

1.1.6.2 调用静态方法

调用静态方法的方法是写出方法名并在后面的括号中列出参数值，用逗号分隔。当调用是表达式的一部分时，方法的返回值将会替代表达式中的方法调用。例如，`BinarySearch` 中调用 `rank()` 返回了一个 `int` 值。仅由一个方法调用和一个分号组成的语句一般用于产生副作用。例如，`BinarySearch` 的 `main()` 函数中对系统方法 `Arrays.sort()` 的调用产生的副作用，是将数组中的所有条目有序地排列。调用方法时，它的参数变量将被初始化为调用时所给出的相应表达式的值。返回语句将结束静态方法并将控制权交还给调用者。如果静态方法的目的是计算某个值，返回语句应该指定这个值（如果这样的静态方法在执行完所有的语句之后都没有返回语句，编译器会报错）。

1.1.6.3 方法的性质

对方法所有性质的完整描述超出了本书的范畴，但以下几点值得一提。

- **方法的参数按值传递：**在方法中参数变量的使用方法和局部变量相同，唯一不同的是参数变量的初始值是由调用方提供的。方法处理的是参数的值，而非参数本身。这种方式产生的结果是在静态方法中改变一个参数变量的值对调用者没有影响。本书中我们一般不会修改参数变量。值传递也意味着数组参数将会是原数组的别名（见 1.1.5.4 节）——方法中使用的参数变量能够引用调用者的数组并改变其内容（只是不能改变原数组变量本身）。例如，`Arrays.sort()` 将能够改变通过参数传递的数组的内容，将其排序。
- **方法名可以被重载：**例如，Java 的 `Math` 包使用这种方法为所有的原始数值类型实现了 `Math.abs()`、`Math.min()` 和 `Math.max()` 函数。重载的另一种常见用法是为函数定义两个版本，其中一个需要一个参数而另一个则为该参数提供一个默认值。
- **方法只能返回一个值，但可以包含多个返回语句：**一个 Java 方法只能返回一个值，它的类型是方法签名中声明的类型。静态方法第一次执行到一条返回语句时控制权将会回到调用代码中。尽管可能存在多条返回语句，任何静态方法每次都只会返回一个值，即被执行的第一条返回语句的参数。
- **方法可以产生副作用：**方法的返回值可以是 `void`，这表示该方法没有返回值。返回值为 `void` 的静态函数不需要明确的返回语句，方法的最后一条语句执行完毕后控制权将会返回给调用方。我们称 `void` 类型的静态方法会产生副作用（接受输入、产生输出、修改数组或者改变系统状态）。例如，我们的程序中的静态方法 `main()` 的返回值就是 `void`，因为它的作用是向外输出。技术上来说，数学方法的返回值都不会是 `void` (`Math.random()` 虽然不接受参数但也有返回值)。

2.1 节所述的实例方法也拥有这些性质，尽管两者在副作用方面大为不同。

1.1.6.4 递归

方法可以调用自己（如果你对递归概念感到奇怪，请完成练习 1.1.16 到练习 1.1.22）。例如，下面给出了 `BinarySearch` 的 `rank()` 方法的另一种实现。我们会经常使用递归，因为递归代码比相应的非递归代码更加简洁优雅、易懂。下面这种实现中的注释就言简意赅地说明了代码的作用。我们可以用数学归纳法证明这段注释所解释的算法的正确性。我们会在 3.1 节中展开这个话题并为二分查找提供一个这样的证明。

编写递归代码时最重要的有以下三点。

- **递归总有一个最简单的情况——方法的第一条语句总是一个包含 `return` 的条件语句。**
- **递归调用总是去尝试解决一个规模更小的子问题，这样递归才能收敛到最简单的情况。**在下面的代码中，第四个参数和第三个参数的差值一直在缩小。

22
23

24

- 递归调用的父问题和尝试解决的子问题之间不应该有交集。在下面的代码中，两个子问题各自操作的数组部分是不同的。

```

public static int rank(int key, int[] a)
{ return rank(key, a, 0, a.length - 1); }

public static int rank(int key, int[] a, int lo, int hi)
{ //如果key存在于a[]中，它的索引不会小于lo且不会大于hi

    if (lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if (key < a[mid]) return rank(key, a, lo, mid - 1);
    else if (key > a[mid]) return rank(key, a, mid + 1, hi);
    else return mid;
}

```

二分查找的递归实现

25

违背其中任意一条都可能得到错误的结果或是低效的代码（见练习 1.1.19 和练习 1.1.27），而坚持这些原则能写出清晰、正确且容易评估性能的程序。使用递归的另一个原因是我们可以使用数学模型的来估计程序的性能。我们会在 3.2 节的二分查找以及其他几个地方分析这个问题。

1.1.6.5 基础编程模型

静态方法库是定义在一个 Java 类中的一组静态方法。类的声明是 `public class` 加上类名，以及用花括号包含的静态方法。存放类的文件名和类名相同，扩展名是 `.java`。Java 开发的基本模式是编写一个静态方法库（包含一个 `main()` 方法）来完成一个任务。输入 `java` 和类名以及一系列字符串就能调用类中的 `main()` 方法，其参数为由输入的字符串组成的一个数组。`main()` 的最后一条语句执行完毕之后程序终止。在本书中，当我们提到用于执行一项任务的 Java 程序时，我们指的是用这种模式开发的代码（可能还包括对数据类型的定义，如 1.2 节所示）。例如，`BinarySearch` 就是一个由两个静态方法 `rank()` 和 `main()` 组成的 Java 程序，它的作用是将输入中所有不在通过命令行指定的白名单中的数字打印出来。

1.1.6.6 模块化编程

这个模型的最重要之处在于通过静态方法库实现了模块化编程。我们可以构造许多个静态方法库（模块），一个库中的静态方法也能够调用另一个库中定义的静态方法。这能够带来许多好处：

- 程序整体的代码量很大时，每次处理的模块大小仍然适中；
- 可以共享和重用代码而无需重新实现；
- 很容易用改进的实现替换老的实现；
- 可以为解决编程问题建立合适的抽象模型；
- 缩小调试范围（请见 1.1.6.7 节关于单元测试的讨论）。

例如，`BinarySearch` 用到了三个独立的库，即我们的 `StdOut` 和 `StdIn` 以及 Java 的 `Arrays`，而这三个库又分别用到了其他的库。

1.1.6.7 单元测试

Java 编程的最佳实践之一就是每个静态方法库中都包含一个 `main()` 函数来测试库中的所有方法（有些编程语言不支持多个 `main()` 方法，因此不支持这种方式）。恰当的单元测试本身也是很具有挑战性的编程任务。每个模块的 `main()` 方法至少应该调用模块中的其他代码并在某种程度上保

证它的正确性。随着模块的成熟，我们可以将 `main()` 方法作为一个开发用例，在开发过程中用它来测试更多的细节；也可以把它编成一个测试用例来对所有代码进行全面的测试。当用例越来越复杂时，我们可能会将它独立成一个模块。在本书中，我们用 `main()` 来说明模块的功能并将测试用例留做练习。

26 1.1.6.8 外部库

我们会使用来自 4 个不同类型的库中的静态方法，重用每种库代码的方式都稍有不同。它们大多都是静态方法库，但也有部分是数据类型的定义并包含了一些静态方法。

- 系统标准库 `java.lang.*`: 这其中包括 `Math` 库，实现了常用的数学函数；`Integer` 和 `Double` 库，能够将字符串转化为 `int` 和 `double` 值；`String` 和 `StringBuilder` 库，我们稍后会在本节和第 5 章中详细讨论；以及其他一些我们没有用到的库。
- 导入的系统库，例如 `java.util.Arrays`: 每个标准的 Java 版本中都含有上千个这种类型的库，不过本书中我们用到的并不多。要在程序的开头使用 `import` 语句导入才能使用这些库（我们也是这样做的）。
- 本书中的其他库：例如，其他程序也可以使用 `BinarySearch` 的 `rank()` 方法。要使用这些库，请在本书的网站上下载它们的源代码并放入你的工作目录中。
- 我们为本书（以及我们的另一本入门教材 *An Introduction to programming in Java: An Interdisciplinary Approach*）开发的标准库 `Std*`: 我们会在下面简要地介绍这些库，它们的源代码和使用方法都能够在本书的网站上找到。

要调用另一个库中的方法（存放在相同或者指定的目录中，或是一个系统标准库，或是在类定义前用 `import` 语句导入的库），我们需要在方法前指定库的名称。例如，`BinarySearch` 的 `main()` 方法调用了系统库 `java.util.Arrays` 的 `sort()` 方法，我们的库 `StdIn` 中的 `readInts()` 方法和 `StdOut` 库中的 `println()` 方法。

27 我们自己及他人使用模块化方式编写的方法库能够极大地扩展我们的编程模型。除了在 Java 的标准版本中可用的所有库之外，网上还有成千上万各种用途的代码库。为了将我们的编程模型限制在一个可控范围之内，以将精力集中在算法上，我们只会使用以下所示的方法库，并在 1.1.7 节中列出了其中的部分方法。

1.1.7 API

模块化编程的一个重要组成部分就是记录库方法的用法并供其他人参考的文档。我们会统一使用应用程序编程接口（API）的方式列出本书中使用的每个库方法名称、签名和简短的描述。我们用用例来指代调用另一个库中的方法的程序，用实现描述实现了某个 API 方法的 Java 代码。

1.1.7.1 举例

在表 1.1.6 的例子中，我们用 `java.lang` 中 `Math` 库常用的静态方法说明 API 的文档格式。这些方法实现了各种数学函数——它们通过参数计算得到某种类型的值（`random()` 除外，它没有对应的数学函数，因为它不接受参数）。它们的参数都是 `double` 类型且返回值也都是 `double` 类型，因此可以将它们看做 `double` 数据类型的扩展——这种扩展的能力正是现代编程语言的特性

系统标准库
<code>Math</code>
<code>Integer</code> [†]
<code>Double</code> [†]
<code>String</code> [†]
<code>StringBuilder</code>
<code>System</code>
导入的系统库
<code>java.util.Arrays</code>
我们的标准库
<code>StdIn</code>
<code>StdOut</code>
<code>StdDraw</code>
<code>StdRandom</code>
<code>StdStats</code>
<code>In</code> [†]
<code>Out</code> [†]

[†] 含有静态方法的数据类型的定义

本书使用的含有静态方法的库

之一。API 中的每一行描述了一个方法，提供了使用该方法所需要知道的所有信息。Math 库也定义了常数 PI（圆周率 π ）和 E（自然对数 e），你可以在自己的程序中通过这些变量名引用它们。例如，`Math.sin(Math.PI/2)` 的结果是 1.0，`Math.log(Math.E)` 的结果也是 1.0（因为 `Math.sin()` 的参数是弧度而 `Math.log()` 使用的是自然对数函数）。

表 1.1.6 Java 的数学函数库的 API（节选）

<code>public class Math</code>	
<code>static double abs(double a)</code>	a 的绝对值
<code>static double max(double a, double b)</code>	a 和 b 中的较大者
<code>static double min(double a, double b)</code>	a 和 b 中的较小者
注 1：abs()、max() 和 min() 也定义了 int、long 和 float 的版本。	
<code>static double sin(double theta)</code>	正弦函数
<code>static double cos(double theta)</code>	余弦函数
<code>static double tan(double theta)</code>	正切函数
注 2：角用弧度表示，可以使用 toDegrees() 和 toRadians() 转换角度和弧度。	
注 3：它们的反函数分别为 asin()、acos() 和 atan()。	
<code>static double exp(double a)</code>	指数函数 (e^a)
<code>static double log(double a)</code>	自然对数函数 (\log_a , 即 $\ln a$)
<code>static double pow(double a, double b)</code>	求 a 的 b 次方 (a^b)
<code>static double random()</code>	[0, 1] 之间的随机数
<code>static double sqrt(double a)</code>	a 的平方根
<code>static double E</code>	常数 e (常数)
<code>static double PI</code>	常数 π (常数)

其他函数请见本书的网站。

28

1.1.7.2 Java 库

成千上万个库的在线文档是 Java 发布版本的一部分。为了更好地描述我们的编程模型，我们只是从中节选了本书所用到的若干方法。例如，BinarySearch 中用到了 Java 的 Arrays 库中的 `sort()` 方法，我们对它的记录如表 1.1.7 所示。

表 1.1.7 Java 的 Arrays 库节选 (java.util.Arrays)

<code>public class Arrays</code>	
<code>static void sort(int[] a)</code>	将数组按升序排序

注：其他原始类型和 Object 对象也有对应版本的方法。

Arrays 库不在 `java.lang` 中，因此我们需要用 `import` 语句导入后才能使用它，与 BinarySearch 中一样。事实上，本书的第 2 章讲的正是数组的各种 `sort()` 方法的实现，包括 `Arrays.sort()` 中实现的归并排序和快速排序算法。Java 和很多其他编程语言都实现了本书讲解的许多基础算法。例如，Arrays 库还包含了二分查找的实现。为避免混淆，我们一般会使用自己的实现，但对于你已经掌握的算法使用高度优化的库实现当然也没有任何问题。

29

1.1.7.3 我们的标准库

为了介绍 Java 编程、为了科学计算以及算法的开发、学习和应用，我们也开发了若干库来提供一些实用的功能。这些库大多用于处理输入输出。我们也会使用以下两个库来测试和分析我们的实

现。第一个库扩展了 `Math.random()` 方法（见表 1.1.8），以根据不同的概率密度函数得到随机值；第二个库则支持各种统计计算（见表 1.1.9）。

表 1.1.8 我们的随机数静态方法库的 API

<code>public class StdRandom</code>	
<code>static void initialize(long seed)</code>	初始化
<code>static double random()</code>	0 到 1 之间的实数
<code>static int uniform(int N)</code>	0 到 N-1 之间的整数
<code>static int uniform(int lo, int hi)</code>	lo 到 hi-1 之间的整数
<code>static double uniform(double lo, double hi)</code>	lo 到 hi 之间的实数
<code>static boolean bernoulli(double p)</code>	返回真的概率为 p
<code>static double gaussian()</code>	正态分布，期望值为 0，标准差为 1
<code>static double gaussian(double m, double s)</code>	正态分布，期望值为 m，标准差为 s
<code>static int discrete(double[] a)</code>	返回 i 的概率为 a[i]
<code>static void shuffle(double[] a)</code>	将数组 a 随机排序

注：库中也包含为其他原始类型和 `Object` 对象重载的 `shuffle()` 函数。

表 1.1.9 我们的数据分析静态方法库的 API

<code>public class StdStats</code>	
<code>static double max(double[] a)</code>	最大值
<code>static double min(double[] a)</code>	最小值
<code>static double mean(double[] a)</code>	平均值
<code>static double var(double[] a)</code>	采样方差
<code>static double stddev(double[] a)</code>	采样标准差
<code>static double median(double[] a)</code>	中位数

StdRandom 的 `initialize()` 方法为随机数生成器提供种子。这样我们就可以重复和随机数有关的实验。以上一些方法的实现请参考表 1.1.10。有些方法的实现非常简单，为什么还要在方法库中实现它们？设计良好的方法库对这个问题的标准回答如下。

- 这些方法所实现的抽象层有助于我们将精力集中在实现和测试本书中的算法，而非生成随机数或是统计计算。每次都自己写完成相同计算的代码，不如直接在用例中调用它们要更简洁易懂。
- 方法库会经过大量测试，覆盖极端和罕见的情况，是我们可以信任的。这样的实现需要大量的代码。例如，我们经常需要使用的各种数据类型的实现，又比如 Java 的 `Arrays` 库针对不同数据类型对 `sort()` 进行了多次重载。

这些是 Java 模块化编程的基础，不过在这里可能有些夸张。但这些方法库的方法名称简单、实现容易，其中一些仍然能作为有趣的算法练习。因此，我们建议你到本书的网站上去学习一下 `StdRandom.java` 和 `StdStats.java` 的源代码并好好利用这些经过验证了的实现。使用这些库（以及检验它们）最简单的方法就是从网站上下载它们的源代码并放入你的工作目录。网站上讲解了在各种系统上使用它们的配置目录的方法。



表 1.1.10 StdRandom 库中的静态方法的实现

期望的结果	实 现
随机返回 $[a,b]$ 之间的一个 double 值	<pre>public static double uniform(double a, double b) { return a + StdRandom.random() * (b-a); }</pre>
随机返回 $[0..N]$ 之间的一个 int 值	<pre>public static int uniform(int N) { return (int) (StdRandom.random() * N); }</pre>
随机返回 $[lo,hi]$ 之间的一个 int 值	<pre>public static int uniform(int lo, int hi) { return lo + StdRandom.uniform(hi - lo); }</pre>
根据离散概率随机返回的 int 值 (出现 i 的概率为 $a[i]$)	<pre>public static int discrete(double[] a) { // a[] 中各元素之和必须等于1 double r = StdRandom.random(); double sum = 0.0; for (int i = 0; i < a.length; i++) { sum = sum + a[i]; if (sum >= r) return i; } return -1; }</pre>
随机将 double 数组中的元素排序 (请见练习 1.1.36)	<pre>public static void shuffle(double[] a) { int N = a.length; for (int i = 0; i < N; i++) { // 将 a[i] 和 a[i..N-1] 中任意一个元素交换 int r = i + StdRandom.uniform(N-i); double temp = a[i]; a[i] = a[r]; a[r] = temp; } }</pre>

1.1.7.4 你自己编写的库

你应该将自己编写的每一个程序都当做一个日后可以重用的库。

- 编写用例，在实现中将计算过程分解成可控的部分。
- 明确静态方法库和与之对应的 API (或者多个库的多个 API)。
- 实现 API 和一个能够对方法进行独立测试的 main() 函数。

这种方法不仅能帮助你实现可重用的代码，而且能够教会你如何运用模块化编程来解决一个复杂的问题。

API 的目的是将调用和实现分离：除了 API 中给出的信息，调用者不需要知道实现的其他细节，而实现也不应考虑特殊的应用场景。API 使我们能够广泛地重用那些为各种目的独立开发的代码。没有任何一个 Java 库能够包含我们在程序中可能用到的所有方法，因此这种能力对于编写复杂的应用程序特别重要。相应地，程序员也可以将 API 看做调用和实现之间的一份契约，它详细说明了每个方法的作用。实现的目标就是能够遵守这份契约。一般来说，做到这一点有很多种方法，而且将调用者的代码和实现的代码分离使我们可以将老算法替换为更新更好的实现。在学习算法的过程中，这也使我们能够感受到算法的改进所带来的影响。

31
32

33

1.1.8 字符串

字符串是由一串字符（`char`类型的值）组成的。一个`String`类型的字面量包括一对双引号和其中的字符，比如“Hello, World”。`String`类型是Java的一个数据类型，但并不是原始数据类型。我们现在就讨论`String`类型是因为它非常基础，几乎所有Java程序都会用到它。

1.1.8.1 字符串拼接

和各种原始数据类型一样，Java内置了一个串联`String`类型字符串的运算符（+）。表1.1.11是对表1.1.2的补充。拼接两个`String`类型的字符串将得到一个新的`String`值，其中第一个字符串在前，第二个字符串在后。

表1.1.11 Java的`String`数据类型

类 型	值 域	举 例	运 算 符	表达式举例	
				表 达 式	值
<code>String</code>	一串字符	"AB"	+ (拼接)	"Hi," + "Bob"	"Hi, Bob"
		"Hello"		"12" + "34"	"1234"
		"2.5"		"1" + "+" + "2"	"1+2"

1.1.8.2 类型转换

字符串的两个主要用途分别是将用户从键盘输入的内容转换成相应数据类型的值以及将各种数据类型的值转化成能够在屏幕上显示的内容。Java的`String`类型为这些操作内置了相应的方法，而且`Integer`和`Double`库还包含了分别和`String`类型相互转化的静态方法（见表1.1.12）。

表1.1.12 `String`值和数字之间相互转换的API

<code>public class Integer</code>		
	<code>static int parseInt(String s)</code>	将字符串s转换为整数
	<code>static String toString(int i)</code>	将整数i转换为字符串
<code>public class Double</code>		
	<code>static double parseDouble(String s)</code>	将字符串s转换为浮点数
	<code>static String toString(double x)</code>	将浮点数x转换为字符串

1.1.8.3 自动转换

我们很少明确使用刚才提到的`toString()`方法，因为Java在连接字符串的时候会自动将任意数据类型的值转换为字符串：如果加号（+）的一个参数是字符串，那么Java会自动将其他参数都转换为字符串（如果它们不是的话）。除了像“`The square root of 2.0 is " + Math.sqrt(2.0)`”这样的使用方式之外，这种机制也使我们能够通过一个空字符串“”将任意数据类型的值转换为字符串值。

1.1.8.4 命令行参数

在Java中字符串的一个重要的用途就是使程序能够接收到从命令行传递来的信息。这种机制很简单。当你输入命令`java`和一个库名以及一系列字符串之后，Java系统会调用库的`main()`方法并将那一系列字符串变成一个数组作为参数传递给它。例如，`BinarySearch`的`main()`方法需要一个命令行参数，因此系统会创建一个大小为1的数组。程序用这个值，也就是`args[0]`，来获取白

名单文件的文件名并将其作为 `StdIn.readInts()` 的参数。另一种在我们的代码中常见的用法是当命令行参数表示的是数字时，我们会用 `parseInt()` 和 `parseDouble()` 方法将其分别转换为整数和浮点数。

字符串的用法是现代程序中的重要部分。现在我们还只是用 `String` 在外部表示为字符串的数字和内部表示为数字类数据类型的值进行转换。在 1.2 节中我们会看到 Java 为我们提供了非常丰富的字符串操作；在 1.4 节中我们会分析 `String` 类型在 Java 内部的表示方法；在第 5 章我们会深入学习处理字符串的各种算法。这些算法是本书中最有趣、最复杂也是影响力最大的一部分算法。

35

1.1.9 输入输出

我们的标准输入、输出和绘图库的作用是建立一个 Java 程序和外界交流的简易模型。这些库的基础是强大的 Java 标准库，但它们一般更加复杂，学习和使用起来都更加困难。我们先来简单地了解一下这个模型。

在我们的模型中，Java 程序可以从命令行参数或者一个名为标准输入流的抽象字符流中获得输入，并将输出写入另一个名为标准输出流的字符流中。

我们需要考虑 Java 和操作系统之间的接口，因此我们要简要地讨论一下大多数操作系统和程序开发环境所提供的相应机制。本书网站上列出了关于你所使用的系统的更多信息。默认情况下，命令行参数、标准输入和标准输出是和应用程序绑定的，而应用程序是由能够接受命令输入的操作系统或是开发环境所支持。我们笼统地用终端来指代这个应用程序提供的供输入和显示的窗口。20 世纪 70 年代早期的 Unix 系统已经证明我们可以用这个模型方便直接地和程序以及数据进行交互。我们在经典的模型中加入了一个标准绘图模块用来可视化表示对数据的分析，如图 1.1.3 所示。

1.1.9.1 命令和参数

终端窗口包含一个提示符，通过它我们能够向操作系统输入命令和参数。本书中我们只会用到几个命令，如表 1.1.13 所示。我们会经常使用 `java` 命令来运行我们的程序。我们在 1.1.8.4 节中提到过，Java 类都会包含一个静态方法 `main()`，它有一个 `String` 数组类型的参数 `args[]`。这个数组的内容就是我们输入的命令行参数，操作系统会将它传递给 Java。Java 和操作系统都默认参数为字符串。如果我们需要的某个参数是数字，我们会使用类似 `Integer.parseInt()` 的方法将其转换为适当的数据类型的值。图 1.1.4 是对命令的分析。

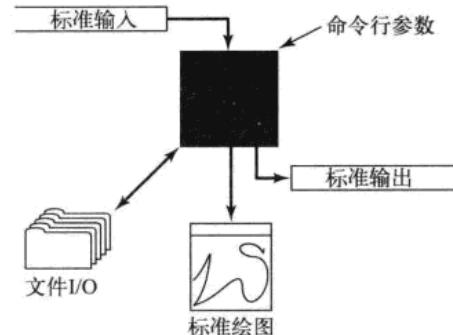


图 1.1.3 Java 程序整体结构

表 1.1.13 操作系统常用命令

命 令	参 数	作 用
<code>javac</code>	<code>java</code> 文件名	编译 Java 程序
<code>java</code>	<code>.class</code> 文件名（不需要扩展名） 和命令行参数	运行 Java 程序
<code>more</code>	任意文本文件名	打印文件内容

36

1.1.9.2 标准输出

我们的 StdOut 库的作用是支持标准输出。一般来说，系统会将标准输出打印到终端窗口。`print()` 方法会将它的参数放到标准输出中；`println()` 方法会附加一个换行符；`printf()` 方法能够格式化输出（见 1.1.9.3 节）。Java 在其 `System.out` 库中提供了类似的方法，但我们会用 StdOut 库来统一处理标准输入和输出（并进行了一些技术上的改进），见表 1.1.4。

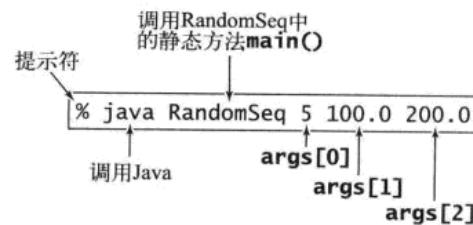


图 1.1.4 命令详解

表 1.1.14 我们的标准输出库的静态方法的 API

<code>public class StdOut</code>	
<code>static void print(String s)</code>	打印 s
<code>static void println(String s)</code>	打印 s 并接一个换行符
<code>static void println()</code>	打印一个换行符
<code>static void printf(String f, ...)</code>	格式化输出

注：其他原始类型和 `Object` 对象也有对应版本的方法。

要使用这些方法，请从本书的网站上将 `StdOut.java` 下载到你的工作目录，并像 `StdOut.println("Hello, World")`；这样在代码中调用它们。左下方的程序就是一个例子。

1.1.9.3 格式化输出

在最简单的情况下 `printf()` 方法接受两个参数。第一个参数是一个格式字符串，描述了第二个参数应该如何在输出中被转换为一个字符串。最简单的格式字符串的第一个字符是 % 并紧跟一个字符表示的转换代码。我们最常使用的转换代码包括 `d`（用于 Java 整型的十进制数）、`f`（浮点型）

```
public class RandomSeq
{
    public static void main(String[] args)
    { // 打印N个(lo, hi)之间的随机值
        int N = Integer.parseInt(args[0]);
        double lo = Double.parseDouble(args[1]);
        double hi = Double.parseDouble(args[2]);
        for (int i = 0; i < N; i++)
        {
            double x = StdRandom.uniform(lo, hi);
            StdOut.printf("%.2f\n", x);
        }
    }
}
```

StdOut 的用例示例

```
% java RandomSeq 5 100.0 200.0
123.43
153.13
144.38
155.18
104.02
```

和 `s`（字符串）。在 % 和转换代码之间可以插入一个整数来表示转换之后的值的宽度，即输出字符串的长度。默认情况下，转换后会在字符串的左边添加空格以达到需要的宽度，如果我们想在右边加入空格则应该使用负宽度（如果转换得到的字符串比设定宽度要长，宽度会被忽略）。在宽度之后我们还可以插入一个小数点以及一个数值来指定转换后的 `double` 值保留的小数位数（精度）或是 `String` 字符串所截取的长度。使用

`printf()` 方法时需要记住的最重要的一点就是，格式字符串中的转换代码和对应参数的数据类型必须匹配。也就是说，Java 要求参数的数据类型和转换代码表示的数据类型必须相同。`printf()` 的第一个 `String` 字符串参数也可以包含其他字符。所有非格式字符串的

字符都会被传递到输出之中，而格式字符串则会被参数的值所替代（按照指定的方式转换为字符串）。例如，这条语句：

```
StdOut.printf("PI is approximately %.2f\n", Math.PI);
```

会打印出：

```
PI is approximately 3.14
```

可以看到，在`printf()`中我们需要明确地在第一个参数的末尾加上`\n`来换行。`printf()`函数能够接受两个或者更多的参数。在这种情况下，在格式化字符串中每个参数都会有对应的转换代码，这些代码之间可能隔着其他会被直接传递到输出中的字符。也可以直接使用静态方法`String.format()`来用和`printf()`相同的参数得到一个格式化字符串而无需打印它。我们可以用格式化打印方便地将实验数据输出为表格形式（这是它们在本书中的主要用途），如表 1.1.15 所示。

表 1.1.15 `printf()` 的格式化方式（更多选项请见本书网站）

数据类型	转换代码	举 例	格式化字符串举例	转换后输出的字符串
<code>int</code>	<code>d</code>	512	"%14d" "%-14d"	" 512" "-512"
<code>double</code>	<code>f</code> <code>e</code>	1595.1680010754388	"%14.2f" "%.7f" "%14.4e"	" 1595.17" "1595.1680011" " 1.5952e+03"
<code>String</code>	<code>s</code>	"Hello, World"	"%14s" "%-14s" "%14.5s"	" Hello, World" "Hello, World " "Hello "

38

1.1.9.4 标准输入

我们的`StdIn`库从标准输入流中获取数据，这些数据可能为空也可能是一系列由空白字符分隔的值（空格、制表符、换行符等）。默认状态下系统会将标准输出定向到终端窗口——你输入的内容就是输入流（由`<ctrl-d>`或`<ctrl-z>`结束，取决于你使用的终端应用程序）。这些值可能是`String`或是Java的某种原始类型的数据。标准输入流最重要的特点是这些值会在你的程序读取它们之后消失。只要程序读取了一个值，它就不能回退并再次读取它。这个特点产生了一些限制，但它反映了一些输入设备的物理特性并简化了对这些设备的抽象。有了输入流模型，这个库中的静态方法大都是自文档化的（它们的签名即说明了它们的用途）。右侧列出了`StdIn`的一个用例。

表 1.1.16 详细说明了标准输入库中的静态方法的 API。

```
public class Average
{
    public static void main(String[] args)
    { // 取StdIn中所有数的平均值
        double sum = 0.0;
        int cnt = 0;
        while (!StdIn.isEmpty())
        { // 读取一个数并计算累计之和
            sum += StdIn.readDouble();
            cnt++;
        }
        double avg = sum / cnt;
        StdOut.printf("Average is %.5f\n", avg);
    }
}
```

