first deck, and the index j for the second deck
int i = 0;
int j = 0;

// the index k traverses the result deck
for (int k = 0; k<result.cards.length; k++) {
    // if d1 is empty, d2 wins; if d2 is empty, d1 wins;
    // otherwise, compare the two cards
    // add the winner to the new deck
}
return result;
}
```

测试 merge 方法的最佳选择就是创建和分发一副牌，用 subdeck 方法形成两手牌，再用上一章的 sort 子程序来对两墩牌排序。最后把两墩牌传递给 merge 看能不能达到目的。

如果能再试试这个简单的实现——mergeSort：

```
public static Deck mergeSort (Deck deck) {
    // find the midpoint of the deck
    // divide the deck into two subdecks
    // sort the subdecks using sortDeck
    // merge the two halves and return the result
}
```

如果还能达到目的，真正有意思的事情才开始了。合并排序的神奇之处在于它是递归的。为什么要调用老的 sort 方法？为什么不试试写一个出色的“新方法” mergeSort 来用？

这不仅是个好主意，而且很有必要，要达到前面许诺的优秀性能就得如此。当然，要让程序正确地工作，就要记得在程序中考虑到递归的基本情形以免方法永远的递归下去。最简单的基本情形就是有 0 或 1 张牌的一个 subdeck，如果 mergesort 方法接收到这样的 subdeck 做为参数，就可以不用做任何修改直接返回，因为 0 或 1 张牌等于已经排好了序。

合并排序的递归版本应该这样写：

```
public static Deck mergeSort (Deck deck) {
    // if the deck is 0 or 1 cards, return it
```

```
// find the midpoint of the deck
// divide the deck into two subdecks
// sort the subdecks using mergesort
// merge the two halves and return the result
}
```

以前说过，对递归程序有两种思路：或者绞尽脑汁去想整个执行流程，或者“掩耳盗铃”。我故意构造这个例子就是为了鼓励你“掩耳盗铃”。

在用 `sortDeck` 对牌墩进行排序的时候，你没有强迫自己去跟踪执行流程对吧？你只是假定 `sortDeck` 能正常工作，因为你已经调试好了。那就行了，你用递归的 `mergeSort` 方法目的仅仅是为了替代另一种排序算法，没有理由要换一种办法读这个程序。

事实上，要保证递归调用能抵达基本情形不得不费点脑筋，但只要保证了这一条，递归版本的程序肯定能很好的工作。希望你顺利地把这些伪代码一一实现。

## 12.7 术语表

**伪代码-pseudocode:** 设计程序的时候把英语和 Java 混合起来写草稿。

**辅助方法-helper method:** 通常是一个很小的方法，本身没有多大用处，但能帮助其它更有用的方法。

## 第 13 章 面向对象编程

### 13.1 编程语言和风格

世界上有很多种编程语言，也有几乎同样多的编程风格（*style*，有时叫做 *paradigm*，**范式**）。这本书里已经出现的三种是：**过程式**，**函数式**和**面向对象**。虽然 Java 通常被认为是一种面向对象的语言，但用 Java 也可以写出任何风格的程序来。我在本书中演示过的风格比较接近过程式的风格。现有的 Java 程序和内建 Java 包都是三种风格混合写成的，但都比这本书里的程序更加面向对象。

要定义什么是面向对象编程并非易事，但可以列举出一些特征来：

- 对象的定义（类）常常对应着现实世界中的相关事物。例如 12.1 节中对 Deck 类的定义就一步步靠近着面向对象编程。

- 大多数方法都是对象方法（为对象而调用）而不是类方法。本书到目前为止写的都是类方法，本章会写对象方法。

- 和面向对象编程最相关的特性是**继承**（*inheritance*）。在本章后面部分会有所涉及。

目前面向对象编程已经广为接受，有人宣称在面向对象在方方面面都优于其它风格。希望通过我展示的众多风格能让你理解这种说法并有自己的评价。

### 13.2 对象和类的方法

Java 中有两种方法，类方法和对象方法。目前我们写过的方法都是类方法。类方法用关键字在第一行标识出来。没有 *static* 关键字的方法都是对象方法。

我们虽然没有写过对象方法，但却调用过。当“在一个对象上”调用方法时，就调用了一个对象方法。例如 `charAt` 方法和其它在 *String* 对象上调用的方法都是对象方法。

只要能写成类方法的方法也可以写成对象方法，反之亦然。只是有时用特定的一个比较自然。稍后就会清楚为什么，对象方法往往比相应的类方法短。

### 13.3 当前对象

当在一个对象上调用方法时，该对象就称为**当前对象**（*current object*）。在方法内部，可以不必指明对象的名称，直接使用当前对象的实例变量名。

还可以用关键字 *this* 指代当前对象。前面看到过 *this* 在构造器中的用法。

实际上可以把构造器看作一种特殊的对象方法。

## 13.4 复数

作为本章剩余部分的主要例子，我们将考虑一个复数的类定义。复数在数学和工程的许多领域都很有用，很多计算也是以复数计算的形式来进行的。复数由实部和虚部的和构成，通常写作  $x + yi$  的形式。 $x$  是实部， $y$  是虚部， $i$  代表-1 的平方根，所以  $i \cdot i = -1$ 。

下面是一个名为 `Complex` 的新对象型别的类定义：

```
class Complex
{
    // instance variables
    double real, imag;
    // constructor
    public Complex () {
        this.real = 0.0; this.imag = 0.0;
    }
    // constructor
    public Complex (double real, double imag) {
        this.real = real;
        this.imag = imag;
    }
}
```

其中没有什么新鲜的东西，实例变量是包含实部和虚部的双精度数，两个构造器也很寻常：

一个没有参数用于为实例变量赋默认值，另一个的参数和实例变量相一致。如前所述，关键字 `this` 用来指代将被初始化的变量。

在 `main` 方法或想创建一个复数对象的任何地方，可以选择先创建对象再设置实例变量也可以同时做这两件事情：

```
Complex x = new Complex ();
x.real = 1.0;
x.imag = 2.0;
Complex y = new Complex (3.0, 4.0);
```

## 13.5 复数函数

看看对复数可能执行的运算。复数的模定义是  $\text{abs}(x^2 + y^2)$ ，`abs` 方法是计算复数模的纯粹函数。写成类的定义是这样：

```
// 类方法
public static double abs (Complex c) {
    return Math.sqrt (c.real * c.real + c.imag * c.imag);
}
```

这个版本的 `abs` 方法计算作为参数接收的复数 `c` 的模。下一个版本是一个对象方法，计算当前对象（调用该方法的对象）的模，所以不需要任何参数：

```
// 对象方法
public double abs () {
    return Math.sqrt (real*real + imag*imag);
}
```

拿走了关键字 `static` 表明这是一个对象方法，同时也摘除了不必要的参数。在方法内部，不必指明对象就可以直接引用实例变量 `real` 和 `imag`。Java 知道引用的是当前对象的实例变量，如果想表达的更详细可以使用关键字 `this`：

```
// 对象方法
public double abs () {
    return Math.sqrt (this.real * this.real + this.imag *
                      this.imag);
}
```

不过这样的代码有点长而且意思不见得更清楚。要使用这个方法，就得用对象来调用它，例如：

```
Complex y = new Complex (3.0, 4.0);
double result = y.abs();
```

## 13.6 另一个复数函数

复数还可以做加法运算，把实部和虚部相加就得到两个复数的和。写成类方法如下：

```
public static Complex add (Complex a, Complex b) {
    return new Complex (a.real + b.real, a.imag + b.imag);
}
```

要调用这个方法，把两个操作数作为参数传递：

```
Complex sum = add (x, y);
```

写成对象方法就只要一个参数，和当前对象相加：

```
public Complex add (Complex b) {  
    return new Complex (real + b.real, imag + b.imag);  
}
```

这里又一次隐含地调用了当前对象的实例变量，不过对 **b** 对象的实例变量则必须用点号来显式地指出对象名 **b**。要使用这个方法，只要对一个操作数调用该方法并把另一个操作数作为参数传递。

```
Complex sum = x.add (y);
```

从这些例子中能看出当前对象 (**this**) 可以代替参数之一的位置，出于这个原因，当前对象有时也叫做**隐含参数** (*implicit parameter*)。

## 13.7 修饰方法

再来看一个例子，**conjugate** 方法，是用来把一个复数转换为它的共轭复数的修饰方法。复数  $x + yi$  的共轭复数是  $x - yi$ 。

写成类方法是这样：

```
public static void conjugate (Complex c) {  
    c.imag = -c.imag;  
}
```

对象方法版本：

```
public void conjugate () {  
    imag = -imag;  
}
```

现在你应该了解到把一种方法转换为另一种方法是一个机械的步骤。稍加练习后不费太多脑筋就可以做到，这样的好处在于不必再为写各种版本的方法伤神。应该具有对二者同样的熟练程度以便在写程序时选择适合相应运算的版本。

例如，加法运算应该写成类方法，因为这种运算对两个操作数是均等的，两个操作数作为参数同时出现则代码的含义比较明显。如果把一个操作数作为参数对一个操作数调用加法，就会显得有点另类。

另一方面，对一个对象的单一操作应该简明地写成对象方法（即使有附加的参数）。

## 13.8 toString 方法

每个对象型别都有一个名为 **toString** 的方法，用来生成一个表示该对象的字



符串。用 `print` 或 `println` 方法打印一个对象的时候，Java 就会调用该对象的 `toString` 方法。`toString` 的默认版本返回一个包含对象型和统一标识符的字符串（见 9.6 节）。定义一个新的对象型别时可以改写默认的做法，只要提供一个描述新的自定义行为的方法就可以了。

`Complex` 类的 `toString` 方法可以写成这样：

```
public String toString () {  
    return real + " + " + imag + "i";  
}
```

返回值的型别自然是 `String`，没有参数。

调用 `toString` 的方法也没有什么特别之处：

```
Complex x = new Complex (1.0, 2.0);
```

```
String s = x.toString ();
```

还可以在 `println` 方法中间调用：

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

这个例子的输出是 `1.0 + 2.0i`。

如果复数的虚部为负，这个版本的 `toString` 方法做的就不够好了。

作为练习，写一个更好的版本。

## 13.9 equals 方法

用 `==` 运算符比较两个对象的真正意图是想知道“二者是不是同一个对象”，就是说想知道两个对象是不是指向同一块内存区域。

对很多型别来说，上述对相等的定义并不适用。例如，两个复数的虚部和实部相等就可以说二者是相等的，却不一定是同一个对象。

定义新对象型别的时候，可以自行定义“相等”，一个名为 `equals` 的对象方法。举 `Complex` 类为例的话该方法可以写成：

```
public boolean equals (Complex b) {  
    return (real == b.real && imag == b.imag);  
}
```

根据约俗，`equals` 作为对象方法总是返回一个布尔值。

官方文档中在 `Object` 类定义部分有对 `equals` 方法的规格描述。应该把这些描述作为编写自定义方法的指南。

`equals` 方法实现了这样一种等价关系：

- 自反的：对任何引用值 `x` 都有 `x.equals(x)` 返回真。
- 对称的：如果对引用值 `x` 和 `y` 有 `x.equals(y)` 为真，则 `y.equals(x)` 也为真。

- 传递的：对任何引用值  $x$ 、 $y$  和  $z$ ，如果  $x.equals(y)$  为真且  $y.equals(z)$  为真，那么  $x.equals(z)$  也为真。

- 持久的：对任意确定的引用值  $x$  和  $y$ ，复合调用  $x.equals(y)$  总是为真或总是为假。

- 对任何引用值  $x$ ， $x.equals(null)$  都返回 `false`。

除了其中一条，前面定义的 `equals` 满足上述所有条件。

是哪一条？作为练习，请纠正。

## 13.10 调用对象方法

也许你已经在想了，没错，在一个对象方法中调用另一个对象方法是合法的。例如，为了规格化一个复数，将实部和虚部同时除以虚数的绝对值（模）。现在还看不出有什么意义，但这是确实有用处的。

现在来把 `normalize` 方法写成一个对象方法，并且是修饰方法。

```
public void normalize () {  
    double d = this.abs();  
    real = real/d;  
    imag = imag/d;  
}
```

第一行对当前对象调用 `abs` 方法调用得到复数的模。这里我特别强调了是当前对象，但以后慢慢的不会再作如此详细的区分。如果在一个方法内调用另一个对象方法，Java 会假定调用的是当前对象。

## 13.11 奇怪的错误

如果在一个类定义中同时有对象方法和类方法，这就容易引起混淆。通常组织类定义的办法是把构造器放在最开始，接下来是对象方法，最后才是类方法。

对象方法和类方法可以有一样的名称，只要二者参数的数目和型别不同。和其它类型的重载一样，Java 通过传递的参数选择相应版本。

现在知道了关键字 `static` 的含义，想必你已经想到 `main` 是一个类方法，这意味着当调用 `main` 时没有“当前对象”。

既然类方法中没有当前对象，使用 `this` 关键字就是错误的，如果试一试就会得到这样一条出错信息：“Undefined variable: this（未定义变量：this）”；同样地，没有提供对象名并使用点号也不能引用实例变量，试一试会得到这样的信息：“Can't make a static reference to nonstatic variable...”（不能对非静态变量作静态引

用)”。这条信息很不友好，因为用的是“非标准”语言。例如“非静态变量”实际的意思是“实例变量”，不过一旦你知道它的意思，就一目了然了。

## 13.12 继承

**继承** (*inheritance*) 是和面向对象编程最密切相关的语言特性，就是对已定义类（包括内建类）进行修改而写成新类。

这种特性最主要的优点是使得不必修改现存的类就可以为之增加新的方法或实例变量。这对内建类来说尤其管用，因为即使你想，也不能修改。

继承之所以称为“继承”是因为新类“遗传 (*inherit*)”了已有类的所有实例变量和方法，顺着这个隐喻的思路，已有的类常常称为**父类** (*parent class*)。

## 13.13 可拖动的矩形

作为继承的例子，下面将取一个已有的 `Rectangle` 类并使之变得“可拖动”。也就是说我们将创建一个名为 `DrawableRectangle` 的新类，它拥有 `Rectangle` 类的所有实例变量和方法，但增加了一个名为 `draw` 的方法，取一个 `Graphics` 对象做参数并绘制一个矩形。

类定义如下：

```
import java.awt.*;

class DrawableRectangle extends Rectangle {
    public void draw (Graphics g) {
        g.drawRect (x, y, width, height);
    }
}
```

没错，整个类定义确实就是这样。第一行导入 `java.awt` 包，其中定义了 `Rectangle` 类和 `Graphics` 类。

接下来的一行指出 `DrawableRectangle` 继承自 `Rectangle`。关键字 `extends` 用来标识所继承的类，即父类。

其余部分是 `draw` 方法的定义，引用了实例变量 `x`、`y`、`width` 和 `height`。引用没有出现在类定义中的实例变量似乎不大正常，但请记住这是由父类继承而来的。

要创建一个 `DrawableRectangle` 对象，可以用下面的语法：

```
public static void draw (Graphics g, int x, int y,
                        int width, int height) {
```

```
DrawableRectangle dr = new DrawableRectangle ();  
dr.x = 10;      dr.y = 10;  
dr.width = 200; dr.height = 200;  
dr.draw (g);  
}
```

draw 的参数是一个 Graphics 对象和要绘制区域的方框（而不是矩形的座标）。

对一个没有构造器的类用 new 命令也有点奇怪，但 DrawableRectangle 继承了父类的默认构造器，所以这里没有问题。

设置 dr 的实例变量和对其调用方法的步骤和以往一样。当调用 draw 方法的时候，Java 就会调用在 DrawableRectangle 中定义的方法，如果对 dr 调用 grow 或其它 Rectangle 的方法，Java 也会知道用父类中定义的方法。

## 13.14 类的层次

Java 中的类都继承自其某个类。最基本的类是 Object 类。Object 类没有实例变量，不过它提供的方法中就包括 equals 和 toString。

许多类对 Object 类进行了扩展，这包括我们写过的几乎所有类和很多内建类，比如 Rectangle。默认一切没有显式指明父类的类都继承自 Object 类。

有的继承链很长，比如附录 D.6 中，Slate 类继承自 Frame 类，Frame 继承自 Window，Window 继承自 Container，Container 继承自 Component，Component 继承自 Object。无论这个链有多长，Object 是所有类的最终父类。

Java 中的所有类可以用一个“家族树”组织成一个类层次。Object 通常出现在最顶端，所有的“孩子类”在下方。以 Frame 类的文档为例，Frame 的一族属于这个大类的一部分。

## 13.15 面向对象设计

继承是一种强大的特性。一些程序虽然可以不使用这个特性就完成，但是使用继承重写却更简洁简单。同时，继承还促进了代码复用，因为可以不用改动内建类就定制其行为。

另一方面，继承可能使得程序难以阅读，因为调用一个方法的时候到哪里找定义不是那么明显。例如，对一个 Slate 对象调用 getBounds，如果想找 getBounds 方法的文档来看看，结果就会发现 getBounds 被定义在 Slate 的父类的父类的父类的父类中。

当然，许多用继承能办到的事情不用继承也能很漂亮（甚至更漂亮）地完成。

## 13.16 术语表

**对象方法-object method:** 对象调用的方法, 作用于当前对象(英语中的“the current object”), Java 中用关键字 `this` 指代。对象方法没有关键字 `static`。

**类方法-class method:** 带有 `static` 关键字的方法。类方法不为对象调用, 也没有当前对象。

**当前对象-current object:** 对象方法调用的对象, 在方法内部, 当前对象用 `this` 指代。(译注: 参考对象方法的定义。)

**this:** 指代当前对象的关键字。

**隐式-implicit:** 没有说出来的或者暗示的内容。在对象方法内部, 可以隐式地引用实例变量(不指明对象名)。

**显式-explicit:** 完整地写出来。在类方法内部, 所有对实例变量的引用都必须是显式的。

## 第 14 章 链表

### 14.1 对象内的引用

上一章提到对象的实例变量可以是数组，也可以是对象。

更有意思的是对象还可以包含对另一个同型别对象的引用。一种常见的数据结构——**表**（*list*）就有这种特性。

表由**节点**（*node*）组成，每个节点包含一个对下一节点的引用。另外，每个节点通常还包含一个数据单元叫做 *cargo*（货物之意）。接下来的第一个例子中的 *cargo* 是一个整数，不过稍后会写一个可以包含任何型别对象的更一般的表。

### 14.2 Node 类

和以往一样，写一个新类的时候从实例变量下手，然后是一两个构造器和 *toString* 方法，这样就能测试基本的创建机制并显示新的型别。

```
public class Node {  
    int cargo;  
    Node next;  
    public Node () {  
        cargo = 0;  
        next = null;  
    }  
    public Node (int cargo, Node next) {  
        this.cargo = cargo;  
        this.next = next;  
    }  
    public String toString () {  
        return cargo + "  
    }  
}
```

实例变量的声明习惯地放在类标识之后，接着按部就班写其它部分。表达式 *cargo + ""* 虽然有点笨拙但能简洁地把一个整数转换为字符串。

在 main 方法中测试目前为止的实现：

```
Node node = new Node (1, null);  
System.out.println (node);
```

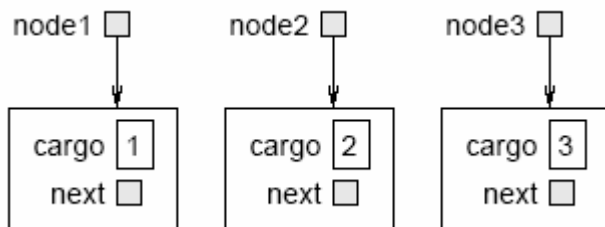
结果很简单：

1

为了更有意思，需要有多节点的表。

```
Node node1 = new Node (1, null);  
Node node2 = new Node (2, null);  
Node node3 = new Node (3, null);
```

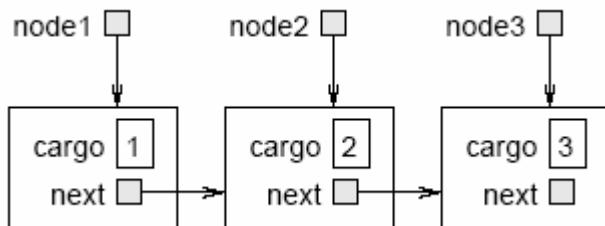
这段代码创建了三节点，但还没有得到表，因为节点还没有被链接 (*linked*)，状态图如下：



要把节点链接起来，就要让第一个节点指向第二个节点、第二个节点指向第三个节点。

```
node1.next = node2;  
node2.next = node3;  
node3.next = null;
```

第三个节点的引用是 null，说明到了表的末端，现在状态图是这样了：



现在知道了如何创建节点并链接起来成为链表，但对这样做的目的可能还不清楚。

### 14.3 作为容器的链表

链表有用的地方就在于提供了一种把多个对象集装在一个实体里的能力，有时这叫做容器。以链表为例，第一个节点就是对整个链表的引用。

如果要把链表作为参数传递，只需要传递对第一个节点的引用。例如 `printList` 方法取一个节点作为参数，从链表的头部起打印每一个节点直到表尾（以 `null` 为标识）。

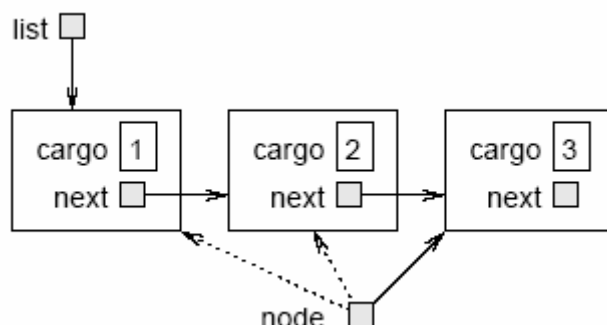
```
public static void printList (Node list) {  
    Node node = list;  
    while (node != null) {  
        System.out.print (node);  
        node = node.next;  
    }  
    System.out.println ();  
}
```

要调用这个方法只要把第一个节点作为引用传递：

```
printList (node1);
```

在 `printList` 内部有一个对链表第一个节点的引用，但没有变量指向其它节点，只有用每个节点的 `next` 值到达下一个节点。

下图示意了链表的值和 `node` 持有的值：



这样在链表中移动的方式叫做**遍历**（*traversal*），和在数组元素中移动的模式相似。常常用 `node` 这样的循环变量来连续地指向链表的每个节点。

这个方法的输出是：

123

一般说来，链表被打印在一个括号里并在每个元素之间加一个逗号，譬如 `(1, 2, 3)`。

作为练习，修改 `printList` 以生成这样的格式。

第二个练习，用 `for` 循环代替 `while` 循环改写 `printList`。

## 14.4 链表与递归

递归和链表常常如影随形。例如这个逆序打印链表的递归算法：



1.把链表划分成两片：第一个节点（叫做表头）和其余部分（叫做表尾）。

2.逆序打印表尾。

3.打印表头。

如果打印方式已知，那么显然，第二步就是递归调用。我们只要假定递归调用能顺利工作——“掩耳盗铃”——就可以说服自己相信这个算法是正确的。

我们需要的仅仅是递归的基本情形，还有就是一个确证任何链表最后都能抵达基本情形的办法。基本情形的一个自然而然的选择是只有一个元素的链表，不过更好的选择是空链表（用 null 表示）：