StdIn 的用例举例

```
% java Average
1.23456
2.34567
3.45678
4.56789
<ctrl-d>
Average is 2.90123
```

表 1.1.16 标准输入库中的静态方法的 API

Public class StdIn		
static boolean	isEmpty()	如果输入流中没有剩余的值则返回 true，否则返回 false
static int	readInt()	读取一个 int 类型的值
static double	readDouble()	读取一个 double 类型的值
static float	readFloat()	读取一个 float 类型的值
static long	readLong()	读取一个 long 类型的值
static boolean	readBoolean()	读取一个 boolean 类型的值
static char	readChar()	读取一个 char 类型的值
static byte	readByte()	读取一个 byte 类型的值
static String	readString()	读取一个 String 类型的值
static boolean	hasNextLine()	输入流中是否还有下一行
static String	readLine()	读取该行的其余内容
static String	readAll()	读取输入流中的其余内容

39

1.1.9.5 重定向与管道

标准输入输出使我们能够利用许多操作系统都支持的命令行的扩展功能。只需要向启动程序的命令中加入一个简单的提示符，就可以将它的标准输出重定向至一个文件。文件的内容既可以永久保存也可以在之后作为另一个程序的输入：

```
% java RandomSeq 1000 100.0 200.0 > data.txt
```

这条命令指明标准输出流不是被打印至终端窗口，而是被写入一个叫做 data.txt 的文件。每次调用 `StdOut.print()` 或是 `StdOut.println()` 都会向该文件追加一段文本。在这个例子中，我们最后会得到一个含有 1000 个随机数的文件。终端窗口中不会出现任何输出：它们都被直接写入了“>”号之后的文件中。这样我们就能将信息存储以备下次使用。请注意不需要改变 `RandomSeq` 的任何内容——它使用的是标准输出的抽象，因此它不会因为我们使用了该抽象的另一种不同的实现而受到影响。类似，我们可以重定向标准输入以使 `StdIn` 从文件而不是终端应用程序中读取数据：

```
% java Average < data.txt
```

这条命令会从文件 `data.txt` 中读取一系列数值并计算它们的平均值。具体来说，“<”号是一个提示符，它告诉操作系统读取文本文件 `data.txt` 作为输入流而不是在终端窗口中等待用户的输入。当程序调用 `StdIn.readDouble()` 时，操作系统读取的是文件中的值。将这些结合起来，将一个程序的输出重定向为另一个程序的输入叫做管道：

40

```
% java RandomSeq 1000 100.0 200.0 | java Average
```

这条命令将 `RandomSeq` 的标准输出和 `Average` 的标准输入指定为同一个流。它的效果是好像在 `Average` 运行时 `RandomSeq` 将它生成的数字输入了终端窗口。这种差别影响非常深远，因为它突破了我们能够处理的输入输出流的长度限制。例如，即使计算机没有足够的空间来存储十亿个数，我们仍然可以将例子中的 1000 换成 1 000 000 000（当然我们还是需要一些时间来处理它们）。当 `RandomSeq` 调用 `StdOut.println()` 时，它就向输出流的末尾添加了一个字符串；当 `Average` 调用 `StdIn.readInt()` 时，它就从输入流的开头删除了一个字符串。这些动作发生的实际顺序取决于操作系统：它可能会先运行 `RandomSeq` 并产生一些输出，然后再运行 `Average`，来消耗这些输出，或者它也可以先运行 `Average`，直到它需要一些输入然后再运行 `RandomSeq` 来产生一些输出。虽然

最后的结果都一样，但我们的程序就不再需要担心这些细节，因为它们只会和标准输入和标准输出的抽象打交道。

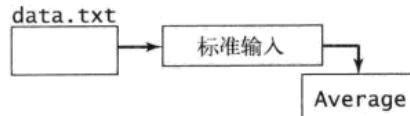
图 1.1.5 总结了重定向与管道的过程。

1.1.9.6 基于文件的输入输出

我们的 In 和 Out 库提供了一些静态方法，来实现向文件中写入或从文件中读取一个原始数据类型（或 String 类型）的数组的抽象。我们会使用 In 库中的 `readInts()`、`readDoubles()` 和 `readStrings()` 以及 Out 库中的 `writeInts()`、`writeDoubles()` 和 `writeStrings()` 方法，参数可以是文件或网页如表 1.1.17 所示。例如，借此我们可以在同一个程序中分别使用文件和标准输入达到两种不同的目的，例如 BinarySearch。In 和 Out 两个库也实现了一些数据类型和它们的实例方法，这使我们能够将多个文件作为输入输出流并将网页作为输入流，我们还会在 1.2 节中再次考察它们。

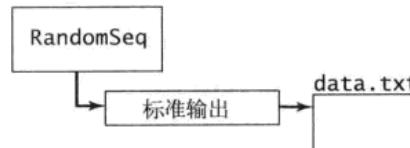
将一个文件重定向为标准输入

% java Average < data.txt



将标准输出重定向到一个文件

% java RandomSeq 1000 100.0 200.0 > data.txt



将一个程序的输出通过管道作为另一个程序的输入

% java RandomSeq 1000 100.0 200.0 | java Average

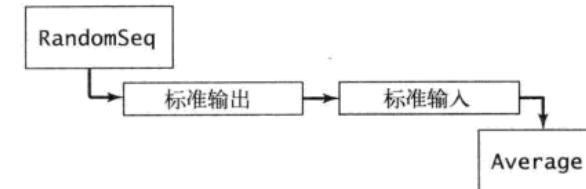


图 1.1.5 命令行的重定向与管道

表 1.1.17 我们用于读取和写入数组的静态方法的 API

public class In		
static int[] readInts(String name)		读取多个 int 值
static double[] readDoubles(String name)		读取多个 double 值
static String[] readStrings(String name)		读取多个 String 值
public class Out		
static void write(int[], String name)		写入多个 int 值
static void write(double[] a, String name)		写入多个 double 值
static void write(String[] a, String name)		写入多个 String 值

注 1：库也支持其他原始数据类型。

注 2：库也支持 StdIn 和 StdOut（忽略 name 参数）。

41

1.1.9.7 标准绘图库（基本方法）

目前为止，我们的输入输出抽象层的重点只有文本字符串。现在我们要介绍一个产生图像输出的抽象层。这个库的使用非常简单并且允许我们利用可视化的方式处理比文字丰富得多的信息。和我们的标准输入输出一样，标准绘图抽象层实现在库 `StdDraw` 中，可以从本书的网站上下载 `StdDraw.java` 到你的工作目录来使用它。标准绘图库很简单：我们可以将它想象为一个抽象的能够在二维画布上画出点和直线的绘图设备。这个设备能够根据程序调用的 `StdDraw` 中的静态方法画出

一些基本的几何图形，这些方法包括画出点、直线、文本字符串、圆、长方形和多边形等。和标准输入输出中的方法一样，这些方法几乎也都是自文档化的：`StdDraw.line()` 能够根据参数的坐标画出一条连接点 (x_0, y_0) 和点 (x_1, y_1) 的线段，`StdDraw.point()`能够根据参数坐标画出一个以 (x, y) 为中心的点，等等，如图 1.1.6 所示。几何图形可以被填充（默认为黑色）。默认的比例尺为单位正方形（所有的坐标均在 0 和 1 之间）。标准的实现会将画布显示为屏幕上的一个窗口，点和线为黑色，背景为白色。

42

表 1.1.18 是对标准绘图库中静态方法 API 的汇总。

表 1.1.18 标准绘图库的静态（绘图）方法的 API

```
public class StdDraw
    static void line(double x0, double y0, double x1, double y1)
    static void point(double x, double y)
    static void text(double x, double y, String s)
    static void circle(double x, double y, double r)
    static void filledCircle(double x, double y, double r)
    static void ellipse(double x, double y, double rw, double rh)
    static void filledEllipse(double x, double y, double rw, double rh)
    static void square(double x, double y, double r)
    static void filledSquare(double x, double y, double r)
    static void rectangle(double x, double y, double rw, double rh)
    static void filledRectangle(double x, double y, double rw, double rh)
    static void polygon(double[] x, double[] y)
    static void filledPolygon(double[] x, double[] y)
```

1.1.9.8 标准绘图库（控制方法）

标准绘图库中还包含一些方法来改变画布的大小和比例、直线的颜色和宽度、文本字体、绘图时间（用于动画）等。可以使用在 `StdDraw` 中预定义的 `BLACK`、`BLUE`、`CYAN`、`DARK_GRAY`、`GRAY`、`GREEN`、`LIGHT_GRAY`、`MAGENTA`、`ORANGE`、`PINK`、`RED`、`BOOK_RED`、`WHITE` 和 `YELLOW` 等颜色常数作为 `setPenColor()` 方法的参数（可以用 `StdDraw.RED` 这样的方式调用它们）。画布窗口的菜单还包含一个选项用于将图像保存为适于在网上传播的文件格式。表 1.1.19 总结了 `StdDraw` 中静态控制方法的 API。

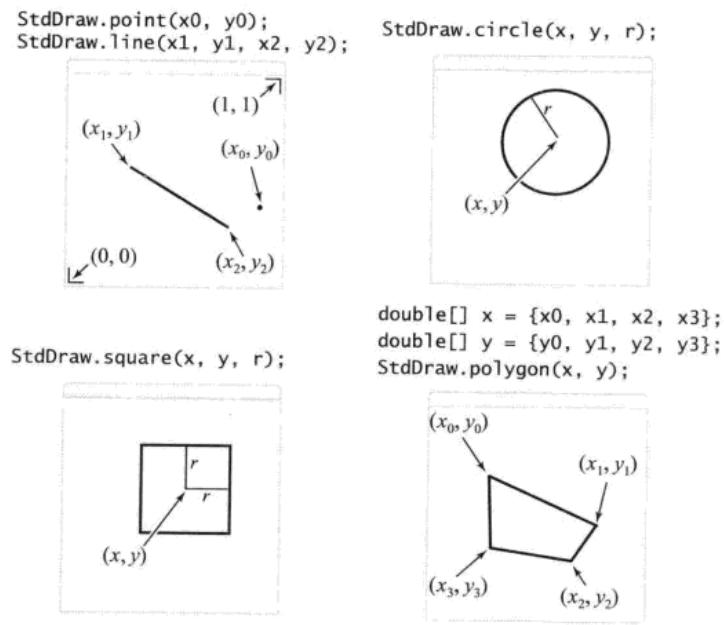
图 1.1.6 `StdDraw` 的用法举例

表 1.1.19 标准绘图库的静态（控制）方法的 API

public class StdDraw	
static void setXscale(double x0, double x1)	将 x 的范围设为 (x_0, x_1)
static void setYscale(double y0, double y1)	将 y 的范围设为 (y_0, y_1)
static void setPenRadius(double r)	将画笔的粗细半径设为 r
static void setPenColor(Color c)	将画笔的颜色设为 c
static void setFont(Font f)	将文本字体设为 f
static void setCanvasSize(int w, int h)	将画布窗口的宽和高分别设为 w 和 h
static void clear(Color c)	清空画布并用颜色 c 将其填充
static void show(int dt)	显示所有图像并暂停 dt 毫秒

43

在本书中，我们会在数据分析和算法的可视化中使用 `StdDraw`。表 1.1.20 是一些例子，我们在本书的其他章节和练习中还会遇到更多的例子。绘图库也支持动画——当然，这个话题只能在本书的网站上展开了。

44

表 1.1.20 StdDraw 绘图举例

数 据	绘图的实现（代码片段）	结 果
函数值	<pre> int N = 100; StdDraw.setXscale(0, N); StdDraw.setYscale(0, N*N); StdDraw.setPenRadius(.01); for (int i = 1; i <= N; i++) { StdDraw.point(i, i); StdDraw.point(i, i*i); StdDraw.point(i, i*Math.log(i)); } </pre>	
随机数组	<pre> int N = 50; double[] a = new double[N]; for (int i = 0; i < N; i++) a[i] = StdRandom.random(); for (int i = 0; i < N; i++) { double x = 1.0*i/N; double y = a[i]/2.0; double rw = 0.5/N; double rh = a[i]/2.0; StdDraw.filledRectangle(x, y, rw, rh); } </pre>	
已排序的随机数组	<pre> int N = 50; double[] a = new double[N]; for (int i = 0; i < N; i++) a[i] = StdRandom.random(); Arrays.sort(a); for (int i = 0; i < N; i++) { double x = 1.0*i/N; double y = a[i]/2.0; double rw = 0.5/N; double rh = a[i]/2.0; StdDraw.filledRectangle(x, y, rw, rh); } </pre>	

45

1.1.10 二分查找

我们要学习的第一个 Java 程序的示例程序就是著名、高效并且应用广泛的二分查找算法，如下所示。这个例子将会展示本书中学习新算法的基本方法。和我们将要学习的所有程序一样，它既是算法的准确定义，又是算法的一个完整的 Java 实现，而且你还能够从本书的网站上下载它。

二分查找

```
import java.util.Arrays;
public class BinarySearch
{
    public static int rank(int key, int[] a)
    { // 数组必须是有序的
        int lo = 0;
        int hi = a.length - 1;
        while (lo <= hi)
        { // 被查找的键要么不存在，要么必然存在于 a[lo..hi] 之中
            int mid = lo + (hi - lo) / 2;
            if (key < a[mid]) hi = mid - 1;
            else if (key > a[mid]) lo = mid + 1;
            else return mid;
        }
        return -1;
    }
    public static void main(String[] args)
    {
        int[] whitelist = In.readInts(args[0]);
        Arrays.sort(whitelist);
        while (!StdIn.isEmpty())
        { // 读取键值，如果不存在于白名单中则将其打印
            int key = StdIn.readInt();
            if (rank(key, whitelist) < 0)
                StdOut.println(key);
        }
    }
}
```

这段程序接受一个白名单文件（一列整数）作为参数，并会过滤掉标准输入中的所有存在于白名单中的条目，仅将不在白名单上的整数打印到标准输出中。它在 `rank()` 静态方法中实现了二分查找算法并高效地完成了这个任务。

关于二分查找算法的完整讨论，包括它的正确性、性能分析及其应用，请见 3.1 节。

```
% java BinarySearch tinyW.txt < tinyT.txt
50
99
13
```

1.1.10.1 二分查找

我们会在 3.2 节中详细学习二分查找算法，但此处先简单地描述一下。算法是由静态方法 `rank()` 实现的，它接受一个整数键和一个已经有序的 `int` 数组作为参数。如果该键存在于数组中则返回它的索引，否则返回 `-1`。算法使用两个变量 `lo` 和 `hi`，并保证如果键在数组中则它一定在 `a[lo..hi]` 中，然后方法进入一个循环，不断将数组的中间键（索引为 `mid`）和被查找的键比较。如果被查找的键等于 `a[mid]`，返回 `mid`；否则算法就将查找范围缩小一半，如果被查找的键小于 `a[mid]` 就继续在左半边查找，如果被查找的键大于 `a[mid]` 就继续在右半边查找。算法找到被查找的键或是查找范围为空时该过程结束。二分查找之所以快是因为它只需检查很少几个条目（相对于数组的大小）就能够找到目标元素（或者确认目标元素不存在）。在有序数组中进行二分查找的示

例如图 1.1.7 所示。

1.1.10.2 开发用例

对于每个算法的实现，我们都会开发一个用例 `main()` 函数，并在书中或是本书的网站上提供一个示例输入文件来帮助读者学习该算法并检测它的性能。在这个例子中，这个用例会从命令行指定的文件中读取多个整数，并会打印出标准输入中所有不存在于该文件中的整数。我们使用了图 1.1.8 所示的几个较小的测试文件来展示它的行为，这些文件也是图 1.1.7 中的跟踪和例子的基础。我们会使用较大的测试文件来模拟真实应用并测试算法的性能（请见 1.1.10.3 节）。

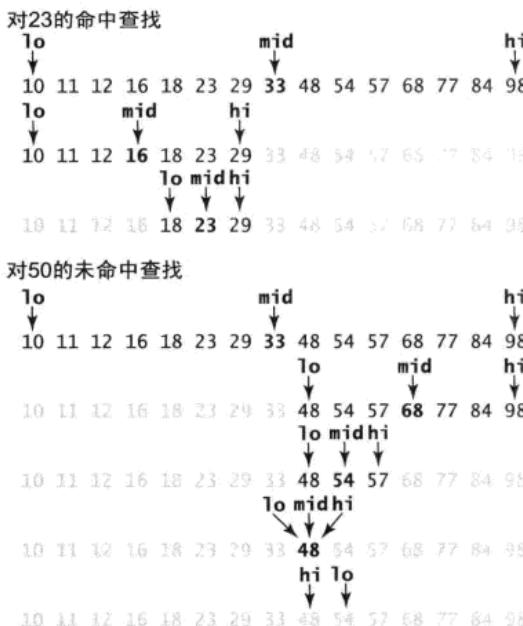


图 1.1.7 有序数组中的二分查找

tinyW.txt tinyT.txt	
84	23
48	50
68	10
10	99
18	18
98	23
12	98
23	84
54	11
57	10
48	48
33	77
16	13
77	54
11	98
29	77
77	68

图 1.1.8 为 BinarySearch 的测试用例准备的小型测试文件

46
47

1.1.10.3 白名单过滤

如果可能，我们的测试用例都会通过模拟实际情况来展示当前算法的必要性。这里该过程被称为白名单过滤。具体来说，可以想象一家信用卡公司，它需要检查客户的交易账号是否有效。为此，它需要：

- 将客户的账号保存在一个文件中，我们称它为白名单；
- 从标准输入中得到每笔交易的账号；
- 使用这个测试用例在标准输出中打印所有与任何客户无关的账号，公司很可能拒绝此类交易。

在一家有上百万客户的大公司中，需要处理数百万甚至更多的交易都是很正常的。为了模拟这种情况，我们在本书的网站上提供了文件 `largeW.txt` (100 万个整数) 和 `largeT.txt` (1000 万个整数) 其基本情况如图 1.1.9 所示。

1.1.10.4 性能

一个程序只是可用往往是不够的。例如，以下 `rank()` 的实现也可以很简单，它会检查数组的每个元素，甚至都不需要数组是有序的：

```
public static int rank(int key, int[] a)
{
    for (int i = 0; i < a.length; i++)
        if (a[i] == key) return i;
    return -1;
}
```

有了这个简单易懂的解决方案，我们为什么还需要归并排序和二分查找呢？你在完成练习 1.1.38 时会看到，计算机用 `rank()` 方法的暴力实现处理大量输入（比如含有 100 万个条目的白名单和 1000 万条交易）非常慢。没有如二分查找或者归并排序这样的高效算法，解决大规模的白名单问题是不可能的。良好的性能常常是极为重要的，因此我们会在 1.4 节中为性能研究做一些铺垫，并会分析我们学习的所有算法的性能特点（包括 2.2 节的归并排序和 3.1 节中的二分查找）。

目前，我们在这里粗略地勾勒出我们的编程模型的目标是，确保你能够在计算机上运行类似于 `BinarySearch` 的代码，使用它处理我们的测试数据并为适应各种情况修改它（比如本节练习中所描述的一些情况）以完全理解它的可应用性。我们的编程模型就是设计用来简化这些活动的，这对各种算法的学习至关重要。

48
49

1.1.11 展望

在本节中，我们描述了一个精巧而完整的编程模型，数十年来它一直在（并且现在仍在）为广大程序员服务。但现代编程技术已经更进一步。前进的这一步被称为数据抽象，有时也被称为面向对象编程，它是下一节的主题。简单地说，数据抽象的主要思想是鼓励程序定义自己的数据类型（一系列值和对这些值的操作），而不仅仅是那些操作预定义的数据类型的静态方法。

面向对象编程在最近几十年得到了广泛的应用，数据抽象已经成为现代程序开发的核心。我们在本书中“拥抱”数据抽象的原因主要有三。

- 它允许我们通过模块化编程复用代码。例如，第 2 章中的排序算法和第 3 章中的二分查找以及其他算法，都允许调用者用同一段代码处理任意类型的数据（而不仅限于整数），包括调用者自定义的数据类型。
- 它使我们可以轻易构造多种所谓的链式数据结构，它们比数组更灵活，在许多情况下都是高效算法的基础。
- 借助它我们可以准确地定义所面对的算法问题。比如 1.5 节中的 union-find 算法、2.4 节中的优先队列算法和第 3 章中的符号表算法，它们解决问题的方式都是定义数据结构并高效地实现它们的一组操作。这些问题都能够用数据抽象很好地解决。

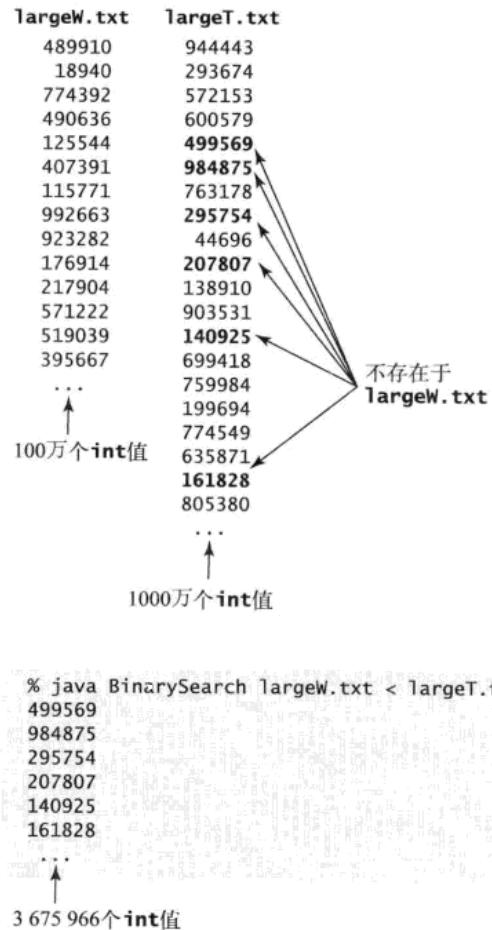
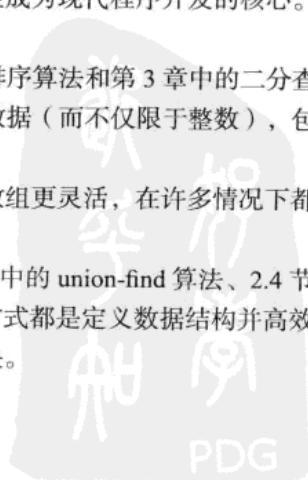


图 1.1.9 为 `BinarySearch` 测试用例准备的大型文件



尽管如此，但我们的重点仍然是对算法的研究。在了解了这些知识以后，我们将学习面向对象编程中和我们的使命相关的另一个重要特性。

50

答疑

问 什么是 Java 的字节码？

答 它是程序的一种低级表示，可以运行于 Java 的虚拟机。将程序抽象为字节码可以保证 Java 程序员的代码能够运行在各种设备之上。

问 Java 允许整型溢出并返回错误值的做法是错误的。难道 Java 不应该自动检查溢出吗？

答 这个问题在程序员中一直是有争议的。简单的回答是它们之所以被称为原始数据类型就是因为缺乏此类检查。避免此类问题并不需要很高深的知识。我们会使用 `int` 类型表示较小的数（小于 10 个十进制位）而使用 `long` 表示 10 亿以上的数。

问 `Math.abs(-2147483648)` 的返回值是什么？

答 `-2147483648`。这个奇怪的结果（但的确是真的）就是整数溢出的典型例子。

问 如何才能将一个 `double` 变量初始化为无穷大？

答 可以使用 Java 的内置常数：`Double.POSITIVE_INFINITY` 和 `Double.NEGATIVE_INFINITY`。

问 能够将 `double` 类型的值和 `int` 类型的值相互比较吗？

答 不通过类型转换是不行的，但请记住 Java 一般会自动进行所需的类型转换。例如，如果 `x` 的类型是 `int` 且值为 3，那么表达式 (`x<3.1`) 的值为 `true`——Java 会在比较前将 `x` 转换为 `double` 类型（因为 3.1 是一个 `double` 类型的字面量）。

问 如果使用一个变量前没有将它初始化，会发生什么？

答 如果代码中存在任何可能导致使用未经初始化的变量的执行路径，Java 都会抛出一个编译异常。

问 Java 表达式 `1/0` 和 `1.0/0.0` 的值是什么？

答 第一个表达式会产生一个运行时除零异常（它会终止程序，因为这个值是未定义的）；第二个表达式的值是 `Infinity`（无穷大）。

51

问 能够使用 `<` 和 `>` 比较 `String` 变量吗？

答 不行，只有原始数据类型定义了这些运算符。请见 1.1.2.3 节。

问 负数的除法和余数的结果是什么？

答 表达式 `a/b` 的商会向 0 取整；`a % b` 的余数的定义是 $(a/b)*b + a \% b$ 恒等于 `a`。例如 `-14/3` 和 `14/-3` 的商都是 `-4`，但 `-14 % 3` 是 `-2`，而 `14 % -3` 是 `2`。

问 为什么使用 `(a && b)` 而非 `(a & b)`？

答 运算符 `&`、`|` 和 `^` 分别表示整数的位逻辑操作与、或和异或。因此，`10|6` 的值为 `14`，`10^6` 的值为 `12`。在本书中我们很少（偶尔）会用到这些运算符。`&&` 和 `||` 运算符仅在独立的布尔表达式中有效，原因是短路求值法则：表达式从左向右求值，一旦整个表达式的值已知则停止求值。

问 嵌套 `if` 语句中的二义性有问题吗？

答 是的。在 Java 中，以下语句：

```
if <expr1> if <expr2> <stmtA> else <stmtB>
```

等价于：

```
if <expr1> { if <expr2> <stmtA> else <stmtB> }
```



即使你想表达的是：

```
if <expr1> { if <expr2> <stmtA> } else <stmtB>
```

避免这种“无主的”`else`陷阱的最好办法是显式地写明所有大括号。

问 一个`for`循环和它的`while`形式有什么区别？

答 `for`循环头部的代码和`for`循环的主体代码在同一个代码段之中。在一个典型的`for`循环中，递增变量一般在循环结束之后都是不可用的；但在和它等价的`while`循环中，递增变量在循环结束之后仍然是可用的。这个区别常常是使用`while`而非`for`循环的主要原因。

52 问 有些Java程序员用`int a[]`而不是`int[] a`来声明一个数组。这两者有什么不同？

答 在Java中，两者等价且都是合法的。前一种是C语言中数组的声明方式。后者是Java提倡的方式，因为变量的类型`int[]`能更清楚地说明这是一个整型的数组。

问 为什么数组的起始索引是0而不是1？

答 这个习惯来源于机器语言，那时要计算一个数组元素的地址需要将数组的起始地址加上该元素的索引。将起始索引设为1要么会浪费数组的第一个元素的空间，要么会花费额外的时间来将索引减1。

问 如果`a[]`是一个数组，为什么`StdOut.println(a)`打印出的是一个十六进制的整数，比如@f62373，而不是数组中的元素呢？

答 问得好。该方法打印出的是这个数组的地址，不幸的是你一般都不需要它。

问 我们为什么不使用标准的Java库来处理输入和图形？

答 我们的确用到了它们，但我们希望使用更简单的抽象模型。`StdIn`和`StdDraw`背后的Java标准库是为实际生产设计的，这些库和它们的API都有些笨重。要想知道它们真正的模样，请查看`StdIn.java`和`StdDraw.java`的代码。

问 我的程序能够重新读取标准输入中的值吗？

答 不行，你只有一次机会，就好像你不能撤销`println()`的结果一样。

问 如果我的程序在标准输入为空之后仍然尝试读取，会发生什么？

答 会得到一个错误。`StdIn.isEmpty()`能够帮助你检查是否还有可用的输入以避免这种错误。

问 这条出错信息是什么意思？

```
Exception in thread "main" java.lang.NoClassDefFoundError: StdIn
```

答 你可能忘记把`StdIn.java`文件放到工作目录中去了。

问 在Java中，一个静态方法能够将另一个静态方法作为参数吗？

53 答 不行，但问得好，因为有很多语言都能够这么做。

练习

1.1.1 给出以下表达式的值：

- a. $(0 + 15) / 2$
- b. $2.0e-6 * 100000000.1$
- c. `true && false || true && true`

1.1.2 给出以下表达式的类型和值：

- a. $(1 + 2.236)/2$
- b. $1 + 2 + 3 + 4.0$



- c. `4.1 >= 4`
 d. `1 + 2 + "3"`

1.1.3 编写一个程序，从命令行得到三个整数参数。如果它们都相等则打印 `equal`，否则打印 `not equal`。

1.1.4 下列语句各有什么问题（如果有的话）？

- a. `if (a > b) then c = 0;`
 b. `if a > b { c = 0; }`
 c. `if (a > b) c = 0;`
 d. `if (a > b) c = 0 else b = 0;`

1.1.5 编写一段程序，如果 `double` 类型的变量 `x` 和 `y` 都严格位于 0 和 1 之间则打印 `true`，否则打印 `false`。

1.1.6 下面这段程序会打印出什么？

```
int f = 0;
int g = 1;
for (int i = 0; i <= 15; i++)
{
    StdOut.println(f);
    f = f + g;
    g = f - g;
}
```

54

1.1.7 分别给出以下代码段打印出的值：

```
a. double t = 9.0;
   while (Math.abs(t - 9.0/t) > .001)
       t = (9.0/t + t) / 2.0;
   StdOut.printf("%.5f\n", t);

b. int sum = 0;
   for (int i = 1; i < 1000; i++)
       for (int j = 0; j < i; j++)
           sum++;
   StdOut.println(sum);

c. int sum = 0;
   for (int i = 1; i < 1000; i *= 2)
       for (int j = 0; j < 1000; j++)
           sum++;
   StdOut.println(sum);
```