```
public static void printBackward (Node list) {  
    if (list == null) return;  
    Node head = list;  
    Node tail = list.next;  
    printBackward (tail);  
    System.out.print (head);  
}
```

第一行处理基本情形——办法是什么也不做。接下来的两行将链表分成 head 和 tail 两部分。最后两行打印链表。

调用这个方法和调用 printList 完全一样：

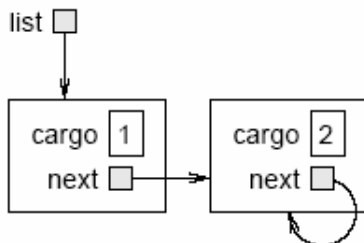
```
printBackward (node1);
```

结果是逆序后的链表。

是否可以证明这个方法总是会终止呢？换句话说，它能抵达基本情形吗？实际上答案是：不行。有的链表会使这个方法失灵。

## 14.5 无穷链表

没有办法防止节点反过来指向链表中更靠前的节点，包括指向自身。例如下图就显示了有两个节点的链表，其中一个节点指向它自身。



如果对这个链表调用 printList 方法，循环将一直进行下去，如果调用 printBackward 方法则递归也永远不会结束。这种行为使无穷链表难以控制。

然而有时链表又很有用。例如，把一个数表示成数字的链表，用无穷链表

来表示无限循环小数。

不管怎样，能否保证 `printList` 方法和 `printBackward` 方法终止都是有疑问的。最好的选择是用假设句，“如果链表不含循环，那么上述方法就会终止。”这种要求叫做前提 (*preconditio*)，它强制性地约束某个参数，并描述当约束条件满足时方法的行为。我们将很快看到更多例子。

## 14.6 基本歧义定理

下面是 `printBackward` 方法中最可能引起怀疑的部分：

```
Node head = list;
Node tail = list.next;
```

初次赋值后，`head` 和 `list` 的型别和值都相同，那为什么要创建两个变量呢？

理由是两个变量扮演的角色不一样。`head` 作为对单个节点的引用，`list` 则是对链表中第一个节点的引用。“角色”并非程序中的概念，而是存在于程序员的头脑中。

第二个赋值语句创建了对第二个节点的新引用，但 `tail` 代表的是链表，所以 `head` 和 `tail` 的角色不一样。

歧义并非一无是处，只不过会把程序弄得比较难读。使用 `node` 和 `list` 这样的英语单词做变量名是为了更好的在程序中使用变量，有时还会另外创造变量来消除歧义。

写 `printBackward` 方法本来可以不用 `head` 和 `tai`，但是这只会让程序难以理解：

```
public static void printBackward (Node list) {
    if (list == null) return;
    printBackward (list.next);
    System.out.print (list);
}
```

注意两个函数调用，要记得区分 `printBackward` 把参数当做链表，而 `print` 把参数当做一个对象

请小心这个基本歧义定理：

指向节点的变量可能把节点当做一个对象，也可能当做链表的第一个节点。

## 14.7 节点的方法

你可能会问 `printList` 和 `printBackward` 方法为什么是类方法。前面说过，类方法能做的事情对象方法也能做，问题在于哪种方法更流畅。

这里有一个选择类方法的恰当理由：把 `null` 当做参数传递给类方法是合法的，而对一个 `null` 对象调用对象方法是非法的。

```
Node node = null;

printList (node); // legal

node.printList (); // NullPointerException
```

这个限制使得用流畅的面向对象风格写链表操纵代码变得很难，不过后面我们会讨论一种绕过这个限制的办法。

## 14.8 修改链表

修改链表最显而易见办法就是改变节点的 `cargo` 中的内容，不过更有意思的操作是增加、删除还有重排节点。

以删除链表第二个节点并返回被删掉节点的引用的方法为例。

```
public static Node removeSecond (Node list) {
    Node first = list;
    Node second = list.next;
    // 使第一个节点指向第三个
    first.next = second.next;
    // 把第二个节点和链表的其余部分分离
    second.next = null;
    return second;
}
```

这里再一次使用了循环变量以增加代码的可读性。下面是这个方法的用法。

```
printList (node1);

Node removed = removeSecond (node1);

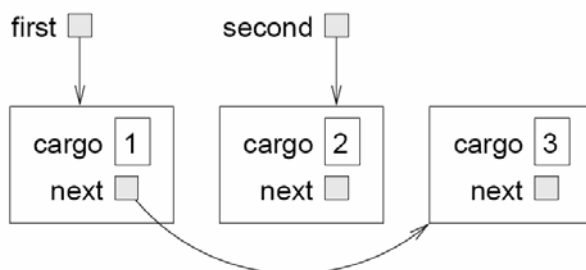
printList (removed);

printList (node1);
```

输出为

(1, 2, 3)	the original list
(2)	the removed node
(1, 3)	the modified list

下图示意了这步操作的效果。



如果调用这个方法并传递一个仅有一个元素的链表(独链表)会怎么样呢? 传递一个空链表又会怎么样呢? 要用这个方法有前提吗?

## 14.9 包装方法和助手方法

对有的链表操作, 将其分为两个方法完成会更好。

例如, 要把一个链表按约定俗成的格式逆向打印出来(象这样: (3,2, 1), 数字代表元素的索引号), 可以用 `printBackwards` 方法打印 3 和 2, 但要一个单独的方法来打印括号和第一个节点, 把这个方法称为 `printBackwardNicely`。

```

public static void printBackwardNicely (Node list) {
    System.out.print "(";
    if (list != null) {
        Node head = list;
        Node tail = list.next;
        printBackward (tail);
        System.out.print (head);
    }
    System.out.println (")");
}

```

同样, 最好还是用特殊条件如空链表或独链表对这个方法进行检查。

在程序其它地方直接调用 `printBackwardNicely` 方法, 它会自动调用 `printBackward` 方法。这种情况 `printBackwardNicely` 方法就是包装方法, `printBackward` 方法则是它的助手方法。

## 14.10 IntList 类

用前面的方式实现的链表有一个不易察觉的细微问题。下面将原因和结果反过来叙述: 我先给出一个替换版本然后解释这个版本解决了什么问题。

首先创建一个名为 `IntList` 的新类。实例变量是一个包含链表长度的整数和对链表第一个节点的引用。`IntList` 对象的作用是借以操纵链表中 `Node` 对象。

```
public class IntList {  
    int length;  
    Node head;  
    public IntList () {  
        length = 0;  
        head = null;  
    }  
}
```

`IntList` 类的一个好处是它自然地留出了位置，以供放置 `printBackwardNicely` 这样的包装方法作为 `IntList` 类中的对象方法。

```
public void printBackward () {  
    System.out.print "(";  
    if (head != null) {  
        Node tail = head.next;  
        Node.printBackward (tail);  
        System.out.print (head);  
    }  
    System.out.println (")");  
}
```

为了故意引起混淆，把 `printBackwardNicely` 换了一个名称。现在有了两个名为 `printBackward` 的方法：一个在 `Node` 类中（作为助手方法），另一个在 `IntList` 类中（作为包装方法）。为了让包装方法顺利调用助手方法，必须显式地指明类（`Node.printBackward`）。

这样一来，`IntList` 类的优点之一就是提供了一个很好的位置来放包装类。另一个优点就是使增加或删除链表的元素变得更容易。例如 `addFirst` 是 `IntList` 的一个对象方法，取一个整数做参数并把它放在链表的开头。

```
public void addFirst (int i) {  
    Node node = new Node (i, head);  
    head = node;  
    length++;  
}
```

和前面一样，应该用特殊情况来检查这段代码。比如说，如果链表一开始

是空的会怎么样？

## 14.11 恒量

有的链表是“格式良好的”，有的则不是。举个例，如果一个链表包含循环，就会导致我们写好的很多方法崩溃。所以要保证链表没有循环；还有一个要求就是 `IntList` 对象的长度值应该等于链表中节点的实际数目。

这样的前提条件叫做**恒量** (*invariant*)，因为在理想情况下对每个对象应该总是成立。为对象说明恒量是一个很好的编程习惯，因为这使得验证代码和数据结构的正确性变得容易，而且也易于诊断错误。

容易引起困惑的一点就是，恒量有时会被违反。例如，在 `addFirst` 对象中间增加一个节点，在没有增加这个节点之前，恒量被违背了。这类违背是可以接受的，实际上要修改对象就免不了会暂时违反恒量。一般说来只要求每个违背了恒量的方法必须调整恒量。

如果对代码进行扩展时违背了恒量，最重要的就是在注释中说清楚，以免执行的操作依赖于恒量。

## 14.12 术语表

**链表-list**: 实现作为一组链接起来的节点的容器的数据结构。

**节点-node**: 链表的一个元素，通常是一个对象，包含了对另一个同型别对象的引用。

**cargo**: 节点中包含的数据项目。

**链接-link**: 对象中嵌入的对象引用。

**通用数据结构-generic data structure**: 能包含任何型别数据的数据结构。

**前提-precondition**: 要让一个对象正确工作就必须满足的断言。

**恒量-invariant**: 对一个对象总是为真的断言（除了对象被修改的片刻）。

**包装方法-wrapper method**: 在调用者和助手方法之间扮演中介的方法，通常提供了比助手方法更清楚的接口。

## 第 15 章 堆栈

### 15.1 抽象数据类型

目前为止学习过的数据类型都是实实在在的，因为它们都有具体实现的完整说明。例如 `Card` 类用两个整数来表示一张纸牌。在前面讨论时候，不止一种途径可以表示纸牌，还有很多的候选实现。

一种抽象数据类型 (*abstract data type*, 简称 *ADT*) 规定了一组操作 (或方法) 和这组操作的语义 (用来做什么), 但是没有规定操作的实现途径。这就是抽象的来源。

抽象数据类型有什么用?

- 简化了说明算法的工作，因为在写算法的时候不必同时考虑操作是如何执行的。
- 能帮助写出可用于任何实现的算法，因为一个 *ADT* 通常有很多实现途径。
- 著名的 *ADT* 例如本章中要讲的堆栈 *ADT* 通常在标准库中已被实现，这就使之达到了一次编写而被许多程序员使用的目的。
- *ADT* 的操作提供了一种用来规定和讨论算法的通用高阶语言。

谈论 *ADT* 通常把使用 *ADT* 的代码 (称为 *client*, 委托者) 和实现 *ADT* 的代码 (提供了一组标准的服务所以叫做 *provider*, 供给者) 区别开来。

### 15.2 堆栈 ADT

这一章将考察一种常用的 *ADT*——堆栈 (*stack*)。堆栈是一个容器，这意味着它是一个包含多个元素的数据结构。我们学过的容器有数组和链表。

如前所述，*ADT* 定义了一组操作。堆栈能执行下面这组操作：

构造器：创建一个新的空堆栈。

`push`：向堆栈增加新项。

`pop`：移出并返回堆栈中的一项，返回的堆栈总是最后增加的项。

`isEmpty`：检查堆栈是否为空。

堆栈常常被叫做“先进先出 (*last in, first out*)”或 *LIFO* 数据结构，因为最后增加的项最先被移出。

## 15.3 Java 中的 Stack 对象

Java 提供了叫做 `Stack` 的内建对象型别来实现 *Stack ADT*。要让二者 (ADT 和 Java 实现) 统一, 得作出一些努力。要用 `Stack` 类先得从 `java.util` 导入。

构造一个新堆栈的语法是

```
Stack stack = new Stack ();
```

初始堆栈是空的, 这可以用 `isEmpty` 方法来确认, 返回一个布尔值:

```
System.out.println (stack.isEmpty ());
```

堆栈是一种通用的数据结构, 就是说可以向其中添加任何型别的项。然而在 Java 的实现中只能添加对象型别。

第一个例子中将使用上一章定义的 `Node` 对象型别。

首先来创建并打印一个很短的链表:

```
IntList list = new IntList ();  
list.addFirst (3);  
list.addFirst (2);  
list.addFirst (1);  
list.print ();
```

输出是 (1, 2, 3)。用 `push` 方法把 `Node` 对象压入 (*put onto*) 堆栈:

```
stack.push (list.head);
```

接下来用一个循环遍历链表并把所有的节点压入堆栈:

```
for (Node node = list.head; node != null; node = node.next) {  
    stack.push (node);  
}
```

可以用 `pop` 方法从堆栈中移出一个元素:

```
Object obj = stack.pop ();
```

`pop` 返回的型别是 `Object`! 这是因为堆栈的实现并不是真的知道包含的对象是什么型别。压入的时候 `Node` 对象自动被转换为 `Object` 型别。所以当从堆栈中取回节点的时候就得转换回到 `Node` 型别:

```
Node node = (Node) obj;  
System.out.println (node);
```

不幸的是这就要辛苦程序员跟踪堆栈中对象的型别并在其被移出的时候转换回正确的型别。如果试图把一个对象转换为错误的型别, 就会引起一个 `ClassCastException` 异常。

下面的循环常用来把堆栈中的元素全部弹出, 当堆栈清空后退出:



```
while (!stack.isEmpty ()) {  
    Node node = (Node) stack.pop ();  
    System.out.print (node + " ");  
}
```

输出是 3 2 1。发现没有，我们刚刚用一个堆栈实现了将链表元素逆序打印！虽然输出不是打印链表的标准格式，但使用堆栈来实现却简单的多。

应该把这段代码和上一章 `printBackward` 方法的实现做个比较。这是递归版本的 `printBackward` 方法和堆栈算法的一个自然而然的对比。区别是 `printBackward` 方法在遍历链表时用运行时的堆栈来跟踪节点，然后在递归的回溯过程中把节点一一打印出来。堆栈算法做的是同一件事情，只不过用一个 `Stack` 对象代替了运行时堆栈。

## 15.4 包装类

对 Java 中的每一个基本型别而言，都有一个内建的对象型别叫做**包装类** (*wrapper class*)。例如 `int` 型别的包装类是 `Integer` 类，`double` 型别的包装类是 `Double`。

包装类很有用，原因如下：

- 可以实体化包装类并创建一个包含基本型别的对象。换句话说，可以把一个基本型别的值包装进一个对象，当调用一个需要对象型别参数的方法时这非常有用。
- 每个包装类包含一些能转换型别的有用方法和一些特殊值(比如该型别的最大或最小值)。

## 15.5 创建包装对象

创建包装对象最直接的途径是用构造器：

```
Integer i = new Integer (17);  
Double d = new Double (3.14159);  
Character c = new Character ('b');
```

技术上讲 `String` 不是包装类，因为没有对应的基本型别，但创建一个 `String` 对象的语法却是一样的：

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

不过一般不会用到 `String` 对象的构造器，因为用赋值语句的效果一样：

```
String s = "fred";
```

## 15.6 再谈创建包装对象

有的包装类拥有的第二个构造器取一个 `String` 对象为参数并将其转换为相应型别。例如：

```
Integer i = new Integer ("17");  
Double d = new Double ("3.14159");
```

这个转换过程并非很牢靠。例如 `String` 对象的格式不正确就会导致一个 `NumberFormatException` 异常。字符串中任何非数字字符包括空格都会导致转换失败。

```
Integer i = new Integer ("17.1"); // 错误!!  
Double d = new Double ("3.1459 "); // 错误!!
```

所以在转换字符的时候要检查格式。

## 15.7 取值

Java 知道如何打印包装对象，所以提取一个值最简单的办法就是打印对象：

```
Integer i = new Integer (17);  
Double d = new Double (3.14159);  
System.out.println (i);  
System.out.println (d);
```

又或者，用 `toString` 方法把包装对象的内容转换为字符串：

```
String istring = i.toString();  
String dstring = d.toString();
```

最后，如果想从对象中提取基本型别值，那么每个包装类中都有一个对象方法来完成这个工作：

```
int iprim = i.intValue ();  
double dprim = d.doubleValue ();
```

还有其它把包装对象转换为基本型别的方法，查看包装类的文档能找到详细情况。

## 15.8 包装类中的有用方法

前面已经提到，包装类包含了属于各种型别的有用方法。例如，`Character` 类包含了很多方法，诸如把字符转换为大写和小写的方法，还有检查一个字符

是数字、字母还是符号的方法。

`String` 类也包含了转换大小写的方法。不过要留心，这些方法都是函数方法而不是修饰方法（见 7.10 节）。

再举一个例子。`Integer` 类包含了解释和打印不同进制整数的方法。如果有一个字符串包含了一个六进制的数，可以用 `parseInt` 方法将其转换为 10 进制数。

```
String base6 = "12345";  
int base10 = Integer.parseInt (base6, 6);  
System.out.println (base10);
```

`parseInt` 是一个类方法，所以用点号把类名和方法名连起来调用。

6 进制可能没什么用处，但十六进制和八进制在计算机科学的相关领域却很常用。

## 15.9 后缀表达式

在大多数编程语言里，数学表达式中的运算符被写在操作数之间，譬如  $1+2$ 。这种格式叫**中缀** (*infix*)。另外一种候选格式是**后缀** (*postfix*)，运算符跟在操作数后面，譬如  $1\ 2+$ 。

后缀有用的地方在于：通过堆栈可以很自然地计算后缀表达式。

●从表达式开头起，每次取一项（运算符或操作数）。

- 如果取得项是操作数则压入堆栈。
- 如果取得项是运算符则从堆栈中弹出两个操作数，对二者执行运算，然后把结果压回堆栈。

●到达表达式结尾的时候，堆栈中应该只有一个操作数，这个操作数就是结果。

作为练习，请按这种算法计算  $1\ 2 + 3 *$ 。

这个例子可以说明后缀的优点之一：不必用括号来控制运算次序。用中缀要得到同样的结果就得写成  $(1 + 2) * 3$ 。作为练习，写一个和  $1 + 2 * 3$  等价的中缀表达式。

## 15.10 解析

为了实现上一节的算法，需要遍历一个字符串并把它分为操作数和运算符。这个过程就是一个解析的例子，而结果——字符串中独立的片断——叫做**符号** (*token*)。

Java 提供了内建类 `StringTokenizer` 来解析字符串并将其打散为符号。使用这个类要从 `java.util` 包中导入。

最简单的形式下，`StringTokenizer` 用空格来分隔符号。标记分隔边界的字符叫做**分界符** (*delimiter*)。

创建 `StringTokenizer` 对象，然后把要解析的字符串作为参数传递。

```
StringTokenizer st = new StringTokenizer ("Here are four tokens.");
```

下面的循环常常用来提取 `StringTokenizer` 对象中的符号。

```
while (st.hasMoreTokens ()) {  
    System.out.println (st.nextToken());  
}
```

输出是：

Here

are

four

tokens

要解析表达式，可以选择是否指定一个用做分界符的附加字符：

```
StringTokenizer st = new StringTokenizer ("11 22+33*", " +-*/*");
```

第二个字符串做参数，其中的所有字符都将被视为分界符。现在的输出是：

11

22

33

这样虽然成功的提取了所有的操作数但是却丢失了运算符。走运的是 `StringTokenizers` 中还有一个选项。

```
StringTokenizer st  
    = new StringTokenizer ("11 22+33*", " +-*/*", true);
```

第三个参数等于说，“是的，把分界符当做符号。”现在结果变成了

11

22

+

33

\*

这串符号正是我们做上一节最后一个练习要用到的。

## 15.11 实现 ADT

ADT 最根本的目标之一就是把供给者（实现 ADT 的代码）和委托者（使用 ADT 的代码）的职责分开来。供给者只要担心实现是否正确——根据 ADT 的规格——和如何使用。

反过来，委托者则假定 ADT 的实现是正确的而不必担心细节。我们使用 Java 内建类的时候，就在舒服地作为委托者考虑问题。

而且供给者实现的 ADT 要由委托者来测试，这个时候程序员必须小心思考自己正在扮演的角色。

解下来的几节我们换档来看看用数组实现堆栈 ADT 的一种方式。首先要扮演一个供给者编写者。

## 15.12 堆栈 ADT 的数组实现

这个实现的实例变量是一个对象组成的数组，其中包含了堆栈的项和一个整数索引号用来跟踪数组中下一处可用空间。初始状态下数组为空索引为 0。

要向堆栈中增加一个元素（压栈），就把它的引用拷贝到堆栈中并将索引号自增 1。要移出一个元素（出栈）先将索引自减 1 再把元素拷出。

类定义如下：

```
public class Stack {  
    Object[] array;  
    int index;  
    public Stack () {  
        this.array = new Object[128];  
        this.index = 0;  
    }  
}
```

和以前说过的一样，一旦确定实例变量，编写构造器就是一个机械的过程。上面的代码默认数组大小是 128 项，后面再考虑如何更好的解决默认大小问题。

检查空堆栈最简单不过：

```
public boolean isEmpty () {  
    return index == 0;  
}
```

不过要注意的是，堆栈的元素数目和数组的大小并不一样。初始化的数组大小是 128，而堆栈的元素数目是 0。

压栈和出栈的实现自然要参照规格说明。

```
public void push (Object item) {
    array[index] = item;
    index++;
}

public Object pop () {
    index--;
    return array[index];
}
```

测试这个方法，要利用前面创建堆栈的委托者。只需要将导入 *java.util* 包的语句行注释掉就行了。接下来，不再使用 *java.util* 包中的堆栈实现而用我们自己刚刚写的供给者。

如果一切按计划进行，程序不会很大改变。这里又说明了使用 ADT 的一个强大之处：修改实现而不必改变委托者。

### 15.13 缩放数组

上面的实现有一个弱点，创建堆栈时选了一个任意的大小。如果用户压栈的项目数超过 128，就会导致一个 *ArrayIndexOutOfBoundsException* 异常。

候补办法是让委托者指定数组大小。这样虽然缓解了问题，但要求委托者提前知道需要的项目数，而这也不是总能满足的。

更好一些的解决方案是检查数组是否已满并在必要时扩大数组。既然不知道需要的数组是多大，那么一种明智的策略就是开始用一个很小的数组，每次溢出就翻一番。

push 的改进版本：

```
public void push (Object item) {
    if (full ()) resize ();
    // at this point we can prove that index < array.length
    array[index] = item;
    index++;
}
```

在往数组压入新项之前，先检查数组是否已满，如果满了就调用 *resize* 方法。在 *if* 语句之后，可能是：（1）数组中还有空间；（2）数组经过缩放，还

有空间。

如果 `full` 方法和 `resize` 方法都是正确的，就可以保证数组长度大于堆栈索引 (`index < array.length`)，这样下一个语句就不会导致异常。

现在该实现 `full` 和 `resize` 了。