1.1.8 下列语句会打印出什么结果？给出解释。

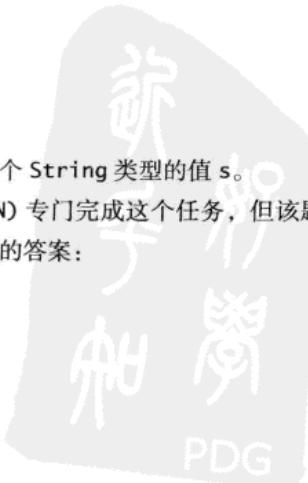
- a. `System.out.println('b');`
 b. `System.out.println('b' + 'c');`
 c. `System.out.println((char) ('a' + 4));`

1.1.9 编写一段代码，将一个正整数 `N` 用二进制表示并转换为一个 `String` 类型的值 `s`。

解答：Java 有一个内置方法 `Integer.toBinaryString(N)` 专门完成这个任务，但该题的目的就是给出这个方法的其他实现方法。下面就是一个特别简洁的答案：

```
String s = "";
for (int n = N; n > 0; n /= 2)
    s = (n % 2) + s;
```

55



1.1.10 下面这段代码有什么问题?

```
int[] a;
for (int i = 0; i < 10; i++)
    a[i] = i * i;
```

解答：它没有用 `new` 为 `a[]` 分配内存。这段代码会产生一个 `variable a might not have been initialized` 的编译错误。

1.1.11 编写一段代码，打印出一个二维布尔数组的内容。其中，使用 * 表示真，空格表示假。打印出行号和列号。

1.1.12 以下代码段会打印出什么结果?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
    a[i] = 9 - i;
for (int i = 0; i < 10; i++)
    a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
    System.out.println(i);
```

1.1.13 编写一段代码，打印出一个 M 行 N 列的二维数组的转置（交换行和列）。1.1.14 编写一个静态方法 `lg()`，接受一个整型参数 N ，返回不大于 $\log_2 N$ 的最大整数。不要使用 `Math` 库。1.1.15 编写一个静态方法 `histogram()`，接受一个整型数组 `a[]` 和一个整数 M 为参数并返回一个大小为 M 的数组，其中第 i 个元素的值为整数 i 在参数数组中出现的次数。如果 `a[]` 中的值均在 0 到 $M-1$ 之间，返回数组中所有元素之和应该和 `a.length` 相等。1.1.16 给出 `exR1(6)` 的返回值：

```
public static String exR1(int n)
{
    if (n <= 0) return "";
    return exR1(n-3) + n + exR1(n-2) + n;
}
```

1.1.17 找出以下递归函数的问题：

```
public static String exR2(int n)
{
    String s = exR2(n-3) + n + exR2(n-2) + n;
    if (n <= 0) return "";
    return s;
}
```

答：这段代码中的基本情况永远不会被访问。调用 `exR2(3)` 会产生调用 `exR2(0)`、`exR2(-3)` 和 `exR2(-6)`，循环往复直到发生 `StackOverflowError`。

1.1.18 请看以下递归函数：

```
public static int mystery(int a, int b)
{
    if (b == 0)      return 0;
    if (b % 2 == 0) return mystery(a+a, b/2);
    return mystery(a+a, b/2) + a;
}
```

`mystery(2, 25)` 和 `mystery(3, 11)` 的返回值是多少？给定正整数 `a` 和 `b`，`mystery(a,b)` 计算的结果是什么？将代码中的 `+` 替换为 `*` 并将 `return 0` 改为 `return 1`，然后回答相同的问题。

1.1.19 在计算机上运行以下程序：

```

public class Fibonacci
{
    public static long F(int N)
    {
        if (N == 0) return 0;
        if (N == 1) return 1;
        return F(N-1) + F(N-2);
    }
    public static void main(String[] args)
    {
        for (int N = 0; N < 100; N++)
            StdOut.println(N + " " + F(N));
    }
}

```

57

计算机用这段程序在一个小时之内能够得到 $F(N)$ 结果的最大 N 值是多少？开发 $F(N)$ 的一个更好的实现，用数组保存已经计算过的值。

- 1.1.20 编写一个递归的静态方法计算 $\ln(N!)$ 的值。
- 1.1.21 编写一段程序，从标准输入按行读取数据，其中每行都包含一个名字和两个整数。然后用 `printf()` 打印一张表格，每行的若干列数据包括名字、两个整数和第一个整数除以第二个整数的结果，精确到小数点后三位。可以用这种程序将棒球球手的击球命中率或者学生的考试分数制成表格。
- 1.1.22 使用 1.1.6.4 节中的 `rank()` 递归方法重新实现 `BinarySearch` 并跟踪该方法的调用。每当该方法被调用时，打印出它的参数 `lo` 和 `hi` 并按照递归的深度缩进。提示：为递归方法添加一个参数来保存递归的深度。
- 1.1.23 为 `BinarySearch` 的测试用例添加一个参数：+ 打印出标准输入中不在白名单上的值；-，则打印出标准输入中在白名单上的值。
- 1.1.24 给出使用欧几里德算法计算 105 和 24 的最大公约数的过程中得到的一系列 p 和 q 的值。扩展该算法中的代码得到一个程序 `Euclid`，从命令行接受两个参数，计算它们的最大公约数并打印出每次调用递归方法时的两个参数。使用你的程序计算 1 111 111 和 1 234 567 的最大公约数。
- 1.1.25 使用数学归纳法证明欧几里德算法能够计算任意一对非负整数 p 和 q 的最大公约数。

58

提高题

- 1.1.26 将三个数字排序。假设 `a`、`b`、`c` 和 `t` 都是同一种原始数字类型的变量。证明以下代码能够将 `a`、`b`、`c` 按照升序排列：


```

if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }

```
- 1.1.27 二项分布。估计用以下代码计算 `binomial(100, 50)` 将会产生的递归调用次数：


```

public static double binomial(int N, int k, double p)
{
    if (N == 0 && k == 0) return 1.0; and if (N < 0 || k < 0) return 0.0;
    return (1.0 - p)*binomial(N-1, k, p) + p*binomial(N-1, k-1);
}

```
- 将已经计算过的值保存在数组中并给出一个更好的实现。
- 1.1.28 删除重复元素。修改 `BinarySearch` 类中的测试用例来删去排序之后白名单中的所有重复元素。

- 1.1.29 等值键。为 `BinarySearch` 类添加一个静态方法 `rank()`，它接受一个键和一个整型有序数组（可能存在重复键）作为参数并返回数组中小于该键的元素数量，以及一个类似的方法 `count()` 来返回数组中等于该键的元素的数量。注意：如果 `i` 和 `j` 分别是 `rank(key, a)` 和 `count(key, a)` 的返回值，那么 `a[i..i+j-1]` 就是数组中所有和 `key` 相等的元素。
- 1.1.30 数组练习。编写一段程序，创建一个 $N \times N$ 的布尔数组 `a[][]`。其中当 `i` 和 `j` 互质时（没有相同因子），`a[i][j]` 为 `true`，否则为 `false`。
- 1.1.31 随机连接。编写一段程序，从命令行接受一个整数 `N` 和 `double` 值 `p`（0 到 1 之间）作为参数，在一个圆上画出大小为 0.05 且间距相等的 `N` 个点，然后将每对点按照概率 `p` 用灰线连接。
59
- 1.1.32 直方图。假设标准输入流中含有一系列 `double` 值。编写一段程序，从命令行接受一个整数 `N` 和两个 `double` 值 `l` 和 `r`。将 (l, r) 分为 `N` 段并使用 `StdDraw` 画出输入流中的值落入每段的数量的直方图。
- 1.1.33 矩阵库。编写一个 `Matrix` 库并实现以下 API：

<code>public class Matrix</code>		
<code>static double dot(double[] x, double[] y)</code>		向量点乘
<code>static double[][] mult(double[][] a, double[][] b)</code>		矩阵和矩阵之积
<code>static double[] transpose(double[][] a)</code>		转置矩阵
<code>static double[] mult(double[][] a, double[] x)</code>		矩阵和向量之积
<code>static double[] mult(double[] y, double[][] a)</code>		向量和矩阵之积

- 编写一个测试用例，从标准输入读取矩阵并测试所有方法。
- 1.1.34 过滤。以下哪些任务需要（在数组中，比如）保存标准输入中的所有值？哪些可以被实现为一个过滤器且仅使用固定数量的变量和固定大小的数组（和 `N` 无关）？在每个问题中，输入都来自于标准输入且含有 `N` 个 0 到 1 的实数。

- 打印出最大和最小的数
- 打印出所有数的中位数
- 打印出第 `k` 小的数，`k` 小于 100
- 打印出所有数的平方和
- 打印出 `N` 个数的平均值
- 打印出大于平均值的数的百分比
- 将 `N` 个数按照升序打印
- 将 `N` 个数按照随机顺序打印

60

实验题

- 1.1.35 模拟掷骰子。以下代码能够计算每种两个骰子之和的准确概率分布：

```
int SIDES = 6;
double[] dist = new double[2*SIDES+1];
for (int i = 1; i <= SIDES; i++)
    for (int j = 1; j <= SIDES; j++)
        dist[i+j] += 1.0;

for (int k = 2; k <= 2*SIDES; k++)
    dist[k] /= 36.0;
```

`dist[i]` 的值就是两个骰子之和为 i 的概率。用实验模拟 N 次掷骰子，并在计算两个 1 到 6 之间的随机整数之和时记录每个值的出现频率以验证它们的概率。 N 要多大才能够保证你的经验数据和准确数据的吻合程度达到小数点后三位？

- 1.1.36 乱序检查。通过实验检查表 1.1.10 中的乱序代码是否能够产生预期的效果。编写一个程序 `ShuffleTest`，接受命令行参数 M 和 N ，将大小为 M 的数组打乱 N 次且在每次打乱之前都将数组重新初始化为 $a[i] = i$ 。打印一个 $M \times M$ 的表格，对于所有的列 j ，行 i 表示的是 i 在打乱后落到 j 的位置的次数。数组中的所有元素的值都应该接近于 N/M 。

- 1.1.37 糟糕的打乱。假设在我们的乱序代码中你选择的是一个 0 到 $N-1$ 而非 i 到 $N-1$ 之间的随机整数。证明得到的结果并非均匀地分布在 $N!$ 种可能性之间。用上一题中的测试检验这个版本。

- 1.1.38 二分查找与暴力查找。根据 1.1.10.4 节给出的暴力查找法编写一个程序 `BruteForceSearch`，在你的计算机上比较它和 `BinarySearch` 处理 `largeW.txt` 和 `largeT.txt` 所需的时间。 61

- 1.1.39 随机匹配。编写一个使用 `BinarySearch` 的程序，它从命令行接受一个整型参数 T ，并会分别针对 $N=10^3$ 、 10^4 、 10^5 和 10^6 将以下实验运行 T 遍：生成两个大小为 N 的随机 6 位正整数数组并找出同时存在于两个数组中的整数的数量。打印一个表格，对于每个 N ，给出 T 次实验中该数量的平均值。 62



1.2 数据抽象

数据类型指的是一组值和一组对这些值的操作的集合。目前，我们已经详细讨论过 Java 的原始数据类型：例如，原始数据类型 `int` 的取值范围是 -2^{31} 到 $2^{31}-1$ 之间的整数，`int` 的操作包括 `+`、`*`、`-`、`/`、`%`、`<` 和 `>`。原则上所有程序都只需要使用原始数据类型即可，但在更高层次的抽象上编写程序会更加方便。在本节中，我们将重点学习定义和使用数据类型，这个过程也被称为数据抽象（它是对 1.1 节所述的函数抽象风格的补充）。

Java 编程的基础主要是使用 `class` 关键字构造被称为引用类型的数据类型。这种编程风格也称为面向对象编程，因为它的核心概念是对象，即保存了某个数据类型的值的实体。如果只有 Java 的原始数据类型，我们的程序会在很大程度上被限制在算术计算上，但有了引用类型，我们就能编写操作字符串、图像、声音以及 Java 的标准库中或者本书的网站上的数百种抽象类型的程序。比各种库中预定义的数据类型更重要的是 Java 编程中的数据类型的种类是无限的，因为你能够定义自己的数据类型来抽象任意对象。

抽象数据类型（ADT）是一种能够对使用者隐藏数据表示的数据类型。用 Java 类来实现抽象数据类型和用一组静态方法实现一个函数库并没有什么不同。抽象数据类型的主要不同之处在于它将数据和函数的实现关联，并将数据的表示方式隐藏起来。在使用抽象数据类型时，我们的注意力集中在 API 描述的操作上而不会去关心数据的表示；在实现抽象数据类型时，我们的注意力集中在数据本身并将实现对该数据的各种操作。

抽象数据类型之所以重要是因为在程序设计上它们支持封装。在本书中，我们将通过它们：

- 以适用于各种用途的 API 形式准确地定义问题；
- 用 API 的实现描述算法和数据结构。

我们研究同一个问题的不同算法的主要原因在于它们的性能特点不同。抽象数据类型正适合于对算法的研究，因为它确保我们可以随时将算法性能的知识应用于实践中：可以在不修改任何用例代码的情况下用一种算法替换另一种算法并改进所有用例的性能。

63
l
64

1.2.1 使用抽象数据类型

要使用一种数据类型并不一定非得知道它是如何实现的，所以我们首先来编写一个使用一种名为 `Counter`（计数器）的简单数据类型的程序。它的值是一个名称和一个非负整数，它的操作有创建对象并初始化为 0、当前值加 1 和获取当前值。这个抽象对象在许多场景中都会用到。例如，这样一个数据类型可以用于电子记票软件，它能够保证投票者所能进行的唯一操作就是将他选择的候选人的计数器加一。我们也可以在分析算法性能时使用 `Counter` 来记录基本操作的调用次数。要使用 `Counter` 对象，首先需要了解应该如何定义数据类型的操作，以及在 Java 语言中应该如何创建和使用某个数据类型的对象。这些机制在现代编程中都非常重要，我们在全书中都会用到它们，因此请仔细学习我们的第一个例子。

1.2.1.1 抽象数据类型的 API

我们使用应用程序编程接口（API）来说明抽象数据类型的行为。它将列出所有构造函数和实例方法（即操作）并简要描述它们的功用，如表 1.2.1 中 `Counter` 的 API 所示。

尽管数据类型定义的基础是一组值的集合，但在 API 可见的仅是对它们的操作，而非它们的意义。因此，抽象数据类型的定义和静态方法库（请见 1.1.6.3 节）之间有许多共同之处：

- 两者的实现均为 Java 类；

- 实例方法可能接受 0 个或多个指定类型的参数，由括号表示并由逗号分隔；
 - 它们可能会返回一个指定类型的值，也可能不会（用 `void` 表示）。
- 当然，它们也有三个显著的不同。
- API 中可能会出现若干个名称和类名相同且没有返回值的函数。这些特殊的函数被称为构造函数。在本例中，`Counter` 对象有一个接受一个 `String` 参数的构造函数。 65
 - 实例方法不需要 `static` 关键字。它们不是静态方法——它们的目的就是操作该数据类型中的值。
 - 某些实例方法的存在是为了尊重 Java 的习惯——我们将此类方法称为继承的方法并在 API 中将它们显示为灰色。

表 1.2.1 计数器的 API

<code>public class Counter</code>		
	<code>Counter(String id)</code>	创建一个名为 <code>id</code> 的计数器
<code>void</code>	<code>increment()</code>	将计数器的值加 1
<code>int</code>	<code>tally()</code>	该对象创建之后计数器被加 1 的次数
<code>String</code>	<code>toString()</code>	对象的字符串表示

和静态方法库的 API 一样，抽象数据类型的 API 也是和用例之间的一份契约，因此它是开发任何用例代码以及实现任意数据类型的起点。在本例中，这份 API 告诉我们可以通过构造函数 `Counter()`、实例方法 `increment()` 和 `tally()`，以及继承的 `toString()` 方法使用 `Counter` 类型的对象。

1.2.1.2 继承的方法

根据 Java 的约定，任意数据类型都能通过在 API 中包含特定的方法从 Java 的内在机制中获益。例如，Java 中的所有数据类型都会继承 `toString()` 方法来返回用 `String` 表示的该类型的值。Java 会在用 + 运算符将任意数据类型的值和 `String` 值连接时调用该方法。该方法的默认实现并不实用（它会返回用字符串表示的该数据类型值的内存地址），因此我们常常会提供实现来重载默认实现，并在此时在 API 中加上 `toString()` 方法。此类方法的例子还包括 `equals()`、`compareTo()` 和 `hashCode()`（请见 1.2.5.5 节）。

1.2.1.3 用例代码

和基于静态方法的模块化编程一样，API 允许我们在不知道实现细节的情况下编写调用它的代码（以及在不知道任何用例代码的情况下编写实现代码）。1.1.7 节介绍的将程序组织为独立模块的机制可以应用于所有的 Java 类，因此它对基于抽象数据类型的模块化编程与对静态函数库一样有效。这样，只要抽象数据类型的源代码 `.java` 文件和我们的程序文件在同一个目录下，或是在标准 Java 库中，或是可以通过 `import` 语句访问，或是可以通过本书网站上介绍的 `classpath` 机制之一访问，该程序就能够使用这个抽象数据类型，模块化编程的所有优势就都能够继续发挥。通过将实现某种数据类型的全部代码封装在一个 Java 类中，我们可以将用例代码推向更高的抽象层次。在用例代码中，你需要声明变量、创建对象来保存数据类型的值并允许通过实例方法来操作它们。尽管你也会注意到它们的一些相似之处，但这种方式和原始数据类型的使用方式非常不同。 66

1.2.1.4 对象

一般来说，可以声明一个变量 `heads` 并将它通过以下代码和 `Counter` 类型的数据关联起来：

```
Counter heads;
```

但如何为它赋值或是对它进行操作呢？这个问题的答案涉及数据抽象中的一个基础概念：对象是能够承载数据类型的值的实体。所有对象都有三大重要特性：状态、标识和行为。对象的状态即数据类型中的值。对象的标识份能够将一个对象区别于另一个对象。可以认为对象的标识就是它在内存中的位置。对象的行为就是数据类型的操作。数据类型的实现的唯一职责就是维护一个对象的身份，这样用例代码在使用数据类型时只需遵守描述对象行为的 API 即可，而无需关注对象状态的表示方法。对象的状态可以为用例代码提供信息，或是产生某种副作用，或是被数据类型的操作所改变。但数据类型的值的表示细节和用例代码是无关的。引用是访问对象的一种方式。Java 使用术语引用类型以示和原始数据类型（变量和值相关联）的区别。不同的 Java 实现中引用的实现细节也各不相同，但可以认为引用就是内存地址，如图 1.2.1 所示（简洁起见，图中的内存地址为三位数）。

1.2.1.5 创建对象

每种数据类型中的值都存储于一个对象中。要创建（或实例化）一个对象，我们用关键字 `new` 并紧跟类名以及 `()`（或在括号中指定一系列的参数，如果构造函数需要的话）来触发它的构造函数。构造函数没有返回值，因为它总是返回它的数据类型的对象的引用。每当用例调用了 `new()`，系统都会：

- 为新的对象分配内存空间；
- 调用构造函数初始化对象中的值；
- 返回该对象的一个引用。

在用例代码中，我们一般都会在一条声明语句中创建一个对象并通过将它和一个变量关联来初始化该变量，和使用原始数据类型时一样。和原始数据类型不同的是，变量关联的是指向对象的引用而非数据类型的值本身。我们可以用同一个类创建无数对象——每个对象都有自己的标识，且所存储的值和另一个相同类型的对象可以相同也可以不同。例如，以下代码创建了两个不同的 `Counter` 对象：

```
Counter heads = new Counter("heads");
Counter tails = new Counter("tails");
```

抽象数据类型向用例隐藏了值的表示细节。可以假定每个 `Counter` 对象中的值是一个 `String` 类型的名称和一个 `int` 计数器，但不能编写依赖于任何特定表示方法的代码（即使知道假定是否正

确——也许计数器是一个 `long` 值呢）对象的创建过程如图 1.2.2 所示。

1.2.1.6 调用实例方法

实例方法的意义在于操作数据类型中的值，因此 Java 语言提供了一种特别的机制来

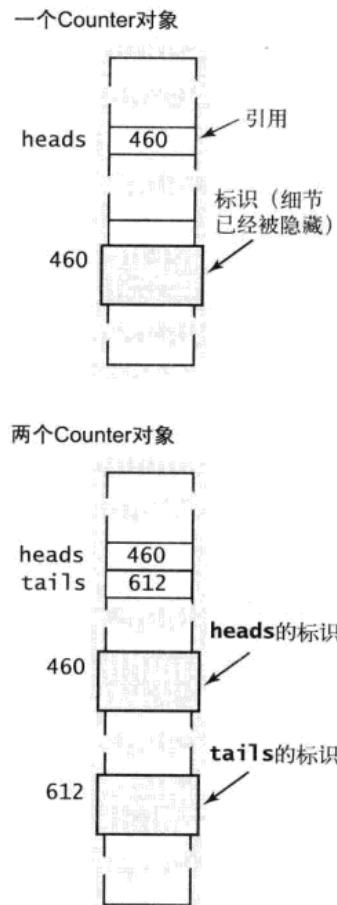


图 1.2.1 对象的表示

图 1.2.2 创建对象

触发实例方法，它突出了实例方法和对象之间的联系。具体来说，我们调用一个实例方法的方式是先写出对象的变量名，紧接着是一个句点，然后是实例方法的名称，之后是0个或多个在括号中并由逗号分隔的参数。实例方法可能会改变数据类型中的值，也可能只是访问数据类型中的值。实例方法拥有我们在1.1.6.3节讨论过的静态方法的所有性质——参数按值传递，方法名可以被重载，方法可以有返回值，它们也许还会产生一些副作用。但它们还有一个特别的性质：方法的每次触发都是和一个对象相关的。例如，以下代码调用了实例方法 `increment()` 来操作 `Counter` 对象 `heads`（在这里该操作会将计数器的值加1）：

```
heads.increment();
```

而以下代码会调用实例方法 `tally()` 两次，第一次操作的是 `Counter` 对象 `heads`，第二次是 `Counter` 对象 `tails`（这里该操作会返回计数器的 int 值）：

```
heads.tally() - tails.tally();
```

以上示例的调用过程见图 1.2.3。

正如这些例子所示，在用例中实例方法和静态方法的调用方式完全相同——可以通过语句（`void` 方法）也可以通过表达式（有返回值的方法）。静态方法的主要作用是实现函数；非静态（实例）方法的主要作用是实现数据类型的操作。两者都可能出现在用例代码中，但很容易就可以区分它们，因为静态方法调用的开头是类名（按习惯为大写），而非静态方法调用的开头总是对象名（按习惯为小写）。表 1.2.2 总结了这些不同之处。

68

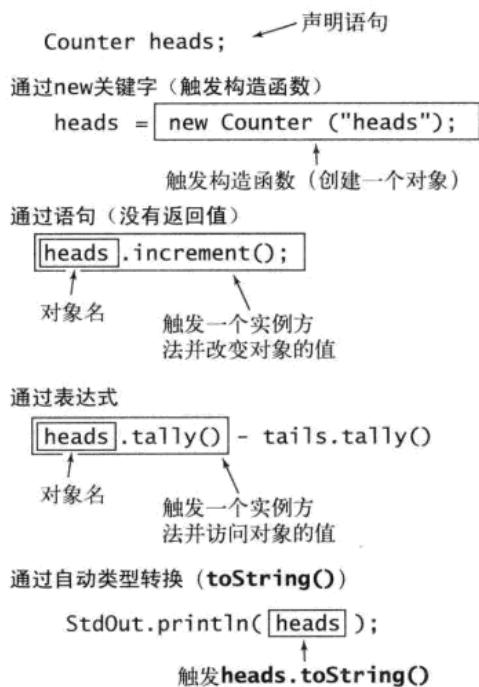


图 1.2.3 触发实例方法的各种方式

表 1.2.2 实例方法与静态方法

	实例方法	静态方法
举例	<code>heads.increment()</code>	<code>Math.sqrt(2.0)</code>
调用方式	对象名	类名
参量	对象的引用和方法的参数	方法的参数
主要作用	访问或改变对象的值	计算返回值

1.2.1.7 使用对象

通过声明语句可以将变量名赋给对象，在代码中，我们不仅可以用该变量创建对象和调用实例方法，也可以像使用整数、浮点数和其他原始数据类型的变量一样使用它。要开发某种给定数据类型的用例，我们需要：

- 声明该类型的变量，以用来引用对象；
- 使用关键字 `new` 触发能够创建该类型的对象的一个构造函数；
- 使用变量名在语句或表达式中调用实例方法。

例如，下面用例代码中的 `Flips` 类就使用了 `Counter` 类。它接受一个命令行参数 `T` 并模拟 `T` 次掷硬币（它还调用了 `StdRandom` 类）。除了这些直接用法外，我们可以和使用原始数据类型的

变量一样使用和对象关联的变量：

- 赋值语句；
- 向方法传递对象或是从方法中返回对象；
- 创建并使用对象的数组。

```
public class Flips
{
    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        Counter heads = new Counter("heads");
        Counter tails = new Counter("tails");
        for (int t = 0; t < T; t++)
            if (StdRandom.bernoulli(0.5))
                heads.increment();
            else tails.increment();
        StdOut.println(heads);
        StdOut.println(tails);
        int d = heads.tally() - tails.tally();
        StdOut.println("delta: " + Math.abs(d));
    }
}
```

Counter 类的用例，模拟 T 次掷硬币

```
% java Flips 10
5 heads
5 tails
delta: 0

% java Flips 10
8 heads
2 tails
delta: 6

% java Flips 1000000
499710 heads
500290 tails
delta: 580
```

接下来将逐个分析它们。你会发现，你需要从引用而非值的角度去考虑问题才能理解这些用法的行为。

1.2.1.8 赋值语句

使用引用类型的赋值语句将会创建该引用的一个副本。赋值语句不会创建新的对象，而只是创建另一个指向某个已经存在的对象的引用。这种情况被称为别名：两个变量同时指向同一个对象。别名的效果可能会出乎你的意料，因为对于原始数据类型的变量，情况不同，你必须理解其中的差异。如果 *x* 和 *y* 是原始数据类型的变量，那么赋值语句 *x* = *y* 会将 *y* 的值复制到 *x* 中。对于引用类型，复制的是引用（而非实际的值）。在 Java 中，别名是 bug 的常见原因，如下例所示（图 1.2.4）：

```
Counter c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
StdOut.println(c1);
```

对于一般的 *toString()* 实现，这段代码将会打印出 "2 ones"。这可能并不是我们想要的，而且乍一看有些奇怪。这种问题经常出现在使用对象经验不足的人所编写的程序之中（可能就是你，所以请集中注意力！）。改变一个对象的状态将会影响到所有和该对象的别名有关的代码。我们习惯于认为两个不同的

```
Counter c1;
c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
```

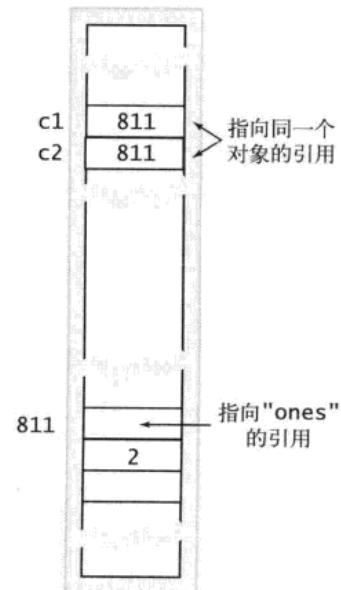


图 1.2.4 别名

原始数据类型的变量是相互独立的，但这种感觉对于引用类型的变量并不适用。

69
70

1.2.1.9 将对象作为参数

可以将对象作为参数传递给方法，这一般都能简化用例代码。例如，当我们使用 `Counter` 对象作为参数时，本质上我们传递的是一个名称和一个计数器，但我们只需要指定一个变量。当我们调用一个需要参数的方法时，该动作在 Java 中的效果相当于每个参数值都出现在了一个赋值语句的右侧，而参数名则在该赋值语句的左侧。也就是说，Java 将参数值的一个副本从调用端传递给了方法，这种方式称为按值传递（请见 1.1.6.3 节）。这种方式的一个重要后果是方法无法改变调用端变量的值。对于原始数据类型来说，这种策略正是我们所期望的（两个变量互相独立），但每当使用引用类型作为参数时我们创建的都是别名，所以就必须小心。换句话说，这种约定将会传递引用的值（复制引用），也就是传递对象的引用。例如，如果我们传递了一个指向 `Counter` 类型的对象的引用，那么方法虽然无法改变原始的引用（比如将它指向另一个 `Counter` 对象），但它能够改变该对象的值，比如通过该引用调用 `increment()` 方法。

1.2.1.10 将对象作为返回值

当然也能够将对象作为方法的返回值。方法可以将它的参数对象返回，如下面的例子所示，也可以创建一个对象并返回它的引用。这种能力非常重要，因为 Java 中的方法只能有一个返回值——有了对象我们的代码实际上就能返回多个值。

```
public class FlipsMax
{
    public static Counter max(Counter x, Counter y)
    {
        if (x.tally() > y.tally()) return x;
        else return y;
    }

    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        Counter heads = new Counter("heads");
        Counter tails = new Counter("tails");
        for (int t = 0; t < T; t++)
            if (StdRandom.bernoulli(0.5))
                heads.increment();
            else tails.increment();

        if (heads.tally() == tails.tally())
            StdOut.println("Tie");
        else StdOut.println(max(heads, tails) + " wins");
    }
}
```

一个接受对象作为参数并将对象作为返回值的静态方法的例子

71

1.2.1.11 数组也是对象

在 Java 中，所有非原始数据类型的值都是对象。也就是说，数组也是对象。和字符串一样，Java 语言对于数组的某些操作有特殊的支持：声明、初始化和索引。和其他对象一样，当我们将数组传递给一个方法或是将一个数组变量放在赋值语句的右侧时，我们都是在创建该数组引用的一个副本，而非数组的副本。对于一般情况，这种效果正合适，因为我们期望方法能够重新安排数组的

条目并修改数组的内容，如 `java.util.Arrays.sort()` 或表 1.1.10 讨论的 `shuffle()` 方法。

1.2.1.12 对象的数组

我们已经看到，数组元素可以是任意类型的数据：我们实现的 `main()` 方法的 `args[]` 参数就是一个 `String` 对象的数组。创建一个对象的数组需要以下两个步骤：

- 使用方括号语法调用数组的构造函数创建数组；
- 对于每个数组元素调用它的构造函数创建相应的对象。

例如，下面这段代码模拟的是掷骰子。它使用了一个 `Counter` 对象的数组来记录每种可能的值的出现次数。在 Java 中，对象数组即是一个由对象的引用组成的数组，而非所有对象本身组成的数组。如果对象非常大，那么在移动它们时由于只需要操作引用而非对象本身，这就会大大提高效率；如果对象很小，每次获取信息时都需要通过引用反而会降低效率。