```
private boolean full () {
    return index == array.length;
}

private void resize () {
    Object[] newArray = new Object[array.length * 2];
    // we assume that the old array is full
    for (int i=0; i<array.length; i++) {
        newArray[i] = array[i];
    }
    array = newArray;
}
```

两个方法都被声明为 `private`，意思就是说不能从其它类调用而只能从当前类中调用。这样做是复合使用 ADT 原则的：首先，委托者没有理由用到这两个函数方法；其次这样做达到了把供给者和委托者的边界划清的目的。

实现 `full` 很简单，仅仅是检查索引号是否越过了有效的编号范围。

`resize` 的实现也很直观，因为假定数组已满。换句话说，这个假定是方法的前提。很容易看到这个前提已被满足，因为只有 `full` 返回 `true` 的时候 `resize` 才会被调用，就是说 `resize` 被调用时数组一定是满的。

在 `resize` 最后用新数组替代了老数组（老的将被当做垃圾回收）。新的 `array.length` 是原来的两倍，索引号没有变，所以 `index < array.length` 肯定成立。“`resize` 方法完成后数组长度一定大于索引号”——这个命题就是前提（数组已满）为真得到的推断（*postcondition*）。

前提、推断还有恒量，这三个工具对分析程序和证明其正确性非常有用。在这个例子中，我示范了一种有助于分析程序的编程风格和一种有助于证明程序正确性的文档风格。

## 15.14 术语表

**抽象数据类型-abstract data type (ADT):** 一种数据类型（通常是对象的容器），定义了一组操作，可以有多种途径实现。

**委托者-client:** 使用 ADT 的程序（或写这种程序的人）。

**供给者-provider:** 实现 ADT 的代码（或写这些代码的人）。

**包装类-wrapper class:** Java 的一种类，比如 Double 和 Integer 类，提供了包含基本型别的对象和对基本型别执行运算的方法。

**private:** Java 关键字，用来指明一个方法或实例变量不能从当前类定义之外的地方访问。

**中缀-infix:** 写数学表达式的时候把运算符放在两个操作数中间的方式。

**后缀-postfix:** 写数学表达式的时候把运算符放在操作数后面的方式。

**解析-parse:** 读取一个字符串并分析它的语法结构。

**符号-token:** 一组被当做一个单元的字符，用来进行解析，像是自然语言中的词。

**分界符-delimiter:** 用来分隔符号的字符，像是自然语言中的标点。

**命题-predicate:** 不为真便为假的数学语句。

**推断-postcondition:** 在方法最后必定为真的命题（开始时前提必须为真才能执行该方法）。



## 第 16 章 队列和优先队列

这一章讨论两个 ADT：队列 (*Queue*) 和优先队列 (*Priority Queue*)。实际生活中的队列可以是一列等候某种服务的顾客。大多数时候，队伍中的第一位顾客是先得到服务。当然也有例外，比如在机场，紧急起飞班机的乘客会被从队列中间带出来，又或者在超市里一位礼貌的顾客会让另一位拿着几件商品的顾客先刷卡。

决定谁最先得到接下来的服务的规则叫做**排队规则** (*queueing discipline*)。最简单的排队规则是叫做“**先进先出** (*first-in-first-out*)”，一般缩写为 *FIFO*。最普通的排队规则是**优先排队** (*priority queueing*)，每个顾客拥有一个优先级，根据到达的先后次序，优先级最高的顾客第一个得到服务。说优先排队是最普通的排队规则是因为优先级可以基于任何事由：航班起飞的时间；顾客手里的货物多少；或者顾客的重要性等等。当然，并非所有的排队规则都是“公平”的，而且公平的标准要视情况而定。

队列 ADT 和优先级队列 ADT 有相同的一组操作并且接口也是一样的。区别在于操作的语义：队列采取 *FIFO* 方式，而优先级队列正如名称暗示的一样采用优先级排队的方式。

和绝大多数 ADT 一样，实现队列的途径有很多种。由于队列是项的容器，就可以利用任何存贮容器的基本机制，包括数组和链表。具体对二者的选择基于两点：性能——执行时间；易用——实现的难易。

### 16.1 队列 ADT

队列 ADT 有如下操作定义：

构造器-**constructor**：创建新的空队列。

**add** 方法：向队列增加一个新项。

**remove** 方法：从队列中移出一项并返回。返回的是最先加入的项。

**isEmpty** 方法：检查队列是否为空。

下面是基于内建类 `java.util.LinkedList` 的一个通用队列的实现：

```
public class Queue {
    private LinkedList list;
    public Queue () {
        list = new LinkedList ();
    }
    public boolean isEmpty () {
```

```
        return list.isEmpty ();
    }
    public void add (Object obj) {
        list.addLast (obj);
    }
}
```

一个队列对象包含单个实例变量，即实现它的链表。对于其它各个方法，只要从 `LinkedList` 类中调用一个就行了。

利用继承可以将同样的实现变得稍微简洁一点：

```
public class Queue extends LinkedList {
    public Object remove () {
        return removeFirst ();
    }
}
```

好，实际上简洁了很多！因为 `Queue` 扩展了 `LinkedList`，也就继承了构造器、`isEmpty` 方法和 `add` 方法。同时还继承了 `remove` 方法，不过从 `LinkedList` 类得到的版本做的事情不是我们期望的，它移出的是链表的最后一个元素而不是第一个。编写一个 `remove` 的新版本覆盖继承的版本可以修正这个问题。

选择何种实现要看几个要素。由于父类中有很多有用的方法，所以继承可以使实现更加简洁。然而这也使实现代码难于阅读和调试，因为继承的方法和新类不在一个地方。还有也可能导致预料之外的行为，因为继承自父类的方法不一定恰好满足需要，这就意味第二个版本的 `Queue` 类中的有 `removeLast` 和 `clear` 这样不属于队列 ADT 组成部分的方法。第一个实现更安全，通过声明一个私有的（`private`）实例变量，防止了委托者访问 `LinkedList` 类的方法。

## 16.2 蒙版

通过 `LinkedList` 来实现 `Queue` 达到了利用现有代码的目的，相当于把 `LinkedList` 的方法翻译成了 `Queue` 的方法。像这样的实现称为**蒙版**（*veneer*）。在实际生活中，蒙版是指做家具时用来蒙住下面较差的木材以遮盖的一块质量很好的薄木板。计算机科学家用这个比喻来描述隐藏实现细节并提供较标准或较简单的接口的一小段代码。

`Queue` 类的例子演示了蒙版的一个好处，这就是易于实现；也暴露了一个陷阱，那就是**性能风险**（*performance hazard*）！

一般说来当调用方法时不用关心实现细节，但只有一个“细节”是我们想

了解的——方法的性能特点。把它视作链表项目数的函数，要花多长时间？

要回答这个问题，就不得不更多的了解实现。假设 `LinkedList` 已经被实现为一个链表，那么 `removeFirst` 的实现可能是这样：

```
public Object removeFirst () {  
    Object result = head;  
    if (head != null) {  
        head = head.next;  
    }  
    return result.cargo;  
}
```

假定 `head` 指向链表的第一个节点，每个节点包含一个 `cargo` 和对下一个节点的引用。此处没有循环和函数调用，所以每次方法的运行时长基本一样。这样的方法叫做固定时长操作。实际上如果链表为空方法可能稍微快一点，因为直接跳过了条件语句的执行，不过这一点区别并不重要。

`addLast` 方法的性能就大不一样了，下面是一个假定的实现：

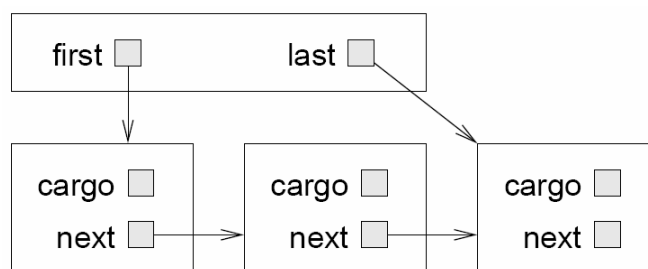
```
public void addLast (Object obj) {  
    // special case: empty list  
    if (head == null) {  
        head = new Node (obj, null);  
        return;  
    }  
    Node last;  
    for (last = head; last.next != null; last = last.next) {  
        // traverse the list to find the last node  
    }  
    last.next = new Node (obj, null);  
}
```

第一个条件句处理向空链表增加新节点这一特殊情况。此时运行时长和链表长度无关。而在一般情况下，就要遍历链表以找到最后一个元素并使其指向新节点。

遍历的时长和链表长度成正比。也就是说运行时长和是链表长度的一个线性函数，这时就说这个方法的耗时是线性的。比起固定时长操作，这类方法不足取。

## 16.3 链队列

我们希望队列 ADT 实现执行的所有操作都是固定时长的。达到目标的途径之一就是实现一个**链队列** (*linked queue*)，这和链表很相似，都由零个或多个链接起来的 Node 对象组成。二者的区别是队列同时拥有指向第一个和最后一个节点的引用，如图所示：



下面是我写的链队列的实现：

```
public class Queue {
    public Node first, last;
    public Queue () {
        first = null;
        last = null;
    }
    public boolean isEmpty () {
        return first == null;
    }
}
```

目前看上去还算直观——在一个空队列中，`first` 和 `last` 都是 `null`。要检查链表是否为空只要检查其中一个。

`add` 方法则稍微复杂一些，因为要处理一些特殊情况：

```
public void add (Object obj) {
    Node node = new Node (obj, null);
    if (last != null) {
        last.next = node;
    }
    last = node;
    if (first == null) {
        first = last;
    }
}
```

```
}  
}
```

第一个条件语句检查 `last` 是否确实指向一个节点，如果没有，就使其指向一个新节点。

第二个条件句处理链表初始化为空的特殊情况，此时 `first` 和 `last` 都指向新节点。

`remove` 也处理几种特殊情况：

```
public Object remove () {  
    Node result = first;  
    if (first != null) {  
        first = first.next;  
    }  
    if (first == null) {  
        last = null;  
    }  
    return result;  
}
```

第一个条件句检查队列中是否有节点。如果有，就把下一个节点拷贝给 `first`。第二个条件句处理链表为空的情况，此时必须将 `last` 赋值为 `null`。

作为练习，画出这些运算的一般和特殊情况，并检验以确认无误。

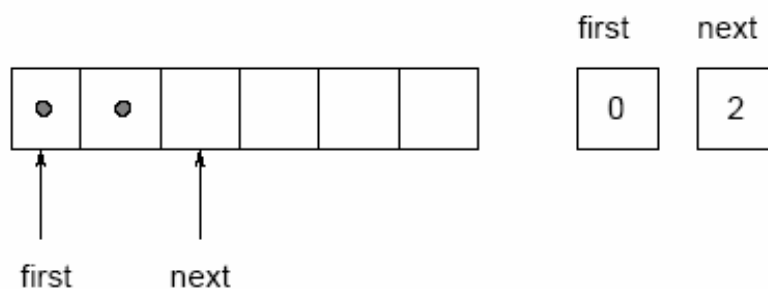
很明显，这个实现比蒙版实现复杂一些，要演示其正确性也更困难。优点是达到了目标：`add` 和 `remove` 都是固定时长操作。

## 16.4 循环缓冲

队列的另一种常用实现是**循环缓冲** (*circular buffer*)。“缓冲”是临时存贮区域的一般叫法，通常指向一个数组，比如在接下来的例子中就是如此。至于为什么是“循环”的缓冲，接下来的几分钟就会明白。

循环缓冲的实现和 15.12 节中的堆栈的数组实现相似。队列项存贮在一个数组中，用索引号来跟踪数组。在堆栈的实现中，有一个唯一的索引号指向下一个可用的空间。在队列实现中则有两个索引号：第一个指向数组中第一个顾客的空间，另外一个指向下一个可用空间。

下图示意了只有两项（用点表示）的队列：



对于变量 `first` 和 `last` 可以从两种角度理解。从字面上看，是整数，它们的值放在右边盒子里；然而抽象地看，二者是数组的索引号，所以又被画成两个指向数组存储区域的箭头。很容易掌握箭头的用法，但是应该记住索引号不是引用，仅仅是整数。

下面是一个未完成的队列的数组实现：

```
public class Queue {
    public Object[] array;
    public int first, next;
    public Queue () {
        array = new Object[128];
        first = 0;
        next = 0;
    }
    public boolean isEmpty () {
        return first == next;
    }
}
```

实例变量和构造器是显而易见的，不过再次遇到了选择任意数组大小的问题。稍后再解决这个问题，方法其实和用堆栈实现一样，如果数组满了就调整大小。

`isEmpty` 的实现可能有点让人吃惊，你也许认为 `first == 0` 指向一个空的队列，不过别忘了队列的头位元素并非在数组的开始。相反，如果 `head` 等于 `next` 那么队列就是空的，没有任何项。等看了 `add` 和 `remove` 的实现，就更能理解这段话的意思了。

```
public void add (Object item) {
    array[next] = item;
    next++;
}
```

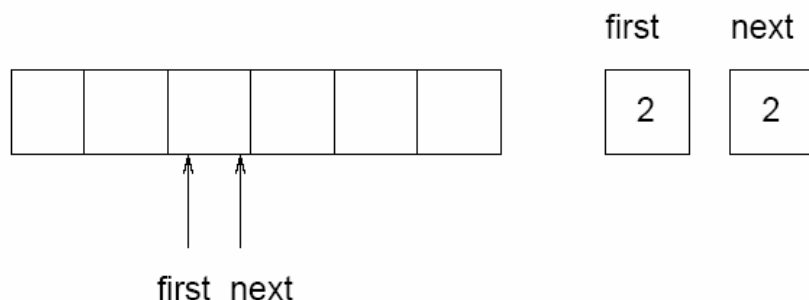
```

public Object remove () {
    Object result = array[first];
    first++;
    return result;
}

```

add 很像 15.12 节的 push，把新项压入下一个可用空间中并将索引号自增 1。

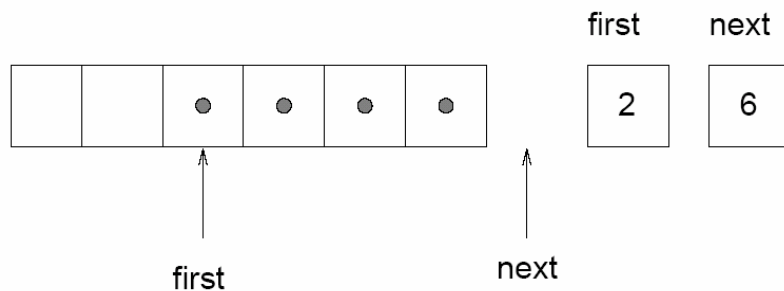
remove 和 add 类似，取队列的第一项 first 并将其自增 1 使之指向队列中新的头元素。下图示意了两项都被移出后队列的情形。



任何时候，next 总是指向一个可用空间。如果 first “追平”了 next 并为之指向同一个空间，那么 first 就会指向一个“空”区域，并且队列也为空。把“空”加上引号是因为 first 指向的空间实际可能含有一个值（代码中没有确认这个空区域是不是含有 null），但因为队列是空的，这个区域就不会被读到，所以抽象的看它就是空的。

这个实现的第二个问题是可能出现空间耗尽。当增加一项时将 next 自增 1，移出一项时将 first 自增 1，但都不会自减。那么到了数组的最后会怎么样呢？

下图示意了再增加四项后队列的情形：



数组已满。没有“下一个可用空间”了，所以 next 没有地方可以指向。一种可能是像实现堆栈时那样扩大数组，但在这里数组应该根据队列的实际项数目而增大。更好一些的方案是围绕着数组是开始部分进行，重用这一区域的空闲。”围绕“就是把这个实现叫做循环缓冲的原因。

把索引进行环绕的办法之一是，当索引号每自增 1 就增加一种特殊情况：

```

next++;

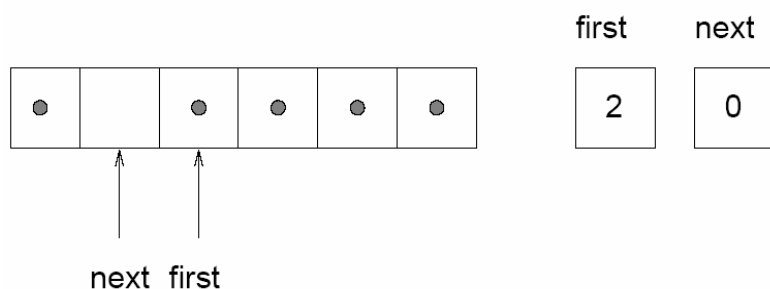
```

```
if (next == array.length) next = 0;
```

而更理想的办法是用取模运算符。

```
next = (next + 1) % array.length;
```

不论哪种办法都还有一个问题要解决。我们如何知道一个队列真的已经满了呢——这意谓着不能在增加新项了。下图示意了队列“满”了的时候的情形。



数组中本来还有一个空间，但如果增加一项就满了，于是就不得不增加 `next` 以使 `next == first`，这样一来队列为空了。

为了避免这种情况发生，要牺牲数组中的一个空间。那么又如何判断队列是满的呢？

```
if ((next + 1) % array.length == first)
```

如果数组满了怎么做？此时，唯一的办法就是增大数组。

## 16.5 优先队列

优先队列 ADT 和队列 ADT 拥有一样的接口，区别只在于语义上。优先队列的接口是：

构造器：创建一个新的空队列。

`add` 方法：向队列中增加一个新项。

`remove` 方法：移出并返回队列中的一项。返回的项是优先级最高的。

`isEmpty` 方法：检查队列是否为空。

语义的区别就在于，被移出的项不必是最先增加的项，而是在队列中任何位置但拥有最高优先级的项。优先级是什么以及如何比较优先级，不是在优先队列中实现的。着要根据队列中的项是什么而定。

打个比方，如果队列中的项是名字，可能就要按字母次序来选；如果是保龄球比赛的分数，可能就从高到低来选；而如果是高尔夫球比赛的分数，应该从低到高。

所以就遇到了一个新的问题。我们希望优先队列的实现尽量通用——应该对任何一种对象都适用，而且同时实现优先队列的代码还要能比较优先队列中的对象。



前面已经看到过用 `Object` 对象实现通用数据结构的例子，可惜那儿的办法不能解决这里的问题，因为除非知道型别，就无法比较 `Object` 对象。

这个问题的答案藏在一种 `Java` 的特性——元类中。

## 16.6 元类

**元类** (*metaclass*) 是一组提供了一些公共接口的类。元类的定义规定了一个类成为一个集合成员所必须满足的要求。

通常元类以“able”结尾的单词为名，以标识元类要具备的基本功能。例如，任何提供名为 `draw` 方法的类可以是名为 `Drawable` 的元类的成员。任何含有方法 `start` 的类可以是 `Runnable` 元类的成员。

`Java` 提供了一个内建元类，可以用来实现优先队列，称为 `Comparable`（能比较）。任何属于 `Comparable` 元类的类必须提供一个方法 `compareTo` 用来比较两个对象并返回一个指出谁大谁小或者大小一样的值。

许多 `Java` 内建类都是 `Comparable` 元类的成员。包括数值包装类，例如 `Integer` 和 `Double` 类。

下一节将展示如何写一个操纵元类的 ADT，同时看看怎么样写一个属于已有元类的新类。下一章就会看到如何定义新元类。

## 16.7 优先队列的数组实现

在优先队列的实现过程中，每声明一个队列中项的型别，就要声明元类 `Comparable`。例如实例变量是一个能比较的数组和一个整数。

```
public class PriorityQueue {  
    private Comparable[] array;  
    private int index;  
}
```

和以前一样，`index` 是数组中下一个可用区域的索引号。实例变量被声明为 `private` 以使其它类不能直接访问。

构造器和 `isEmpty` 方法和前面的也很类似，而数组的初始大小也是随意的。

```
public PriorityQueue () {  
    array = new Comparable [16];  
    index = 0;  
}  
  
public boolean isEmpty () {
```

```
    return index == 0;
}
```

add 和 push 类似:

```
public void add (Comparable item) {
    if (index == array.length) {
        resize ();
    }
    array[index] = item;
    index++;
}
```

这个类中唯一有点实际意义的方法是 remove, 用来遍历数组找出并移出最大项:

```
public Comparable remove () {
    if (index == 0) return null;
    int maxIndex = 0;
    // find the index of the item with the highest priority
    for (int i=1; i<index; i++) {
        if (array[i].compareTo (array[maxIndex]) > 0) {
            maxIndex = i;
        }
    }
    Comparable result = array[maxIndex];
    // move the last item into the empty slot
    index--;
    array[maxIndex] = array[index];
    return result;
}
```

遍历数组的时候, maxIndex 跟踪最大的元素的索引号。“最大”的意思是经过 compareTo 比较得到的。此处的 compareTo 方法由 Integer 类提供, 作用很直观——返回两个数中较大的一个。

## 16.8 优先队列的委托者 (client)

上一节中的优先队列完全通过 Comparable 对象来实现, 然而实际并不存在一个 Comparable 对象。不信试试创建一个:

```
Comparable comp = new Comparable (); // 错误
```

Java 将会给出一条编译时信息说“java.lang.Comparable 是一个接口，不能被实例化”之类的话。元类在 Java 中叫做接口（*interface*）。我一直没有使用这个词的原因是因为它同时有很多意思，不过到了这里不能不用了。

为什么元类不能被实例化？因为元类只规定了要求（必须有一个 compareTo 方法）但不提供实现。

要创建一个 Comparable 对象就不得不创建一个属于 Comparable 集合的对象，比如 Integer。接下来就可以在任何 Comparable 对象被调用的地方使用这个 Integer 对象了。

```
PriorityQueue pq = new PriorityQueue ();
Integer item = new Integer (17);
pq.add (item);
```

这段代码创建一个空的新队列和一个新的 Integer 对象。然后把这个 Integer 对象增加到队列。add 方法取一个 Comparable 对象做参数，所以 Integer 将完全满足要求。如果试图传递一个不属于 Comparable 元类的 Rectangle 对象，就会得到一条编译信息类似“型别与方法矛盾，必须显式地将 java.awt.Rectangle 转换为 java.lang.Comparable。”

编译器说的很清楚，想进行转换就要显式地做。那么就照办好了：

```
Rectangle rect = new Rectangle ();
pq.add ((Comparable) rect);
```

不过这样却会得到一个运行时错误，产生一个 ClassCastException 异常。当 Rectangle 试图作为 Comparable 对象传递时，运行时系统检查是否满足要求并将其拒绝。这就是按照编译器的建议做的结果。

要从队列中移出项，就要把过程反过来：

```
while (!pq.isEmpty ()) {
    item = (Integer) pq.remove ();
    System.out.println (item);
}
```

这个循环移出队列中所有项并打印出来。这里假定队列中的项都是整数，如果不是，就会得到一个 ClassCastException 异常。

## 16.9 Golfer（高尔夫球手）类

最后来看看如何创建一个属于 Comparable 元类的新类。为了举例说明“最高”的反常定义，采用高尔夫球手：

```
public class Golfer implements Comparable {  
    String name;  
    int score;  
    public Golfer (String name, int score) {  
        this.name = name;  
        this.score = score;  
    }  
}
```

类定义和构造器还是和以前的方式一样，区别在于必须声明“**Golfer implements Comparable**”。这儿的关键字 `implements` 意思就是 `Golfer` 实现了规定的 `Comparable` 接口。

如果打算编译此时的 *Golfer.java* 文件，将会得到类似“**Golfer 类必须被声明为 abstract；没有定义接口 java.lang.Comparable 中的 compareTo 方法**”这样的出错信息。换句话说，`Golfer` 要是个 `Comparable` 对象，就必须提供一个 `compareTo` 方法。下面就来写一个：

```
public int compareTo (Object obj) {  
    Golfer that = (Golfer) obj;  
    int a = this.score;  
    int b = that.score;  
    // for golfers, low is good!  
    if (a<b) return 1;  
    if (a>b) return -1;  
    return 0;  
}
```

这段代码中有两点值得注意。首先，参数是一个 `Object` 对象。这是因为一般情况下调用者并不知道将要比较的对象的型别。例如，在 *PriorityQueue.java* 中调用 `compareTo` 时传递一个 `Comparable` 元类对象作为参数，不必知道它究竟是一个 `Integer` 对象还是 `Golfer` 对象或者别的什么对象。

其次在 `compareTo` 方法中必须把参数由 `Object` 对象转换为 `Golfer` 对象。和以往一样这要冒一点风险：如果转换到的型别是错误的就会导致一个异常。

最后才可以创建一些高尔夫球员对象：

```
Golfer tiger = new Golfer ("Tiger Woods", 61);  
Golfer phil = new Golfer ("Phil Mickelson", 72);  
Golfer hal = new Golfer ("Hal Sutton", 69);
```

将这些对象压入队列：

```
pq.add (tiger);
```

```
pq.add (phil);
```

```
pq.add (hal);
```

把上面的对象全部弹出：

```
while (!pq.isEmpty ()) {
    golfer = (Golfer) pq.remove ();
    System.out.println (golfer);
}
```

结果球员名字按分数降序排列：

Tiger Woods          61

Hal Sutton            69

Phil Mickelson       72

当从 `Integer` 对象转换到 `Golfer` 对象时，不必将 `PriorityQueue.java` 做任何改动。这样就成功的在 `PriorityQueue` 类和使用它的类之间架起了一个抽象护栏 (*abstraction barrier*)，使得我们不必作任何修改就重用了代码。此外，我们能够根据 `compareTo` 方法的定义直接操纵委托者代码，这使 `PriorityQueue` more versatile.

## 16.10 术语表

**队列-queue:** 一组等候某种服务的有序对象。

**排队规则-queueing discipline:** 决定队列中哪位成员将是下一个最先被移出的规则。

**FIFO:** “first in, first out”，先进先出，最先到达的成员最先被移出的排队规则。

**优先队列-priority queue:** 每个成员拥有一个由外在条件决定的优先级的排队规则。优先级最高的成员将在下一次最先被移出。

**Priority Queue:** 定义了可以对优先队列执行的操作的 ADT。

**蒙版-veneer:** 通过定义方法以调用其它类中方法来实现一种 ADT 的类，有时也将调用的类稍作修改。蒙版不用来做真正重要的事情，只是提高或者将接口标准化以便于委托者调用。

**性能风险-performance hazard:** 一些方法可能以委托者不易发现的低效率方式实现，这种风险往往和蒙版联系在一起。

**固定时长-constant time:** 操作的运行时长不依赖于数据结构的大小。

**线性时长-linear time:** 操作的运行时长是数据结构大小的一个线性函数。

**链接队列-linked queue:** 通过链表和对头尾节点的引用实现的队列。

**循环缓冲-circular buffer:** 通过数组和数值中第一个元素的索引号以及下一个可用空间的索引号实现的队列。

**元类-metaclass:** 一组类。元类规定了要称为其成员必须满足的一系列要求。

**接口-interface:** Java 中对元类的说法。注意不要和广义的接口混淆。

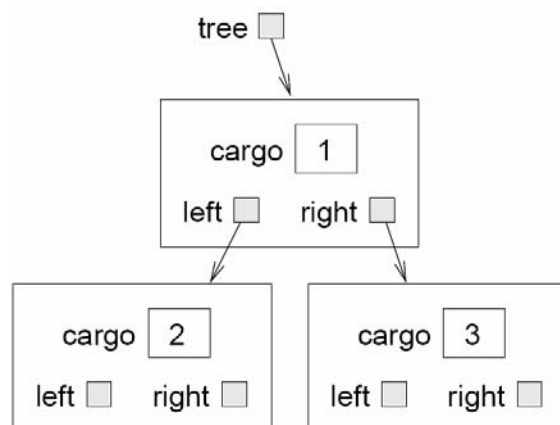
## 第 17 章 树

### 17.1 树节点

和链表相似，**树**(*tree*)也由**节点**(*node*)组成。最普通的树是**二叉树**(*binary tree*)，每个节点包含有一个指向另外两个节点的引用（可能是 `null`）。类定义如下：

```
public class Tree {  
    Object cargo;  
    Tree left, right;  
}
```

和链表类似，树节点也包含 `cargo`，在这里是一个泛指的 `Object` 型别对象。还有两个实例变量 `left` 和 `right`，在树的标准图示方法中分别代表左节点和右节点。



树的顶点（指代树的节点）叫做**根节点**(*root*)。顺着对“树”的比喻，其它节点叫做分支，其中末端没有引用对象的节点叫做**叶节点**(*leave*)。把根画在最上方而叶画在底部似乎有悖常理，不过这还不是最奇怪的事情。

更糟的是，计算机科学家们还混入了另一个比喻：族谱。顶节点有时叫做父节点，而它指向的节点叫做孩子节点，有同一个父节点的节点叫做兄弟节点，等等。

最后，要掌握树，还有一些几何术语要了解。上面提到了左(*left*)和右(*right*)，而实际上还有上（朝着父节点/根结点的方向）和下（朝着孩子节点/叶节点的方向）。还有，同一棵树同一级别上的所有节点到根节点的距离相等。

我不清楚为什么要同时借助那么多比喻来谈论树，但事实就是这样。

## 17.2 建立树

组装树节点的过程和组装链表的过程相似。下面是为树节点初始化实例变量的构造器。

```
public Tree (Object cargo, Tree left, Tree right) {  
    this.cargo = cargo;  
    this.left = left;  
    this.right = right;  
}
```

首先分配孩子节点：

```
Tree left = new Tree (new Integer(2), null, null);  
Tree right = new Tree (new Integer(3), null, null);
```

同时还可以创建父节点并将其链接到孩子节点上：

```
Tree tree = new Tree (new Integer(1), left, right);
```

这些代码生成了上一节的图中示意的树。

## 17.3 遍历树

遍历树最自然的办法是递归。例如要把一棵树上的所有整数加起来，可以写一个这样的类方法：

```
public static int total (Tree tree) {  
    if (tree == null) return 0;  
    Integer cargo = (Integer) tree.cargo;  
    return cargo.intValue() + total (tree.left)  
                           + total (tree.right);  
}
```

用类方法的原因是希望用 `null` 来表示一个空树，并把这个空树当做递归的起始情况。如果树是空的，方法返回 `0`。否则，方法将递归调用两次以求出两个孩子节点的和，最后加上根结点的 `cargo` 中的值并返回总和。

虽然这个方法可以使用，但是要与面向对象设计相适应还有些困难，因为作为 `Tree` 类中的方法不应该要求 `cargo` 只能是 `Integer` 对象。如果在 `Tree.java` 中作这样的假定就失去了通用数据结构的有点。

换句话说，这段代码访问树节点的实例变量的时候，了解的内容超过了应该知道的实现细节。如果稍后改变了实现细节，这段代码就会变得无用。

本章后面会讨论介绍这个问题的办法，允许委托者代码遍历包含任何型别

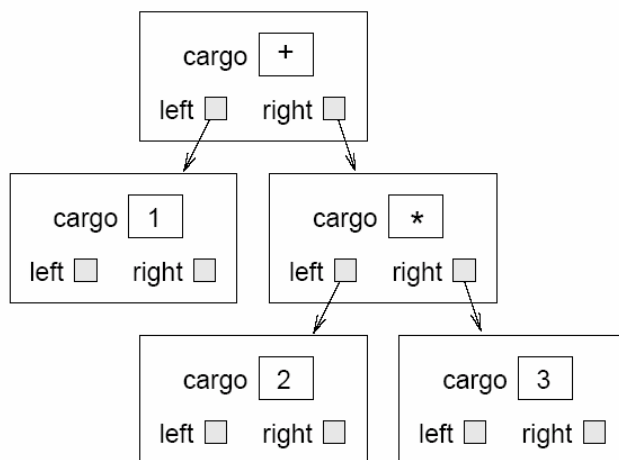


对象的树，以达到不破坏委托者代码和实现之间隔离的目的。在这之前，先看树的一个应用。

## 17.4 表达式的树

树是表达数学表达式的一种自然途径。不像其它记法，树可以毫不含糊的表示计算。举个例，如果不确定先做乘法再做加法，中缀表达式  $1 + 2 * 3$  的意思就不明确。

下图则明确无误的表示了这个算式：



节点可以是操作数如 1 或 2，也可以是运算符+或\*。操作数是叶节点，运算符节点包含了指向操作数的引用（这几个运算符都是二元的，也就是说只能有两个操作数）。

根据这幅图的描述，操作的次序是毫无疑问的：先做乘法再做加法。

象这样的表达式的树有很多用处，接下来将看看用于格式（后缀/中缀）之间转换的例子。类似的树还用于编译器内部对程序进行解析、优化和翻译。

## 17.5 遍历

前面已经提到了递归是遍历树的自然途径。

可以像下面这样打印一个表达式的树的内容：

```
public static void print (Tree tree) {
    if (tree == null) return;
    System.out.print (tree + " ");
    print (tree.left);
    print (tree.right);
}
```

这段代码说明，要打印一个树，首先打印根结点的内容，然后打印整个左子树，接下来再打印整个右子树。这种遍历树的方式叫做**先序** (*preorder*) 遍历，因为根结点的内容先于子树的内容出现。

对例子表达式的输出是+ 1 \* 2 3，这与后缀和中缀都不一样，是一种新的记法叫做前缀，其中运算符写在它的操作数后。

可以推测如果用不同的顺序遍历树得到的表达式又是不用的记法。例如先打印子树再打印根结点：

```
public static void printPostorder (Tree tree) {  
    if (tree == null) return;  
    printPostorder (tree.left);  
    printPostorder (tree.right);  
    System.out.print (tree + " ");  
}
```

得到的是后缀式的表达式 (1 2 3 \* +)! 正如方法名暗示的，这种遍历方式叫做**后序** (*postorder*) 遍历。最后来对树进行**中序** (*inorder*) 遍历，先打印左子树，然后是根结点，最后是右子树：

```
public static void printInorder (Tree tree) {  
    if (tree == null) return;  
    printInorder (tree.left);  
    System.out.print (tree + " ");  
    printInorder (tree.right);  
}
```

结果为 1 + 2 \* 3，是中缀表达式。

为了不产生误导，还要指出一个额外的但重要的问题。有的中序表达式不得不使用括号来确保运算符的次序，所以中序遍历对于生成中缀表达式来说不是非常有效。

虽然如此，通过采用一些增补措施，表达式的树和这三种递归的遍历方法仍然提供了翻译表达式格式的最常用的办法。

## 17.6 封装

前面提到过当时遍历树的办法有一个问题：破坏了委托者代码（使用树的程序）和供给者代码（树的实现）之间的隔离。理想情况下树的代码应该是通用的，不必也不该和表达式的树有任何关系。而生成和遍历表达式的树的代码也不该了解树的实现。这种设计标准叫做**对象封装** (*object encapsulation*) ——

注意和 6.6 节的方法封装区别开来。

在当前版本中，Tree 类的代码对于委托者代码了解太多。相反的，Tree 类应该提供多种遍历树的通用能力，遍历的时候执行委托者代码指定的操作。

为了使委托者和供给者代码更加彻底的分离，下面创建一个新的元类 Visitable。树中存贮的项要求是**可见的**（*visitable*），就是说这些项定义一个方法 visit 完成委托者代码希望对每个节点做的操作。这样一来，Tree 执行遍历而委托者代码执行节点操作。

要在委托者和供给者之间植入元类需要如下一些步骤：

1. 定义一个用于对供给者的组件进行调用的方法的元类。
2. 根据新元类来编写供给者代码，和一般的 Object 对象相反。
3. 定义一个术语元类的类，根据委托者要求实现相应的方法。
4. 编写使用新类的委托者代码。

接下来的几节演示上述步骤。

## 17.7 定义元类

Java 中有两种途径实现一个元类——接口或**抽象类**（*abstract class*）。二者的区别现在不重要，所以下面将定义一个接口。

接口定义看上去很像类定义，不过有两点不一样：

- 关键词 class 被换成了 interface
- 方法定义没有方法体

接口定义说明了一个类要称为元类的成员必须实现的方法。说明还包括每个方法的名称、参数型别和返回值。

Visitable 的定义是：

```
public interface Visitable {  
    public void visit ();  
}
```

就这么简单！visit 方法的定义看上去和别的类差不多，只是没有方法体。这个定义说明任何实现 Visitable 接口的类都必须有一个不带参数返回值为 void 的名为 visit 的方法。

和其它类相似，接口定义被保存在和类名相同的文件里（这里就是 *Visitable.java*）。

## 17.8 实现元类

如果用一个表达式的树来生成中缀表达式，那么“访问（*visiting*）“一个节

点的意思就打印其中的内容。因为一个表达式的树中的内容是符号，我们将创建一个名为 `Token` 的新类来实现 `Visitable` 元类：

```
public class Token implements Visitable {
    String str;
    public Token (String str) {
        this.str = str;
    }
    public void visit () {
        System.out.print (str + " ");
    }
}
```

编译这个类定义（位于 `Token.java` 文件中）的时候，编译器会检查提供的方法是否满足元类说明的要求。如果不是，将产生一条出错信息。举个例，如果把方法名 `visit` 拼错了，就会得到类似这样的消息：“`Token` 必须被声明为抽象类，没有定义接口 `Visitable` 规定的 `void visit()` 方法”。这是出错后编译器给出的众多建议的一种。它说类“必须是抽象的”，实际意思是说你必须修改类以正确的实现接口。有时候我在想，写这些信息的人真该被痛扁一顿。

下一步是修改这一小段代码以将 `Token` 对象而不是字符串放进树中。下面是简单的例子：

```
String expr = "1 2 3 * +";
StringTokenizer st = new StringTokenizer (expr, " +-*/", true);
String token = st.nextToken();
Tree tree = new Tree (new Token (token), null, null);
```

这些代码取得字符串中的第一个符号并将它包装进一个 `Token` 对象然后将这个 `Token` 放进一个树节点中。如果 `Tree` 要求 `cargo` 是 `Visitable` 型别的，这段代码也会把 `Token` 转换为一个 `Visitable` 对象。当我们从树中移出 `Visitable` 对象的时候，就得把它转换回 `Token` 对象。

执行 `visitPreorder` 这样的方法的流程有点不寻常。委托者调用一个 `Tree` 的实现提供的方法，然后树的实现调用一个由委托者提供的方法。这样的模式叫做回叫（*callback*），是一种使供给者代码不破坏抽象护栏而且通用性更强的办法。

## 17.9 Vector 类

`Vector`（向量）类是一个 Java 的 `java.util` 包中内建的类。它是一个 `Object`

数组的实现，另外还有可以自动伸缩的特性，所以我们不必担心数组的大小问题。

在使用 `Vector` 类之前，应该弄清楚几个概念。每个 `Vector` 对象都有一个 `capacity`（容积），是被分配来存贮值的空间的大小；还有一个 `size`（尺寸），是向量中实际的值的数目。

下图是一个含有三个元素的向量的示意图，该向量的容积是 7。



访问向量元素的方法有两种类型。它们提供不同的语义和不同的错误检查能力，而且两者很容易混淆。

最简单的访问方法是 `get` 和 `set`，语义上和数组索引号运算符相近。`get` 方法取一个整数索引号并返回相应位置的元素。`set` 取一个索引号和一个元素，把新元素存贮在相应位置替换已有的元素。

`get` 和 `set` 不改变向量的尺寸（元素数目），委托者代码有责任在调用 `set` 或 `get` 之前确认向量是否有足够的尺寸。`size` 方法返回 `Vector` 对象的元素数目。如果试图访问一个不存在的元素（本例中索引号为 3 到 6 的元素），将得到一个 `ArrayIndexOutOfBoundsException` 异常。

另一种类型的方法包括几个版本的 `add` 方法和 `remove` 方法。这些方法修改 `Vector` 的尺寸，在必要时也修改容积。`add` 的一个版本取一个元素作为参数并将其加到 `Vector` 对象的最后，这个方法是安全的，因为它不会导致异常。

`add` 的另一个版本取一个索引号和一个元素，像 `set` 一样将新元素放在指定位置。区别在于 `add` 方法不替换原有元素，而是增加 `Vector` 的尺寸并将元素右移为新元素腾出空间。所以，`v.add(0, elt)` 就是将新元素放在 `Vector` 的开始。不幸的是这个方法既不安全也不高效，反而很有可能会在运行时导致一个 `ArrayIndexOutOfBoundsException` 异常，而且在大多数实现中，这个方法的性能都是线性的（与向量的尺寸成正比）。

多数时候委托者代码不必担心容积。每当向量尺寸改变，容积都会自动增加。考虑到性能的因素，一些程序要控制容积增加的过程，就需要另外编写的增加和减少容积的方法。

因为委托者代码没有访问向量的实现，也就不清楚如何进行遍历。很容易想到，一种可能就是用循环变量作为向量的索引：

```
for (int i=0; i<v.size(); i++) {  
    System.out.println (v.get(i));  
}
```

这样写完全可以，但还有一种途径：可以借助 `Iterator` 类。向量类提供了

一个名为 `iterator` 的方法，返回一个 `Iterator` 对象，这使遍历数组成为可能。

## 17.10 Iterator 类

`Iterator` 是 `java.util` 包中的接口，它规定了三个方法：

`hasNext`：这次迭代还有元素吗？

`next`：返回下一个元素，如果没有就抛出异常。

`remove`：移出遍历的数据结构中最新的元素。

下面的例子用一个 `iterator` 对象遍历并打印一个向量的元素。

```
Iterator it = vector.it ();
while (it.hasNext ()) {
    System.out.println (it.next ());
}
```

一旦创建了 `Iterator`，它就是和原有向量独立的一个对象。`Vector` 对象的后继改变不影响 `Iterator` 对象。事实上，如果创建一个 `Iterator` 对象后再修改 `Vector` 对象，这个 `Iterator` 对象就变成了无效的，如果再次访问这个 `Iterator` 对象，将导致一个 `ConcurrentModification` 异常。

前面一小节我用 `Visitable` 元类以使委托者得以遍历一个数据结构而不必知道其实现细节。迭代器则提供了另一个途径做同样的事情。第一种情形下供给者执行迭代并调用委托代码以“拜访”每个元素。第二种情形下供给者向委托者提供一个对象，这个对象每次能选择一个元素（按照委托者控制的顺序）。

## 17.11 Glossary

**二叉树**-*binary tree*：每个节点指向 0、1 或 2 个不等的附属节点。

**根节点**-*root*：一棵树最上面的节点，没有别的节点指向根节点。

**叶节点**-*leaf*：一棵树最底部的节点，不指向任何别的节点。

**父节点**-*parent*：指向某个指定节点的节点。

**孩子节点**-*child*：被另一个节点所指向的节点。

**层**-*level*：到根节点等距的节点的集合。

**前缀**-*prefix notation*：将每个运算符写在其操作数前面的数学表达式。

**先序**-*preorder*：遍历树的一种方式，先访问父节点再访问它的孩子节点。

**后序**-*postorder*：遍历树的方式，在访问，每个节点之前先访问它的孩子节点。

**中序**-*inorder*：遍历树的方式，先访问左子树，然后是根节点，最后是右子树。

**类变量**-*class variable*：在任何方法之外声明的 `static` 变量，可以由任何方法访

问。

**二元运算符-binary operator:** 取两个操作数的运算符。

**对象封装-object encapsulation:** 设计这种技术的目的是为了保持两个对象的实现尽可能独立。两个对象的类都不必知道对方的实现细节。

**方法封装-method encapsulation:** 这种技术的设计目的是为了保持方法的接口和实现细节相互独立。

**回叫-callback:** 供给代码调用委托者提供的方法的执行流程。

## 第 18 章 堆

### 18.1 树的数组实现

“实现”一个树的意思是什么呢？迄今为止我们只看过一种树的实现——和链表类似的链接数据结构，但还有许多结构也应当被视作树。任何能执行树的基本操作的结构都应该被视作树。

那么树的操作是些什么呢？如何定义树 ADT？

构造器：创建一棵空树。

getLeft 方法：返回当前节点的左孩子。

getRight 方法：返回当前节点的右孩子。

getParent 方法：返回当前节点的父节点。

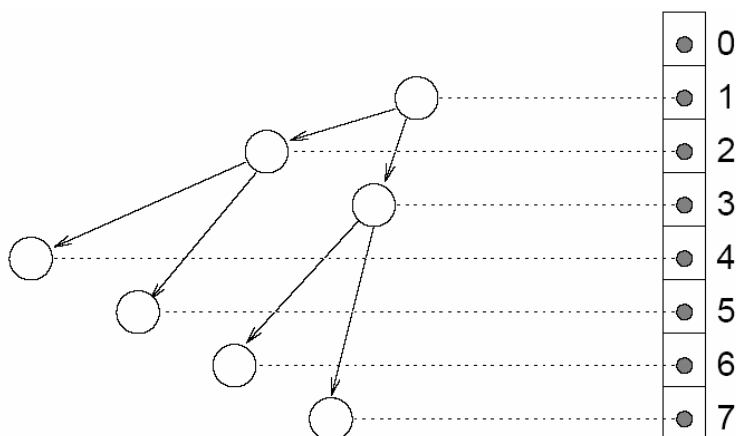
getCargo 方法：返回当前节点的 cargo 对象。

setCargo 方法：将一个 cargo 对象赋值给节点（并在必要时创建一个节点）。

在链接型的实现中，空树用特殊值 null 表示。getLeft 方法和 getRight 方法通过访问节点的实例变量来执行，getCargo 和 setCargo 也是这样。在链接版本中没有实现 getParent 方法（或许你可以想想怎么做）。

另外一种树的实现不是用对象和引用，而是用数组和索引号。为了看看到底是怎么做的，首先从一个数组和对象的混合实现开始。

下图示意的树和前面看过的那棵很像，只不过这里是倾斜画的。右边是一个数组，数组元素是指向节点中 cargo 的引用。



树的每一个节点都有唯一的索引号。此外，索引号按照预先设计的方式被赋值给节点，以达到下面的目标：

1. 索引号为  $i$  的节点的左孩子的索引号为  $2i$ 。
2. 索引号为  $i$  的节点的右孩子的索引号为  $2i+1$ 。
3. 索引号为  $i$  的节点的父节点的索引号为  $i/2$ （四舍五入）。



利用这几个公式，只要做点数学运算就可以实现 `getLeft`、`getRight` 和 `getParent` 方法，而根本用不到引用。

既然用不到引用，就可以弃之不理，就是说树的节点现在只有 `cargo` 而没有别的东西了。换句话说可以把树实现为一个 `cargo` 对象的数组，不需要节点。

下面是实现的例子：