```
public class Rolls
{
    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        int SIDES = 6;
        Counter[] rolls = new Counter[SIDES+1];
        for (int i = 1; i <= SIDES; i++)
            rolls[i] = new Counter(i + "'s");

        for (int t = 0; t < T; t++)
        {
            int result = StdRandom.uniform(1, SIDES+1);
            rolls[result].increment();
        }
        for (int i = 1; i <= SIDES; i++)
            StdOut.println(rolls[i]);
    }
}
```

```
% java Rolls 1000000
167308 1's
166540 2's
166087 3's
167051 4's
166422 5's
166592 6's
```

72 模拟 T 次掷骰子的 `Counter` 对象的用例

有了这些对象的知识，运用数据抽象的思想编写代码（定义和使用数据类型，将数据类型的值封装在对象中）的方式称为面向对象编程。刚才学习的基本概念是我们面向对象编程的起点，因此有必要对它们进行简单的总结。数据类型指的是一组值和一组对值的操作的集合。我们会将数据类型实现在独立的 Java 类模块中并编写它们的用例。对象是能够存储任意该数据类型的值的实体，或数据类型的实例。对象有三大关键性质：状态、标识和行为。一个数据类型的实现所支持的操作如下。

- 创建对象（创造它的标识）：使用 `new` 关键字触发构造函数并创建对象，初始化对象中的值并返回对它的引用。
- 操作对象中的值（控制对象的行为，可能会改变对象的状态）：使用和对象关联的变量调用实例方法来对对象中的值进行操作。
- 操作多个对象：创建对象的数组，像原始数据类型的值一样将它们传递给方法或是从方法中返回，只是变量关联的是对象的引用而非对象本身。

73 这些能力是这种灵活且应用广泛的现代编程方式的基础，也是我们在本书中对算法研究的基础。

1.2.2 抽象数据类型举例

Java 语言内置了上千种抽象数据类型，我们也会为了辅助算法研究创建许多其他抽象数据类型。实际上，我们编写的每一个 Java 程序实现的都是某种数据类型（或是一个静态方法库）。为了控制复杂度，我们会明确地说明在本书中用到的所有抽象数据类型的 API（实际上并不多）。

在本节中，我们会举一些抽象数据类型的例子，以及它们的一些用例。在某些情况下，我们会节选一些含有数十个方法的 API 的一部分。我们将会用这些 API 展示一些实例以及在本书中会用到的一些方法，并用它们说明要使用一个抽象数据类型并不需要了解其实现细节。

作为参考，下页显示了我们在本书中将会用到或开发的所有数据类型。它们可以被分为以下几类。

- `java.lang.*` 中的标准系统抽象数据类型，可以被任意 Java 程序调用。
- Java 标准库中的抽象数据类型，如 `java.swt`、`java.net` 和 `java.io`，它们也可以被任意 Java 程序调用，但需要 `import` 语句。
- I/O 处理类抽象数据类型，和 `StdIn` 和 `StdOut` 类似，允许我们处理多个输入输出流。
- 面向数据类抽象数据类型，它们的主要作用是通过封装数据的表示简化数据的组织和处理。稍后在本节中我们将介绍在计算几何和信息处理中的几个实际应用的例子，并会在以后将它们作为抽象数据类型用例的范例。
- 集合类抽象数据类型，它们的主要用途是简化对同一类型的一组数据的操作。我们将会在 1.3 节中介绍基本的 `Bag`、`Stack` 和 `Queue` 类，在第 2 章中介绍优先队列（PQ）及其相关的类，在第 3 章和第 5 章中分别介绍符号表（ST）和集合（SET）以及相关的类。
- 面向操作的抽象数据类型，我们用它们分析各种算法，如 1.4 节和 1.5 节所述。
- 图算法相关的抽象数据类型，它们包括一些用来封装各种图的表示的面向数据的抽象数据类型，和一些提供图的处理算法的面向操作的抽象数据类型。

这个列表中并没有包含我们将在练习中遇到的某些抽象数据类型，读者可以在本书的索引中找到它们。另外，如 1.2.4.1 节所述，我们常常通过描述性的前缀来区分各种抽象数据类型的多种实现。从整体上来说，我们使用的抽象数据类型说明组织并理解你所使用的数据结构是现代编程中的重要因素。

一般的应用程序可能只会使用这些抽象数据类型中的 5 ~ 10 个。在本书中，开发和组织抽象数据类型的主要目标是使程序员们在编写用例时能够轻易地利用它们的一小部分。

74

1.2.2.1 几何对象

面向对象编程的一个典型例子是为几何对象设计数据类型。例如，表 1.2.3 至表 1.2.5 中的 API 为三种常见的几何对象定义了相应的抽象数据类型：`Point2D`（平面上的点）、`Interval1D`（直线上的间隔）、`Interval2D`（平面上的二维间隔，即和数轴对齐的长方形）。和以前一样，这些 API 都是自文档化的，它们的用例十分容易理解，列在了表 1.2.5 的后面。这段代码从命令行读取一个 `Interval2D` 的边界和一个整数 T ，在单位正方形内随机生成 T 个点并统计落在间隔之内的点数（用来估计该长方形的面积）。为了表现效果，用例还画出了间隔和落在间隔之外的所有点。这种计算方法是一个模型，它将计算几何图形的面积和体积的问题转化为了判定一个点是否落在该图形中（稍稍简单，但仍然不那么容易）。我们当然也能为其他几何对象定义 API，比如线段、三角形、多边形、圆等，不过实现它们的相关操作可能十分有挑战性。本节末尾的练习会考察其中几个例子。



java.lang 中的标准 Java 系统类型	
Integer	<code>int</code> 的封装类
Double	<code>double</code> 的封装类
String	可由索引访问的 <code>char</code> 值序列
StringBuilder	字符串构造类
其他 Java 数据类型	
<code>java.awt.Color</code>	颜色
<code>java.awt.Font</code>	字体
<code>java.net.URL</code>	URL
<code>java.io.File</code>	文件
我们的标准 I/O 类型	
<code>In</code>	输入流
<code>Out</code>	输出流
<code>Draw</code>	绘图类
用于用例的面向数据的数据类型	
<code>Point2D</code>	平面上的点
<code>Interval1D</code>	一维间隔
<code>Interval2D</code>	二维间隔
<code>Date</code>	日期
<code>Transaction</code>	换位
用于算法分析的数据类型	
<code>Counter</code>	计数器
<code>Accumulator</code>	累加器
<code>VisualAccumulator</code>	可视累加器
<code>Stopwatch</code>	计时器

集合类数据类型	
<code>Stack</code>	下压栈
<code>Queue</code>	先进先出 (FIFO) 队列
<code>Bag</code>	包
<code>MinPQ, MaxPQ</code>	优先队列
<code>IndexMinPQ, IndexMaxPQ</code>	索引优先队列
<code>ST</code>	符号表
<code>SET</code>	集合
<code>StringST</code>	符号表 (字符串键)
面向数据的图数据类型	
<code>Graph</code>	无向图
<code>Digraph</code>	有向图
<code>Edge</code>	边 (加权)
<code>EdgeWeightedGraph</code>	无向图 (加权)
<code>DirectedEdge</code>	边 (有向, 加权)
<code>EdgeWeightedDigraph</code>	图 (有向, 加权)
面向操作的图数据类型	
<code>UF</code>	动态连通性
<code>DepthFirstPaths</code>	路径的深度优先搜索
<code>CC</code>	连通分量
<code>BreadthFirstPaths</code>	路径的广度优先搜索
<code>DirectedDFS</code>	有向图路径的深度优先搜索
<code>DirectedBFS</code>	有向图路径的广度优先搜索
<code>TransitiveClosure</code>	所有路径
<code>Topological</code>	拓扑排序
<code>DepthFirstOrder</code>	深度优先搜索顶点被访问的顺序
<code>DirectedCycle</code>	环的搜索
<code>SCC</code>	强连通分量
<code>MST</code>	最小生成树
<code>SP</code>	最短路径

75

本书中使用的部分抽象数据类型

表 1.2.3 平面上的点的 API

<code>public class Point2D</code>	
<code>Point2D(double x, double y)</code>	创建一个点
<code>double x()</code>	<code>x</code> 坐标
<code>double y()</code>	<code>y</code> 坐标
<code>double r()</code>	极径 (极坐标)
<code>double theta()</code>	极角 (极坐标)
<code>double distTo(Point2D that)</code>	从该点到 <code>that</code> 的欧几里德距离
<code>void draw()</code>	用 <code>StdDraw</code> 绘出该点

表 1.2.4 直线上间隔的 API

<code>public class Interval1D</code>		
	<code>Interval1D(double lo, double hi)</code>	创建一个间隔
<code>double length()</code>		间隔长度
<code>boolean contains(double x)</code>		<code>x</code> 是否在间隔中
<code>boolean intersect(Interval1D that)</code>		该间隔是否和间隔 <code>that</code> 相交
<code>void draw()</code>		用 <code>StdDraw</code> 绘出该间隔

表 1.2.5 平面上的二维间隔的 API

<code>public class Interval2D</code>		
	<code>Interval2D(Interval1D x, Interval1D y)</code>	创建一个二维间隔
<code>double area()</code>		二维间隔的面积
<code>boolean contains(Point2D p)</code>		<code>p</code> 是否在二维间隔中
<code>boolean intersect(Interval2D that)</code>		该间隔是否和二维间隔 <code>that</code> 相交
<code>void draw()</code>		用 <code>StdDraw</code> 绘出该二维间隔

```

public static void main(String[] args)
{
    double xlo = Double.parseDouble(args[0]);
    double xhi = Double.parseDouble(args[1]);
    double ylo = Double.parseDouble(args[2]);
    double yhi = Double.parseDouble(args[3]);
    int T = Integer.parseInt(args[4]);

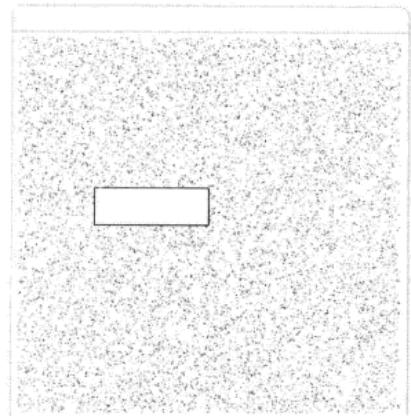
    Interval1D xinterval = new Interval1D(xlo, xhi);
    Interval1D yinterval = new Interval1D(ylo, yhi);
    Interval2D box = new Interval2D(x, y);
    box.draw();

    Counter c = new Counter("hits");
    for (int t = 0; t < T; t++)
    {
        double x = Math.random();
        double y = Math.random();
        Point2D p = new Point(x, y);
        if (box.contains(p)) c.increment();
        else                  p.draw();
    }

    StdOut.println(c);
    StdOut.println(box.area());
}

```

Interval2D 的测试用例



```
% java Interval2D .2 .5 .5 .6 10000
297 hits
.03
```

处理几何对象的程序在自然世界模型、科学计算、电子游戏、电影等许多应用的计算中有着广泛的应用。此类程序的研发已经发展成了计算几何学的这门影响深远的研究学科。在贯穿全书的众多例子中你会看到，我们在本书中学习的许多算法在这个领域都有应用。在这里我们要说明的是直接表示几何对象的抽象数据类型的定义并不困难且在用例中的应用也十分简洁。本书网站和本节末尾的若干练习都证明了这一点。

1.2.2.2 信息处理

无论是需要处理数百万信用卡交易的银行，还是需要处理数十亿点击的网络分析公司，或是需

要处理数百万实验观察结果的科学小组，无数应用的核心都是组织和处理信息。抽象数据类型是组织信息的一种自然方式。虽然没有给出细节，表 1.2.6 中的两份 API 也展示了商业应用程序中的一种典型做法。这里的主要思想是定义和真实世界中的物体相对应的对象。一个日期就是一个月、日和年的集合，一笔交易就是一个客户、日期和金额的集合。这只是两个例子，我们也可以为客户、时间、地点、商品、服务和其他任何东西定义对象以保存相关的信息。每种数据类型都包含能够创建对象的构造函数和用于访问其中数据的方法。为了简化用例的代码，我们为每个类型都提供了两个构造函数，一个接受适当类型的数据，另一个则能够解析字符串中的数据（细节请见练习 1.2.19）。和以前一样，用例并不需要知道数据的表示方法。用这种方式组织数据最常见的理由是将一个对象和它相关的数据变成一个整体：我们可以维护一个 `Transaction` 对象的数组，将 `Date` 值作为参数或是某个方法的返回值等。这些数据类型的重点在于封装数据，同时它们也可以确保用例的代码不依赖于数据的表示方法。我们不会深究这种组织信息的方式，需要注意的只是这种做法，以及实现继承的方法 `toString()`、`compareTo()`、`equals()` 和 `hashCode()` 可以使我们的算法处理任意类型的数据。我们会在 1.2.5.4 节中详细讨论继承的方法。例如，我们已经注意到，根据 Java 的习惯，在数据结构中包含一个 `toString()` 的实现可以帮助用例打印出由对象中的值组成的一个字符串。我们会在 1.3 节、2.5 节、3.4 节和 3.5 节中用 `Date` 类和 `Transaction` 类作为例子考察其他继承的方法所对应的习惯用法。1.3 节给出了有关数据类型和 Java 语言的类型参数（泛型）机制的几个经典例子，它们都遵循了这些习惯用法。第 2 章和第 3 章也都利用了泛型和继承的方法来实现可以处理任意数据类型的高效排序和查找算法。

表 1.2.6 商业应用程序中的示例 API（日期和交易）

<code>public class Date implements Comparable<Date></code>	
<code> Date(int month, int day, int year)</code>	创建一个日期
<code> Date(String date)</code>	创建一个日期（解析字符串的构造函数）
<code> int month()</code>	月
<code> int day()</code>	日
<code> int year()</code>	年
<code> String toString()</code>	对象的字符串表示
<code> boolean equals(Object that)</code>	该日期和 <code>that</code> 是否相同
<code> int compareTo(Date that)</code>	将该日期和 <code>that</code> 比较
<code> int hashCode()</code>	散列值
<code>public class Transaction implements Comparable<Transaction></code>	
<code> Transaction(String who, Date when,</code>	
<code> double amount)</code>	
<code> Transaction(String transaction)</code>	创建一笔交易（解析字符串的构造函数）
<code> String who()</code>	用户名
<code> Date when()</code>	交易日期
<code> double amount()</code>	交易金额
<code> String toString()</code>	对象的字符串表示
<code> boolean equals(Object that)</code>	该笔交易和 <code>that</code> 是否相同
<code> int compareTo(Transaction that)</code>	将该笔交易和 <code>that</code> 比较
<code> int hashCode()</code>	散列值

每当遇到逻辑上相关的不同类型的数据时，你都应该考虑像刚才的例子那样定义一个抽象数据类型。这么做能够帮助我们组织数据并在一般应用程序中极大地简化使用者的代码。它是我们在通向数据抽象之路上迈出的重要一步。

1.2.2.3 字符串

Java 的 `String` 是一种重要而实用的抽象数据类型。一个 `String` 值是一串可以由索引访问的 `char` 值。`String` 对象拥有许多实例方法，如表 1.2.7 所示。

表 1.2.7 Java 的字符串 API（部分）

Public class <code>String</code>	
<code>String()</code>	创建一个空字符串
<code>int length()</code>	字符串长度
<code>int charAt(int i)</code>	第 <code>i</code> 个字符
<code>int indexOf(String p)</code>	<code>p</code> 第一次出现的位置（如果没有则返回 -1）
<code>int indexOf(String p, int i)</code>	<code>p</code> 在 <code>i</code> 个字符后第一次出现的位置（如果没有则返回 -1）
<code>String concat(String t)</code>	将 <code>t</code> 附在该字符串末尾
<code>String substring(int i, int j)</code>	该字符串的子字符串（第 <code>i</code> 个字符到第 <code>j-1</code> 个字符）
<code>String[] split(String delim)</code>	使用 <code>delim</code> 分隔符切分字符串
<code>int compareTo(String t)</code>	比较字符串
<code>boolean equals(String t)</code>	该字符串的值和 <code>t</code> 的值是否相同
<code>int hashCode()</code>	散列值

`String` 值和字符数组类似，但两者是不同的。数组能够通过 Java 语言的内置语法访问每个字符，`String` 则为索引访问、字符串长度以及其他许多操作准备了实例方法。另一方面，Java 语言为 `String` 的初始化和连接提供了特别的支持：我们可以直接使用字符串字面量而非构造函数来创建并初始化一个字符串，还可以直接使用 + 运算符代替 `concat()` 方法。我们不需要了解实现的细节，但是在第 5 章中你会看到，了解某些方法的性能特点在开发字符串处理算法时是非常重要的。为什么不直接使用字符数组代替 `String` 值？对于任何抽象数据类型，这个问题的答案都是一样的：为了使代码更加简洁清晰。有了 `String` 类型，我们可以写出清晰干净的用例代码而无需关心字符串的表示方式。先看一下右侧这段短小的列表，其中甚至含有一些需要我们在第 5 章才会学到的高级算法才能实现的强大操作。例如，`split()` 方法的参数可以是正则表达式（请见 5.4 节），“典型的字符串处理代码”（显示在下页）中 `split()` 的参数是 “`\s+`”，它表示“一个或多个制表符、空格、换行符或回车”。

方法	返回值
<code>a.length()</code>	7
<code>a.charAt(4)</code>	i
<code>a.concat(c)</code>	"now is to"
<code>a.indexOf("is")</code>	4
<code>a.substring(2, 5)</code>	"w i"
<code>a.split(" ")[0]</code>	"now"
<code>a.split(" ")[1]</code>	"is"
<code>b.equals(c)</code>	false

字符串操作举例

任 务	实 现
判断字符串是否是一条回文	<pre>public static boolean isPalindrome(String s) { int N = s.length(); for (int i = 0; i < N/2; i++) if (s.charAt(i) != s.charAt(N-1-i)) return false; return true; }</pre>
从一个命令行参数中提取文件名和扩展名	<pre>String s = args[0]; int dot = s.indexOf("."); String base = s.substring(0, dot); String extension = s.substring(dot + 1, s.length());</pre>
打印出标准输入中所有含有通过命令行指定的字符串的行	<pre>String query = args[0]; while (!StdIn.isEmpty()) { String s = StdIn.readLine(); if (s.contains(query)) StdOut.println(s); }</pre>
以空白字符为分隔符从 StdIn 中创建一个字符串数组	<pre>String input = StdIn.readAll(); String[] words = input.split("\\s+");</pre>
检查一个字符串数组中的元素是否已按照字母表顺序排列	<pre>public boolean isSorted(String[] a) { for (int i = 1; i < a.length; i++) { if (a[i-1].compareTo(a[i]) > 0) return false; } return true; }</pre>

81

典型的字符串处理代码

1.2.2.4 再谈输入输出

1.1节中的 StdIn、StdOut 和 StdDraw 标准库的一个缺点是对于任意程序，我们只能接受一个输入文件、向一个文件输出或是产生一幅图像。有了面向对象编程，我们就能定义类似的机制来在一个程序中同时处理多个输入流、输出流和图像。具体来说，我们的标准库定义了数据类型 **In**、**Out** 和 **Draw**，它们的 API 如表 1.2.8 至表 1.2.10 所示。当使用一个 **String** 类型的参数调用它们的构造函数时，**In** 和 **Out** 会首先尝试在当前目录下查找指定的文件。如果找不到，它会假设该参数是一个网站的名称并尝试连接到那个网站（如果该网站不存在，它会抛出一个运行时异常）。无论哪种情况，指定的文件或网站都会成为被创建的输入或输出流对象的来源或目标，所有 **read***(*O*) 和 **print***(*O*) 方法都会指向那个文件或网站（如果你使用的是无参数的构造函数，对象将会使用标准的输入输出流）。这种机制使得单个程序能够处理多个文件和图像；你也能将这些对象赋给变量，将它们当做方法的参数、作为方法的返回值或是创建它们的数组，可以像操作任何类型的对象那样操作它们。下页所示的程序 **Cat** 就是一个 **In** 和 **Out** 的用例，它使用了多个输入流来将多个输入文件归并到同一个输出文件中。**In** 和 **Out** 类也包括将仅含 **int**、**double** 或 **String** 类型值的文件读取为一个数组的静态方法（请见 1.3.1.5 节和练习 1.2.15）。



```

public class Cat
{
    public static void main(String[] args)
    { // 将所有输入文件复制到输出流（最后一个参数）中
        Out out = new Out(args[args.length-1]);
        for (int i = 0; i < args.length - 1; i++)
        { // 将第i个输入文件复制到输出流中
            In in = new In(args[i]);
            String s = in.readAll();
            out.println(s);
            in.close();
        }
        out.close();
    }
}

```

```

% more in1.txt
This is
% more in2.txt
a tiny
test.
% java Cat in1.txt in2.txt out.txt
% more out.txt
This is
a tiny
test.

```

82

In和Out的用例示例

表 1.2.8 我们的输入流数据类型的 API

public class In		
In()	从标准输入创建输入流	
In(String name)	从文件或网站创建输入流	
boolean isEmpty()	如果输入流为空则返回 true，否则返回 false	
int readInt()	读取一个 int 类型的值	
double readDouble()	读取一个 double 类型的值	
...		
void close()	关闭输入流	

注：In 对象也支持 StdIn 所支持的所有操作。

表 1.2.9 我们的输出流数据类型的 API

public class Out		
Out()	从标准输出创建输出流	
Out(String name)	从文件创建输出流	
void print(String s)	将 s 添加到输出流中	
void println(String s)	将 s 和一个换行符添加到输出流中	
void println()	将一个换行符添加到输出流中	
void printf(String f, ...)	格式化并打印到输出流中	
void close()	关闭输出流	

注：Out 对象也支持 StdOut 所支持的所有操作。

表 1.2.10 我们的绘图数据类型的 API

public class Draw		
Draw()		
void line(double x0, double y0, double x1, double y1)		
void point(double x, double y)		
...		

注：Draw 对象也支持 StdDraw 所支持的所有操作。

83

1.2.3 抽象数据类型的实现

和静态方法库一样，我们也要使用 Java 的类（`class`）实现抽象数据类型并将所有代码放入一个和类名相同并带有`.java`扩展名的文件中。文件的第一部分语句会定义表示数据类型的值的实例变量。它们之后是实现对数据类型的值的操作的构造函数和实例方法。实例方法可以是公共的（在 API 中说明）或是私有的（用于辅助计算，用例无法使用）。一个数据类型的定义中可能含有多个构造函数，而且也可能含有静态方法，特别是单元测试用例`main()`，它通常在调试和测试中很实用。作为第一个例子，我们来学习 1.2.1.1 节定义的`Counter`抽象数据类型的实现。它的完整实现（带有注释）如图 1.2.5 所示，在对它的各个部分的讨论中，我们还将该图作为参考。本书后面开发的每个抽象数据类型的实现都会含有和这个简单例子相同的元素。

```
public class Counter
{
    // 实例变量的声明
    private final String name;
    private int count;
    ...
}
```

抽象数据类型中的实例变量是私有的

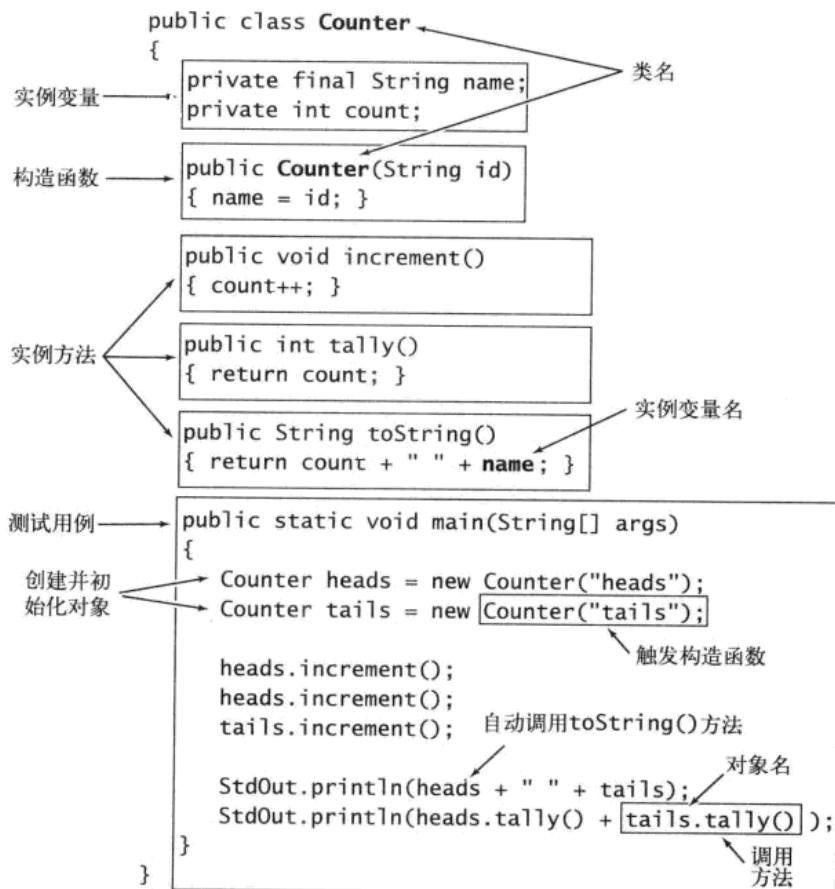


图 1.2.5 详解数据类型的定义类

1.2.3.1 实例变量

要定义数据类型的值（即每个对象的状态），我们需要声明实例变量，声明的方式和局部变量差不多。实例变量和你所熟悉的静态方法或是某个代码段中的局部变量最关键的区别在于：每一时刻每个局部变量只会有一个值，但每个实例变量则对应着无数值（数据类型的每个实例对象都会有一个）。

这并不会产生二义性，因为我们在访问实例变量时都需要通过一个对象——我们访问的是这个对象的值。同样，每个实例变量的声明都需要一个可见性修饰符。在抽象数据类型的实现中，我们会使用 `private`，也就是使用 Java 语言的机制来保证向使用者隐藏抽象数据类型中的数据表示，如下面的示例所示。如果该值在初始化之后不应该再被改变，我们也会使用 `final`。`Counter` 类型含有两个实例变量，一个 `String` 类型的值 `name` 和一个 `int` 类型的值 `count`。如果我们使用 `public` 修饰这些实例变量（在 Java 中是允许的），那么根据定义，这种数据类型就不再是抽象的了，因此我们不会这么做。

1.2.3.2 构造函数

每个 Java 类都至少含有一个构造函数以创建一个对象的标识。构造函数类似于一个静态方法，但它能够直接访问实例变量且没有返回值。一般来说，构造函数的作用是初始化实例变量。每个构造函数都将创建一个对象并向调用者返回一个该对象的引用。构造函数的名称总是和类名相同。我们可以和重载方法一样重载这个名称并定义签名不同的多个构造函数。如果没有定义构造函数，类将会隐式定义一个默认情况下不接受任何参数的构造函数并将所有实例变量初始化为默认值。原始数字类型的实例变量默认值为 0，布尔类型变量为 `false`，引用类型变量为 `null`。我们可以在声明语句中初始化这些实例变量并改变这些默认值。当用例使用关键字 `new` 时，Java 会自动触发一个构造函数。重载构造函数一般用于将实例变量由默认值初始化为用例提供的值。例如，`Counter` 类型有个接受一个参数的构造函数，它将实例变量 `name` 初始化为由参数给定的值（实例变量 `count` 仍将被初始化为默认值 0）。构造函数解析如图 1.2.6 所示。

1.2.3.3 实例方法

实现数据类型的实例方法（即每个对象的行为）的代码和 1.1 节中实现静态方法（函数）的代码完全相同。每个实例方法都有一个返回值类型、一个签名（它指定了方法名、返回值类型和所有参数变量的名称）和一个主体（它由一系列语句组成，包括一个返回语句来将一个返回类型的值传递给调用者）。当调用者触发了一个方法时，方法的参数（如果有）均会被初始化为调用者所提供的值，方法的语句会被执行，直到得到一个返回值并且将该值返回给调用者。它的效果就好像调用者代码中的函数调用被替换为了这个返回值。实例方法的所有这些行为都和静态方法相同，只有一点关键的不同：它们可以访问并操作实例变量。如何指定我们希望使用的对象的实例变量？只要稍加思考，就能够得到合理的答案：在一个实例方法中对变量的引用指的是该方法的变量中的值。当我们调用 `heads.increment()` 时，`increment()` 方法中的代码访问的是 `heads` 中的实例变量。换句话说，面向对象编程为 Java 程序增加了另一种使用变量的重要方式。

□ 通过触发一个实例方法来操作该对象的值。

这与调用静态方法仅仅是语法上的区别（请见答疑），但在许多情况下它颠覆了现代程序员对程序开发的思维方式。你会看到，这种方式与算法和数据结构的研究非常契合。实例方法解析如图 1.2.7 所示。

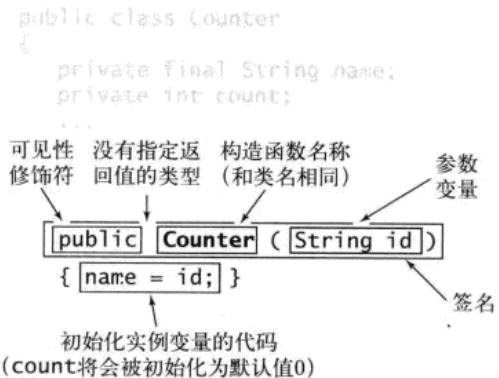


图 1.2.6 详解构造函数

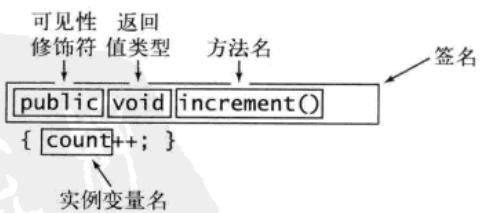


图 1.2.7 详解实例方法

1.2.3.4 作用域

总的来说，我们在实现实例方法的 Java 代码中使用了三种变量：

- 参数变量；
- 局部变量；
- 实例变量。

在静态方法中前两者的用法没有变化：

方法的签名定义了参数变量，在方法被调用时参数方法会被初始化为调用者提供的值；局部变量的声明和初始化都在方法的主体中。参数变量的作用域是整个方法；局部变量的作用域是当前代码段中它的定义之后的所有语句。实例变量则完全不同（如右侧示例所示）：它们为该类的对象保存了数据类型的值，它们的作用域是整个类（如果出现二义性，可以使用 `this` 前缀来区别实例变量）。

理解实例方法中这三种变量的区别是理解面向对象编程的关键。

87

1.2.3.5 API、用例与实现

这些都是你要在 Java 中构造并使用抽象数据类型所需要理解的基本组件。我们将要学习的每个抽象数据类型的实现都会是一个含有若干私有实例变量、构造函数、实例方法和一个测试用例的 Java 类。要完全理解一个数据类型，我们需要它的 API、典型的用例和它的实现。`Counter` 类型的总结请见表 1.2.11。为了强调用例和实现的分离，我们一般会将用例独立成为含有一个静态方法 `main()` 的类，并将数据类型定义中的 `main()` 方法预留为一个用于开发和最小单元测试的测试用例（至少调用每个实例方法一次）。我们开发的每种数据类型都会遵循相同的步骤。我们思考的不是应该采取什么行动来达成某个计算性的目的（如同我们第一次学习编程时那样），而是用例的需求。我们会按照下面三步走的方式用抽象数据类型满足它们。

- 定义一份 API：API 的作用是将使用和实现分离，以实现模块化编程。我们制定一份 API 的目标有二：第一，我们希望用例的代码清晰而正确，事实上，在最终确定 API 之前就编写一些用例代码来确保所设计的数据类型操作正是用例所需要的是很好的主意；第二，我们希望能够实现这些操作，定义一些无法实现的操作是没有意义的。
- 用一个 Java 类实现 API 的定义：首先我们选择适当的实例变量，然后再编写构造函数和实例方法。
- 实现多个测试用例来验证前两步做出的设计决定。

表 1.2.11 一个简单计数器的抽象数据类型

API	public class Counter	
	Counter(String id)	创建一个名为 id 的计数器
	void increment()	将计数器的值加 1
	int tally()	计数器的值
	String toString()	对象的字符串表示