```
public class ArrayTree {
    Object[] array;
    int size;
    public ArrayTree () {
        array = new Object [128];
    }
}
```

没有让人惊讶的事情发生。唯一的实例变量是包含树的 `cargo` 的 `Object` 对象的数组。构造器专门负责初始化，生成一个空树。

这里是 `getCargo` 和 `setCargo` 最简单的实现。

```
public Object getCargo (int i) {
    return array[i];
}

public void setCargo (int i, Object obj) {
    array[i] = obj;
}
```

这两个方法不做任何错误检查，因此如果参数是错误的，将导致一个 `ArrayIndexOutOfBoundsException` 异常。

`getLeft`、`getRight` 和 `getParent` 的实现仅仅是数学运算：

```
public int getLeft (int i) { return 2*i; }
public int getRight (int i) { return 2*i + 1; }
public int parent (int i) { return i/2; }
```

最后准备好该创建一个树了，在一个类（委托者代码）中可以这么写：

```
ArrayTree tree = new ArrayTree ();
```

```
tree.setCargo (1, "cargo for root");
```

构造器创建一个空树，调用 `setCargo` 方法把字符串 `cargo for root` 放在根节点。

向根节点增加孩子节点：

```
tree.setCargo (tree.getLeft(1), "cargo for left");
tree.setCargo (tree.getRight(1), "cargo for right");
```

在树类中应该提供一个按先序遍历打印树的内容的方法：

```
public void print (int i) {  
    Object cargo = tree.getCargo (i);  
    if (cargo == null) return;  
    System.out.println (cargo);  
    print (getRight (i));  
    print (getLeft (i));  
}
```

要调用这个方法就得将根节点的索引号作为参数传递：

```
tree.print (1);
```

输出是：

cargo for root

cargo for left

cargo for right

这个实现提供了树定义的基本操作，前面已经指出，链接型的实现中提供了同样的操作，只是语法不同。

从某种角度来说，数组实现有点笨拙。其中一个问题是，假定值为 `null` 的 `cargo` 对象指向一个不存在的节点，实际就是说不能把一个 `null` 对象放入树中作为一个 `cargo`。

另一个问题是，子树并非表示为一个对象，而是用数组的索引号来表示的。要把树节点当做参数传递，就得传递一个引用给树对象一个索引号给数组。最后，一些在链接实现中易于执行的操作，比如移出整个子树，在数组实现中就较难。

话说回来，这种实现节约了空间，因为节点之间没有链接，并且也有几种比较简单快速的操作。这几种操作正是实现堆（*Heap*）要用到的。

堆是基于树的数组实现的一个优先级队列 ADT，这种实现比前面讨论过的实现更高效。

只要几步就可以证明这种说法。首先，需要一种比较不同实现的性能的办法；接下来，考察堆执行的操作；最后比较优先队列的堆实现和其它实现（数组和链表），看看为什么认为堆特别高效。

## 18.2 性能分析

比较算法的时候，需要一种办法来分辨谁更快、使用的空间更小，或者占用的其它资源更少。很难给出这个问题的详细答案，因为算法用的时间和空间

依赖于算法的实现、具体解决的问题，还有运行程序的硬件平台。

这一节的目标就是找到一个办法来讨论独立于以上因素，只依赖于算法本身的性能。让我们从运行时着手，后面再讨论其它资源。

下面几天约束条件会指导我们的选择：

**1.**首先，算法的性能依赖于运行的硬件，所以谈论运行时长通常不用秒这样的常规概念，而是对抽象的数学运算次数进行统计。

**2.**其次，性能通常依赖于要解决的特定问题——一些问题比另一些简单。为了比较算法，通常关注最坏情形和平均（或普通）情形。

**3.**第三，性能依赖于问题的规模（通常但并非绝对，是指容器中元素的数目）。这种依赖性通过将运行时长表示为问题规模的函数来说明。

**4.**最后，性能依赖于实现的细节，例如分配对象的额外开销和方法调用的额外开销。这些细节通常被忽略，因为随着问题规模的增大不影响抽象操作的比率。

为了更好的理解整个过程，考虑两个前面见过的分别对数组和整数排序的算法。第一个是选择排序，这个算法我们在 12.3 节看过。下面是当时使用的伪代码：

```
selectionsort (array) {  
    for (int i=0; i<array.length; i++) {  
        // find the lowest item at or to the right of i  
        // swap the ith item and the lowest item  
    }  
}
```

为了执行伪代码中说明的操作要写两个助手方法，findLowest 和 swap。用伪代码写的 findLowest 方法是这样的：

```
// find the index of the lowest item between  
// i and the end of the array  
findLowest (array, i) {  
    // lowest contains the index of the lowest item so far  
    lowest = i;  
    for (int j=i+1; j<array.length; j++) {  
        // compare the jth item to the lowest item so far  
        // if the jth item is lower, replace lowest with j  
    }  
    return lowest;  
}
```

swap 方法是这样的：

```
swap (i, j) {  
    // store a reference to the ith card in temp  
    // make the ith element of the array refer to the jth card  
    // make the jth element of the array refer to temp  
}
```

要分析这个算法的性能，第一步是选择要统计的操作。显而易见，程序做的事情包括：自增  $i$ ；将  $i$  与 `deck` 的长度做比较；搜索数组中最大的元素，等等。到底什么是应该统计的东西还目前还不清楚。

实践证明统计比较两个项目所花费的时间是一个很好的选择。其它还有很多选择最后得到的结果是一样的，但这种办法做起来很简单。下面我们将看到用这种办法比较排序算法有多么容易。

写好了 `findLowest` 方法，接下来就要定义“问题规模”。这里自然该选数组的大小，设为  $n$ 。

最后，根据前面的分析，写一个  $n$  的函数——用来告诉我们应该做多少次抽象操作（这里是比较）的表达式。

下面从分析助手方法着手。`swap` 方法拷贝一些引用，但不执行任何比较，所以忽略用于执行 `swap` 的时间。`findLowest` 方法从  $i$  开始遍历数组，将每一项和 `lowest` 比较。`findLowest` 方法比较的项数是  $n - i$ ，所以总的比较次数是  $n - i - 1$ 。

接着考虑 `findLowest` 方法调用了多少次，每次  $i$  的值是什么。最后一次调用时  $i$  的值是  $n - 2$ ，所以比较的次数是 1；这之前的一次迭代执行 2 次比较；依此类推。

第一次迭代过程中， $i$  为 0，比较次数为  $n - 1$ 。

比较次数的总和是  $1 + 2 + \cdots + n - 1$ ，这个式子求和的结果等于  $n^2/2 - n/2$ 。

描述这个算法一般忽略低次项 ( $n/2$ ) 而说工作总量和  $n^2/2$  成正比。因为多项式的高次项是二次方，也可以说这个算法是二次时长 (*quadratic time*) 的。

### 18.3 合并排序的分析

在 12.6 节提到合并排序耗时和  $n \log n$  成正比，但当时没有说是为什么，现在来看看原因。





还是先从看算法的伪代码开始，对于合并排序，就是：

```

mergeSort (array) {
    // find the midpoint of the array
    // divide the array into two halves
    // sort the halves recursively
    // merge the two halves and return the result
}

```

在递归的每个层次，将数组分成两半，做两次递归调用，然后将两半合并起来。用图形示意这个过程如下：

	# arrays	items per array	# merges	comparisons per merge	total work
	1	$n$	1	$n-1$	$\sim n$
	2	$n/2$	2	$n/2-1$	$\sim n$
⋮	⋮	⋮	⋮	⋮	⋮
	$n/2$	2	$n/2$	$2-1$	$\sim n$
	$n$	1	0	0	

图中的每一行是一层的递归。在顶层数组被一分为二，在底层  $n$  个只有一个元素的数组被合并为  $n/2$  个有 2 个元素的数组中。

表的前两列显示每一层有多少数组和每个数组有多少项元素。第三列显示在每层递归中发生的合并次数有多少。再下一列代表的意思就值得好好思考了：每次合并需要执行多少次比较。

如果再去看看合并算法的伪代码（或者实现），你会发现最坏的情况要进行  $m-1$  次比较， $m$  是将要合并的项数。

接下来将每个层次的合并次数乘以每次合并的平均工作量，结果就是每个层次的工作总量。这里取了一点巧，既然我们最后感兴趣的是结果的高次项，那么就可以直接忽略每次合并中比较次数的-1次方项。如果照这么算，每层的工作量就是  $n$ 。

再接下来需要得到层数关于  $n$  的函数。我们是从一个  $n$  项的数组开始的，将其不断的二分直到获得  $n$  个只有一项的数组。实际上这个过程也可以这样描述：从 1 开始不断乘 2 直到获得结果  $n$ 。换句话说，我们想知道得到  $n$  要乘多少次 2。答案是：层数 1 是以 2 为底  $n$  的对数。

最后，将每层的工作量  $n$  乘以层数  $\log_2 n$  得到  $n \log_2 n$ ，和 12.6 说的一样。这种函数没有一个很好的叫法，很多时候就念作“en log en”。

开始  $n$  比较小的时候  $n \log_2 n$  比起  $n^2$  来优势不明显，但对一个值较大的  $n$ ，

优势就显示出来了。作为练习，写一个程序打印在一个区间内  $n$  的函数  $n \log_2 n$  和  $n^2$ 。

## 18.4 额外开销

进行性能分析往往要采取一些简化措施。开始我们忽略了程序的大多数操作只统计比较次数，后来又只考虑最坏的情况。在分析的时候可以任意的舍弃一些东西，在最后常常丢弃低次项。

在解释这些分析结果的时候要提醒自己有简化措施的作用。因为合并排序的性能是  $n \log_2 n$ ，因此我们认为合并排序是一个比选择排序更好的算法，但这并不意味着合并排序总是更快。只是可以说，如果排序的数组越来越大，合并排序最后将更胜一筹。

至于花费的时间则依赖于实现的细节，包括我们考察的比较次数之外的，每个算法都会做的一些工作。这些附带的工作叫做**额外开销**（*overhead*）。这些额外开销并不影响性能分析，但要影响算法的运行时长。

例如我们实现的合并排序在作递归之前分配了一些子数组，当合并完成后就将它们交给了垃圾收集管理。回过头再看看合并排序的图，可以看到分配的空间总和与  $n \log_2 n$  成正比，分配的对象总数约为  $2n$ 。而这些分配都要消耗时间。

即使这样，一个好算法的坏实现也比一个坏算法的好实现要强。因为，对于值较大的  $n$  来说好算法总是要好些，而当  $n$  值较小的时候则无所谓，因为二者都足够好。

## 18.5 优先队列的实现

在 16 章中我们讨论了一个基于数组的优先队列实现。数组中的项没有经过排序，所以很容易增加新项（在末尾），但移出一项就较为困难，因为不得不搜索优先级最高的项。

候选办法是基于一个已排序链表来实现。在这个例子中，每当增加一项的时候就遍历链表并将新项放入正确的位置。这个实现充分利用了链表的优点，即：易于在中间增加新节点。类似地，移出具有最高优先级的项也容易，只要保持该项一直在链表的开头。

这些操作的性能分析不费什么周折。不管项数多少，在数组最后增加一项

或者从链表的开头移出一个节点花费的时间是一样的。所以两种操作都是固定时长的。

遍历数组或链表的时候，对每个元素的操作总是固定时长的，运行时长和项数成正比。因此，从数组移出项和增加节点到链表中的操作都是线性时长的。

那么在优先队列中先增加  $n$  项再移出要耗费多少时间呢？对于数组实现来说，增加  $n$  项花费的时间和  $n$  成正比，不过移出操作耗费的时间要长些。第一次移出必须遍历全部  $n$  项，第二次要遍历  $n - 1$  项（次），以此类推直到最后一次移出操作只要移出最后 1 项。因此，总共耗时为  $1 + 2 + \cdots + n$ ，等于  $n^2/2 + n/2$ 。所以增加和移出的耗时为一个线性函数和一个二次函数的和，近似地，被视作一个二次函数。

对链表实现的分析与此类似。第一次增加操作不需要作遍历，但此后每增加一项都不得不遍历至少部分链表。一般来说都不知道要遍历链表的多少项，因为这依赖于增加的数据和增加的次序。不过我们可以假定平均每次遍历链表的一半。不幸的是，即使遍历链表的一半也仍然是一个线性操作。

所以和数组一样，在链表实现中增加和移出  $n$  项的耗时都与  $n^2$  成正比。因此，根据上面这些分析是无法判断哪种实现更好的，数组和链表都是二次时长的实现。

如果我们使用一堆来实现一个优先队列，将使增加和移出操作的时长都和  $\log n$  成正比，远远比  $n^2$  强。这就是本章开始说堆能特别高效地实现优先队列的原因。

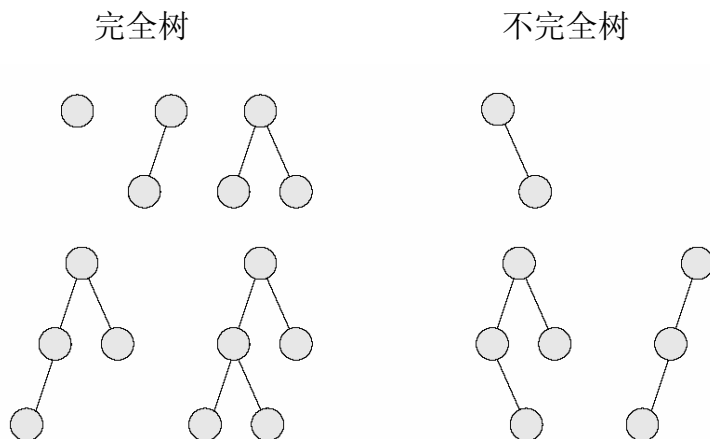
## 18.6 堆的定义

堆是一种特殊的树。它有两条一般树没有的属性——

**完整性 (completeness)**：一棵完整的树的节点是按从上到下从左到右的顺序来增加的，没有一个留空。

**堆属性 (heapness)**：树中优先级最高的项位于顶端，而且对每个子树也是如此。

这两个属性都需要一点解释。下面的图示意了一些完全树和不完整的树：



一棵空树也被认为是完全的。我还可以通过比较子树的高度来更严格的定义完整性。回想一下，树的高度就是层数。

从根开始，如果树是完全的，那么左子树的高度和右子树的高度应该是相等的，或者左子树可以高 1。除此之外任何情况，树都不是完全的。更进一步讲，如果树是完全的，那么对树上每个节点的子树之间的关系都应当满足上述条件。

堆属性和递归有些神似。要使一棵树成为堆，树上最大的值必须位于根节点，每个子树也是如此。做为练习，写一个方法来检验一棵树是否具有堆属性。

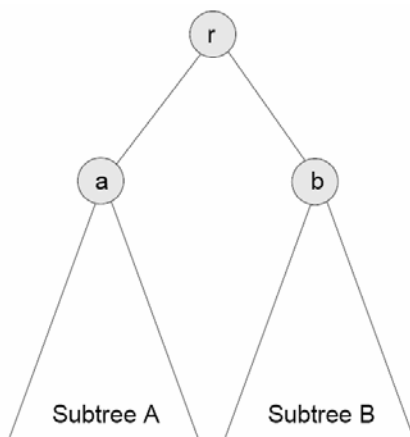
## 18.7 堆的 remove 方法

在没有增加任何东西之前就要从堆中移出似乎有点古怪，但我认为移出操作更容易讲解。

一眼看上去，从堆中移出项似乎是一个固定时长操作，因为拥有最高优先级的项总是在根节点上。问题在于一旦移出了根节点，剩下的部分就不再是一个堆了。在返回值之前，我们不得不还原堆属性。我把这个操作叫做**堆重整 (reheapify)**。



下图示意了上述情形：



根节点的优先级是  $r$ ，具有两个子树 A 和 B。子树 A 根节点的值是  $a$  子树 B 根节点的值是  $b$ 。

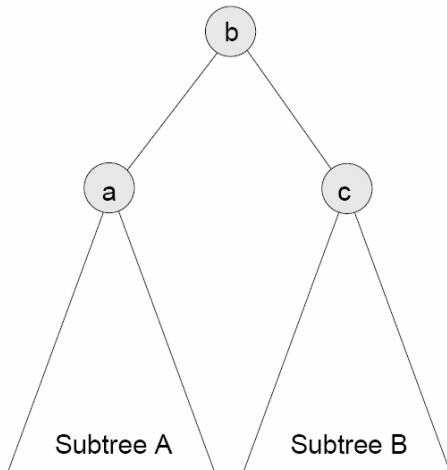
假设在移出  $r$  之前树是一个堆。这暗示了  $r$  是堆中最大的值，而  $a$  和  $b$  则是各自子树中的最大值。

一旦移出  $r$ ，就必须让留下的树再次成为堆。换句话说我们要确保其完整性和堆属性。

确保完整性的最好办法是，移出最靠底部和右边的节点。将该节点命名为  $c$  并将其值放在根节点中。在树的一般实现中，为了找整个  $c$  节点将不得不遍历整个树；但在数组实现中，可以在固定时长内找到该节点，因为它总是数组的最后一个（非空）元素。

当然，绝大多数时候上述值不会是优先级最高的，于是将它放在根节点的位置上就会破坏堆属性。幸运的是要恢复也很容易。根据堆的属性知道，剩下的部分最大的不是  $a$  就是  $b$ 。于是可以将二者中较大的选出来和根节点的值交换。

任意举个例说  $b$  较大。既然我们知道  $b$  是堆中最大的值，就可以将它放在根节点并将  $c$  放在子树 B 的顶部。此时的情形如图：



问题又来了， $c$  的值是刚刚从数组的最后一个元素中拷贝来的，而  $b$  是堆中剩下的最大的值。既然根本没有改动过  $A$  子树，就可以确信它仍然是堆。唯一的问题就是不知道现在的  $B$  子树是否还是一个堆，因为刚刚将一个值（可能是最小的那个）压入了它的根节点。

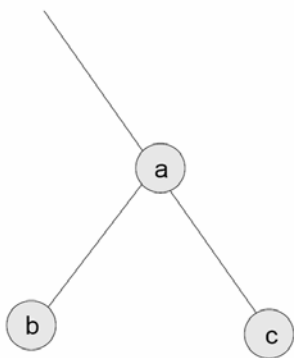
如果有一个可以对子树  $B$  进行堆重整的方法不是就好了吗？别急……会有有的。

## 18.8 add 方法

向堆中增加新项和移出是类似的操作，只不过不再是依次从顶部向下操作，而是从底部依次向上操作。

为了保证完整性，还是将新增元素放在树上最底部和最右边的位置，同时这也是数组中的下一个可用空间。

然后，为了恢复堆属性，将新的值与其相邻的元素比较。此时的情形是这样：



新的值是  $c$ 。通过将  $c$  和  $a$  作比较可以恢复该子树的堆属性。如果  $c$  较小，堆属性已经满足。如果  $c$  较大则将  $c$  和  $a$  交换，交换后也满足了堆属性，因为  $c > a$  且  $a > b$ ，所以  $c$  必定也大于  $b$ 。

既然可以对子树进行堆重整，就可以将这个办法层层往上使用直到到达根节点。

## 18.9 堆的性能

无论增加还是移出，在实际中都是执行的固定时长操作，但随后不得不将树进行堆重整。有时我们从根节点开始向下操作，比较各项，然后递归地将各个子树进行堆重整。有时从一个叶节点开始向上操作，比较树的每一层。

和以前做的性能分析一样，这里有数种操作可以纳入统计的目标，比如比较操作和交换操作。两种选择都可以，真正的问题在于树的层数和在每层上的

工作量。在两种情形中，都要不断检测树的各层直到恢复堆属性——有时可能只需要访问某一层，而最糟的情况下可能不得不访问所有的层。下面来讨论一下最糟的情形。

在每一层上，执行的都是固定时长操作，例如比较操作和交换操作。所以工作总量与树的层数（又叫树的高）是正比关系。

因此可以说这些操作与树的高是线性相关的，但是我们关心的“问题大小”并不是高，而是堆的项数。

树的高对于  $n$  的函数是  $\log_2 n$ 。这个公式并非对所有的树都正确，但对完全树是成立的。要知道为什么成立，考虑一下树的每层上的节点数。第一层上节点数为 1，第二层为 2，第三层为 4，以此类推，第  $i$  层有  $2^i$  个节点，所有  $i$  层上的节点总数为  $2^i - 1$ 。换句话说， $2^h = n$ ，也就是说  $h = \log_2 n$ 。所以说，增加和移出操作都是对数时长的。增加和移出  $n$  项所耗费的时间都和  $n \log_2 n$  成正比。

## 18.10 堆排序

上一节的结论还暗含了另一个排序算法。设有  $n$  项，要将这些项增加到堆中然后再移出。根据堆的语义它们是有序的。前面已经表明这个算法（堆排序 *heapsort*）耗费的时长和  $n \log_2 n$  成正比，比选择排序快而和合并排序一样。

随着  $n$  增大，堆排序应该比选择排序更快。但性能分析无法告诉我们堆排序是否也比合并排序快。只可以说这两个算法的增长公式是一样的，因为它们的运行时长以同样形式的函数增长。同样意思的换句话说就是二者属于同样的复杂等级（*complexity class*）。

复杂度常常被写成“大  $O$  记号（*big-O notation*）”的形式。形如  $O(n^2)$ ，念作“ $n$  的平方的  $o$ ”。 $O(n^2)$  是一个集合，所有对于一个大值  $n$  的增长速度相当于  $n^2$  的函数都属于这个集合。说一个算法是  $O(n^2)$  的就等于说该算法是二次时长的。我们见过的算法的复杂度按递增的顺序排列如下：

- $O(1)$       固定时长（*constant time*）
- $O(\log n)$     对数时长（*logarithmic*）
- $O(n)$       线性时长（*linear*）
- $O(n \log n)$     $n$  倍  $\log n$  时长（对数时长）
- $O(n^2)$       二次时长（*quadratic*）
- $O(2^n)$       指数时长（*exponential*）

实际上到目前为止我们讨论过的算法没有一个是指数时长的。对于一个大值  $n$  而言，指数时长的各种算法很快就会变得不切实际。虽然如此，“指数增长 (*exponential growth*)”这个词却仍然频繁地出现在一些非技术语言中出现。这个词被误用滥用了，所以这里我想阐述一下它的技术含义。

人们常常用“**指数级**的 (*exponential*)”来描述一切增长和加速的曲线（这包括各种具有正斜率和正曲率的曲线）。当然，除了指数曲线还有很多别的曲线也满足这样的描述，包括二次函数（还有高阶多项式）甚至一些像  $n \log n$  这么平缓的函数。但绝大多数这些曲线都没有指数曲线那样爆炸性的（往往也是破坏性的）增长速度。

## 18.11 术语表

**选择排序**-*selection sort*: 一种简单的排序算法。参见 12.3 节。

**合并排序**-*mergesort*: 较合并排序更好的排序算法。参见 12.6 节。

**堆排序**-*heapsort*: 又一个排序算法（译注：参见本章）。

**复杂等级**-*complexity class*: 具有同样的性能增长公式（通常是运行时长）的一算法集合。

**增长公式**-*order of growth*: 有相同最高次项的一组函数，因此对于一个值较大的  $n$  有一样性质的行为。

**额外开销**-*overhead*: 在对程序的性能分析中没有考虑但消耗掉的额外运行时间或资源。

## 第 19 章 映射

### 19.1 数组、向量和映射

总的来说数组是一个很有用的数据结构，但它有两个严重的局限：

- 数组的大小不依赖于其中的项数多少。如果数组太大，就会浪费空间，太小又可能导致错误，或者不得不写代码修改数组的大小。

- 虽然数组可以包含任何型别的项，但数组的索引必须是整数。因为不能用诸如字符串来指明一个数组中的元素。

在 17.9 节中我们用内建的 `vector` 类解决了一个问题。当委托代码增加项时 `vector` 对象会自动扩展。也可以收缩一个 `vector` 对象以使其容量和当前需要的大小一致。

不过 `vector` 类对第二个问题也无能为力，索引号仍然是整数。而这正是**映射** (*Map*) ADT 大展拳脚的地方。映射和向量很相似，区别在于映射可以用任何对象型别作为索引，不论是什么型别这些索引号都叫**键** (*key*)。

正如在向量中用索引号来访问值，键是用来访问映射中的值。每个键和一个值相关联，所以映射常常也叫做**关联数组** (*associative array*)。一个特定键与一个特定值的关联叫做一个**记录** (*entry*)。

映射的一个常见例子是字典，其中的每个字（键）映射到它的定义（值）上。因为这个例子，映射常常又叫做**字典** (*Dictionary*)。还要补充一点，映射有时还叫做**表** (*Table*)。

### 19.2 Map ADT

和我们讨论过的其它 ADT 类似，`Map` 的定义支持一组操作，这包括：

构造器：创建一个新的空映射。

`put` 方法：创建一个记录使一个值和一个键相关联。

`get` 方法：对给定的键，找出相应的值。

`containsKey` 方法：如果映射中对给定的键的相应记录，就返回 `true`。

`keySet` 方法：返回映射中所有键的集合。

### 19.3 内建的 HashMap

Java 提供了一个 `Map` ADT 的实现称为 `java.util.HashMap`。本章稍后我们会明白为什么叫做 `HashMap`（哈希映射/散列映射）。

为了演示哈希映射的用法，我们将写一个短小的程序，遍历一个字符串并统计每个单词出现的次数。

我们将创建一个新的类叫做 `WordCount`，用来创建映射并打印其中的内容。每个 `WordCount` 对象包含一个 `HashMap` 对象作为实例变量：