```
public class Example
{
    private int var;           实例变量

    ...
}

private void method1()
{
    int var;                 局部变量
    ...
    ... var ...
    ... this.var ...
    ...
}

private void method2()
{
    ...
    ... var ...
    ...
}
```

实例方法中的实例变量和局部变量的作用范围

(续)

典型的用例

```
public class Flips
{
    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        Counter heads = new Counter("heads");
        Counter tails = new Counter("tails");
        for (int t = 0; t < T; t++)
            if (StdRandom.bernoulli(0.5))
                heads.increment();
            else tails.increment();
        StdOut.println(heads);
        StdOut.println(tails);
        int d = heads.tally() - tails.tally();
        StdOut.println("delta: " + Math.abs(d));
    }
}
```

数据类型的实现

```
public class Counter
{
    private final String name;
    private int count;
    public Counter(String id)
    { name = id; }
    public void increment()
    { count++; }
    public int tally()
    { return count; }
    public String toString()
    { return count + " " + name; }
}
```

使用方法

```
% java Flips 1000000
500172 heads
499828 tails
delta: 344
```

用例一般需要什么操作？数据类型的值应该是什么才能最好地支持这些操作？这些基本的判断是我们开发的每种实现的核心内容。

88
l
89

1.2.4 更多抽象数据类型的实现

和任何编程概念一样，理解抽象数据类型的威力和用法的最好办法就是仔细研究更多的例子和实现。本书中大量代码是通过抽象数据类型实现的，因此你的机会很多，但是一些更简单的例子能够帮助我们为研究抽象数据类型打好基础。

1.2.4.1 日期

表 1.2.12 是我们在表 1.2.6 中定义的 Date 抽象数据类型的两种实现。简单起见，我们省略了解析字符串的构造函数（请见练习 1.2.19）和继承的方法 equals()（请见 1.2.5.8 节）、compareTo()（请见 2.1.1.4 节）和 hashCode()（请见练习 3.4.22）。表 1.2.12 中左侧的简单实现将日、月和年设为实例变量，这样实例方法就可以直接返回适当的值；右侧的实现更加节省空间，仅使用了一个 int 变量来表示一个日期。它将 d 日、m 月和 y 年的一个日期表示为一个混合进制的整数 $512y+32m+d$ 。用例分辨这两种实现的区别的一种方法可能是打破我们对日期的隐式假设：第

二种实现的正确性基于日的值在0到31之间，月的值在0到15之间，年的值为正（在实际应用中，两种实现都应该检查月份的值是否在1到12之间，日的值是否在1到31之间，以及例如2009年6月31日和2月29日这样的非法日期，尽管这么做要费些工夫）。这个例子的主要意思是说明我们在API中极少完整地指定对实现的要求（一般来说我们都会尽力而为，这里还可以做得更好）。用例要分辨出这两种实现的区别另一种方法是性能：右侧的实现中保存数据类型的值所需的空间较少，代价是在向用例按照约定的格式提供这些值时花费的时间更多（需要进行一两次算数运算）。这种交换是很常见的：某些用例可能偏爱其中一种实现，而另一些用例可能更喜欢另一种，因此我们两者都要满足。事实上，本书中反复出现的一个主题就是我们需要理解各种实现对空间和时间的需求以及它们对各种用例的适用性。在实现中使用数据抽象的一个关键优势是我们可以将一种实现替换为另一种而无需改变用例的任何代码。

表 1.2.12 一种封装日期的抽象数据类型以及它的两种实现

API	public class Date		
	Date(int month, int day, int year)	创建一个日期	
	int day()	日	
	int month()	月	
	int year()	年	
	String toString()	对象的字符串表示	

测试用例

```
public static void main(String[] args)
{
    int m = Integer.parseInt(args[0]);
    int d = Integer.parseInt(args[1]);
    int y = Integer.parseInt(args[2]);
    Date date = new Date(m, d, y);
    StdOut.println(date);
}
```

使用方法

```
% java Date 12 31 1999
12/31/1999
```

数据类型的实现

```
public class Date
{
    private final int month;
    private final int day;
    private final int year;
    public Date(int m, int d, int y)
    {
        month = m; day = d; year = y;
    }
    public int month()
    {
        return month;
    }
    public int day()
    {
        return day;
    }
    public int year()
    {
        return year;
    }
    public String toString()
    {
        return month() + "/" + day()
               + "/" + year();
    }
}
```

数据类型的另一种实现

```
public class Date
{
    private final int value;
    public Date(int m, int d, int y)
    {
        value = y*512 + m*32 + d;
    }
    public int month()
    {
        return (value / 32) % 16;
    }
    public int day()
    {
        return value % 32;
    }
    public int year()
    {
        return value / 512;
    }
    public String toString()
    {
        return month() + "/" + day()
               + "/" + year();
    }
}
```

1.2.4.2 维护多个实现

同一份 API 的多个实现可能会产生维护和命名问题。在某些情况下，我们可能只是想将较老的实现替换为改进的实现。而在另一些情况下，我们可能需要维护两种实现，一种适用于某些用例，另一种适用于另一些用例。实际上，本书的一个主要目标就是深入讨论若干种基本抽象数据结构的实现并衡量它们的性能的不同。在本书中，我们经常会比较同一份 API 的两种不同实现在同一个用例中的性能表现。为此，我们通常采用一种非正式的命名约定。

- 通过前缀的描述性修饰符区别同一份 API 的不同实现。例如，我们可以将表 1.2.12 中的 `Date` 实现命名为 `BasicDate` 和 `SmallDate`，我们可能还希望实现一种能够验证日期是否合法的 `SmartDate`。
- 维护一个没有前缀的参考实现，它应该适合于大多数用例的需求。在这里，大多数用例应该直接会使用 `Date`。

在一个庞大的系统中，这种解决方案并不理想，因为它可能会需要修改用例的代码。例如，如果需要开发一个新的实现 `ExtraSmallDate`，那么我们只能修改用例的代码或是让它成为所有用例的参考实现。Java 有许多高级语言特性来保证在无需修改用例代码的情况下维护多个实现，但我们很少会使用它们，因为即使 Java 专家使用起它们来也十分困难（有时甚至是争议的），尤其是同我们极为需要的其他高级语言特性（泛型和迭代器）一起使用时。这些问题很重要（例如，忽略它们会导致千禧年著名的 Y2K 问题，因为许多程序使用的都是它们自己对日期的抽象实现，且并没有考虑到年份的头两位数字），但是深究它们会使我们大大偏离对算法的研究。

1.2.4.3 累加器

表 1.2.13 中的累加器 API 定义了一种能够为用例计算一组数据的实时平均值的抽象数据类型。例如，本书中经常会使用该数据类型来处理实验结果（请见 1.4 节）。它的实现很简单：它维护一个 `int` 类型的实例变量来记录已经处理过的数据值的数量，以及一个 `double` 类型的实例变量来记录所有数据值之和，将和除以数据数量即可得到平均值。请注意该实现并没有保存数据的值——它可以用子处理大规模的数据（甚至是在一个无法全部保存它们的设备上），而一个大型系统也可以大量使用

表 1.2.13 一种能够累加数据的抽象数据类型

API	<code>public class Accumulator</code>	
	<code>Accumulator()</code>	创建一个累加器
	<code>void addDataValue(double val)</code>	添加一个新的数据值
	<code>double mean()</code>	所有数据值的平均值
	<code>String toString()</code>	对象的字符串表示
典型的用例		使用方法
	<pre>public class TestAccumulator { public static void main(String[] args) { int T = Integer.parseInt(args[0]); Accumulator a = new Accumulator(); for (int t = 0; t < T; t++) a.addDataValue(StdRandom.random()); StdOut.println(a); } }</pre>	<pre>% java TestAccumulator 1000 Mean (1000 values): 0.51829 % java TestAccumulator 1000000 Mean (1000000 values): 0.49948 % java TestAccumulator 1000000 Mean (1000000 values): 0.50014</pre>

(续)

数据类型的实现

```

public class Accumulator
{
    private double total;
    private int N;
    public void addDataValue(double val)
    {
        N++;
        total += val;
    }
    public double mean()
    {
        return total/N;
    }
    public String toString()
    {
        return "Mean (" + N + " values): "
            + String.format("%7.5f", mean());
    }
}

```

92
93 累加器。这种性能特点很容易被忽视，所以也许应该在 API 中注明，因为一种存储所有数据值的实现可能会使调用它的应用程序用光所有内存。

1.2.4.4 可视化的累加器

表 1.2.14 所示的可视化累加器的实现继承了 `Accumulator` 类并展示了一种实用的副作用：它用 `StdDraw` 画出了所有数据（灰色）和实时的平均值（红色），见图 1.2.8。完成这项任务最简单的方法是添加一个构造函数来指定需要绘出的点数和它们的最大值（用于调整图像的比例）。严格说来，`VisualAccumulator` 并不是 `Accumulator` 的 API 的实现（它的构造函数的签名不同且产生

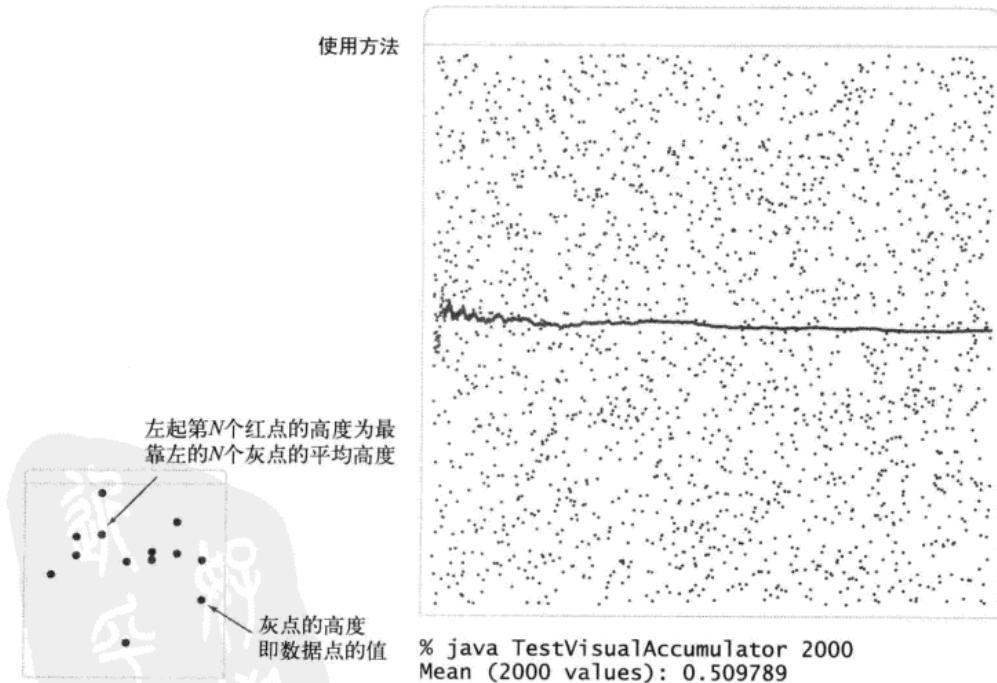


图 1.2.8 可视化累加器图像（另见彩插）

了一种不同的副作用）。一般来说，我们会仔细而完整地设计 API，并且一旦定型就不愿再对它做任何改动，因为这有可能会涉及修改无数用例（和实现）的代码。但添加一个构造函数来取得某些功能有时能够获得通过，因为它对用例的影响和改变类名所产生的变化相同。在本例中，如果已经开发了一个使用 `Accumulator` 的用例并大量调用了 `addDataValue()` 和 `mean()`，只需改变用例的一行代码就能享受到 `VisualAccumulator` 的优势。

表 1.2.14 一种能够累加数据的抽象数据类型（可视版本，另见彩插）

API	<code>public class VisualAccumulator</code>
	<code> VisualAccumulator(int trials, double max)</code>
	<code> void addDataValue(double val)</code> 添加一个新的数据值
	<code> double mean()</code> 所有数据的平均值
	<code> String toString()</code> 对象的字符串表示
典型的用例	<pre>public class TestVisualAccumulator { public static void main(String[] args) { int T = Integer.parseInt(args[0]); VisualAccumulator a = new VisualAccumulator(T,1.0); for (int t = 0; t < T; t++) a.addDataValue(StdRandom.random()); StdOut.println(a); } }</pre>
数据类型的实现	<pre>public class VisualAccumulator { private double total; private int N; public VisualAccumulator(int trials, double max) { StdDraw.setXscale(0, trials); StdDraw.setYscale(0, max); StdDraw.setPenRadius(.005); } public void addDataValue(double val) { N++; total += val; StdDraw.setPenColor(StdDraw.DARK_GRAY); StdDraw.point(N, val); StdDraw.setPenColor(StdDraw.RED); StdDraw.point(N, total/N); } public double mean() public String toString() // 和 Accumulator 相同 }</pre>

1.2.5 数据类型的设计

抽象数据类型是一种向用例隐藏内部表示的数据类型。这种思想强有力地影响了现代编程。我们遇到过的众多例子为我们研究抽象数据类型的高级特性和它们的 Java 实现打下了基础。简单看来，下面的许多话题和算法的学习关系不大，因此你可以跳过本节，在今后实现抽象数据类型中遇到特定问题时再回过头来参考它。我们的目的是将关于设计数据类型的重要知识集中起来以供参考，并为本书中的所有抽象数据类型的实现做铺垫。

1.2.5.1 封装

面向对象编程的特征之一就是使用数据类型的实现封装数据，以简化实现和隔离用例开发。封装实现了模块化编程，它允许我们：

- 独立开发用例和实现的代码；
- 切换至改进的实现而不会影响用例的代码；
- 支持尚未编写的程序（对于后续用例，API 能够起到指南的作用）。

封装同时也隔离了数据类型的操作，这使我们可以：

- 限制潜在的错误；
- 在实现中添加一致性检查等调试工具；
- 确保用例代码更明晰。

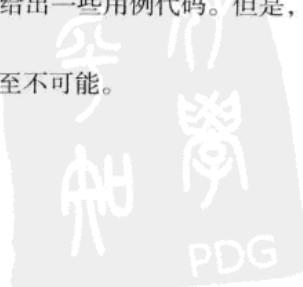
一个封装的数据类型可以被任意用例使用，因此它扩展了 Java 语言。我们所提倡的编程风格是将大型程序分解为能够独立开发和调试的小型模块。这种方式将修改代码的影响限制在局部区域，改进了我们的软件质量。它也促进了代码复用，因为我们可以用某种数据类型的新实现实代替老的实现来改进它的性能、准确度或是内存消耗。同样的思想也适用于许多其他领域。我们在使用系统库时常常从封装中受益。Java 系统的新实现往往更新了多种数据类型或静态方法库的实现，但它们的 API 并没有变化。在算法和数据结构的学习中，我们总是希望开发出更好的算法，因为只需用抽象数据类型的改进实现替换老的实现即可在不改变任何用例代码的情况下改进所有用例的性能。模块化编程成功的关键在于保持模块之间的独立性。我们坚持将 API 作为用例和实现之间唯一的依赖点来做到这一点。并不需要知道一个数据类型是如何实现的才能使用它，实现数据类型时也应该假设使用者除了 API 什么也不知道。封装是获得所有这些优势的关键。

96

1.2.5.2 设计 API

构建现代软件最重要也最有挑战的一项任务就是设计 API。它需要经验、思考和反复的修改，但设计一份优秀的 API 所付出的所有时间都能从调试和代码复用所节省的时间中获得回报。为一个小程序给出一份 API 似乎有些多余，但你应该按照能够复用的方式编写每个程序。理想情况下，一份 API 应该能够清楚地说明所有可能的输入和副作用，然后我们应该先写出检查实现是否与 API 相符的程序。但不幸的是，计算机科学理论中一个叫做说明书问题（specification problem）的基础结论说明这个目标是不可能实现的。简单地说，这样一份说明书应该用一种类似于编程语言的形式语言编写。而从数学上可以证明，判定这样两个程序进行的计算是否相同是不可能的。因此，我们的 API 将是与抽象数据类型相关联的值以及一系列构造函数和实例方法的目的和副作用的自然语言描述。为了验证我们的设计，我们会在 API 附近的正文中给出一些用例代码。但是，这些宏观概述之中也隐藏着每一份 API 设计都可能落入的无数陷阱。

- API 可能会难以实现：实现的开发非常困难，甚至不可能。



- API 可能会难以使用：用例代码甚至比没有 API 时更复杂。
 - API 的范围可能太窄：缺少用例所需的方法。
 - API 的范围可能太宽：包含许多不会被任何用例调用的方法。这种缺陷可能是最常见的，并且也是最难避免的。API 的大小一般会随着时间而增长，因为向已有的 API 中添加新方法很简单，但在不破坏已有用例程序的前提下从中删除方法却很困难。
 - API 可能会太粗略：无法提供有效的抽象。
 - API 可能会太详细：抽象过于细致或是发散而无法使用。
 - API 可能会过于依赖某种特定的数据表示：用例代码可能会因此无法从数据表示的细节中解脱出来。要避免这种缺陷也是很困难的，因为数据表示显然是抽象数据类型实现的核心。97
- 这些考虑有时又被总结为另一句格言：只为用例提供它们所需要的，仅此而已。

1.2.5.3 算法与抽象数据类型

数据抽象天生适合算法研究，因为它能够为我们提供一个框架，在其中能够准确地说明一个算法的目的以及其他程序应该如何使用该算法。在本书中，算法一般都是某个抽象数据类型的一个实例方法的实现。例如，本章开头的白名单例子就很自然地被实现为一个抽象数据类型的用例。它进行了以下操作：

- 由一组给定的值构造了一个 SET（集合）对象；
- 判定一个给定的值是否存在于该集合中。

这些操作封装在 `StaticSETofInts` 抽象数据类型中，和 `Whitelist` 用例一起显示在表 1.2.15 中。`StaticSETofInts` 是更一般也更有用的符号表抽象数据类型的一种特殊情况，符号表抽象数据类型将是第 3 章的重点。在我们研究过的所有算法中，二分查找是较为适合用于实现这些抽象数据类型的一种。和 1.1.10 节中的 `BinarySearch` 实现比较起来，这里的实现所产生的用例代码更加清晰和高效。例如，`StaticSETofInts` 强制要求数组在 `rank()` 方法被调用之前排序。有了抽象数据类型，我们可以将抽象数据类型的调用和实现区分开来，并确保任意遵守 API 的用例程序都能受益于二分查找算法（使用 `BinarySearch` 的程序在调用 `rank()` 之前必须能够将数组排序）。白名单应用是众多二分查找算法的用例之一。

每个 Java 程序都是一组静态方法和（或）一种数据类型的实现的集合。在本书中我们主要关注的是抽象数据类型的实现中的操作和向用例隐藏其中的数据表示，例如 `StaticSETofInts`。正如这个例子所示，数据抽象使我们能够：

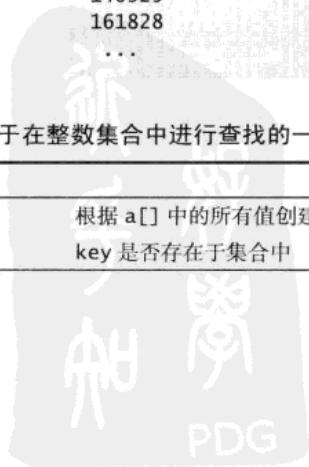
- 准确定义算法能为用例提供什么；
- 隔离算法的实现和用例的代码；
- 实现多层抽象，用已知算法实现其他算法。

应用

```
% java Whitelist largeW.txt <
largeT.txt
499569
984875
295754
207807
140925
161828
...
...
```

表 1.2.15 将二分查找重写为一段面向对象的程序（用于在整数集合中进行查找的一种抽象数据类型）

API	<code>public class StaticSETofInts</code>	
	<code> StaticSETofInts(int[] a)</code>	根据 a[] 中的所有值创建一个集合
	<code> boolean contains(int key)</code>	key 是否存在于集合中



(续)

典型的用例

```
public class Whitelist
{
    public static void main(String[] args)
    {
        int[] w = In.readInts(args[0]);
        StaticSETofInts set = new StaticSETofInts(w);
        while (!StdIn.isEmpty())
        {
            // 读取键，如果不在白名单中则打印它
            int key = StdIn.readInt();
            if (!set.contains(key))
                StdOut.println(key);
        }
    }
}
```

数据类型的实现

```
import java.util.Arrays;
public class StaticSETofInts
{
    private int[] a;
    public StaticSETofInts(int[] keys)
    {
        a = new int[keys.length];
        for (int i = 0; i < keys.length; i++)
            a[i] = keys[i]; // 保护性复制
        Arrays.sort(a);
    }
    public boolean contains(int key)
    { return rank(key) != -1; }
    private int rank(int key)
    { // 二分查找
        int lo = 0;
        int hi = a.length - 1;
        while (lo <= hi)
        {
            // 键要么存在于 a[lo..hi] 中，要么不存在
            int mid = lo + (hi - lo) / 2;
            if (key < a[mid]) hi = mid - 1;
            else if (key > a[mid]) lo = mid + 1;
            else return mid;
        }
        return -1;
    }
}
```

无论是使用自然语言还是伪代码描述算法，这些都是我们所希望拥有的性质。使用 Java 的类机制来支持数据的抽象将使我们收获良多：我们编写的代码将能够测试算法并比较各种用例程序的性能。

1.2.5.4 接口继承

Java 语言为定义对象之间的关系提供了支持，称为接口。程序员广泛使用这些机制，如果上过软件工程的课程那么你可以详细地研究一下它们。我们学习的第一种继承机制叫做子类型。它允许我们通过指定一个含有一组公共方法的接口为两个本来并没有关系的类建立一种联系，这两个类都必须实现这些方法。例如，如果不使用我们的非正式 API，也可以为 Date 声明一个接口：

```
public interface Datable
{
    int month();
    int day();
    int year();
}
```

并在我们的实现中引用该接口：

```
public class Date implements Datable
{
    // 实现代码（和以前一样）
}
```

这样，Java 编译器就会检查该实现是否和接口相符。为任意实现了 `month()`、`day()` 和 `year()` 的类添加 `implements Datable` 保证了所有用例都能用该类的对象调用这些方法。这种方式称为接口继承——实现类继承的是接口。接口继承使得我们的程序能够通过调用接口中的方法操作实现该接口的任意类型的对象（甚至是还未被创建的类型）。我们可以在更多非正式的 API 中使用接口继承，但为了避免代码依赖于和理解算法无关的高级语言特性以及额外的接口文件，我们并没有这么做。在某些情况下 Java 的习惯用法鼓励我们使用接口：我们用它们进行比较和迭代，如表 1.2.16 所示。我们会在接触那些概念时再详细研究它们。

100

表 1.2.16 本书中所用到的 Java 接口

接 口	方 法	章 节
比较	<code>java.lang.Comparable</code>	<code>compareTo()</code> 2.1
	<code>java.util.Comparator</code>	<code>compare()</code> 2.5
迭代	<code>java.lang.Iterable</code>	<code>iterator()</code> 1.3
	<code>java.util.Iterator</code>	<code>hasNext()</code> 1.3
	<code>next()</code>	
	<code>remove()</code>	

1.2.5.5 实现继承

Java 还支持另一种继承机制，被称为子类。这种非常强大的技术使程序员不需要重写整个类就能改变它的行为或者为它添加新的功能。它的主要思想是定义一个新类（子类，或称为派生类）来继承另一个类（父类，或称为基类）的所有实例方法和实例变量。子类包含的方法比父类更多。另外，子类可以重新定义或者重写父类的方法。子类继承被系统程序员广泛用于编写所谓可扩展的库——任何一个程序员（包括你）都能为另一个程序员（或者也许是一个系统程序员团队）创建的库添加方法。这种方法能够有效地重用潜在的十分庞大的库中的代码。例如，这种方法被广泛用于图形用户界面的开发，因此实现用户所需要的各种控件（下拉菜单，剪切一粘贴，文件访问等）的大量代码都能够被重用。子类继承的使用在系统程序员和应用程序员之间是有争议的（它和接口继承之间的优劣还没有定论）。在本书中我们会避免使用它，因为它会破坏封装。但这种机制是 Java 的一部分，因此它的残余是无法避免的：具体来说，每个类都是 Java 的 `Object` 类的子类。这种结构意味着每个类都含有 `getClass()`、`toString()`、`equals()`、`hashCode()`（见表 1.2.17）和另外几个我们不会在本书中用到的方法的实现。实际上，每个类都通过子类继承从 `Object` 类中继承了这些方法，因此任何用例都可以在任意对象中调用这些方法。我们通常会重载新类的 `toString()`、`equals()` 和 `hashCode()` 方法，因为 `Object` 类的默认实现一般无法提供所需的行为。接下来我们将讨论 `toString()` 和 `equals()`，在 3.4 节中讨论 `hashCode()`。

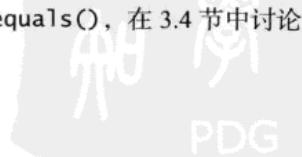


表 1.2.17 本书中所使用的由 Object 类继承得到的方法

方 法	作 用	章 节
Class getClass()	该对象的类是什么	1.2
String toString()	该对象的字符串表示	1.1
boolean equals(Object that)	该对象是否和 that 相等	1.2
int hashCode()	该对象的散列值	3.4

101

1.2.5.6 字符串表示的习惯

按照习惯，每个 Java 类型都会从 Object 继承 `toString()` 方法，因此任何用例都能够调用任意对象的 `toString()` 方法。当连接运算符的一个操作数是字符串时，Java 会自动将另一个操作数也转换为字符串，这个约定是这种自动转换的基础。如果一个对象的数据类型没有实现 `toString()` 方法，那么转换会调用 `Object` 的默认实现。默认实现一般都没有多大实用价值，因为它只会返回一个含有该对象内存地址的字符串。因此我们通常会为我们的每个类实现并重写默认的 `toString()` 方法，如下面代码框的 `Date` 类中加粗的部分所示。由代码可以看到，`toString()` 方法的实现通常很简单，只需隐式调用（通过 +）每个实例变量的 `toString()` 方法即可。

1.2.5.7 封装类型

Java 提供了一些内置的引用类型，称为封装类型。每种原始数据类型都有一个对应的封装类型：`Boolean`、`Byte`、`Character`、`Double`、`Float`、`Integer`、`Long` 和 `Short` 分别对应着 `boolean`、`byte`、`char`、`double`、`float`、`int`、`long` 和 `short`。这些类主要由类似于 `parseInt()` 这样的静态方法组成，但它们也含有继承得到的实例方法 `toString()`、`compareTo()`、`equals()` 和 `hashCode()`。在需要的时候 Java 会自动将原始数据类型转换为封装类型，如 1.3.1.1 节所述。例如，当一个 `int` 值需要和一个 `String` 连接时，它的类型会被转换为 `Integer` 并触发 `toString()` 方法。

1.2.5.8 等价性

两个对象相等意味着什么？如果我们用相同类型的两个引用变量 `a` 和 `b` 进行等价性测试 (`a == b`)，我们检测的是它们的标识是否相同，即引用是否相同。一般用例希望能够检查数据类型的值（对象的状态）是否相同或者实现某种针对该类型的规则。Java 为我们开了个头，为 `Integer`、`Double` 和 `String` 等标准数据类型以及一些如 `File` 和 `URL` 的复杂数据类型提供了实现。在处理这些类型的数据时，可以直接使用内置的实现。例如，如果 `x` 和 `y` 均为 `String` 类型的值，那么当且仅当 `x` 和 `y` 的长度相同且每个位置的字符均相同时 `x.equals(y)` 的返回值为 `true`。当我们在定义自己的数据类型时，比如 `Date` 或 `Transaction`，需要重载 `equals()` 方法。Java 约定 `equals()` 必须是一种等价性关系。它必须具有：

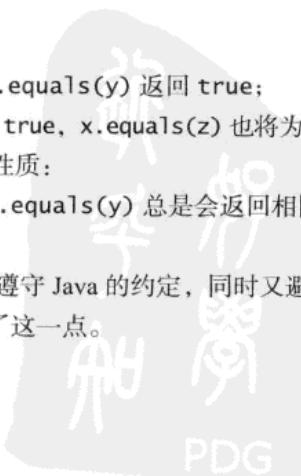
- 自反性，`x.equals(x)` 为 `true`；
- 对称性，当且仅当 `y.equals(x)` 为 `true` 时，`x.equals(y)` 返回 `true`；
- 传递性，如果 `x.equals(y)` 和 `y.equals(z)` 均为 `true`，`x.equals(z)` 也将为 `true`。

另外，它必须接受一个 `Object` 为参数并满足以下性质：

- 一致性，当两个对象均未被修改时，反复调用 `x.equals(y)` 总是会返回相同的值；
- 非空性，`x.equals(null)` 总是返回 `false`。

102

这些定义都是自然合理的，但确保这些性质成立并遵守 Java 的约定，同时又避免在实现时做无用功却不容易，如 `Date` 所示。它通过以下步骤做到了这一点。



- 如果该对象的引用和参数对象的引用相同，返回 `true`。这项测试在成立时能够免去其他所有测试工作。
- 如果参数为空 (`null`)，根据约定返回 `false`（还可以避免在下面的代码中使用空引用）。
- 如果两个对象的类不同，返回 `false`。要得到一个对象的类，可以使用 `getClass()` 方法。请注意我们会使用 `==` 来判断 `Class` 类型的对象是否相等，因为同一种类型的所有对象的 `getClass()` 方法一定能够返回相同的引用。
- 将参数对象的类型从 `Object` 转换到 `Date`（因为前一项测试已经通过，这种转换必然成功）。
- 如果任意实例变量的值不相同，返回 `false`。对于其他类，等价性测试方法的定义可能不同。例如，我们只有在两个 `Counter` 对象的 `count` 变量相等时才会认为它们相等。

```

public class Date
{
    private final int month;
    private final int day;
    private final int year;

    public Date(int m, int d, int y)
    { month = m; day = d; year = y; }

    public int month()
    { return month; }

    public int day()
    { return day; }

    public int year()
    { return year; }

    public String toString()
    { return month() + "/" + day() + "/" + year(); }

    public boolean equals(Object x)
    {
        if (this == x) return true;
        if (x == null) return false;
        if (this.getClass() != x.getClass()) return false;
        Date that = (Date) x;
        if (this.day != that.day)           return false;
        if (this.month != that.month)      return false;
        if (this.year != that.year)        return false;
        return true;
    }
}

```

在数据类型的定义中重写 `toString()` 和 `equals()` 方法

你可以使用上面的实现作为实现任意数据类型的 `equals()` 方法的模板。只要实现一次 `equals()` 方法，下一次就不会那么困难了。

1.2.5.9 内存管理

103

我们可以为一个引用变量赋予一个新的值，因此一段程序可能会产生一个无法被引用的对象。例如，请看图 1.2.9 中所示的三行赋值语句。在第三行赋值语句之后，不仅 `a` 和 `b` 会指向同一个 `Date` 对象（1/1/2011），而且不存在能够引用初始化变量 `b` 的那个 `Date` 对象的引用了。本来该对

象的唯一引用就是变量 b，但是该引用被赋值语句覆盖了，这样的对象被称为孤儿。对象在离开作用域之后也会变成孤儿。Java 程序经常会创建大量对象（以及许多保存原始数据类型值的变量），但在某个时刻程序只会需要它们之中的一小部分。因此，编程语言和系统需要某种机制来在必要时为数据类型的值分配内存，而在不需要时释放它们的内存（对于一个对象来说，有时是在它变成孤儿之后）。内存管理对于原始数据类型更容易，因为内存分配所需的所有信息在编译阶段就能够获取。Java（以及大多数其他系统）会在声明变量时为它们预留内存空间，并会在它们离开作用域后释放这些空间。对象的内存管理更加复杂：系统会在创建一个对象时为它分配内存，但是程序在执行时的动态性决定了一个对象何时才会变为孤儿，系统并不能准确地知道应该何时释放一个对象的内存。在许多语言中（例如 C 和 C++），分配和释放内存是程序员的责任。众所周知，这种操作既繁琐又容易出错。Java 最重要的一个特性就是自动内存管理。它通过记录孤儿对象并将它们的内存释放到内存池中将程序员从管理内存的责任中解放出来。这种回收内存的方式叫做垃圾回收。Java 的一个特点就是它不允许修改引用的策略。这种策略使 Java 能够高效自动地

回收垃圾。程序员们至今仍在争论，为获得无需为内存管理操心的方便而付出的使用自动垃圾回收的代价是否值得。

1.2.5.10 不可变性

不可变数据类型，例如 Date，指的是该类型的对象中的值在创建之后就无法再被改变。与此相反，可变数据类型，例如 Counter 或 Accumulator，能够操作并改变对象中的值。Java 语言通过 final 修饰符来强制保证不可变性。当你将一个变量声明为 final 时，也就保证了只会对它赋值一次，可以用赋值语句，也可以用构造函数。试图改变 final 变量的值的代码将会产生一个编译时错误。在我们的代码中，我们用 final 修饰值不会改变的实例变量。这种策略就像文档一样，说明了这个变量的值不会再发生改变，它能够预防意外修改，也能使程序的调试更加简单。像 Date 这样实例变量均为原始数据类型且被 final 修饰的数据类型（按照约定，在不使用子类继承的代码中）是不可变的。数据类型是否可变是一个重要的设计决策，它取决于当前的应用场景。对于类似于 Date 的数据类型，抽象的目的是封装不变的值，以便将其和原始数据类型一样用于赋值语句、作为函数的参数或返回值（而不必担心它们的值会被改变）。程序员在使用 Date 时可能会写出操作两个 Date 类型的变量的代码 d = d0，就像操作 double 或者 int 值一样。但如果 Date 类型是可变的且 d 的值在 d = d0 之后可以被改变，那么 d0 的值也会被改变（它们都是指向同一个对象的引用）！从另一方面来说，对于类似于 Counter 和 Accumulator 的数据类型，抽象的目的是封装变化中的值。作为用例程序员，你在使用 Java 数组（可变）和 Java 的 String 类型（不可变）时就已经遇到了这种区别。将一个 String 传递给一个方法时，你不会担心该方法会改变字符串中的字符顺序，但当你把一个数组传递给一个方法时，方法可以自由改变数组的内容。String 对象是

```
Date a = new Date(12, 31, 1999);
Date b = new Date(1, 1, 2011);
a = b;
```

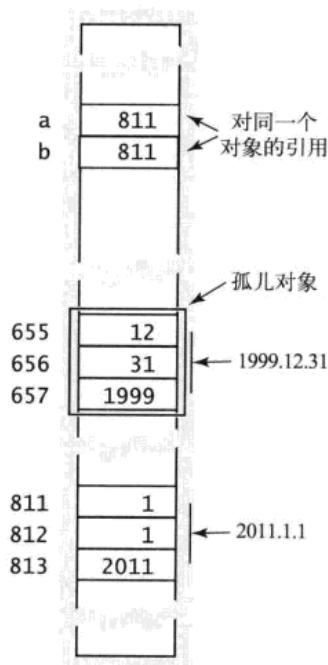


图 1.2.9 孤儿对象

不可变的，因为我们一般都不希望 `String` 的值改变，而 Java 数组是可变的，因为我们一般的确希望改变数组中的值。但也存在我们希望使用可变字符串（这就是 Java 的 `StringBuilder` 类存在的目的）和不可变数组（这就是稍后讨论的 `Vector` 类存在的目的）的情况。一般来说，不可变的数据类型比可变的数据类型使用更容易，误用更困难，因为能够改变它们的值的方式要少得多。调试使用不可变类型的代码更简单，因为我们更容易确保用例代码中使用它们的变量的状态前后一致。在使用可变数据类型时，必须时刻关注它们的值会在何时何地发生变化。而不可变性的缺点在于我们需要为每个值创建一个新对象。这种开销一般是可以接受的，因为 Java 的垃圾回收器通常都为此进行了优化。不可变性的另一个缺点在于，`final` 非常不幸地只能用来保证原始数据类型的实例变量的不可变性，而无法用于引用类型的变量。如果一个应用类型的实例变量含有修饰符 `final`，该实例变量的值（某个对象的引用）就永远无法改变了——它将永远指向同一个对象，但对象的值本身仍然是可变的。例如，这段代码并没有实现一个不可变的数据类型：

```
public class Vector
{
    private final double[] coords;
    public Vector(double[] a)
    { coords = a; }
    ...
}
```

用例程序可以通过给定的数组创建一个 `Vector` 对象，并在构造函数执行之后（绕过 API）改变 `Vector` 中的元素的值：

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
a[0] = 0.0; // 绕过了公有 API
```

实例变量 `coords[]` 是 `private` 和 `final` 的，但 `Vector` 是可变的，因为用例拥有指向数据的一个引用。任何数据类型的设计都需要考虑到不可变性，而且数据类型是否是不可变的则应该在 API 中说明，这样使用者才能知道该对象中的值是无法改变的。在本书中，我们对不可变性的主要兴趣在于用它保证我们的算法的正确性。例如，如果一个二分查找算法所使用的数据的类型是可变的，那么算法的用例就可能破坏我们对二分查找中的数组已经有序的假设。可变数据与不可变数据的示例见表 1.2.18。

1.2.5.11 契约式设计

在最后，我们将简要讨论 Java 语言中能够在程序运行时检验程序状态的一些机制。为此我们将使用两种 Java 的语言特性：

- 异常（Exception），一般用于处理不受我们控制的不可预见的错误；
- 断言（Assertion），验证我们在代码中做出的一些假设。

大量使用异常和断言是很好的编程实践。为了节约版面我们在本书中极少使用它们，但你在本书网站上的所有代码中都会找到它们。这些代码中的每个和异常条件以及断言恒等式有关的算法周围都有大量的注释。

1.2.5.12 异常与错误

异常和错误都是在程序运行中出现的破坏性事件。Java 采取的行动称为抛出异常或是抛出错误。我们已经在学习 Java 的基本特性的过程中遇到过 Java 系统方法抛出的异常：

表 1.2.18 可变与不可变数据类型举例

可变数据类型	不可变数据类型
Counter	Date
Java 数组	String

105
106

`StackOverflowError`、`ArithmaticException`、`ArrayIndexOutOfBoundsException`、`OutOfMemoryError` 和 `NullPointerException` 都是典型的例子。你也可以创建自己的异常，最简单的一种是 `RuntimeException`，它会中断程序的执行并打印出一条出错信息：

```
throw new RuntimeException("Error message here.");
```

一种叫做快速出错的常规编程实践提倡，一旦出错就立刻抛出异常，使定位出错位置更容易（这和忽略错误并将异常推迟到以后处理的方式相反）。

1.2.5.13 断言

断言是一条需要在程序的某处确认为 `true` 的布尔表达式。如果表达式的值为 `false`，程序将会终止并报告一条出错信息。我们使用断言来确定程序的正确性并记录我们的意图。例如，假设你计算得到一个值并可以将它作为索引访问一个数组。如果该值为负数，稍后它将会产生一条 `ArrayIndexOutOfBoundsException` 异常。但如果代码中有一句 `assert index >= 0;`，你就能找到出错的位置。还可以选择性地加上一条详细的消息来辅助定位 bug，例如：

```
assert index >= 0 : "Negative index in method X";
```

默认设置没有启用断言，可以在命令行下使用 `-enableassertions` 标志（简写为 `-ea`）启用断言。

[107] 学习使用断言来保证代码永远不会被系统错误终止或是进入死循环。一种叫做契约式设计的编程模型采用的就是这种思想。数据类型的设计者需要说明前提条件（用例在调用某个方法前必须满足的条件）、后置条件（实现在方法返回时必须达到的要求）和副作用（方法可能对对象状态产生的任何其他变更）。在开发过程中，这些条件可以用断言进行测试。

1.2.5.14 小结

本节所讨论的语言机制说明实用数据类型的设计中所遇到的问题并不容易解决。专家们仍然在讨论支持某些我们已经学习过的设计理念的最佳方法。为什么 Java 不允许将函数作为参数？为什么 Matlab 会复制作参数传递给函数的数组？正如本章前文所述，如果你总是抱怨编程语言的特性，那么你只能自己设计编程语言了。如果你不希望这样，最好的策略就是使用应用最广泛的编程语言。大多数系统都含有大量的库，在适当的时候你应该能用到它们，但通常你都能够通过构造易于移植到其他编程语言的抽象层来简化用例代码并进行自我保护。设计数据类型是你的主要目标，从而使大多数工作都能在抽象层次完成，且和手头的问题匹配。

表 1.2.19 总结了我们讨论过的各种 Java 类。

表 1.2.19 Java 类（数据类型的实现）

类的类别	举 例	特 点
静态方法	<code>Math StdIn StdOut</code>	没有实例变量
不可变的抽象数据类型	<code>Date Transaction String Integer</code>	实例变量均为 <code>private</code> 实例变量均为 <code>final</code> 保护性复制引用类型数据 注意：这些都是必要但不充分条件
可变的抽象数据类型	<code>Counter Accumulator</code>	实例变量均为 <code>private</code> 并非所有实例变量均为 <code>final</code>
具有 I/O 副作用的抽象数据类型	<code>VisualAccumulator In Out Draw</code>	实例变量均为 <code>private</code> 实例方法会处理 I/O

答疑

问 为什么要使用数据抽象？

答 它能够帮助我们编写可靠而正确的代码。例如，在2000年的美国总统竞选中，Al Gore在弗罗里达州的Volusia县的一个电子计票机上得到了-16022张选票——显然电子计票机软件中的选票计数器的封装不正确！

问 为什么要区别原始数据类型和引用类型？为什么不只用引用类型？

答 因为性能。Java提供了Integer、Double等和原始数据类型对应的引用类型，以供希望忽略这些类型的区别的程序员使用。原始数据类型更接近计算机硬件所支持的数据类型，因此使用它们的程序比使用引用类型的程序运行得更快。

问 数据类型必须是抽象的吗？

答 不。Java也支持public和protected来帮助用例直接访问实例变量。如正文所述，允许用例代码直接访问数据所带来的好处比不上对数据的特定表示方式的依赖所带来的坏处，因此我们代码中所有的实例变量都是私有的(private)，有时也会使用私有实例方法在公有方法之间共享代码。

问 如果我在创建一个对象时忘记使用new关键字会发生什么？

答 对于Java，这种代码看起来就好像你希望调用一个静态方法，却得到一个对象类型的返回值。因为并没有定义这样一个方法，你得到的错误信息和引用一个未定义的符号是一样的。如果编译这段代码：

```
Counter c = Counter("test");
```

会得到这条错误信息：

```
cannot find symbol
symbol : method Counter(String)
```

如果你提供给构造函数的参数数量不对，也会得到相同的出错信息。

110

问 如果我在创建一个对象数组时忘记使用new关键字会发生什么？

答 创建每个对象都需要使用new，所以要创建一个含有N个对象的数组，需要使用N+1次new关键字：创建数组需要一次，创建每个对象各需要一次。如果忘了创建数组：

```
Counter[] a;
a[0] = new Counter("test");
```

你得到的错误信息和尝试为一个未初始化的变量赋值是一样的：

```
variable a might not have been initialized
    a[0] = new Counter("test");
    ^
```

但如果在创建数组中的一个对象时忘了使用new，然后又尝试调用它的方法，会得到一个NullPointerException：

```
Counter[] a = new Counter[2];
a[0].increment();
```

问 为什么不用StdOut.println(x.toString())来打印对象？

答 这条语句也可以，但Java能够自动调用任意对象的toString()方法来帮我们省去这些麻烦，因为println()接受的参数是一个Object对象。

问 指针是什么？

答 问得好。或许上面那个异常应该叫做NullReferenceException。和Java的引用一样，可以把指针看做机器地址。在许多编程语言中，指针是一种原始数据类型，程序员可以用各种方法操作它。

但众所周知，指针的编程非常容易出错，因此需要精心设计指针类的操作以帮助程序员避免错误。

Java 将这种观点发挥到了极致（许多主流编程语言的设计者也赞同这种做法）。在 Java 中，创建引用的方法只有一种（`new`），且改变引用的方法也只有一种（赋值语句）。也就是说，程序员能对引用进行的操作只有创建和复制。在编程语言的行话里，Java 的引用被称为安全指针，因为 Java 能够保证每个引用都会指向某种类型的对象（而且它能找出无用的对象并将其回收）。习惯于编写直接操作指针的程序员认为 Java 完全没有指针，但人们仍在为是否真的需要不安全的指针而争论。

问 我在哪里能够找到 Java 如何实现引用和进行垃圾收集的细节？

答 Java 系统的实现各有不同。例如，实现引用的一种自然方式是使用指针（机器地址）；而另一种使用的则可能是句柄（指针的指针）。前者访问数据的速度更快，而后者则能够更好地实现垃圾回收。

问 导入（`import`）一个对象名意味着什么？

答 没什么，只是可以少打一些字。如果不使用 `import` 语句，你也可以在代码中用 `java.util.Arrays` 替代所有的 `Arrays`。

问 实现继承有什么问题？

答 子类继承阻碍模块化编程的原因有两点。第一，父类的任何改动都会影响它的所有子类。子类的开发不可能和父类无关。事实上，子类是完全依赖于父类的。这种问题被称为脆弱的基类问题。第二，子类代码可以访问所有实例变量，因此它们可能会扭曲父类代码的意图。例如，用于选票统计系统的 `Counter` 类的设计者可能会尽最大努力保证 `Counter` 每次只能将计数器加一（还记得 Al Gore 的问题吗）。但它的子类可以完全访问这个实例变量，因此可以将它改变为任意值。

问 怎样才能使一个类不可变？

答 要保证含有一个可变类型的实例变量的数据类型的不可变性，需要得到一个本地副本，这被称为保护性复制，但这也不一定能够达到目的。得到副本是一个方面，保证没有任何实例方法能够改变数据的值是另一方面。

问 什么是空（`null`）？

答 它是一个不指向任何对象的字面量。引用 `null` 调用一个方法是没有意义的，并且会产生 `NullPointerException`。如果你得到了这条错误信息，请检查并确认构造函数是否正确地初始化了类的所有实例变量。

问 实现某种数据类型的类中能否存在静态方法？

答 当然可以。例如，我们实现的所有类中都含有一个 `main()` 方法。另外，对于涉及多个对象的操作，如果它们都不是触发该方法的合适对象，那么就应该考虑添加一个静态方法。例如，我们可以在 `Point` 类中定义如下静态方法：

```
public static double distance(Point a, Point b)
{
    return a.distTo(b);
}
```

这种方法常常能够简化用例代码。

问 除了参数变量、局部变量和实例变量外还有其他种类的变量吗？

答 如果你在类的声明中包含了关键字 `static`（在其他类型之前），就创建了一种称为静态变量的完全不同的变量。和实例变量一样，类中的所有方法都可以访问静态变量，但静态变量却并不和任何具体的对象相关联。在较老的编程语言中，这种变量被称为全局变量，因为它们的作用域是全局的。在现代编程中，我们希望限制变量的作用域，因此很少使用这种变量。在使用它们时会非常小心。

问 什么是弃用 (deprecated) 的方法?

答 不再被支持但为了保持兼容性而留在 API 中的方法叫做弃用的方法。例如，Java 曾经包含了一个 `Character.isSpace()` 的方法，程序员也使用这个方法编写了一些程序。当 Java 的设计者们后来希望支持 Unicode 空白字符时，他们无法既改变 `isSpace()` 的行为又不损害用例程序。因此他们添加了一个新方法 `Character.isWhiteSpace()` 并放弃了老的方法。随着时间的推移，这种方式显然会使 API 更复杂。有时候甚至整个类都会被弃用。例如，Java 为了更好地支持国际化就将它的 `java.util.Date` 标记为弃用。

113

练习

1.2.1 编写一个 `Point2D` 的用例，从命令行接受一个整数 N 。在单位正方形中生成 N 个随机点，然后计算两点之间的最近距离。

1.2.2 编写一个 `Interval1D` 的用例，从命令行接受一个整数 N 。从标准输入中读取 N 个间隔（每个间隔由一对 `double` 值定义）并打印出所有相交的间隔对。

1.2.3 编写一个 `Interval2D` 的用例，从命令行接受参数 N 、`min` 和 `max`。生成 N 个随机的 2D 间隔，其宽和高均匀地分布在单位正方形中的 `min` 和 `max` 之间。用 `StdDraw` 画出它们并打印出相交的间隔对的数量以及有包含关系的间隔对数量。

1.2.4 以下这段代码会打印出什么？

```
String string1 = "hello";
String string2 = string1;
string1 = "world";
StdOut.println(string1);
StdOut.println(string2);
```

1.2.5 以下这段代码会打印出什么？

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
StdOut.println(s);
```

答： "hello World"。 `String` 对象是不可变的——所有字符串方法都会返回一个新的 `String` 对象（但它们不会改变参数对象的值）。这段代码忽略了返回的对象并直接打印了原字符串。要打印出 "WORLD"，请用 `s = s.toUpperCase()` 和 `s = s.substring(6, 11)`。

1.2.6 如果字符串 `s` 中的字符循环移动任意位置之后能够得到另一个字符串 `t`，那么 `s` 就被称为 `t` 的回环变位 (circular rotation)。例如，ACTGACG 就是 TGACCGAC 的一个回环变位，反之亦然。判定这个条件在基因组序列的研究中是很重要的。编写一个程序检查两个给定的字符串 `s` 和 `t` 是否互为回环变位。提示：答案只需要一行用到 `indexOf()`、`length()` 和字符串连接的代码。

114

1.2.7 以下递归函数的返回值是什么？

```
public static String mystery(String s)
{
    int N = s.length();
    if (N <= 1) return s;
    String a = s.substring(0, N/2);
    String b = s.substring(N/2, N);
    return mystery(b) + mystery(a);
}
```

- 1.2.8 设 `a[]` 和 `b[]` 均为长数百万的整形数组。以下代码的作用是什么？有效吗？

```
int[] t = a; a = b; b = t;
```

答：这段代码会将它们交换。它的效率不可能再高了，因为它复制的是引用而不需要复制数百万个元素。

- 1.2.9 修改 `BinarySearch`（请见 1.1.10.1 节中的二分查找代码），使用 `Counter` 统计在有查找中被检查的键的总数并在查找全部结束后打印该值。提示：在 `main()` 中创建一个 `Counter` 对象并将它作为参数传递给 `rank()`。

- 1.2.10 编写一个类 `VisualCounter`，支持加一和减一操作。它的构造函数接受两个参数 `N` 和 `max`，其中 `N` 指定了操作的最大次数，`max` 指定了计数器的最大绝对值。作为副作用，用图像显示每次计数器变化后的值。

- 1.2.11 根据 `Date` 的 API 实现一个 `SmartDate` 类型，在日期非法时抛出一个异常。

- 1.2.12 为 `SmartDate` 添加一个方法 `dayOfTheWeek()`，为日期中每周的日返回 `Monday`、`Tuesday`、`Wednesday`、`Thursday`、`Friday`、`Saturday` 或 `Sunday` 中的适当值。你可以假定时间是 21 世纪。

- 1.2.13 用我们对 `Date` 的实现（请见表 1.2.12）作为模板实现 `Transaction` 类型。

- 115
116 1.2.14 用我们对 `Date` 中的 `equals()` 方法的实现（请见 1.2.5.8 节中的 `Date` 类代码框）作为模板，实现 `Transaction` 中的 `equals()` 方法。

提高题

- 1.2.15 文件输入。基于 `String` 的 `split()` 方法实现 `In` 中的静态方法 `readInts()`。

解答：

```
public static int[] readInts(String name)
{
    In in = new In(name);
    String input = StdIn.readAll();
    String[] words = input.split("\\s+");
    int[] ints = new int[words.length];
    for(int i = 0; i < words.length; i++)
        ints[i] = Integer.parseInt(words[i]);
    return ints;
}
```

我们会在 1.3 节中学习另一个不同的实现（请见 1.3.1.5 节）。

- 1.2.16 有理数。为有理数实现一个不可变数据类型 `Rational`，支持加减乘除操作。

<code>public class Rational</code>	
<code> Rational(int numerator, int denominator)</code>	
<code> Rational plus(Rational b)</code>	该数与 <code>b</code> 之和
<code> Rational minus(Rational b)</code>	该数与 <code>b</code> 之差
<code> Rational times(Rational b)</code>	该数与 <code>b</code> 之积
<code> Rational divides(Rational b)</code>	该数与 <code>b</code> 之商
<code> boolean equals(Rational that)</code>	该数与 <code>that</code> 相等吗
<code> String toString()</code>	对象的字符串表示

无需测试溢出（请见练习 1.2.17），只需使用两个 `long` 型实例变量表示分子和分母来控制溢出

的可能性。使用欧几里德算法来保证分子和分母没有公因子。编写一个测试用例检测你实现的所有方法。

117

- 1.2.17 有理数实现的健壮性。在 `Rational` (请见练习 1.2.16) 的开发中使用断言来防止溢出。
- 1.2.18 累加器的方差。以下代码为 `Accumulator` 类添加了 `var()` 和 `stddev()` 方法，它们计算了 `addDataValue()` 方法的参数的方差和标准差，验证这段代码。

```
public class Accumulator
{
    private double m;
    private double s;
    private int N;
    public void addDataValue(double x)
    {
        N++;
        s = s + 1.0 * (N-1) / N * (x - m) * (x - m);
        m = m + (x - m) / N;
    }
    public double mean()
    { return m; }
    public double var()
    { return s/(N - 1); }
    public double stddev()
    { return Math.sqrt(this.var()); }
}
```

与直接对所有数据的平方求和的方法相比较，这种实现能够更好地避免四舍五入产生的误差。

118

- 1.2.19 字符串解析。为你在练习 1.2.13 中实现的 `Date` 和 `Transaction` 类型编写能够解析字符串数据的构造函数。它接受一个 `String` 参数指定的初始值，格式如表 1.2.20 所示：

表 1.2.20 被解析的字符串的格式

类 型	格 式	举 例
<code>Date</code>	由斜杠分隔的整数	5/22/1939
<code>Transaction</code>	客户、日期和金额，由空白字符分隔	Turing 5/22/1939 11.99

部分解答：

```
public Date(String date)
{
    String[] fields = date.split("/");
    month = Integer.parseInt(fields[0]);
    day   = Integer.parseInt(fields[1]);
    year  = Integer.parseInt(fields[2]);
}
```

119



1.3 背包、队列和栈

许多基础数据类型都和对象的集合有关。具体来说，数据类型的值就是一组对象的集合，所有操作都是关于添加、删除或是访问集合中的对象。在本节中，我们将学习三种这样的数据类型，分别是背包（Bag）、队列（Queue）和栈（Stack）。它们的不同之处在于删除或者访问对象的顺序不同。

背包、队列和栈数据类型都非常基础并且应用广泛。我们在本书的各种实现中也会不断用到它们。除了这些应用以外，本节中的实现和用例代码也展示了我们开发数据结构和算法的一般方式。

本节的第一个目标是说明我们对集合中的对象的表示方式将直接影响各种操作的效率。对于集合来说，我们将会设计适于表示一组对象的数据结构并高效地实现所需的方法。

本节的第二个目标是介绍泛型和迭代。它们都是简单的 Java 概念，但能极大地简化用例代码。它们是高级的编程语言机制，虽然对于算法的理解并不是必需的，但有了它们我们能够写出更加清晰、简洁和优美的用例（以及算法的实现）代码。

本节的第三个目标是介绍并说明链式数据结构的重要性，特别是经典数据结构链表，有了它我们才能高效地实现背包、队列和栈。理解链表是学习各种算法和数据结构中最关键的第一步。

对于这三种数据结构，我们都会学习其 API 和用例，然后再讨论数据类型的值的所有可能的表示方法以及各种操作的实现。这种模式会在全书中反复出现（且数据结构会越来越复杂）。这里的实现是下文所有实现的模板，值得仔细研究。

120

1.3.1 API

照例，我们对集合型的抽象数据类型的讨论从定义它们的 API 开始，如表 1.3.1 所示。每份 API 都含有一个无参数的构造函数、一个向集合中添加单个元素的方法、一个测试集合是否为空的方法和一个返回集合大小的方法。Stack 和 Queue 都含有一个能够删除集合中的特定元素的方法。除了这些基本内容之外，我们将在以下几节中解释这几份 API 反映出的两种 Java 特性：泛型与迭代。

表 1.3.1 泛型可迭代的基础集合数据类型的 API

背包

<code>public class Bag<Item> implements Iterable<Item></code>	
<code> Bag()</code>	创建一个空背包
<code> void add(Item item)</code>	添加一个元素
<code> boolean isEmpty()</code>	背包是否为空
<code> int size()</code>	背包中的元素数量

先进先出（FIFO）队列

<code>public class Queue<Item> implements Iterable<Item></code>	
<code> Queue()</code>	创建空队列
<code> void enqueue(Item item)</code>	添加一个元素
<code> Item dequeue()</code>	删除最近添加的元素
<code> boolean isEmpty()</code>	队列是否为空
<code> int size()</code>	队列中的元素数量

(续)

下压（后进先出，LIFO）栈

<code>public class Stack<Item> implements Iterable<Item></code>	
<code> Stack()</code>	创建一个空栈
<code> void push(Item item)</code>	添加一个元素
<code> Item pop()</code>	删除最近添加的元素
<code> boolean isEmpty()</code>	栈是否为空
<code> int size()</code>	栈中的元素数量

[121]

1.3.1.1 泛型

集合类的抽象数据类型的一个关键特性是我们应该可以用它们存储任意类型的数据。一种特别的 Java 机制能够做到这一点，它被称为泛型，也叫做参数化类型。泛型对编程语言的影响非常深刻，许多语言并没有这种机制（包括早期版本的 Java）。在这里我们对泛型的使用仅限于一点额外的 Java 语法，非常容易理解。在每份 API 中，类名后的 `<Item>` 记号将 `Item` 定义为一个类型参数，它一个象征性的占位符，表示的是用例将会使用的某种具体数据类型。可以将 `Stack<Item>` 理解为某种元素的栈。在实现 `Stack` 时，我们并不知道 `Item` 的具体类型，但用例可以用我们的栈处理任意类型的数据，甚至是在我们的实现之后才出现的数据类型。在创建栈时，用例会提供一种具体的数据类型：我们可以将 `Item` 替换为任意引用数据类型（`Item` 出现的每个地方都是如此）。这种能力正是我们所需要的。例如，可以编写如下代码来用栈处理 `String` 对象：