```
public class WordCount {  
    HashMap map;  
    public WordCount () {  
        map = new HashMap ();  
    }  
}
```

`WordCount` 类的唯一一个公共方法是 `processLine`，取一个字符串并将其中的单词增加到映射中；还有一个 `print` 方法，用来在最后打印结果。

`processLine` 方法用一个 `StringTokenizer` 方法将字符串打散成单词并将每个单词传递给 `processWord` 方法：

```
public void processLine (String s) {  
    StringTokenizer st = new StringTokenizer (s, " ,.");  
    while (st.hasMoreTokens()) {  
        String word = st.nextToken();  
        processWord (word.toLowerCase ());  
    }  
}
```

最有趣的部分在于 `processWord` 做的工作：

```
public void processWord (String word) {  
    if (map.containsKey (word)) {  
        Integer i = (Integer) map.get (word);  
        Integer j = new Integer (i.intValue() + 1);  
        map.put (word, j);  
    } else {  
        map.put (word, new Integer (1));    }  
}
```

如果单词已经在映射中，则用 `get` 方法取得该单词的计数器并将计数器增 1，用 `put` 方法放置新的计数器值。否则，只要用 `put` 方法放置一个新的记录并将其计数器置为 1。

要打印映射中的记录就要能够遍历映射中的键。幸运的是，`HashMap` 对象提供了一个方法 `keySet`，该方法返回一个 `Set` 对象，`Set` 对象提供了一个方法

iterator——正如其名称所暗示，该方法返回一个**迭代器** (*Iterator*)。

下面是如何用 `keySet` 方法来打印该 `HashMap` 的内容：

```
public void print () {  
    Set set = map.keySet();  
    Iterator it = set.iterator();  
    while (it.hasNext ()) {  
        String key = (String) it.next ();  
        Integer value = (Integer) map.get (key);  
        System.out.println ("{ " + key + ", " + value + " }");  
    }  
}
```

迭代器的每个元素都是一个 `Object` 型别的对象，不过由于它们是键，所以要进行型别浇铸转换为 `String` 型别对象。我们从映射中取得的值元素是 `Object` 型别的，但它们应该是计数器，所以要进行型别浇铸转换为整数。

最后，统计一个字符串中的单词数：

```
WordCount wc = new WordCount ();  
wc.processLine ("you spin me right round baby " +  
                "right round like a record baby " +  
                "right round round round");  
wc.print ();
```

输出为：

```
{ you, 1 }  
{ round, 5 }  
{ right, 3 }  
{ me, 1 }  
{ like, 1 }  
{ baby, 2 }  
{ spin, 1 }  
{ record, 1 }  
{ a, 1 }
```

迭代器的元素没有按任何特定的顺序排列，只能保证映射中所有的键都不会漏掉。

## 19.4 一个向量实现

实现 Map ADT 的一个简单办法是用一个 Vector 对象做记录，此时每个 Entry 都是一个包含键和值的对象。Entry 类的定义可以写成这样：

```
class Entry {
    Object key, value;
    public Entry (Object key, Object value) {
        this.key = key;
        this.value = value;
    }
    public String toString () {
        return "{ " + key + ", " + value + " }";
    }
}
```

而映射的实现可以是这样：

```
public class Map {
    Vector v;
    public Map () {
        v = new Vector ();
    }
}
```

要在映射中放置一个记录，只要用 add 方法向 Vector 对象增加一个新的 Entry：

```
public void put (Object key, Object value) {
    v.add (new Entry (key, value));
}
```

最后，要在映射中查询一个键就不得不遍历 Vector 对象以找出和键匹配的

Entry：

```
public Object get (Object key) {
    Iterator it = v.iterator ();
    while (it.hasNext ()) {
        Entry entry = (Entry) it.next ();
        if (key.equals (entry.key)) {
            return entry.value;
        }
    }
}
```



```
    }  
    }  
    return null;  
}
```

遍历一个向量的代码范例在 17.10 节中已经给出。比较键的时候要用深比较（`equals` 方法）而不用浅比较（`==` 运算符）。这样做允许键的类找到“相等”的定义。在本例中，键是字符串，所以 `get` 方法会用 `String` 类中的 `equals` 方法。

对大多数内建类来说，`equals` 方法实现的是深比较。不过对于有的类却不容易定义什么才是深比较。举个例可以看一下 `Double` 型别中 `equals` 方法的文档。

因为 `equals` 是一个对象方法，如果一个键为 `null` 那么这个实现中的 `get` 方法就不会工作。为了处理 `null` 键，必须增加一种 `get` 方法的特殊情况或者写一个类方法来比较键并安全地处理 `null` 参数。不过这里我们将跳过这个细节。

有了 `get` 方法就可以写出 `put` 方法一个更完整的版本。如果对给定的键在映射中已经有有了一个记录，`put` 方法就应该更新这个记录（赋与一个新值）并返回旧值（或者如果没有的话就返回 `null`）。下面这个实现提供了上述特性：

```
public Object put (Object key, Object value) {  
    Object result = get (key);  
    if (result == null) {  
        Entry entry = new Entry (key, value);  
        v.add (entry);  
    } else {  
        update (key, value);  
    }  
    return result;  
}  
  
private void update (Object key, Object value) {  
    Iterator it = v.iterator ();  
    while (it.hasNext ()) {  
        Entry entry = (Entry) it.next ();  
        if (key.equals (entry.key)) {  
            entry.value = value;  
            break;  
        }  
    }  
}
```

```
}  
}
```

`update` 方法并非 `Map ADT` 的一部分，所以被声明为 `private`。该方法遍历向量直到找出了正确的记录然后更新字段值。注意这里没有修改向量本身而只是改动了它包含的对象之一。

现在只有 `containsKey` 和 `keySet` 方法没有实现。`containsKey` 方法和 `get` 方法几乎完全一样，除了返回的值是 `true` 或 `false` 而不是一个对象引用或者 `null`。

## 19.5 List 元类

`java.util` 包定义了一个元类名为 `List`，声明了某个类要被（抽象地）认可是一个链表就必须实现的一组操作。当然，这就意味着并非每个实现 `List` 元类的类必须是一个链表。

如果说内建的 `LinkedList` 类是 `List` 元类的成员想必不会你不会吃惊，但令人惊讶的是 `Vector` 类也是 `List` 元类的成员。

`List` 元类中定义的方法包括 `add`、`get` 和 `iterator`。事实上 `Vector` 中所有用来实现 `Map` 的方法都是 `List` 元类中定义的。这就是除了 `Vector` 型别的对象也可以用任何属于 `List` 元类的类。在我们实现的映射中可以用 `LinkedList` 对象来替代 `Vector` 对象，而程序照样能工作！

这样一种类型通性对于调节程序性能是非常有用的。你可以按照一个 `List` 这样的元类写一个程序，然后测试多种不同的实现以发现可以得到的最佳性能。

## 19.6 HashMap 实现

内建的 `Map ADT` 的实现被叫做 `HashMap` 的原因是，该实现使用了一种特别高效的方法实现了映射，这种实现叫做**哈希表**（*hash table* 注：又译“散列表”）。

为了搞懂哈希表的实现，并理解为什么高效，我们首先分析 `List` 实现的性能。

考察 `put` 方法的实现，可以看到有两种情形。如果键没有在映射中，直只需要创建一个新的记录并将其增加到 `List` 对象中。两种操作都是固定时长的。

在第二种情形中，不得不遍历 `List` 对象以找到存在的记录。这是一个线性时长的操作。出于同样的理由，`get` 方法和 `containsKey` 方法也是线性时长的。

虽然很多时候线性时长操作就足够好了，但这里还可以做的更好。事实证明有一种途径实现 `Map ADT` 以使 `put` 和 `get` 两个方法都成为固定时长的操作。

关键在于要意识到遍历一个链表耗费的时长和链表的长度成线性关系。如果可以为链表的长度加一个上限，就可以将遍历时长也加上一个上限，而任何有固定上限的操作都可以被视为固定时长的。

但如何才能限制链表的长度而不限制映射中包含的项数呢？——通过增加链表的数目。不再是只使用一个长链表，而是很多短链表。

只要知道要查找的是哪些链表，就可以给查找操作设定一个上限。

## 19.7 哈希函数

到了这里要引入一个新的概念——**哈希函数** (*hash function*)。我们需要某种办法来避免查找就判断出一个键应该在哪个链表中。假设这些短链表放在一个数组（或向量）中，就可以通过所有来引用它们。

解决方案是找出键值和链表的索引之间的某种映射关系——几乎任何一种映射关系都行。对每一个可能的键有且只有一个索引号与之对应，但同一个索引号可以有多个键与之对应。

举个例，假设有一个由 8 个链表组成的数组，一个由整数键和字符串值构成的映射。如果用 `intValue` 方法将字符串转换为整数做索引，虽然型别匹配，但有很多整数并不在 0 到 7 之间，而合法的索引号却只有这八个数。

取模运算符提供了一个简单（从代码角度看）而高效（从运行时长看）的途径来将所有整数映射到 (0, 7) 区间上。表达式

```
key.intValue() % 8
```

保证了产生的值在 -7 到 7 之间（包括两端）。如果取该结果的绝对值（用 `Math.abs` 方法），就能得到一个合法的索引号。

对于别的型别，可以采用同样的手法。例如，要将一个字符转换为整数，可以使用内建的 `Character.getNumericValue` 方法，对于双精度数则用 `intValue` 方法。对于字符串我们可以获得每个字符对应的数值并将这些值加起来，或者还可以用**移位和** (*shifted sum*)。要计算移位和有两种选择，向**累加器** (*accumulator*) 中新增一个值或者将累加器左移。将累加器“左移”的意思是乘以一个常量。

为了弄清楚这样做的工作原理，以一个链表为例，其中的元素是 1、2、3、4、5、6，接下来就要计算这些值的移位和。首先初始化累加器为 0，然后：

1. 将累加器乘以 10。
2. 将链表中的下一个元素加到累加器上。
3. 重复上两步直到完成对链表中所有元素的操作。

作为练习，写一个方法，以 16 为乘数计算一个字符串中每个字符对应数值

的移位和。

对每个型别，都可以找到一个函数，取该型别的值作参数并生成相应的整数值。这样的函数就叫哈希函数，因为这类函数调用常常将对象的部件打散（*hash*）。每个对象的整数值就是它的**哈希代码**（*hash code*）。

还有一种为 Java 对象生成哈希代码的办法。每个 Java 对象都提供了一个方法叫做 `hashCode`，返回该对象对应的一个整数。对于内建的型别，`hashCode` 方法的实现使得如果两个对象含有同样的数据（深比较相等），它们就有同样的哈希代码。这类方法的文档解释了哈希函数是什么，你应该查看一下。

对于自定义型别，应该由实现来提供相应的哈希函数。`Object` 类提供的默认哈希函数常常根据对象的位置来生成哈希代码，所以在这里“相等”的概念就是浅比较相等。但多数时候在映射中查找一个键我们想做的却不是浅比较。

不管哈希代码是如何生成的，最后一步都是用模和绝对值函数来将哈希代码映射到合法的索引序列上。

## 19.8 缩放哈希映射

先来复习一下。哈希表由一个数组（或向量）中的 `List`（链表）对象组成，其中的每个 `List` 对象包含数目不多的几个记录。要向映射中新增一个记录，就要计算新键的哈希代码并将记录插入到相应的 `List` 中。

要查询一个键，就得将映射再次打散然后查找相应的链表。如果链表的长度是有界的，那么查找时间也是有界的。

那么要如何确保链表的短小精干呢？没错，一种途径就是尽可能设法让所有链表保持平衡，这样一来就不会出现当一些链表是空的时候一些链表却很长。要做到完美是很难的——这依赖于选择的哈希函数是否足够好——不过我们通常可以做的足够好。

即使有了绝对的平衡，链表的平均长度还是和记录数的增长成线性关系增长，必须阻止这种增长势头。

解决办法是跟踪每个链表中平均记录数目，这个量叫做**负荷比**（*load factor*）。如果负荷比过高，就必须修改哈希表的大小（缩放操作）。

为了进行缩放操作要创建一个新的哈希表，一般是原来大小的两倍，然后将原来所有的记录取出，再次打散，然后将这些记录放进新的哈希表中。通常不必使用新的哈希函数，只要用一个新的值来做取模运算就行了。

## 19.9 哈希表调整操作的性能

修改一个哈希表的大小需要多少时长？很明显这和记录数是成线性关系

的。这就是说大多数时候 `put` 方法消耗的是固定时长，但每当修改哈希表大小的片刻里，`put` 消耗的时长就是线性的。

乍一听似乎很糟。这不是等于悄悄地破坏了我前面说的可以在固定时长内执行 `put` 操作的断言？好吧，老实说是这样。不过只要稍微用点心思就可以解决这个问题。

由于有的压入 (`put`) 消耗的时间比别的压入操作长，首先要设法算出抽象的 `put` 操作平均消耗的时长。设这个平均时长为  $c$ ，这是一个简单的 `put` 操作要耗费的固定时长；附加项  $p$  是调整操作占总调整时间的比例， $kn$  是调整操作的耗时。这样就得到如下的计算公式：

$$t(n) = c + p \cdot kn \quad (19.1)$$

我们不知道  $c$  和  $k$  到底是什么、值为多少，但可以弄清楚  $p$  是什么。假设调整哈希映射的操作仅仅是将其扩大一倍，如果有  $n$  项，那么就可以在调整之前增加  $n$  项。这样需要调整的时长所占比例就是  $\frac{1}{n}$ 。

植入上面的式子中就得到

$$t(n) = c + \frac{1}{n} \cdot kn = c + k \quad (19.2)$$

换句话说， $t(n)$  是固定时长的！

## 19.10 术语表

**映射-map**：定义了一组记录上的操作的 ADT。

**记录-entry**：映射中的一个元素，包含一个键和一个值。

**键-key**：一个索引，可以是任何型别，用来在找出映射中的值。

**值-value**：映射中存贮的一个元素，可以是任何型别。

**字典-dictionary**：映射的另一种说法。

**关联数组-associative array**：字典的又一种说法。

**哈希映射-hash map**：映射的一种特别高效的实现。

**哈希函数-hash function**：将一种特定型别的值映射到整数的函数。

**哈希代码-hash code**：和一个给定值对应的整数值。

**移位和-shifted sum**：一个简单的哈希函数，常常用来混和字符串这样的对象。

**负荷比-load factor**：用哈希映射中的记录数除以链表数，换言之，每个链表中的平均记录数。

## 第 20 章 霍夫曼编码

### 20.1 长短编码

如果熟悉莫尔斯电码 (Morse code)，你应该知道它是用一系列点和划为字母表编码的系统。例如有名的...---...信号表示 SOS 这三个字母——构成了国际通用的求救信号。下面的表给出了莫尔斯电码的编码：

A .-	N -. 1 .----	. .-..-
B -...	O --- 2 ..---	, ---..-
C -. .	P -. . 3 ...--	? ..--..
D -..	Q --.- 4 ....-	( -. -. .
E .	R .-. 5 .....	) -. -. .-
F ..-.	S ... 6 -....	- -....-
G --.	T - 7 --...	" .-.-.
H ....	U ..- 8 ---..	_ ..-..-
I ..	V ...- 9 ----.	' .----.
J .---	W .-- 0 -----	: ----..
K -.-	X -.- / -. -. .	; -. -. .
L .-.	Y -. - + .-. .	\$ ...-. -
M --	Z --.. = -.- -	

注意有的编码比别的长。经过设计，最常用的字母有最短的代码。由于短小的代码有限，这就意味着不那么常用的字母和符号有较长的编码。一个典型的信息中的短编码比长编码要多，这就使传送每封信的平均时间得到了最小化。

像这样的编码叫做**长短编码** (*variable-length code*)。在本章中将讨论一个产生长短编码的算法，叫做**霍夫曼编码** (*Huffman code*)。它的确是个有趣的算法，而且也可以作为一个很好的练习，因为它的实现用到了很多我们学习过的数据结构。

下面是接下来几节的一个轮廓描述：

- 首先，我们用一个英文文本的例子来生成一个**频率表** (*frequency table*)。频率表有点像一个柱状图，用来统计文本例子中每个字母出现的次数。

- 霍夫曼编码的核心是**霍夫曼树** (*Huffman tree*)。我们会用频率表来构建一个霍夫曼树，然后用这个树来编码或解码某些序列。

- 最后，遍历霍夫曼树并构建一个编码表，包含每个字母的点和划的序列。

## 20.2 频率表

既然目标是要将最短的代码分配给最常用的字母，就必须知道每个字母出现了多少次。在埃德加·爱伦·坡（Edgar Allen Poe）的小故事“金色的虫子（The Gold Bug）”中，有个角色利用字母的出现频率来破解了一个密码，他解释道：

“现代英语中出现频率最高的字母是 e，其后的次序是：a o i d h n r s t u y c f g l m w b k p q x z。E 的优势如此明显，以致罕有一个句子其中 E 的出现不占有最多的次数。”

那么我们的第一个任务就是看看坡说的对不对。为了检验他的话，我将“The Gold Bug”的文本自身作为示例，这些文本是从一个利用了公共域名之便的网站上下载回来的。

## 20.3 霍夫曼树

接下来就要**装配**（*assemble*）霍夫曼树。树的每个节点包含一个字母和它的出现频率，还有两个分别指向左右节点的**指针**（*pointer*）。

我们从创建一组**孤儿树**（*singleton tree*）开始来装配霍夫曼树，每个孤儿树对应频率表中的一个记录。然后自底向上创建这棵树，从出现频率最低的字母开始反复地加入子树直到获得一棵包含所有字母的树。

下面是这个算法的详细描述：

1. 对频率表中的每个记录，创建一个霍夫曼孤儿树并将这棵树增加到一个优先队列 `PriorityQueue` 中。当我们将一棵树从优先队列中移出的时候就将该树的优先级设置为最低。
2. 从优先队列 `PriorityQueue` 中移出两棵树，通过创建一个新的父节点指向移出的树来将二者连接起来。父节点的频率是两个孩子节点的频率的和。
3. 如果优先队列 `PriorityQueue` 为空，操作即完成。否则，将新树放入优先队列中然后调到第 2 步。

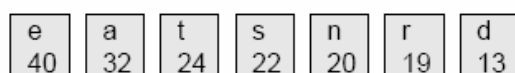
举个例会得将这一过程说明得更清楚。为了方便解说，这里用一个简单的文本，仅包含 `adenrst` 这几个字母：

Eastern Tennessee anteaters ensnare and eat red ants, detest ant antennae (a tart taste) and dread Antarean anteater-eaters. Rare Andean deer eat tender sea reeds, aster seeds and rats' ears. Dessert? Rats' asses.

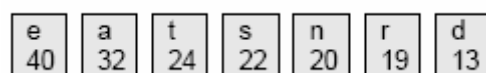
这段提及了数种生物以及它们之间食物链关系的精彩论述产生了如下的频率表：

e 40  
a 32  
t 24  
s 22  
n 20  
r 19  
d 13

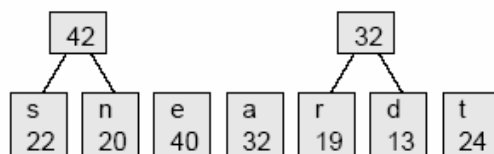
经过第一步之后，优先队列 PriorityQueue 是这样的：



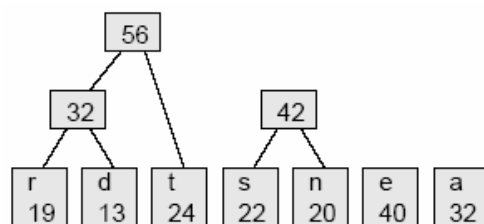
在第 2 步中，把频率最低的两棵树（r 和 d）移出并创建一个频率为 32 的父节点将二者连接起来。节点内是哪个字母并不重要，所以图中忽略了对应的字母。将新树放回 PriorityQueue 之后结果是这样的：



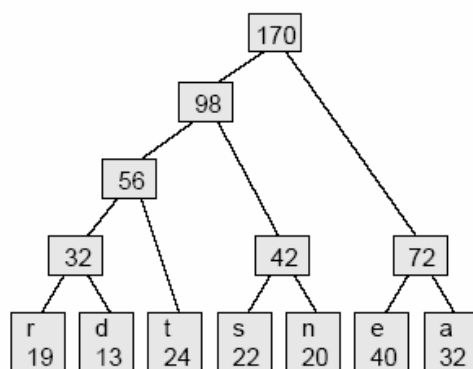
重复这个步骤（迭代），将 s 和 n 连起来：



经过两次迭代之后，得到下面图示的树的集合（*collection of trees*）。顺便提一下，树的集合叫做**森林**（*forest*）。



再经过两次迭代，只剩下一棵树：





这就是示例文本的霍夫曼树。这段文本的霍夫曼树实际上不止这一个，因为每次将两棵进行连接是时候，哪棵在左哪棵在右是任意的，还有当优先队列中存在一个**结**（*tie*）时，选择也是任意的。所以一个简单的例子也有很多可能的树。

那么究竟如何从霍夫曼树中取得编码呢？每个字母的编码取决于从树的根到包含该字母的叶节点的路径。例如，从根节点到 *s* 的路径是左-右-左，如果用 *.* 表示左-表示右（也可以有任意别的选择），就可以得到下面的编码表：