```
Stack<String> stack = new Stack<String>();
stack.push("Test");
...
String next = stack.pop();
```

并在以下代码中使用队列处理 `Date` 对象：

```
Queue<Date> queue = new Queue<Date>();
queue.enqueue(new Date(12, 31, 1999));
...
Date next = queue.dequeue();
```

如果你尝试向 `stack` 变量中添加一个 `Date` 对象（或是任何其他非 `String` 类型的数据）或者向 `queue` 变量中添加一个 `String` 对象（或是任何其他非 `Date` 类型的数据），你会得到一个编译时错误。如果没有泛型，我们必须为需要收集的每种数据类型定义（并实现）不同的 API。有了泛型，我们只需要一份 API（和一次实现）就能够处理所有类型的数据，甚至是在未来定义的数据类型。你很快将会看到，使用泛型的用例代码很容易理解和调试，因此全书中我们都会用到它。

1.3.1.2 自动装箱

类型参数必须被实例化为引用类型，因此 Java 有一种特殊机制来使泛型代码能够处理原始数据类型。我们还记得 Java 的封装类型都是原始数据类型所对应的引用类型：`Boolean`、`Byte`、`Character`、`Double`、`Float`、`Integer`、`Long` 和 `Short` 分别对应着 `boolean`、`byte`、`char`、`double`、`float`、`int`、`long` 和 `short`。在处理赋值语句、方法的参数和算术或逻辑表达式时，Java 会自动在引用类型和对应的原始数据类型之间进行转换。在这里，这种转换有助于我们同时使用泛型和原始数据类型。例如：

[122]

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);           // 自动装箱 (int -> Integer)
int i = stack.pop();      // 自动拆箱 (Integer -> int)
```

自动将一个原始数据类型转换为一个封装类型被称为自动装箱，自动将一个封装类型转换为一个原始数据类型被称为自动拆箱。在这个例子中，当我们把一个原始类型的值 17 传递给 `push()` 方法时，Java 将它的类型自动转换（自动装箱）为 `Integer`。`pop()` 方法返回了一个 `Integer` 类型的值，Java 在将它赋予变量 `i` 之前将它的类型自动转换（自动拆箱）为了 `int`。

1.3.1.3 可迭代的集合类型

对于许多应用场景，用例的要求只是用某种方式处理集合中的每个元素，或者叫做迭代访问集合中的所有元素。这种模式非常重要，在 Java 和其他许多语言中它都是一级语言特性（不只是库，编程语言本身就含有特殊的机制来支持它）。有了它，我们能够写出清晰简洁的代码且不依赖于集合类型的具体实现。例如，假设用例在 `Queue` 中维护一个交易集合，如下：

```
Queue<Transaction> collection = new Queue<Transaction>();
```

如果集合是可迭代的，用例用一行语句即可打印出交易的列表：

```
for (Transaction t : collection)
{ StdOut.println(t); }
```

这种语法叫做 `foreach` 语句：可以将 `for` 语句看做对于集合中的每个交易 `t(foreach)`，执行以下代码段。这段用例代码不需要知道集合的表示或实现的任何细节，它只想逐个处理集合中的元素。相同的 `for` 语句也可以处理交易的 `Bag` 对象或是任何可迭代的集合。很难想象还有比这更加清晰和简洁的代码。你将会看到，支持这种迭代需要在实现中添加额外的代码，但这些工作是值得的。

有趣的是，`Stack` 和 `Queue` 的 API 的唯一不同之处只是它们的名称和方法名。这让我们认识到无法简单地通过一列方法的签名说明一个数据类型的所有特点。在这里，只有自然语言的描述才能说明选择被删除元素（或是在 `foreach` 语句中下一个被处理的元素）的规则。这些规则的差异是 API 的重要组成部分，而且显然对用例代码的开发十分重要。

123

1.3.1.4 背包

背包是一种不支持从中删除元素的集合数据类型——它的目的就是帮助用例收集元素并迭代遍历所有收集到的元素（用例也可以检查背包是否为空或者获取背包中元素的数量）。迭代的顺序不确定且与用例无关。要理解背包的概念，可以想象一个非常喜欢收集弹子球的人。他将所有的弹子球都放在一个背包里，一次一个，并且会不时在所有的弹子球中寻找某一颗拥有某种特点的弹子球。使用 `Bag` 的 API，用例可以将元素添加进背包并根据需要随时使用 `foreach` 语句访问所有的元素。用例也可以使用栈或是队列，但使用 `Bag` 可以说明元素的处理顺序不重要。图 1.3.1 所示的 `Stats` 类是 `Bag` 的一个典型用例。

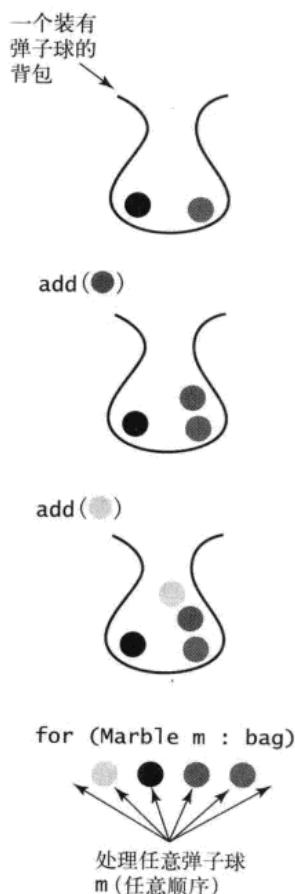


图 1.3.1 背包的操作(另见彩插)

它的任务是简单地计算标准输入中的所有 `double` 值的平均值和样本标准差。如果标准输入中有 N 个数字，那么平均值为它们的和除以 N ，样本标准差为每个值和平均值之差的平方之和除以 $N-1$ 之后的平方根。在这些计算中，数的计算顺序和结果无关，因此我们将它们保存在一个 `Bag` 对象中并使用 `foreach` 语法来计算每个和。注意：不需要保存所有的数也可以计算标准差（就像我们在 `Accumulator` 中计算平均值那样——请见练习 1.2.18）。用 `Bag` 对象保存所有数字是更复杂的统计计算所必需的。

124

以下代码框列出的是常用的背包用例。

背包的典型用例

```
public class Stats
{
    public static void main(String[] args)
    {
        Bag<Double> numbers = new Bag<Double>();
        while (!StdIn.isEmpty())
            numbers.add(StdIn.readDouble());
        int N = numbers.size();
        double sum = 0.0;
        for (double x : numbers)
            sum += x;
        double mean = sum/N;
        sum = 0.0;
        for (double x : numbers)
            sum += (x - mean)*(x - mean);
        double std = Math.sqrt(sum/(N-1));
        StdOut.printf("Mean: %.2f\n", mean);
        StdOut.printf("Std dev: %.2f\n", std);
    }
}
```

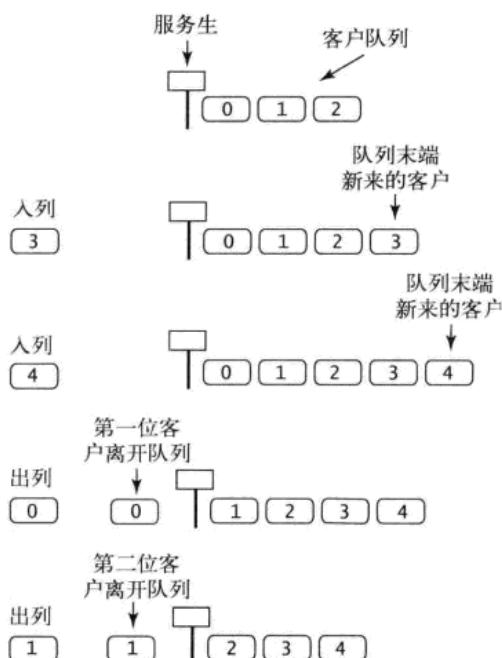
使用方法

```
% java Stats
100
99
101
120
98
107
109
81
101
90
Mean: 100.60
Std dev: 10.51
```

125

1.3.1.5 先进先出队列

先进先出队列（或简称队列）是一种基于先进先出（FIFO）策略的集合类型，如图 1.3.2 所示。按照任务产生的顺序完成它们的策略我们每天都会遇到：在剧院门前排队的人们、在收费站前排队的汽车或是计算机上某种软件中等待处理的任务。任何服务性策略的基本原则都是公平。在提到公平时大多数人的第一个想法就是应该优先服务等待最久的人，这正是先进先出策略的准则。队列是许多日常现象的自然模型，它也是无数应用程序的核心。当用例使用 `foreach` 语句迭代访问队列中的元素时，元素的处理顺序就是它们被添加到队列中的顺序。在应用程序中使用队列的主要原因是在用集合保存元素的同时保存它们的相对顺序：使它们入列顺序和出列顺序相同。例如，下页的用例是我们的 `In` 类的静态方法 `readInts()` 的一种实现。这个方法为用例解决的问题是用例无需预先知道文件的大小即可将文件中的所有整数读入一个数组中。我们首先将所有的整数读入队列中，然后使用 `Queue` 的 `size()` 方法得到所需数组的大小，创建数组并将队列中的所有整数移动到数组中。队列之所以合适是因为它能够将整数按照文件中的顺序放入数组中（如果该顺序并不重要，也可以使用 `Bag` 对象）。这段代码使用了自动装箱和拆箱来转换用例中的 `int` 原始数据类型和队列的 `Integer` 封装类型。



126 图 1.3.2 一个典型的先进先出队列

1.3.1.6 下压栈

下压栈（或简称栈）是一种基于后进先出（LIFO）策略的集合类型，如图 1.3.3 所示。当你的邮件在桌上放成一叠时，使用的就是栈。新邮件来到时你将它们放在最上面，当你有空时你会一封一封地从上到下阅读它们。现在人们应付的纸质品比以前少得多，但计算机上的许多常用程序遵循相同的组织原则。例如，许多人仍然用栈的方式存放电子邮件——在收信时将邮件压入（push）最顶端，在取信时从最顶端将它们弹出（pop），且第一封一定是最新的邮件（后进，先出）。这种策略的好处是我们能够及时看到感兴趣的邮件，坏处是如果你不把栈清空，某些较早的邮件可能永远也不会被阅读。你在网上冲浪时很可能遇到栈的另一个例子。点击一个超链接，浏览器会显示一个新的页面（并将它压入一个栈）。你可以不断点击超链接并访问新页面，但总是可以通过点击“回退”按钮重新访问以前的页面（从栈中弹出）。栈的后进先出策略正好能够提供你所需要的行为。当用例使用 `foreach` 语句迭代遍历栈中的元素时，元素的处理顺序和它们被压入

```
public static int[] readInts(String name)
{
    In in = new In(name);
    Queue<Integer> q = new Queue<Integer>();
    while (!in.isEmpty())
        q.enqueue(in.readInt());
    int N = q.size();
    int[] a = new int[N];
    for (int i = 0; i < N; i++)
        a[i] = q.dequeue();
    return a;
}
```

Queue 的用例

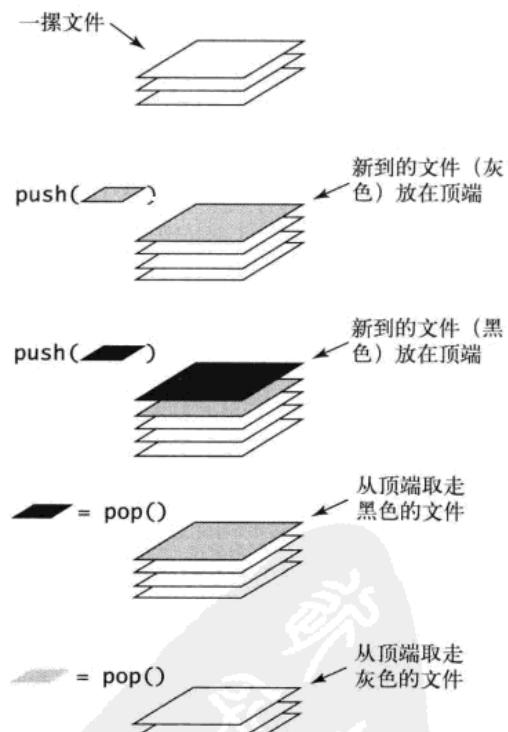


图 1.3.3 下压栈的操作

的顺序正好相反。在应用程序中使用栈迭代器的一个典型原因是在用集合保存元素的同时颠倒它们的相对顺序。例如，右侧的用例 Reverse 将会把标准输入中的所有整数逆序排列，同样它也无需预先知道整数的多少。在计算机领域，栈具有基础而深远的影响，下一节我们会仔细研究一个例子，以说明栈的重要性。

1.3.1.7 算术表达式求值

我们要学习的另一个栈用例同时也是展示泛型的应用的一个经典例子。我们在 1.1 节中最初学习的几个程序之一就是用来计算算术表达式的值的，例如：

$$(1 + ((2 + 3) * (4 * 5)))$$

如果将 4 乘以 5，把 3 加上 2，取它们的积然后加 1，就得到了 101。但 Java 系统是如何完成这些运算的呢？不需要研究 Java 系统的构造细节，我们也可以编写一个 Java 程序来解决这个问题。它接受一个输入字符串（表达式）并输出表达式的值。为了简化问题，首先来看一下这份明确的递归定义：算术表达式可能是一个数，或者是由一个左括号、一个算术表达式、一个运算符、另一个算术表达式和一个右括号组成的表达式。简单起见，这里定义的是未省略括号的算术表达式，它明确地说明了所有运算符的操作数——你可能更熟悉形如 $1 + 2 * 3$ 的表达式，省略了括号，而采用优先级规则。我们将要学习的简单机制也能处理优先级规则，但在这里我们不想把问题复杂化。为了突出重点，我们支持最常见的二元运算符 *、+、- 和 /，以及只接受一个参数的平方根运算符 sqrt。我们也可以轻易支持更多数量和种类的运算符来计算多种大家熟悉的数学表达式，包括三角函数、指数和对数函数。我们的重点在于如何解析由括号、运算符和数字组成的字符串，并按照正确的顺序完成各种初级算术运算操作。如何才能够得到一个（由字符串表示的）算术表达式的值呢？E.W.Dijkstra 在 20 世纪 60 年代发明了一个非常简单的算法，用两个栈（一个用于保存运算符，一个用于保存操作数）完成了这个任务，其实现过程见下页，求值算法的轨迹如图 1.3.4 所示。

表达式由括号、运算符和操作数（数字）组成。我们根据以下 4 种情况从左到右逐个将这些实体送入栈处理：

- 将操作数压入操作数栈；
- 将运算符压入运算符栈；
- 忽略左括号；
- 在遇到右括号时，弹出一个运算符，弹出所需数量的操作数，并将运算符和操作数的运算结果压入操作数栈。

在处理完最后一个右括号之后，操作数栈上只会有一个值，它就是表达式的值。这种方法乍一看有些难以理解，但要证明它能够计算得到正确的值很简单：每当算法遇到一个被括号包围并由一个运算符和两个操作数组成的子表达式时，它都将运算符和操作数的计算结果压入操作数栈。这样的结果就好像在输入中用这个值代替了该子表达式，因此用这个值代替子表达式得到的结果和原表达式相同。我们可以反复应用这个规律并得到一个最终值。例如，用该算法计算以下表达式得到的结果都是相同的：

```
public class Reverse
{
    public static void main(String[] args)
    {
        Stack<Integer> stack;
        stack = new Stack<Integer>();
        while (!StdIn.isEmpty())
            stack.push(StdIn.readInt());
        for (int i : stack)
            StdOut.println(i);
    }
}
```

127

Stack 的用例

128



```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
( 1 + ( 5 * ( 4 * 5 ) ) )
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

上一页中的 Evaluate 类是该算法的一个实现。这段代码是一个简单的“解释器”：一个能够解释给定字符串所表达的运算并计算得到结果的程序。

Dijkstra 的双栈算术表达式求值算法

```
public class Evaluate
{
    public static void main(String[] args)
    {
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty())
        { // 读取字符，如果是运算符则压入栈
            String s = StdIn.readString();
            if (s.equals("(")) ;
            else if (s.equals("+")) ops.push(s);
            else if (s.equals("-")) ops.push(s);
            else if (s.equals("*")) ops.push(s);
            else if (s.equals("/")) ops.push(s);
            else if (s.equals("sqrt")) ops.push(s);
            else if (s.equals(")")))
            { // 如果字符为 "}", 弹出运算符和操作数，计算结果并压入栈
                String op = ops.pop();
                double v = vals.pop();
                if (op.equals("+")) v = vals.pop() + v;
                else if (op.equals("-")) v = vals.pop() - v;
                else if (op.equals("*")) v = vals.pop() * v;
                else if (op.equals("/")) v = vals.pop() / v;
                else if (op.equals("sqrt")) v = Math.sqrt(v);
                vals.push(v);
            } // 如果字符既非运算符也不是括号，将它作为 double 值压入栈
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}
```

这段 Stack 的用例使用了两个栈来计算表达式的值。它展示了一种重要的计算模型：将一个字符串解释为一段程序并执行该程序得到结果。有了泛型，我们只需实现 Stack 一次即可使用 String 值的栈和 Double 值的栈。简单起见，这段代码假设表达式没有省略任何括号，数字和字符均以空白字符相隔。

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0

% java Evaluate
( ( 1 + sqrt ( 5.0 ) ) / 2.0 )
1.618033988749895
```

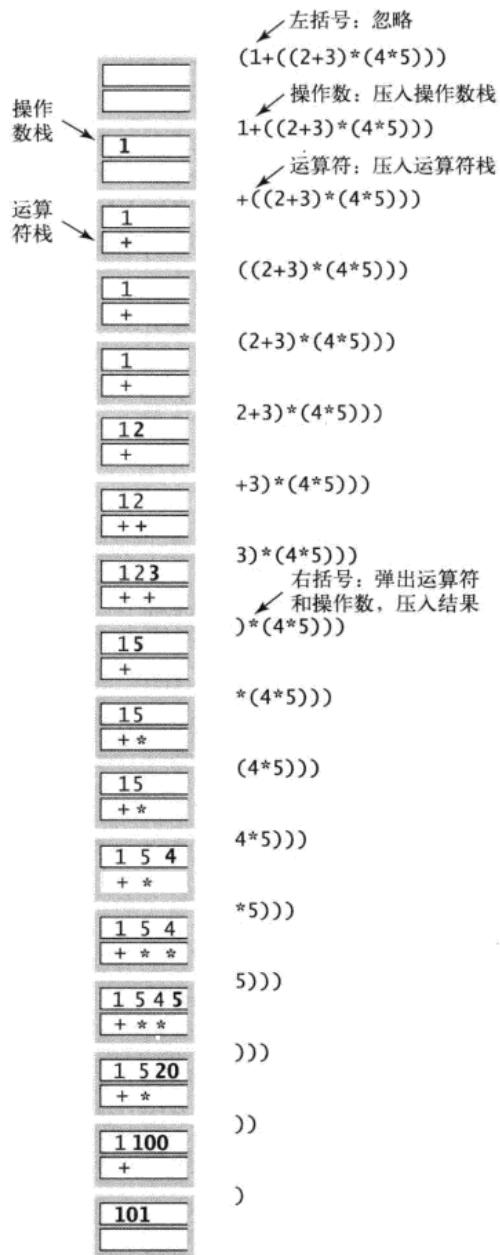


图 1.3.4 Dijkstra 的双栈算术表达式求值算法的轨迹

131

1.3.2 集合类数据类型的实现

在讨论 Bag、Stack 和 Queue 的实现之前，我们会先给出一个简单而经典的实现，然后讨论它的改进并得到表 1.3.1 中的 API 的所有实现。

1.3.2.1 定容栈

作为热身，我们先来看一种表示容量固定的字符串栈的抽象数据类型，如表 1.3.2 所示。它的 API 和 Stack 的 API 有所不同：它只能处理 String 值，它要求用例指定一个容量且不支持迭代。

实现一份 API 的第一步就是选择数据的表示方式。对于 `FixedCapacityStackOfStrings`，我们显然可以选择 `String` 数组。由此我们可以得到表 1.3.2 中底部的实现，它已经是简单得不能再简单了（每个方法都只有一行）。它的实例变量为一个用于保存栈中的元素的数组 `a[]`，和一个用于保存栈中的元素数量的整数 `N`。要删除一个元素，我们将 `N` 减 1 并返回 `a[N]`。要添加一个元素，我们将 `a[N]` 设为新元素并将 `N` 加 1。这些操作能够保证以下性质：

表 1.3.2 一种表示定容字符串栈的抽象数据类型

API	<code>public class FixedCapacityStackOfStrings</code>	
	<code> FixedCapacityStackOfStrings(int cap)</code>	创建一个容量为 <code>cap</code> 的空栈
<code>void</code>	<code> push(String item)</code>	添加一个字符串
<code>String</code>	<code> pop()</code>	删除最近添加的字符串
<code>boolean</code>	<code> isEmpty()</code>	栈是否为空
<code>int</code>	<code> size()</code>	栈中的字符串数量

测试用例	<code>public static void main(String[] args)</code>
	{ <code> FixedCapacityStackOfStrings s;</code> <code> s = new FixedCapacityStackOfStrings(100);</code> <code> while (!StdIn.isEmpty())</code> { <code> String item = StdIn.readString();</code> <code> if (!item.equals("-"))</code> <code> s.push(item);</code> <code> else if (!s.isEmpty()) StdOut.print(s.pop() + " ");</code> } <code> StdOut.println("(" + s.size() + " left on stack)");</code> }

使用方法	<code>% more tobe.txt</code> <code>to be or not to - be - - that - - - is</code> <code>% java FixedCapacityStackOfStrings < tobe.txt</code> <code>to be not that or be (2 left on stack)</code>
------	---

数据类型的实现	<code>public class FixedCapacityStackOfStrings</code>
	{ <code> private String[] a; // stack entries</code> <code> private int N; // size</code> <code> public FixedCapacityStackOfStrings(int cap)</code> { <code> a = new String[cap];</code> <code> N = 0;</code> } <code> public boolean isEmpty() { return N == 0; }</code> <code> public int size() { return N; }</code> <code> public void push(String item)</code> { <code> a[N++] = item;</code> } <code> public String pop()</code> { <code> return a[--N];</code> }

- 数组中的元素顺序和它们被插入的顺序相同；
- 当 N 为 0 时栈为空；
- 栈的顶部位于 a[N-1]（如果栈非空）。

和以前一样，用恒等式的方式思考这些条件是检验实现正常工作的最简单的方式。请你务必完全理解这个实现。做到这一点的最好方法是检验一系列操作中栈内容的轨迹，如表 1.3.3 所示。测试用例会从标准输入读取多个字符串并将它们压入一个栈，当遇到 - 时它会将栈的内容弹出并打印结果。这种实现的主要性能特点是 push 和 pop 操作所需的时间独立于栈的长度。许多应用会因为这种简洁性而选择它。但几个缺点限制了它作为通用工具的潜力，我们要改进的也是这一点。经过一些修改（以及 Java 语言机制的一些帮助），我们就能给出一个适用性更加广泛的实现。这些努力是值得的，因为这个实现是本书中其他许多更强大的抽象数据类型的模板。

132
133

表 1.3.3 FixedCapacityStackOfStrings 的测试用例的轨迹

StdIn (push)	StdOut (pop)	N	a[]				
			0	1	2	3	4
		0					
to	1	to					
be	2	to be					
or	3	to be or					
not	4	to be or not					
to	5	to be or not to					
-	to 4	to be or not to					
be	5	to be or not be					
-	be 4	to be or not be					
-	not 3	to be or not be					
that	4	to be or that be					
-	that 3	to be or that be					
-	or 2	to be or that be					
-	be 1	to be or that be					
is	2	to is or not to					

1.3.2.2 泛型

FixedCapacityStackOfStrings 的第一个缺点是它只能处理 String 对象。如果需要一个 double 值的栈，你就需要用类似的代码实现另一个类，也就是把所有的 String 都替换为 double。这还算简单，但如果我们要实现 Transaction 类型的栈或者 Date 类型的队列等，情况就很棘手了。如 1.3.1.1 节的讨论所示，Java 的参数类型（泛型）就是专门用来解决这个问题的，而且我们也看过了几个用例的代码（请见 1.3.1.4 节、1.3.1.5 节、1.3.1.6 节和 1.3.1.7 节）。但如何才能实现一个泛型的栈呢？表 1.3.4 中的代码展示了实现的细节。它实现了一个 FixedCapacityStack 类，该类和 FixedCapacityStackOfStrings 类的区别仅在于加粗部分的代码——我们把所有的 String 都替换为 Item（一个地方除外，会在稍后讨论）并用下面这行代码声明了该类：

```
public class FixedCapacityStack<Item>
```

Item 是一个类型参数，用于表示用例将会使用的某种具体类型的象征性的占位符。可以将 FixedCapacityStack<Item> 理解为某种元素的栈，这正是我们想要的。在实现 FixedCapacityStack 时，我们并不知道 Item 的实际类型，但用例只要能在创建栈时提供具体的数据类型，它就能用栈处理任意数据类型。实际的类型必须是引用类型，但用例可以依靠自动装箱将原始数据类型转换为相应的封装类型。Java 会使用类型参数 Item 来检查类型不匹配的错误——尽管具体的数据类型还不知道，赋予 Item 类型变量的值也必须是 Item 类型的，等等。在这里有一个细节非常重要：我们希望用以下代码在 FixedCapacityStack 的构造函数的实现中创建一个泛型的数组：

```
a = new Item[cap];
```

由于某些历史和技术原因（不在本书讲解范围之内），创建泛型数组在 Java 中是不允许的。我

们需要使用类型转换：

```
a = (Item[]) new Object[cap];
```

这段代码才能够达到我们所期望的效果（但Java编译器会给出一条警告，不过可以忽略它），

[134] 我们在本书中会一直使用这种方式（Java系统库中类似抽象数据类型的实现中也使用了相同的方式）。

表 1.3.4 一种表示泛型定容栈的抽象数据类型

API	public class FixedCapacityStack<Item>
	FixedCapacityStack(int cap)
void	push(Item item)
Item	pop()
boolean	isEmpty()
int	size()

测试用例

```
public static void main(String[] args)
{
    FixedCapacityStack<String> s;
    s = new FixedCapacityStack<String>(100);
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (!item.equals("-"))
            s.push(item);
        else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
    }
    StdOut.println("(" + s.size() + " left on stack)");
}
```

使用方法

```
% more tobe.txt
to be or not to - be - - that - - is
% java FixedCapacityStack < tobe.txt
to be not that or be (2 left on stack)
```

数据类型的实现

```
public class FixedCapacityStack<Item>
{
    private Item[] a; // stack entries
    private int N; // size
    public FixedCapacityStack(int cap)
    { a = (Item[]) new Object[cap]; }
    public boolean isEmpty() { return N == 0; }
    public int size() { return N; }
    public void push(Item item)
    { a[N++] = item; }
    public Item pop()
    { return a[--N]; }
}
```

1.3.2.3 调整数组大小

选择用数组表示栈内容意味着用例必须预先估计栈的最大容量。在 Java 中，数组一旦创建，其大小是无法改变的，因此栈使用的空间只能是这个最大容量的一部分。选择大容量的用例在栈为空或几乎为空时会浪费大量的内存。例如，一个交易系统可能会涉及数十亿笔交易和数千个交易的集合。即使这种系统一般都会限制每笔交易只能出现在一个集合中，但用例必须保证所有集合都有能力保存所有的交易。另一方面，如果集合变得比数组更大那么用例有可能溢出。为此，`push()` 方法需要在代码中检测栈是否已满，我们的 API 中也应该含有一个 `isFull()` 方法来允许用例检测栈是否已满。我们在此省略了它的实现代码，因为我们希望用例从处理栈已满的问题中解脱出来，如我们的原始 `Stack` API 所示。因此，我们修改了数组的实现，动态调整数组 `a[]` 的大小，使得它既足以保存所有元素，又不至于浪费过多的空间。实际上，完成这些目标非常简单。首先，实现一个方法将栈移动到另一个大小不同的数组中：

```
private void resize(int max)
{ // 将大小为 N <= max 的栈移动到一个新的大小为 max 的数组中
    Item[] temp = (Item[]) new Object[max];
    for (int i = 0; i < N; i++)
        temp[i] = a[i];
    a = temp;
}
```

现在，在 `push()` 中，检查数组是否太小。具体来说，我们会通过检查栈大小 `N` 和数组大小 `a.length` 是否相等来检查数组是否能够容纳新的元素。如果没有多余的空间，我们会将数组的长度加倍。然后就可以和从前一样用 `a[N++] = item` 插入新元素了：

```
public void push(String item)
{ // 将元素压入栈顶
    if (N == a.length) resize(2*a.length);
    a[N++] = item;
}
```

类似，在 `pop()` 中，首先删除栈顶的元素，然后如果数组太大我们就将它的长度减半。只要稍加思考，你就明白正确的检测条件是栈大小是否小于数组的四分之一。在数组长度被减半之后，它的状态约为半满，在下次需要改变数组大小之前仍然能够进行多次 `push()` 和 `pop()` 操作。

```
public String pop()
{ // 从栈顶删除元素
    String item = a[--N];
    a[N] = null; // 避免对象游离（请见下节）
    if (N > 0 && N == a.length/4) resize(a.length/2);
    return item;
}
```

136

在这个实现中，栈永远不会溢出，使用率也永远不会低于四分之一（除非栈为空，那种情况下数组的大小为 1）。我们会在 1.4 节中详细分析这种实现方法的性能特点。

`push()` 和 `pop()` 操作中数组大小调整的轨迹见表 1.3.5。

1.3.2.4 对象游离

Java 的垃圾收集策略是回收所有无法被访问的对象的内存。在我们对 `pop()` 的实现中，被弹出的元素的引用仍然存在于数组中。这个元素实际上已经是一个孤儿了——它永远也不会再被访问了，但 Java 的垃圾收集器没法知道这一点，除非该引用被覆盖。即使用例已经不再需要这个元素了，数组中的引用仍然可以让它继续存在。这种情况（保存一个不需要的对象的引用）

称为游离。在这里，避免对象游离很容易，只需将被弹出的数组元素的值设为 `null` 即可，这将覆盖无用的引用并使系统可以在用例使用完被弹出的元素后回收它的内存。

表 1.3.5 一系列 `push()` 和 `pop()` 操作中数组大小调整的轨迹

<code>push()</code>	<code>pop()</code>	N	<code>a.length</code>	<code>a[]</code>							
				0	1	2	3	4	5	6	7
		0	1	null							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	null				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	null	null	null
-	to	4	8	to	be	or	not	null	null	null	null
be		5	8	to	be	or	not	be	null	null	null
-	be	4	8	to	be	or	not	null	null	null	null
-	not	3	8	to	be	or	null	null	null	null	null
that		4	8	to	be	or	that	null	null	null	null
-	that	3	8	to	be	or	null	null	null	null	null
-	or	2	4	to	be	null	null				
-	be	1	2	to	null						
is		2	2	to	is						

137

1.3.2.5 迭代

本节开头已经提过，集合类数据类型的基本操作之一就是，能够使用 Java 的 `foreach` 语句通过迭代遍历并处理集合中的每个元素。这种方式的代码既清晰又简洁，且不依赖于集合数据类型的具体实现。在讨论迭代的实现之前，我们先看一段能够打印出一个字符串集合中的所有元素的用例代码：

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
    StdOut.println(s);
...
```

这里，`foreach` 语句只是 `while` 语句的一种简写方式（就好像 `for` 语句一样）。它本质上和以下 `while` 语句是等价的：

```
Iterator<String> i = collection.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```

这段代码展示了一些在任意可迭代的集合数据类型中我们都需要实现的东西：

- 集合数据类型必须实现一个 `iterator()` 方法并返回一个 `Iterator` 对象；
- `Iterator` 类必须包含两个方法：`hasNext()`（返回一个布尔值）和 `next()`（返回集合中的一个泛型元素）。

在 Java 中，我们使用接口机制来指定一个类所必须实现的方法（请见 1.2.5.4 节）。对于可迭代的集合数据类型，Java 已经为我们定义了所需的接口。要使一个类可迭代，第一步就是在它的声

明中加入 `implements Iterable<Item>`, 对应的接口 (即 `java.lang.Iterable`) 为:

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

然后在类中添加一个方法 `iterator()` 并返回一个迭代器 `Iterator<Item>`。迭代器都是泛型的, 因此我们可以使用参数类型 `Item` 来帮助用例遍历它们指定的任意类型的对象。对于一直使用的数组表示法, 我们需要逆序迭代遍历这个数组, 因此我们将迭代器命名为 `ReverseArrayIterator`, 并添加了以下方法:

```
public Iterator<Item> iterator()
{
    return new ReverseArrayIterator();
}
```

迭代器是什么? 它是一个实现了 `hasNext()` 和 `next()` 方法的类的对象, 由以下接口所定义 (即 `java.util.Iterator`):

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();
}
```

尽管接口指定了一个 `remove()` 方法, 但在本书中 `remove()` 方法总为空, 因为我们希望避免在迭代中穿插能够修改数据结构的操作。对于 `ReverseArrayIterator`, 这些方法都只需要一行代码, 它们实现在栈类的一个嵌套类中:

```
private class ReverseArrayIterator implements Iterator<Item>
{
    private int i = N;

    public boolean hasNext() { return i > 0; }
    public Item next() { return a[--i]; }
    public void remove() { }
}
```

请注意, 嵌套类可以访问包含它的类的实例变量, 在这里就是 `a[]` 和 `N` (这也是我们使用嵌套类实现迭代器的主要原因)。从技术角度来说, 为了和 `Iterator` 的结构保持一致, 我们应该在两种情况下抛出异常: 如果用例调用了 `remove()` 则抛出 `UnsupportedOperationException`, 如果用例在调用 `next()` 时 `i` 为 0 则抛出 `NoSuchElementException`。因为我们只会在 `foreach` 语法中使用迭代器, 这些情况都不会出现, 所以我们省略了这部分代码。还剩下一个非常重要的细节, 我们需要在程序的开头加上下面这条语句:

```
import java.util.Iterator;
```

因为 (某些历史原因) `Iterator` 不在 `java.lang` 中 (尽管 `Iterable` 是 `java.lang` 的一部分)。现在, 使用 `foreach` 处理该类的用例能够得到的行为和使用普通的 `for` 循环访问数组一样, 但它不知道数据的表示方法是数组 (即实现细节)。对于我们在这本书中学习的和 Java 库中所包含的所有类似于集合的基础数据类型的实现, 这一点非常重要。例如, 我们无需改变任何用例代码就可以随意切换不同的表示方法。更重要的是, 从用例的角度来说, 无需知晓类的实现细节用例也能使用迭代。

算法 1.1 是 `Stack` API 的一种能够动态改变数组大小的实现。用例能够创建任意类型数据的栈, 并支持用例用 `foreach` 语句按照后进先出的顺序迭代访问所有栈元素。这个实现的基础

138

139

是 Java 的语言特性，包括 `Iterable` 和 `Iterator`，但我们没有必要深究这些特性的细节，因为代码本身并不复杂，并且可以用做其他集合数据类型的实现的模板。

例如，我们在实现 `Queue` 的 API 时，可以使用两个实例变量作为索引，一个变量 `head` 指向队列的开头，一个变量 `tail` 指向队列的结尾，如表 1.3.6 所示。在删除一个元素时，使用 `head` 访问它并将 `head` 加 1；在插入一个元素时，使用 `tail` 保存它并将 `tail` 加 1。如果某个索引在增加之后越过了数组的边界则将它重置为 0。实现检查队列是否为空、是否充满并需要调整数组大小的细节是一项有趣而又实用的编程练习（请见练习 1.3.14）。

表 1.3.6 ResizingArrayQueue 的测试用例的轨迹

StdIn (入列)	StdOut (出列)	N	head	tail	a[]							
					0	1	2	3	4	5	6	7
-	to	5	0	5	to	be	or	not	to			
be		4	1	5	to	be	or	not	to			
-	be	5	1	6	to	be	or	not	to	be		
-	be	4	2	6	to	be	or	not	to	be		
-	or	3	3	6	to	be	or	not	to	be		

在算法的学习中，算法 1.1 十分重要，因为它几乎（但还没有）达到了任意集合类数据类型的实现的最佳性能：

- 每项操作的用时都与集合大小无关；
- 空间需求总是不超过集合大小乘以一个常数。

`ResizingArrayQueue` 的缺点在于某些 `push()` 和 `pop()` 操作会调整数组的大小：这项操作的耗时和栈大小成正比。下面，我们将学习一种克服该缺陷的方法，使用一种完全不同的方式来组织数据。

140

算法 1.1 下压 (LIFO) 栈 (能够动态调整数组大小的实现)

```
import java.util.Iterator;
public class ResizingArrayStack<Item> implements Iterable<Item>
{
    private Item[] a = (Item[]) new Object[1]; // 栈元素
    private int N = 0; // 元素数量
    public boolean isEmpty() { return N == 0; }
    public int size() { return N; }
    private void resize(int max)
    { // 将栈移动到一个大小为 max 的新数组
        Item[] temp = (Item[]) new Object[max];
        for (int i = 0; i < N; i++)
            temp[i] = a[i];
        a = temp;
    }
    public void push(Item item)
    { // 将元素添加到栈顶
        if (N == a.length) resize(2*a.length);
        a[N++] = item;
    }
    public Item pop()
    { // 从栈顶删除元素
        Item item = a[--N];
    }
}
```

```

        a[N] = null; // 避免对象游离(请见 1.3.2.4 节)
        if (N > 0 && N == a.length/4) resize(a.length/2);
        return item;
    }
    public Iterator<Item> iterator()
    {
        return new ReverseArrayIterator();
    }
    private class ReverseArrayIterator implements Iterator<Item>
    {
        // 支持后进先出的迭代
        private int i = N;
        public boolean hasNext() { return i > 0; }
        public Item next() { return a[--i]; }
        public void remove() {}
    }
}

```

这份泛型的可迭代的 `Stack` API 的实现是所有集合类抽象数据类型实现的模板。它将所有元素保存在数组中，并动态调整数组的大小以保持数组大小和栈大小之比小于一个常数。

141

1.3.3 链表

现在我们来学习一种基础数据结构的使用，它是在集合类的抽象数据类型实现中表示数据的合适选择。这是我们构造非 Java 直接支持的数据结构的第一个例子。我们的实现将成为本书中其他更加复杂的数据结构的构造代码的模板。所以请仔细阅读本节，即使你已经使用过链表。

定义。链表是一种递归的数据结构，它或者为空(`null`)，或者是指向一个结点(`node`)的引用，该结点含有一个泛型的元素和一个指向另一条链表的引用。

在这个定义中，结点是一个可能含有任意类型数据的抽象实体，它所包含的指向结点的应用显示了它在构造链表之中的作用。和递归程序一样，递归数据结构的概念一开始也令人费解，但其实它的简洁性赋予了它巨大的价值。

1.3.3.1 结点记录

在面向对象编程中，实现链表并不困难。我们首先用一个嵌套类来定义结点的抽象数据类型：

```

private class Node
{
    Item item;
    Node next;
}

```

一个 `Node` 对象含有两个实例变量，类型分别为 `Item` (参数类型) 和 `Node`。我们会在需要使用 `Node` 类的类中定义它并将它标记为 `private`，因为它不是为用例准备的。和任意数据类型一样，我们通过 `new Node()` 触发 (无参数的) 构造函数来创建一个 `Node` 类型的对象。调用的结果是一个指向 `Node` 对象的引用，它的实例变量均被初始化为 `null`。`Item` 是一个占位符，表示我们希望用链表处理的任意数据类型 (我们将会使用 Java 的泛型使之表示任意引用类型)；`Node` 类型的实例变量显示了这种数据结构的链式本质。为了强调我们在组织数据时只使用了 `Node` 类，我们没有定义任何方法且会在代码中直接引用实例变量：如果 `first` 是一个指向某个 `Node` 对象的变量，我们可以使用 `first.item` 和 `first.node` 访问它的实例变量。这种类型的类有时也被称为记录。它们实现的不是抽象数据类型因为我们会直接使用其实例变量。但是在我们的实现中，`Node` 和它的用例代码都会被封装在相同的类中且无法被该类的用例访问，所以

[142] 我们仍然能够享受数据抽象的好处。

1.3.3.2 构造链表

现在，根据递归定义，我们只需要一个 `Node` 类型的变量就能表示一条链表，只要保证它的值是 `null` 或者指向另一个 `Node` 对象且该对象的 `next` 域指向了另一条链表即可。例如，要构造一条含有元素 `to`、`be` 和 `or` 的链表，我们首先为每个元素创造一个结点：

```
Node first = new Node();
Node second = new Node();
Node third = new Node();
```

并将每个结点的 `item` 域设为所需的值（简单起见，我们假设在这些例子中 `Item` 为 `String`）：

```
first.item = "to";
second.item = "be";
third.item = "or";
```

然后设置 `next` 域来构造链表：

```
first.next = second;
second.next = third;
```

（注意：`third.next` 仍然是 `null`，即对象创建时它被初始化的值。）结果是，`third` 是一条链表（它是一个结点的引用，该结点指向 `null`，即一个空链表），`second` 也是一条链表（它是一个结点的引用，且该结点含有一个指向 `third` 的引用，而 `third` 是一条链表），`first` 也是一条链表（它是一个结点的引用，且该结点含有一个指向 `second` 的引用，而 `second` 是一条链表）。图 1.3.5 所示的代码以不同的顺序完成了这些赋值语句。

链表表示的是一列元素。在我们刚刚考察过的例子中，`first` 表示的序列是 `to`、`be`、`or`。我们也可以用一个数组来表示一列元素。例如，可以用以下数组表示同一列字符串：

```
String[] s = { "to", "be", "or" };
```

不同之处在于，在链表中向序列插入元素或是从

序列中删除元素都更方便。下面，我们来学习完

[143] 成这些任务的代码。

在追踪使用链表和其他链式结构的代码时，我们会使用可视化表示方法：

- 用长方形表示对象；
- 将实例变量的值写在长方形中；
- 用指向被引用对象的箭头表示引用关系。

这种表示方式抓住了链表的关键特性。方便起见，我们用术语链接表示对结点的引用。简单起见，当元素的值为字符串时（如我们的例子所示），我们会将字符串写在长方形之内，而非使用 1.2 节中所讨论的更准确的方式表示字符串对象和字符数组。这种可视化的表示方式使我们能够将注意力集中在链表上。

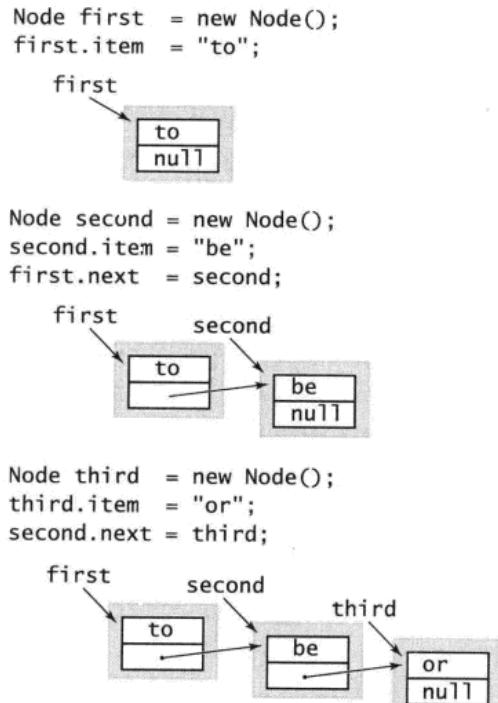


图 1.3.5 用链接构造一条链表

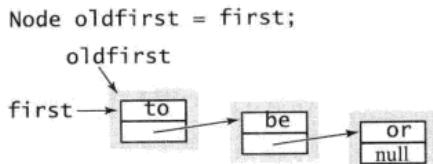
1.3.3.3 在表头插入结点

首先，假设你希望向一条链表中插入一个新的结点。最容易做到这一点的地方就是链表的开头。例如，要在首结点为 `first` 的给定链表开头插入字符串 `not`，我们先将 `first` 保存在 `oldfirst` 中，然后将一个新结点赋予 `first`，并将它的 `item` 域设为 `not`，`next` 域设为 `oldfirst`。以上过程如图 1.3.6 所示。这段在链表开头插入一个结点的代码只需要几行赋值语句，所以它所需的时间和链表的长度无关。

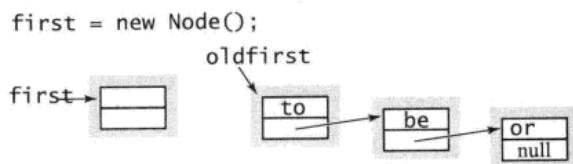
1.3.3.4 从表头删除结点

接下来，假设你希望删除一条链表的首结点。这个操作更简单：只需将 `first` 指向 `first.next` 即可。一般来说你可能会希望在赋值之前得到该元素的值，因为一旦改变了 `first` 的值，就再也无法访问它曾经指向的结点了。曾经的结点对象变成了一个孤儿，Java 的内存管理系统最终将回收它所占用的内存。和以前一样，这个操作只含有一条赋值语句，因此它的运行时间和链表的长度无关。此过程如图 1.3.7 所示。

保存指向链表的链接



创建新的首结点



设置新结点中的实例变量

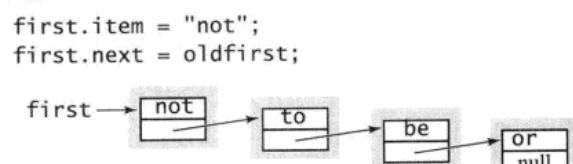


图 1.3.6 在链表的开头插入一个新结点

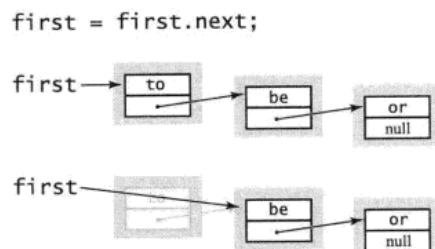
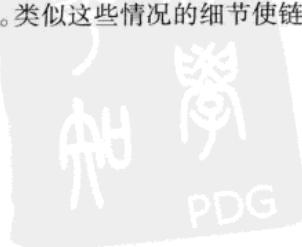


图 1.3.7 删除链表的首结点

144

1.3.3.5 在表尾插入结点

如何才能在链表的尾部添加一个新结点？要完成这个任务，我们需要一个指向链表最后一个结点的链接，因为该结点的链接必须被修改并指向一个含有新元素的新结点。我们不能在链表代码中草率地决定维护一个额外的链接，因为每个修改链表的操作都需要添加检查是否要修改该变量（以及作出相应修改）的代码。例如，我们刚刚讨论过的删除链表首结点的代码就可能改变指向链表的尾结点的引用，因为当链表中只有一个结点时，它既是首结点又是尾结点！另外，这段代码也无法处理链表为空的情况（它会使用空链接）。类似这些情况的细节使链表代码特别难以调试。在链表结尾插入新结点的过程如图 1.3.8 所示。



1.3.3.6 其他位置的插入和删除操作

总的来说，我们已经展示了在链表中如何通过若干指令实现以下操作，其中我们可以通过 `first` 链接访问链表的首结点并通过 `last` 链接访问链表的尾结点：

- 在表头插入结点；
- 从表头删除结点；
- 在表尾插入结点。

其他操作，例如以下几种，就不那么容易实现了：

- 删除指定的结点；
- 在指定结点前插入一个新结点。

例如，我们怎样才能删除链表的尾结点呢？`last` 链接帮不上忙，因为我们需要将链表尾结点的前一个结点中的链接（它指向的正是 `last`）值改为 `null`。在缺少其他信息的情况下，唯一的解决办法就是遍历整条链表并找出指向 `last` 的结点（请见下文以及练习 1.3.19）。这种解决

[145] 方案并不是我们想要的，因为它所需的时间和链表的长度成正比。实现任意插入和删除操作的标准解决方案是使用双向链表，其中每个结点都含有两个链接，分别指向不同的方向。我们将实现这些操作的代码留做练习（请见练习 1.3.31）。我们的所有实现都不需要双向链表。

1.3.3.7 遍历

要访问一个数组中的所有元素，我们会使用如下代码来循环处理 `a[]` 中的所有元素：

```
for (int i = 0; i < N; i++)
{
    // 处理 a[i]
}
```

访问链表中的所有元素也有一个对应的方式：将循环的索引变量 `x` 初始化为链表的首结点，然后通过 `x.item` 访问和 `x` 相关联的元素，并将 `x` 设为 `x.next` 来访问链表中的下一个结点，如此反复直到 `x` 为 `null` 为止（这说明我们已经到达了链表的结尾）。这个过程被称为链表的遍历，可以用以下循环处理链表的每个结点的代码简洁表达，其中 `first` 指向链表的首结点：

```
for (Node x = first; x != null; x = x.next)
{
    // 处理 x.item
}
```

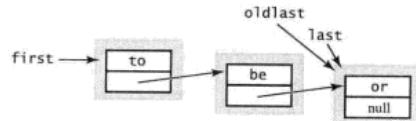
[146] 这种方式和迭代遍历一个数组中的所有元素的标准方式一样自然。在我们的实现中，它是迭代器使用的基本方式，它使用例能够迭代访问链表的所有元素而无需知道链表的实现细节。

1.3.3.8 栈的实现

有了这些预备知识，给出我们的 `Stack API` 的实现就很简单了，如 94 页的算法 1.2 所示。它将栈保存为一条链表，栈的顶部即为表头，实例变量 `first` 指向栈顶。这样，当使用 `push()` 压入一

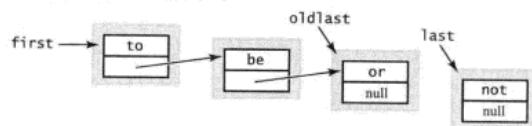
保存指向尾结点的链接

```
Node oldlast = last;
```



创建新的尾结点

```
last = new Node();
last.item = "not";
last.next = null;
```



将尾链接指向新结点

```
oldlast.next = last;
```

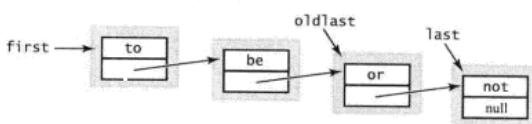
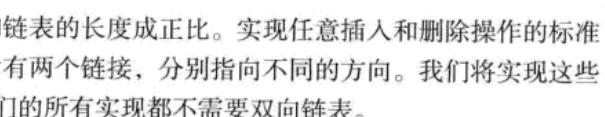


图 1.3.8 在链表的结尾插入一个新结点



个元素时，我们会按照 1.3.3.3 节所讨论的代码将该元素添加在表头；当使用 `pop()` 删除一个元素时，我们会按照 1.3.3.4 节讨论的代码将该元素从表头删除。要实现 `size()` 方法，我们用实例变量 `N` 保存元素的个数，在压入元素时将 `N` 加 1，在弹出元素时将 `N` 减 1。要实现 `isEmpty()` 方法，只需检查 `first` 是否为 `null`（或者可以检查 `N` 是否为 0）。该实现使用了泛型的 `Item`——你可以认为类名后的 `<Item>` 表示的是实现中所出现的所有 `Item` 都会替换为用例所提供的任意数据类型的名称（请见 1.3.2.2 节）。我们暂时省略了关于迭代的代码并将它们留到算法 1.4 中继续讨论。图 1.3.9 显示了我们所常用的测试用例的轨迹（测试用例代码放在了图后面）。链表的使用达到了我们的最优设计目标：

- 它可以处理任意类型的数据；
- 所需的空间总是和集合的大小成正比；
- 操作所需的时间总是和集合的大小无关。

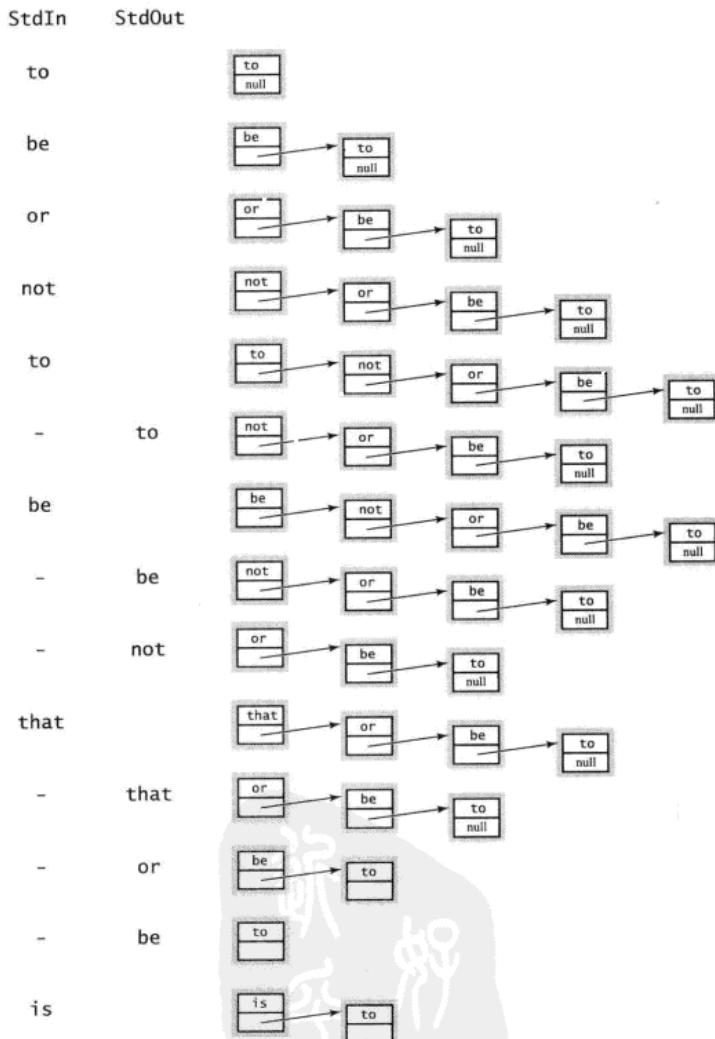


图 1.3.9 stack 的开发用例的轨迹

```

public static void main(String[] args)
    // 创建一个栈并根据StdIn中的指示压入或弹出字符串
    Stack<String> s = new Stack<String>();
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (item.equals("-"))
            s.push(item);
        else if (s.isEmpty()) StdOut.print(s.pop() + " ");
    }
    StdOut.println("(" + s.size() + " left on stack)");
}

```

Stack的测试用例

这份实现是我们对许多算法的实现的原型。它定义了链表数据结构并实现了供用例使用的方法 `push()` 和 `pop()`，仅用了少量代码就取得了所期望的效果。算法和数据结构是相辅相成的。在本例中，算法的实现代码很简单，但数据结构的性质却并不简单，我们用了好几页纸来说明这些性质。这种数据结构的定义和算法的实现的相互作用很常见，也是本书中我们对抽象数据类型的实现重点。

147
148

算法 1.2 下压堆栈（链表实现）

```

public class Stack<Item> implements Iterable<Item>
{
    private Node first; // 栈顶（最近添加的元素）
    private int N;      // 元素数量
    private class Node
    { // 定义了结点的嵌套类
        Item item;
        Node next;
    }
    public boolean isEmpty() { return first == null; } // 或: N == 0
    public int size() { return N; }
    public void push(Item item)
    { // 向栈顶添加元素
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
        N++;
    }
    public Item pop()
    { // 从栈顶删除元素
        Item item = first.item;
        first = first.next;
        N--;
        return item;
    }
    // iterator() 的实现请见算法 1.4
    // 测试用例 main() 的实现请见本节前面部分
}

```

这份泛型的 `Stack` 实现的基础是链表数据结构。它可以用于创建任意数据类型的栈。要支持迭代，请添加算法 1.4 中为 `Bag` 数据类型给出的加粗部分的代码。

149

```

% more tobe.txt
to be or not to - be -- that -- is
% java Stack < tobe.txt
to be not that or be (2 left on stack)

```



1.3.3.9 队列的实现

基于链表数据结构实现 Queue API 也很简单，如算法 1.3 所示。它将队列表示为一条从最早插入的元素到最近插入的元素的链表，实例变量 `first` 指向队列的开头，实例变量 `last` 指向队列的结尾。这样，要将一个元素入列（`enqueue()`），我们就将它添加到表尾（请见图 1.3.8 中讨论的代码，但是在链表为空时需要将 `first` 和 `last` 都指向新结点）；要将一个元素出列（`dequeue()`），我们就删除表头的结点（代码和 Stack 的 `pop()` 方法相同，只是当链表为空时需要更新 `last` 的值）。`size()` 和 `isEmpty()` 方法的实现和 Stack 相同。和 Stack 一样，Queue 的实现也使用了泛型参数 `Item`。这里我们省略了支持迭代的代码并将它们留到算法 1.4 中继续讨论。下面所示的是一个开发用例，它和我们在 Stack 中使用的用例很相似，它的轨迹如算法 1.3 所示。Queue 的实现使用的数据结构和 Stack 相同——链表，但它实现了不同的添加和删除元素的算法，这也是用例所看到的后进先出和先进后出的区别所在。和刚才一样，我们用链表达到了最优设计目标：它可以处理任意类型的数据，所需的空间总是和集合的大小成正比，操作所需的时间总是和集合的大小无关。^①

```
public static void main(String[] args)
{ // 创建一个队列并操作字符串入列或出列
    Queue<String> q = new Queue<String>();
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (!item.equals("-"))
            q.enqueue(item);
        else if (!q.isEmpty()) StdOut.print(q.dequeue() + " ");
    }
    StdOut.println("(" + q.size() + " left on queue)");
}
```

Queue 的测试用例

```
% more tobe.txt
to be or not to - be -- that --- is
% java Queue < tobe.txt
to be or not to be (2 left on queue)
```

150

算法 1.3 先进先出队列

```
public class Queue<Item> implements Iterable<Item>
{
    private Node first; // 指向最早添加的结点的链接
    private Node last; // 指向最近添加的结点的链接
    private int N; // 队列中的元素数量
    private class Node
    { // 定义了结点的嵌套类

```

^① 这里原书应该是因为版面原因没有使用列表，如果版面允许可以使用和 Stack 部分相同的列表显示这三个目标。

——译者注