```
e  -.
a  --
t  ..-
s  .-.
n  .--
r  ....
d  ...-
```

注意目标已经达到：最常用的字母编码最短。

## 20.4 super 方法

实现霍夫曼树（下面的 `HuffTree` 类）的办法之一是由扩展 `Exercise 20.13` 中的 `Pair` 方法：

```
public class HuffTree extends Pair implements Comparable {
    HuffTree left, right;
    public HuffTree (int freq, String letter,
                     HuffTree left, HuffTree right) {
        this.freq = freq;
        this.letter = letter;
        this.left = left;
        this.right = right;
    }
}
```

`HuffTree` 类实现了 `Comparable` 元类，所以我们可以将这个霍夫曼树放进一个优先队列中。为了实现 `Comparable`，必须提供一个 `compareTo` 方法。可以从头开始写一个，但简单的办法是利用 `Pair` 类中的 `compareTo` 版本。

不幸的是已有的方法不完全符合要求。对于 `Pair` 类的对象，将高优先级分配给高频率，而对霍夫曼树，要将高优先级分配给低频率。当然，可以写一个

新版本的 `compareTo` 方法，但这会覆盖父类中的版本，而我们更乐意调用父类中的版本。

显然我们不会是第一个遇到这个小小的两难处境（*Catch 22*）的人，因为可爱的 Java 发明人已经提供了解决办法。关键字 `super` 允许调用一个被覆盖掉的方法。因为父类有时也叫做**超类**（*superclass*），所以这种机制也用 `super` 关键字来表示。

下面的代码是我实现的霍夫曼树中的示例：

```
public int compareTo (Object obj) {  
    return -super.compareTo (obj);  
}
```

对一个 `HuffTree` 对象调用 `compareTo` 方法时，该方法会转而调用被覆盖掉版本的 `compareTo`，然后将能导致优先级反转的结果否决掉。

如果一个**孩子类**（*child class*，或叫**子类** *subclass*）还重载了构造器，它还可以用 `super` 关键字调用父类的构造器：

```
public HuffTree (int freq, String letter,  
    HuffTree left, HuffTree right) {  
    super (freq, letter);  
    this.left = left;  
    this.right = right;  
}
```

在本例中，父类的构造器初始化变量 `freq` 和 `letter`，而后子类的构造器初始化 `left` 和 `right`。

虽然这种特性很有用，但也容易导致错误。你会遇到一些奇怪的限制——父类的构造器必须在初始化其它变量前被调用——还有一些你甚至不想知道的牛角尖问题。通常这种机制就像一个急救箱。如果真的遇到麻烦，这个箱子能帮你解决问题。但你知道什么才是更好的？——别陷入麻烦。在本例中，在子类的构造器中初始化所有四个实体变量更为简便。

## 20.5 解码（Decoding）

当收到一条用霍夫曼编码的信息，就用霍夫曼树来解码。算法如下：

1. 从霍夫曼树的根开始。
2. 如果下一个符号是`.`，转到左孩子，否则转到右孩子。
3. 如果处于叶节点就取出节点中的字母并将其添加到结果中。回到根节点。

#### 4. 跳到第 2 步。

以编码`..--`为例。从树的顶部开始，经过左-左-右的路径得到字母 `t`。然后又从根节点开始，经过右-左路径得到字母 `e`。回到顶部，经过右-右得到字母 `a`。如果代码是经过良好格式化的，当代码走完的时候应该处于一个叶节点上。这里这条信息是我偏好的饮料——`tea`（茶）。

## 20.6 编码 Encoding

从某种程度上讲，编码一条信息比解码难，因为对给定的字母不得不在树上查找包含该字母的叶节点，然后找出从根到该叶节点的路径。

如果采取以下措施这个过程可以更加高效：对树进行一次遍历，计算所有的节点，并创建一个字母到编码的映射。

到现在为止我们见过了足够多的树的遍历，但这次有点不一样，因为当我们在整棵树里移动的时候，需要跟踪记录经过的路径。一开始这看起来可能较难，但有一种自然的办法递归地执行这个计算。经过仔细观察发现：如果从根节点到一个给定节点的路径（`path`）被表示为一个点和划的字符串，那么到它的左孩子节点的路径就是 `path + '-'` 而到它的右孩子节点的路径是 `path + '.'`。

## 20.7 术语表

**森林-forest**：树的集合（毫无新意！）

**两难-Catch 22**：问题的一个显而易见的解决方案不可避免地妨害到其它的解决方案。这个词汇来自 Joseph Heller 的同名小说（由 Stanley Kubrick 导演了同名电影）。

**超类-superclass**：父类的另一种叫法。

**子类-subclass**：孩子类的另一种叫法。

**super**：用于覆盖父类方法的关键字。

## 附录 A 程序开发计划

如果花费了大量的时间在调试上，很可能是因为没有一个有效的程序开发计划。

一个典型的不好的程序开发计划就像这样：

1. 编写一个完整的方法。
2. 编写更多的方法。
3. 编译程序。
4. 花一个小时来找语法错误。
5. 花一个小时来找运行时错误。
6. 花三个小时来找语义错误。

显而易见，问题出在头两步。如果写了一个方法甚至很多方法都不调试，那么得到的代码可能已经多得让你无法调试了。

如果遇到这种情况，唯一的解决办法就是删除代码直到再次获得一个可以工作的程序，然后再慢慢将程序增加回来。编程新手往往不希望这么干，因为他们精心编写的代码实在是太宝贝了。可是为了高效的进行调试，你不得不残忍起来！

下面是一个较好的程序开发计划：

1. 从一个能做一些直观事情（比如打印一些东西）的程序开始。
2. 每次增加少许几行代码，并且每次改动都测试程序是否正确。
3. 重复前两步直到程序满足预期的要求。

每次改动后的程序都应该产生一些验证新添代码的可见效果。这种编程方式能节省许多时间。因为一次只增加少许几行代码，所以容易发现语法错误；程序的每个版本产生一些可见的结果，这就使你能不断测试自己头脑中关于程序是如何工作的模型。如果头脑中的模型是错误的，在写出一大堆错误代码之前你将面对矛盾（并且也有了改正的机会）。

这种方式的问题是常常难于找出下手的地方并得到一个完整正确的程序。

我将通过开发一个名为 `isIn` 的方法来演示这种方式。这个方法取一个字符串和一个字符为参数，返回一个布尔值：如果字符出现在字符串中就返回 `true`，否则返回 `false`。

1. 第一步，写一个尽量短但可以编译、运行并做一些可见的事情的方法：

```
public static boolean isIn (char c, String s) {  
    System.out.println ("isIn");  
    return false;  
}
```

当然，要测试这个方法就要调用它，需要在 `main` 方法或在一个正常工作的程序中什么地方创建一个简单的测试用例。

先来看一个字符串中出现了字符的用例（期望得到的结果是 `true`）：

```
public static void main (String[] args) {  
    boolean test = isIn ('n', "banana");  
    System.out.println (test);  
}
```

如果一切按照计划进行，代码将顺利编译、运行然后打印单词 `isIn` 和值 `false`。当然，答案是不对的，但目前我们知道方法被调用并且返回了值。

在个人的编程生涯里，我浪费了太多太多的时间调试某个方法，结果却发现它根本没有被调用。如果当时我采用这种开发方式，这种事情是不应该发生的。

## 2. 第二步，检查方法取得的参数：

```
public static boolean isIn (char c, String s) {  
    System.out.println ("isIn looking for " + c);  
    System.out.println ("in the String " + s);  
    return false;  
}
```

第一个打印语句允许我们确认 `isIn` 方法找的是正确的字母，第二句用来确认找的是正确的位置。

现在输出是：

```
isIn looking for b  
in the String banana
```

既然已经知道它们的作用，再打印参数似乎有点傻。关键在于确认它们是否和我们设想的一样。

**3.** 为了遍历字符串，可以利用 7.3 节的代码。一般来说，重用代码片断比全部从头开始更好。

```
public static boolean isIn (char c, String s) {  
    System.out.println ("isIn looking for " + c);  
    System.out.println ("in the String " + s);  
    int index = 0;  
    while (index < s.length()) {  
        char letter = s.charAt (index);  
        System.out.println (letter);  
        index = index + 1;  
    }
```

```
    }  
    return false;  
}
```

现在运行这个程序，它将一次打印字符串中的一个字符。如果一切进展良好，可以确定这个循环检测了字符串中的每个字母。

**4.** 到这里还没有充分思考过这个方法到底要做什么，此时我们最需要的可能就是找到一种算法。最简单的算法是一个线性查找，即遍历向量并将每个元素和目标键进行比较。

令人愉快的是前面已经写过了遍历向量的代码。和以往一样，每次增加几行：

```
public static boolean isIn (char c, String s) {  
    System.out.println ("isIn looking for " + c);  
    System.out.println ("in the String " + s);  
    int index = 0;  
    while (index < s.length()) {  
        char letter = s.charAt (index);  
        System.out.println (letter);  
        if (letter == c) {  
            System.out.println ("found it");  
        }  
        index = index + 1;  
    }  
    return false;  
}
```

遍历字符串的时候，将每一个字母和目标键做比较。如果找到了目标，就打印一些信息，这样我们就能知道新增代码执行的时候产生了一些可见的效果。

**5.** 现在已经很接近正确工作的代码了。如果找到了要找的内容那么下一个改动是要从方法中返回：

```
public static boolean isIn (char c, String s) {  
    System.out.println ("isIn looking for " + c);  
    System.out.println ("in the String " + s);  
    int index = 0;  
    while (index < s.length()) {  
        char letter = s.charAt (index);  
        System.out.println (letter);
```

```
        if (letter == c) {  
            System.out.println ("found it");  
            return true;  
        }  
        index = index + 1;  
    }  
    return false;  
}
```

如果找到了目标字符，返回 `true`；如果经历了整个循环也没有找到，正确的返回值就应该是 `false`。

现在运行程序应该得到：

```
isIn looking for n  
in the String banana  
b  
a  
n  
found it  
true
```

**6.** 下一步是要确认别的测试用例能正确的工作。首先应该确认如果字符没有在字符串中则方法返回 `false`。然后要检查一些典型的容易招致麻烦的情况，例如空字符串`""`，或者只有一个字符的字符串。

一般说来这种测试可以帮我们找到存在的 **bug**，但不能判断方法是否正确。

**7.** 倒数第二步要移出或注释掉打印语句。

```
public static boolean isIn (char c, String s) {  
    int index = 0;  
    while (index < s.length()) {  
        char letter = s.charAt (index);  
        if (letter == c) {  
            return true;  
        }  
        index = index + 1;  
    }  
    return false;  
}
```

如果稍后还要查看这个方法，那么将打印语句注释掉是一个好主意。但如

果这个方法是最终版本并且你能确信它是正确无误的，就可以移出这些打印语句了。

移出注释可以让代码更加干净，也有助于发现遗留的问题。

如果代码的用意并不是很明显，就应该增加注释来解释清除。要抵制逐行翻译代码的诱惑。例如，这样做是毫无必要的：

```
// if letter equals c, return true
if (letter == c) {
    return true;
}
```

注释应该用来解释含义不明显的代码，提示容易导致错误的情况和说明包含在代码中的假设。还有，在每个方法之前给出该方法的用途也是一个很好的做法。

**8.** 最后一步是检测代码并确认它是正确的。在这里我们知道方法的语法是正确的，因为编译顺利通过。要检查运行时错误，只有找出每个可能导致错误的语句和条件。

在这方法中唯一能导致运行时错误的语句是 `s.charAt (index)`。如果 `s` 是 `null` 或者索引超越了边界那么这条语句就将失败。因为 `s` 是一个参数，就不能确保它不是 `null`，所以只有检查。一般说来方法最好都要确认参数的合法性。`while` 循环的结构保证了 `index` 总是在 0 到 `s.length-1` 之间。如果检查全部有问题的条件，或者证明这些条件不可能发生，那么就证明了方法不会导致运行时错误。

我们还没有证明这个方法的语义是正确的，但在逐步递增的过程中，避免了很多可能的错误。例如已经知道方法能正确取得参、循环遍历了整个字符串。我们还知道这个方法成功的比较了字符，如果目标在字符串中就返回 `true`。最后我们知道，如果循环存在就表明目标不在字符串中。

在没有正式证明的情况下，这可能是我们能做到的最好情况。



## 附录 B 调试

程序中可能出现的错误不过数种，区别加以对待能让调试更加迅速。

● 编译时错误是编译器产生的错误，通常表明程序的语法有错误，比如忽略了语句结尾的分号。

● 运行时错误由运行时系统产生，表明程序运行的时候出了错。大多数运行时错误是一些异常，例如一个无穷递归最后将导致一个 `StackOverflowException` 异常。

● 语义错误是这样的：程序既能编译也能运行，但做的事情却不是该做的。比如一个表达式没有按照你预料的顺序执行于是得到了一个不符合期望的结果。

调试的第一步是找出要处理的是哪种错误。虽然接下来的几节按照错误类型来安排，但有些技术对不止一种情形适用。

### B.1 运行时错误

最好的调试是不调试，因为一开始就避免了错误。在附录 A 中给出了一种最小化错误并使查错很容易的程序开发计划。关键在于从一个正确工作的程序开始每次增加一小点代码。这样一来一旦出现错误也很容易想到错在哪里。

虽然如此，你可能会遇到下面一些情形的一种。对每种情况我给出一些处理建议。

#### 编译器给出错误信息

如果编译器报告了 100 条错误信息，那并不意味着程序中有 100 个错误。当编译器遇到一个错误的时候，往往离开执行流程一会儿。它在遇到第一个错误之后试图纠正并继续下去，但这种尝试常常失败，于是就报告错误的信息。

一般来说，只有第一条出错信息是可靠的。我的建议是每次只改正一个错误，然后重新编译程序。你可能会发现一个分号就“改正”了 100 个错误。当然，如果看到数条合乎逻辑的错误信息，每次编译前也可以改正多个 bug。

#### 得到一条古怪的编译信息而且怎么也无法避免

首先，仔细阅读出错信息。其中尽是精练的行话，但常常有一个被精心隐藏起来的核心信息。

如果不出意外，这条信息会告诉你问题出在程序的哪个地方。准确地说，该信息告诉你发现问题时编译器编译到了何处，而错误出在哪里却并非要点。用编译器给的信息做为指示，但如果在编译器指向的地方没有发现错误，就扩

大查错的范围。

一般来说错误位于出错信息指向位置的前面，但有时也可能完全在别的地方。举个例，如果调用方法时得到一条错误信息，真实的错误可能是在方法定义中。

如果采用步增法开发程序，你应该很容易想到出错的位置在哪里——就在你增加的最后一行代码。

如果你从一本书里抄代码，首先要非常仔细的将抄下来的代码和书中的进行比较。检查每一个字符。同时，记住书中也可能有错，所以如果发现语法错误，那极可能就是。

如果没有很快地发现错误，做一个深呼吸，然后把眼光放宽点看看整个程序。首先看看程序缩进是否恰当，不能说好的缩进一定能使查找语法错误变得容易，但坏的缩进一定会使之更困难。

现在，开始检查代码中常见的语法错误。

**1.**检查所有的圆括号和各种括号是否成对并且恰当的进行了嵌套。所有方法定义都应该嵌套在一个类定义中。所有的程序语句都应该在一个方法定义中。

**2.**记住大写字母和小写字母是不一样的。

**3.**检查确保分号在语句结尾（而且花括弧后面没有分号）。

**4.**确保代码中的所有字符串被配对的引号围起来。确保字符串用双引号，字符用单引号。

**5.**对每个赋值语句，确保左边的型别和右边一样。确保左边的表达式是一个变量名或其它可以为之赋值的表达式（比如一个数组的元素）。

**6.**对每个方法调用，确保提供参数是按正确的次序和正确的型别，并且调用该方法的对象是正确的型别。

**7.**如果调用的是一个带返回值的方法，确认返回的值是有用处的。

**8.**如果调用一个对象方法，确认调用的对象是正确的型别。如果不是从定义类方法的地方调用该方法，确保指明类名。

**9.**在对象方法内部可以不指明对象而直接引用实体变量。如果在类方法中这么做，将得到一条让人困惑的信息，类似“静态引用指向非静态变量”之类的话。

如果上述步骤都没有用，那么看下一节吧...

## 无论怎么做都不能使程序通过编译

如果编译器说有错但是你却没有找到这个错误，可能是因为你和编译器查看的不是同样的代码。检查开发环境以确认你正在编辑的程序和编译器在编译的是同一个程序。如果不是很确定，试试在程序的开头故意放一个语法错误。再次编译，如果编译器不能发现新增的错误，那么你建立工程的方式可能有问

题。

否则，如果你已经彻底检查了代码，那就只有孤注一掷了。从一个可以编译通过的程序开始，逐渐的将代码加回去。

- 将正在处理的文件做一个备份。如果正在调试 *Fred.java*，做一个备份叫做 *Fred.java.old*。

- 从 *Fred.java* 中删掉一半代码，再试着编译一次。

- 如果程序通过了编译，就可以推断出错误出在另一半中。将删掉的代码取一半回来加到正确的代码中，再编译一次。

- 如果程序还是不能通过编译，错误必定就在增加回来的这段代码中。将这段代码删掉一半再编译一次。

- 一旦找到并修正了错误，可以逐步将删掉的代码加回来，一次一小点。

这种做法叫做“二分调试”。还有一个办法，不用删除代码，只要将大块的代码注释掉就行了。但我认为删除更可靠——不必担心注释的语法，而且程序变小了更易于阅读；相反的，注释却有许多想当麻烦的语法问题。

## 照编译器说的去做，但没有用

有的编译器信息给了一些零碎的建议，比如：“*Golfer* 类必须声明为抽象的。它没有定义 `java.lang.Comparable` 接口中的 `int` 型别方法 `compareTo(java.lang.Object)`。”听上去编译器是告诉你应该把 *Golfer* 声明为一个抽象类，但是如果你对 Java 的了解仅限于本书，那你很可能既不知道什么这句话的意思也不知道怎么做。

那你走运了，编译器是错的。遇到这种情况的解决办法是确认 *Golfer* 类是否有一个名为 `compareTo` 的方法取一个 `Object` 做参数。

一般来说我们不能让编译器牵着鼻子走。出错信息只是表明什么地方出了错，但可能产生误导，其“建议”通常是错误的。

## B.2 运行时错误

### 程序卡住了

如果一个程序停下来什么也不做，就称为“卡住了 (hang)”。通常这意味着程序陷入了一个无穷循环或无穷递归中。

- 如果找到问题可能存在的某个循环，立即在循环之前和之后分别增加一个打印语句输出“进入 (或退出) 循环”。运行程序，如果只得到第一条信息而没有得到第二条，那就是遇到无穷循环了。请参考“无穷循环”一节。

- 大多数时候无穷递归将导致程序运行一会儿然后就产生一个

`StackOverflowException` 异常。如果这种情况发生，请参考“无穷递归”一节。

如果没有得到一个 `StackOverflowException` 异常，但又怀疑递归方法有问题，那么还是可以用无穷递归一节中讲的技术。

- 如果该节中讲的方法都不起作用，那就测试别的循环和递归方法。

- 如果上述建议都没有帮助，可能是你没有弄懂程序的执行流程。请参考“执行流程”一节。

## 无穷循环

如果程序中有无穷循环并且你认为自己知道是什么循环导致了问题，那就在循环最后增加一条打印语句来打印条件变量的值和条件的值。

例如：

```
while (x > 0 && y < 0) {  
    // do something to x  
    // do something to y  
    System.out.println ("x: " + x);  
    System.out.println ("y: " + y);  
    System.out.println ("condition: " + (x > 0 && y < 0));  
}
```

现在运行程序，每次循环会看到三行输出。在最后一次循环中，条件将变为 `false`。如果循环继续进行，就可以看见 `x` 和 `y` 的值，也就可以弄清楚它们为什么没正确的更新。

## 无穷递归

大多数时候无穷递归会导致程序运行一会儿然后产生一个 `StackOverflowException` 异常。

如果知道有一个方法导致了无穷递归，首先要做的是确认是否存在一个基本情形（`base case`）。换句话说，应该有些条件下方法不做递归调用就返回了。如果没有基本情形，就需要重新审视算法并明确基本情形。

如果有基本情形但程序不能抵达，那么在方法的开头增加一个打印语句来打印参数。现在运行程序，每次方法被调用将会有几行输出，可以看到参数的值。如果参数没有逐步接近基本情形，你也能获得一些为什么不能到达基本情形的原因。

## 执行流程

如果不能确定程序中的执行流程是如何运作的，那就在每个方法的前面增加一条打印语句，打印的内容类似“进入方法 `foo`”，意思是该打印语句之后执

行的方法名称为 `foo`。

现在运行程序将跟踪打印出每个被调用的方法。

每个方法被调用时都将其取得的参数打印出来是很有用的。当运行程序时，检查参数是否合理，看看是否有一个典型的错误——以错误的次序提供了参数。

## 运行程序时得到一个异常

如果运行时出现什么错误，Java 运行时系统会打印一条包含异常名称的信息，还有问题发生的程序所在的行以及一个跟踪堆栈。

跟踪堆栈包括当前运行的方法，以及调用该方法的方法，以及调用这个调用方法的方法，依次类推。换句话说，跟踪堆栈跟随记录了异常发生之前的所有方法调用。

第一步就要检查程序中发生错误的地方，看看能否弄明白到底发生了什么。

- `NullPointerException` 异常：试图访问一个 `null` 对象的实体变量或对其调用方法。这时应该找出哪个变量的值是 `null` 并弄清楚它是如何称为 `null` 的。请记住当声明一个对象型别的变量时，在被赋值之前初始化是 `null` 对象。例如这段代码就会导致一个 `NullPointerException` 异常：

```
Point blank;  
System.out.println (blank.x);
```

- `ArrayIndexOutOfBoundsException` 异常：用来访问数组的索引号是负的或者大于 `array.length-1`。如果找到问题所在，立即在前面增加一条打印语句将索引号的值和数组长度打印出来。数组的大小正确吗？索引号的值正确吗？现在重新看一遍程序前面的部分找出数组和索引号来自何处。再找出最近的赋值语句看看是否做了正确的事情。如果索引和数组大小都没有参数，那就到方法被调用的地方看看值是从哪里来的。
- `StackOverflowException`：参见“无穷递归”一节。

## 增加了许多打印语句，输出太多

使用打印语句给调试带来的问题之一就是被输出淹没掉。有两种办法继续下去：简化输出或简化程序。

要简化输出，可以将没有帮助的打印语句移出或注释掉，或者将这些语句组合起来，或者将它们的输出格式统一起来便于理解。开发程序的时候，应该构思能够让程序执行流程更加可见的办法，并开发代码来生成简洁、含义丰富的可见结果。

为了简化程序，有几件事情要做。首先，缩小程序正在处理的问题。举个

例，将一个数组排序，一个名为 `small` 的数组。如果程序从用户处取得输入，那就输入一个能导致错误的最简单的数据。

然后，清理一下程序。移出无用的代码并使程序尽可能的易读。举个例，当你假设错误处在程序中一个深度嵌套的内部，那就试试用简单一些的结构重写这个部分。如果怀疑一个庞大的方法有错，就试试将它分割写成一些较小的方法然后分别进行测试。

往往寻找最小测试用例的过程能引导你找出 `bug`。举个例，如果发现当数组有偶数个元素时能正常工作而元素数目为奇数时则不能，这就为找出问题的原因提供了一个线索。

类似地，重写一段代码能帮助你找出一个微妙的 `bug`。如果做一个你认为不会影响程序的改动却产生了影响，这已经向你泄露了错误的秘密所在。

## B.3 语义错误

### 程序不能正确工作

从某种意义上讲语义错误最难解决，因为编译器和运行系统都不会对错在何处给出任何信息。只有你知道程序应该做什么，也只有你知道它没有做应该做的事情。

第一步要找出程序文本和你看见的行为之间的联系。需要假设程序实际做的是什么事情。难于修改语义错误的原因之一是计算机运行太快。你常常希望能将程序放慢到符合人的速度，但是没有这么简单的途径，而且即使有，也实在不是调试的好办法。

请问你自己几个问题。

- 有没有程序应该做但是似乎没有发生的事情？找到执行这些功能的代码片断，确认当你认为它应该工作时是否真的执行。在有疑点的方法前面加一条打印语句。

- 有没有发生一些不该发生的事情？找到执行这些功能的代码片断，看看当它不该工作时是否被执行。

- 有没有一段代码产生了一个你不希望的作用？确认你自己明白问题代码的意思，特别是当这段代码涉及到调用 `Java` 内建方法的时候，要阅读所调用方法的文档。通过直接用简单的测试用例来调用该内建方法，并检查结果。

要编程，头脑中就得有一个关于程序如何工作的模型。如果程序没有做你期望的事情，往往问题不在程序中，而在你脑中的模型里。

修正脑中模型的最好办法是将程序打散成组成它的部件（通常是类和方法），然后单独测试每个部件。一旦找到模型和实现的矛盾之处，就能解决问题。

当然，开发程序的时候就该右边构造部件一遍进行测试。这样即使遇到问题，也只是为数不多的新增代码的某部分需要改正。

下面是一些应该注意检查的常见语义错误：

●如果使用赋值运算符=替代了等号==，在 if、while 或 for 语句中，将得到一个语法正确但做的事情与期望不一致的表达式。

●如果将等号==用在一个对象上，它将进行浅比较。如果你的本意是要做深比较，就应该用 equals 方法（或者为自定义的对象定义一个用来进行比较运算的方法）。

●有的 Java 库要求自定义对象定义 equals 这样的方法。如果不进行定义，将从父类中集成默认行为，而这很可能不是你所想要的。

●继承可能导致微妙的语义错误，因为你可能无意识地执行了一些继承得来的代码。要确认自己是否能理解程序的执行流程，请参见“执行流程”一节。

## 一个又臭又长的表达式不按预期的工作

只要还可以读懂，一个复杂表达式完全可以写的尽量长，不过可能难以调试。将一个复杂表达式打散为一些相关的赋值语句和临时变量是一个不错的主意。

例如：

```
rect.setLocation(rect.getLocation().translate  
                (-rect.getWidth(), -rect.getHeight()));
```

可以写成

```
int dx = -rect.getWidth();  
int dy = -rect.getHeight();  
Point location = rect.getLocation();  
Point newLocation = location.translate (dx, dy);  
rect.setLocation (newLocation);
```

较详尽的版本也较易阅读，因为变量名提供了一些额外的说明；详尽的版本也更易调试，因为可以检查中间变量的型别并显示它们的值。

庞大的表达式可能产生的另一个问题是其执行次序也许和你期望的不一样。举个例，如果将  $\frac{x}{2\pi}$  这个式子翻译成 Java 中的相应形式，你可能会这么写：

```
double y = x / 2 * Math.PI;
```

然而这并不对，因为乘法和除法的优先级是一样的，执行时将从左到右来。

所以上面的表达式实际是  $\frac{x\pi}{2}$ 。

调试表达式的一个好办法是增加括号以使执行顺序更加清晰。

```
double y = x / (2 * Math.PI);
```

任何时候如果不清楚运算顺序就用括号。不仅程序不会出错（从按照你的要求来说），而且对不记得优先级的人来说也更具有可读性。

## 一个方法没有返回期望的值

如果有一条含有复杂表达式的返回语句，那就没有机会在返回之前打印该值。临时变量再次派上了用场。举个例，下面的代码

```
public Rectangle intersection (Rectangle a, Rectangle b) {  
    return new Rectangle (  
        Math.min (a.x, b.x),  
        Math.min (a.y, b.y),  
        Math.max (a.x+a.width, b.x+b.width)  
            -Math.min (a.x, b.x),  
        Math.max (a.y+a.height, b.y+b.height)  
            -Math.min (a.y, b.y) );  
}
```

可以写成

```
public Rectangle intersection (Rectangle a, Rectangle b) {  
    int x1 = Math.min (a.x, b.x);  
    int y2 = Math.min (a.y, b.y);  
    int x2 = Math.max (a.x+a.width, b.x+b.width);  
    int y2 = Math.max (a.y+a.height, b.y+b.height);  
    Rectangle rect = new Rectangle (x1, y1, x2-x1, y2-y1);  
    return rect;  
}
```

这次有了在返回前显示中间变量的可能。

## 打印语句什么也没有做

如果用 `println` 方法，输出立即就会被显示出来，但如果用的是 `print`，在有的环境中输出会被存贮起来直到有了换行字符才显示。如果程序没有产生换行字符就终止，将看不到存贮的输出。

如果怀疑发生了这种情况，试试把部分或全部 `print` 语句都改成 `println`。

## 实在被套牢了，需要帮助

首先，试试离开计算机几分钟。计算机放射出影响大脑的波导致以下症状：

- 沮丧和/或盛怒。
- 迷信和神秘的想法（例如“计算机讨厌我”或者“只有我反戴帽子的时



候程序才能正常工作”）。

● 不管三七二十一就写一大堆程序(试图写出每种可能的程序并从中选出正确的一个)。

如果你发现自己有上述症状中的任何一种，那就起来出去走走吧。等你冷静下来，再考虑程序。它在做什么？是什么引起了这种行为？最近什么时候的程序是正常工作的、之后做了什么修改吗？

找到一个 bug 往往要的只是时间。有时当我离开计算机让大脑散散步却找到了一些 bug。坐地铁，淋浴，还有睡前躺在床上，这些都是找到 bug 的好去处。

## 真的需要帮助

这种情况是可能的。即使最好的程序员有时也会被套牢。有时你在一个程序上思考太久以致于看不见错误。这时候需要的正是一双清醒的眼睛。

在找别人之前，请确认已经用尽了前面描述的技术。程序应该尽可能的简单，应该用能引起错误的最小输入。在适当的地方应该放上打印语句（其输出结果应该是可以理解的）。对问题应该有足够充分的理解以便简明扼要的描述它。

当找来一个帮忙的人时，务必给予他需要的信息。

● 什么类型的 bug？编译时、运行时还是语义错误？

● 如果 bug 发生在编译时或运行时，出错信息是什么、这些信息指出的是程序的哪部分？

● 在错误发生之前你最后做了些什么？你写的最后一行代码是什么？最近失败的测试是什么？

● 到请他之前你尝试了一些什么办法、从中得到了什么结论？

你常常会发现在向别人解释问题的时候你自己就看到了答案。这种现象如此普遍，以致于有人提出了一种叫做“橡皮鸭”的调试技术。下面是做法：

1.买一只符合流行标准的橡皮鸭。

2.当你确实被一个问题套牢的时候，把橡皮鸭拿出来放在面前的桌子上对它说：“橡皮鸭，我被一个问题缠住了。事情是这样的……”

3.给橡皮鸭解释这个问题。

4.看看解决方案。

5.谢谢橡皮鸭。

## 找到 bug 了！

大多数时候，找到一个 bug 之后怎样修正就是很明显的了。但并非总是如此。有时一个 bug 也许表明你不甚理解程序，或者你的算法有问题。这种时候，

必须重新考虑算法，或者调整头脑中关于程序的模型。离开计算机想想程序，心算一些测试用例，或者画一张图来表示计算。

当你修正了一个 **bug**，别只顾着立即又一头扎进程序然后开始犯新的错误。花点时间来想想这个 **bug** 是什么类型、第一次是为什么犯这种错误的？这种错误的表现是什么、应该怎么做以更快的找到这种错误？下次遇到类似的问题，你将能更加迅速的找到这种 **bug**。

## 附录 C Java 中的输入/输出

### System 对象

`System` 是一个内建类的名称，它包含了用来从键盘取得输入、向屏幕打印文本以及进行文件输入/输出（I/O）的方法和对象。

`System.out` 对象用来在屏幕上显示文本。当调用 `print` 和 `println` 方法的时候，实际是对 `System.out` 对象调用了这两个方法。

有趣的是，`System.out` 对象也可以打印出来：

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

输出为：

```
java.io.PrintStream@80cc0e5
```

和已往一样，Java 打印一个对象的时候，打印内容包括对象的型别（`PrintStream`）、定义型别的包（`java.io`）和对象的唯一标识码。在我的机器上这个标识码是 `80cc0e5`，但如果你运行同样的代码，可能将得到不同的结果。

同样，也有一个名为 `System.in` 的对象，型别为 `BufferedInputStream`。`System.in` 对象使从键盘获取输入成为可能，不幸的是，它没有使从键盘获得输入变容易。

### 键盘输入

首先必须用 `System.in` 对象来创建一个新的 `InputStreamReader` 对象：

```
InputStreamReader in = new InputStreamReader (System.in);
```

然后用 `in` 对象创建一个新的 `BufferedReader` 对象：

```
BufferedReader keyboard = new BufferedReader (in);
```

这个操作的关键是一个能对 `BufferedReader` 对象调用的方法，名为 `readLine`，从键盘获得一个输入并将其转换为一个字符串，例如：

```
String s = keyboard.readLine ();
```

```
System.out.println (s);
```

从键盘读取一行输入并打印结果。

只是有一个问题。调用 `readLine` 方法时可能出错，将导致一个 `IOException` 异常。Java 中对可能导致异常的方法有一个处理规则。语法是这样的：

```
public static void main (String[] args) throws IOException {  
    // body of main  
}
```

这表明 `main` 方法可能“抛出（throw）”一个 `IOException` 异常。你可以把抛出异常想成好似发脾气。

## 文件输入

从一个文件读取输入和从键盘取得输入同样傻气。下面是一个例子：

```
public static void main (String[] args)
    throws FileNotFoundException, IOException {
    processFile ("/usr/dict/words");
}

public static void processFile (String filename)
    throws FileNotFoundException, IOException {
    FileReader fileReader = new FileReader (filename);
    BufferedReader in = new BufferedReader (fileReader);
    while (true) {
        String s = in.readLine();
        if (s == null) break;
        System.out.println (s);
    }
}
```

这个程序将指定文件（/usr/dict/words）中的每一行读入一个字符串并将它打印出来。声明又一次抛出了 `FileNotFoundException` 异常，编译器要求必须有 `IOException` 异常。对象型别 `FileReader` 和 `BufferedReader` 是 Java 的类体系的一部分，真不敢相信 Java 有这么疯狂的类体系仅仅是用来做些平常简单的事情。除了这么叹上一口气，上面这段凌乱的代码的工作细节实在没有什么研究价值。

## 附录 D Graphics 类

### D.1 Slate 和 Graphics 对象

Java 中有许多途径可以创建图形，有的复杂有的简单。为了简化讨论的手续，我创建了一个对象型别名为 **Slate**（石板），代表一个可以在其上画画的表面。当创建一个 **Slate** 的时候，会出现一个新的空窗口。**Slate** 对象包含一个 **Graphics** 对象，用来在石板上画画。

属于 **Graphics** 对象的方法已经被定义在内建的 **Graphics** 类中。属于 **Slate** 对象的方法则定义在 **Slate** 类中，这将会在 **Section D.6** 给出。

用 **new** 运算符创建一个新的 **Slate** 对象：

```
Slate slate = new Slate (500, 500);
```

参数是窗口的宽和高。返回值被赋与一个名为 **slate** 的变量。在类名（以大写的“**S**”开头）和变量名（以小写的“**s**”开头）之间没有矛盾。

接下来需要一个方法 **getSlateGraphics**，用来返回一个 **Graphics** 对象。可以把 **Graphics** 对象想像成一只粉笔。

```
Graphics g = slate.getSlateGraphics ();
```

用 **g** 做变量名纯粹为了方便，实际上可以用任何名称。

### D.2 对一个 Graphics 对象调用方法

为了在屏幕上画画，就要对 **Graphics** 型别的对象调用方法。

```
g.setColor (Color.black);
```

**setColor** 方法改变当前颜色，这里是变为黑色。除非再次调用 **setColor** 方法，所有画出来的东西都是黑色的。

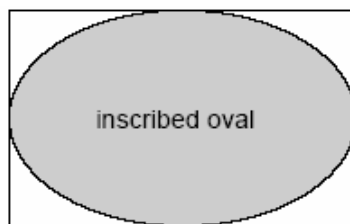
**Color.black** 是由类 **Color** 提供的一个特殊值，正如 **Math.PI** 是由类 **Math** 提供的特殊值一样。下面你会高兴地看到类 **Color** 像调色板一样提供了一些现成的颜色，包括：

```
black    blue    cyan    darkGray    gray    lightGray  
magenta  orange  pink    red        white   yellow
```

要在 **Slate** 上画画，可以对 **Graphics** 型别的对象调用 **draw** 方法，比如：

```
g.drawOval (x, y, width, height);
```

**drawOval** 方法取四个整数做为参数。这几个参数描述了一个方框——一个用来在其中的画椭圆的矩形（如图所示）。方框本身不画出来，只有椭圆被画出来了。方框类似辅助线，总是水平或者垂直方向，不会是东倒西歪的。

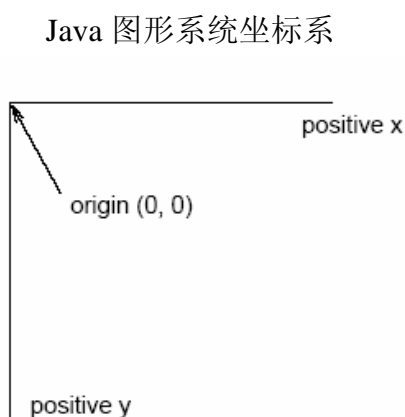
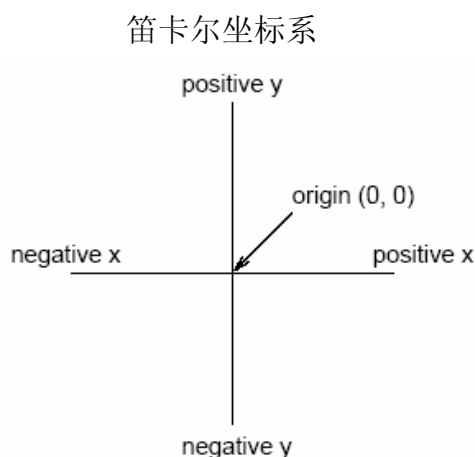


想想看，描述矩形位置和大小有很多办法。有了高和宽，可以给出中心的位置或者任意一个角的位置。或者还可以给出两个对角的位置。选择是随意的，不过任何一种情况需要知道的参数都是一样多：四个。

一般来说，描述一个方框最常用的办法是给出左上角的位置和长与宽。而常用于表示位置的方法是坐标系。

### D.3 坐标

你可能早已熟悉二维的笛卡尔坐标，其中的每一个点都由一个  $x$  坐标（离  $x$  轴的距离）和一个  $y$  坐标（离  $y$  轴的距离）来确定。一般来说笛卡尔坐标向右方和下方增大，如图所示：



烦人的是，一般来说计算机图形系统用的是笛卡尔坐标的变种，其中的原点在屏幕或窗口的左上角，而且  $y$  轴的正轴向下。Java 也遵循这种约俗。

计量的单位叫做像素 (*pixel*)，一个典型的屏幕大约有 1000 像素宽。坐标一般是整数，如果想用一个浮点数的值做为坐标，必须将其舍入为整数（参见 3.2 节）。

### D.4 未完成的米老鼠

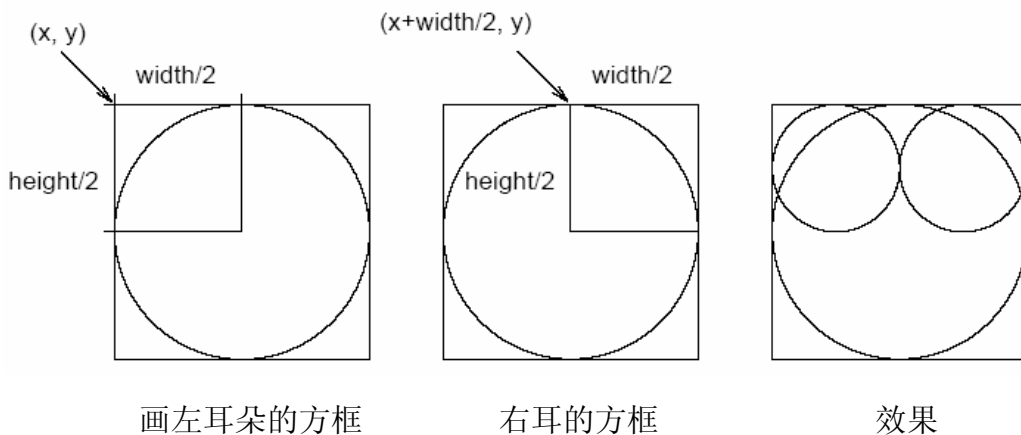
假设要画一张米老鼠的图，我们可以用刚刚画的椭圆做脸，然后再增加耳朵。开始动手前最好将程序分为两个方法。main 方法创建一个 `Slate` 对象和一个 `Graphics` 对象，并调用 `draw` 方法，`draw` 方法才真正的进行绘画。

调用了 `draw` 方法之后, 必须再调用 `slate.repaint` 方法使改动呈现在屏幕上:

```
public static void main (String[] args) {
    int width = 500;
    int height = 500;
    Slate slate = Slate.makeSlate (width, height);
    Graphics g = Slate.getGraphics (slate);
    g.setColor (Color.black);
    draw (g, 0, 0, width, height);
    slate.repaint ();
}

public static void draw
    (Graphics g, int x, int y, int width, int height) {
    g.drawOval (x, y, width, height);
    g.drawOval (x, y, width/2, height/2);
    g.drawOval (x+width/2, y, width/2, height/2);
}
```

`draw` 方法的参数是 `Graphics` 对象和方框, 为了画米老鼠的脸和两只耳朵, `draw` 三次调用了 `drawOval` 方法。下图示意了画耳朵的方框:



如图所示, 左上角用来画左边耳朵的方框的坐标是  $(x, y)$ 。用来画右边耳朵的是  $(x+width/2, y)$ 。两种情况下耳朵的宽和高都是原方框的一半。

注意, 画耳朵的两个方框和原来 (画脸) 的方框的位置 ( $x$  和  $y$ ) 和大小 (宽和高) 都密切相关。因此, 可以用 `draw` 方法在屏幕上画一个任意大小的米老鼠 (不过没有完成)。

## D.5 其它绘图命令

和 `drawOval` 方法有同样参数的另一个方法是

```
drawRect (int x, int y, int width, int height)
```

这里我用一种标准的格式来说明方法的名称和参数。这些信息有时叫做方法的接口或**原型** (*prototype*)。看到原型就能知道参数的型别和用处 (根据参数名)。下面是另一个例子:

```
drawLine (int x1, int y1, int x2, int y2)
```

参数名称是 `x1`、`x2`、`y1` 和 `y2`, 暗示了 `drawLine` 方法从点 (`x1`, `y1`) 到点 (`x2`, `y2`) 画一条线。

也许你还想试试这个方法:

```
drawRoundRect (int x, int y, int width,  
               int height, int arcWidth, int arcHeight)
```

前四个参数描述了矩形方框, 其余两个参数说明矩形的角是弧, 并描述了水平和垂直方向的直径和弧度。

这个命令也有一个“填充”版本, 不仅画轮廓线条, 还将内部填满。接口完全是一样的, 只是方法的名称变了:

```
fillOval (int x, int y, int width, int height)
```

```
fillRect (int x, int y, int width, int height)
```

```
fillRoundRect (int x, int y, int width, int height,  
              int arcWidth, int arcHeight)
```

没有什么 `fillLine` 方法——填充直线是没有意义的。



## D.6 Slate 类

```
import java.awt.*;

class Example {
    // 演示 Slate 类的简单用法
    public static void main (String[] args) {
        int width = 500;
        int height = 500;
        Slate slate = new Slate (width, height);
        Graphics g = slate.getSlateGraphics ();
        g.setColor (Color.blue);
        draw (g, 0, 0, width, height);
        slate.repaint ();
        anim (slate);
    }
    // draw 方法演示了递归模式
    public static void draw (Graphics g, int x, int y, int width,
        int height) {
        if (height < 3) return;
        g.drawOval (x, y, width, height);
        draw (g, x, y+height/2, width/2, height/2);
        draw (g, x+width/2, y+height/2, width/2, height/2);
    }
    // anim 演示了一个简单的动画
    public static void anim (Slate slate) {
        Graphics g = slate.image.getGraphics ();
        g.setColor (Color.red);
        for (int i=-100; i<500; i+=10) {
            g.drawOval (i, 100, 100, 100);
            slate.repaint ();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {}
        }
    }
}
```

```
}  
class Slate extends Frame {  
    // image 做为缓存，当 Slate 的用户画画时，实际是  
    // 在缓存上画，当 Slate 被画好后再将 image 中的内  
    // 容拷贝到屏幕上。  
    Image image;  
    public Slate (int width, int height) {  
        setBounds (100, 100, width, height);  
        setBackground (Color.white);  
        setVisible (true);  
        image = createImage (width, height);  
    }  
    // 当一个 Slate 用户需要一个 Graphics 对象的时候，  
    // 就从屏幕上看不见的缓存中取出一个给它。  
    public Graphics getSlateGraphics () {  
        return image.getGraphics ();  
    }  
    // 一般情况下 update 方法先清理屏幕上的内容然后调用 paint 方法，  
    // 不过既然总是要覆写整个屏幕，将 update 改为不清理屏幕将使  
    // 程序稍微快一点儿  
    public void update (Graphics g) {  
        paint (g);  
    }  
    // paint 方法将屏幕之外的缓存中的内容拷贝到屏幕上  
    public void paint (Graphics g) {  
        if (image == null) return;  
        g.drawImage (image, 0, 0, null);  
    }  
}
```