翻译—伤寒明理论 包含全部章节

Chapter 1
Introducing Karel the Robot
In the 1970s, a Stanford graduate student named Rich Pattis decided that it would be easier to teach the fundamentals of programming if students could somehow learn the basic ideas in a simple environment free from the complexities that characterize most programming languages. Drawing inspiration from the success of Seymour Papert's LOGO project at MIT, Rich designed an introductory programming environment in which students teach a robot to solve simple problems. That robot was named Karel, after the Czech playwright KarelCapek, whose 1923 play R.U.R. (Rossum's Universal Robots) gave the word robot to the English language.

第一章：

机器人卡雷尔简介

在二十世纪七十年代，一位名字叫 Rich Pattis 的斯坦福研究生觉得，在编程基础的教学中，如果学生可以在某种简单的环境中，摆脱大多数编程语言复杂的特性，学习基本的编程思想，可以取得更好的效果。麻省理工 Seymour Papert's LOGO 计划的成功，启发了灵感，Rich 设计了一个入门编程环境，（这个编程环境)让学生教一个机器人来解决简单的问题。这个机器人名字叫卡雷尔。因为捷克剧作家 KarelCapek 在 1923 年公演了 R.U.R (Rossum's Universal Robots)后,为英语带来了机器人这个英语单词--Robot 。

Karel the Robot was quite a success. Karel was used in introductory computer science courses all across the country, to the point that Rich's textbook sold well over 100,000 copies. Many generations of CS106A students learned how programming works by putting Karel through its paces. But nothing lasts forever. In the middle of the 1990s, the simulator we had been using for Karel the Robot stopped working. We were, however, soon able to get a version of Karel up and running in the Thetis interpreter we were using at the time. But then, a year ago, CS106A switched to Java, and Karel again vanished from the scene. For the last three quarters, the hole in the curriculum left by Karel's departure has been competently filled by Nick Parlante's Binky world, but it seems about time to bring Karel back. The new implementation of Karel is designed to be compatible with both Java and the Eclipse programming environment, which means that you'll get to practice using the Eclipse editor and debugger from the very beginning of the course.

机器人卡雷尔相当成功。 卡雷尔被用于全国的计算机科学入门课程，到了 Rich 的教科书畅销超过 10 万份的地步。许多学习 CS106A 的学生，通过设计卡雷尔的行为，学会了如何让程序工作。在 20 世纪 90 年代中期，我们曾经使用的机器人卡雷尔模拟器停止工作了。但是，我们很快就得到了一个 Thetis 编译的卡雷尔升级版供那时使用。但是，一年以前，CS106A 课程转向到 Java，卡雷尔再次从课堂上消失了。虽然在过去的三个季度，由于卡雷尔的离去产生的空白，已经完全被 Nick Parlante 的 Binky world 填补了。但现在是带卡雷尔回来的时候了。新完工的卡雷尔设计得完全兼容 Java 和 Eclipse 编程环境，这就意味着，你将在这门课程的开始，就可以练习使用 Eclipse 的编辑器和调试器。

What is Karel?

Karel is a very simple robot living in a very simple world. By giving Karel a set of commands, you can direct it to perform certain tasks within its world. The process of specifying those commands is called programming. Initially, Karel understands only a very small number of predefined commands, but an

important part of the programming process is teaching Karel new commands that extend its capabilities.

卡雷尔是啥？

卡雷尔是一个生活在非常简单的世界中的非常简单的机器人。在它的世界中，你可以通过给卡雷尔一组命令，直接让卡雷尔执行某些任务。指定这些命令的过程称为编程。最初，卡雷尔只明白极少数预定义的命令，但编程（学习）过程的一个重要内容，就是教卡雷尔可以扩展它的能力的新命令。

When you program Karel to perform a task, you must write out the necessary commands in a very precise way so that the robot can correctly interpret what you have told it to do. In particular, the programs you write must obey a set of syntactic rules that define what commands and language forms are legal. Taken together, the predefined commands and syntactic rules define the Karel programming language. The Karel programming language is designed to be as similar as possible to Java so as to ease the transition to the language you will be using all quarter. Karel programs have much the same structure and involve the same fundamental elements as Java programs do. The critical difference is that Karel's programming language is extremely small, in the sense that it has very few commands and rules. It is easy, for example, to teach the entire Karel language in just a couple of hours, which is precisely what we do in CS106A. At the end of that time, you will know everything that Karel can do and how to specify those actions in a program. The details are easy to master. Even so, you will discover that solving a problem can be extremely challenging. Problem solving is the essence of programming; the rules are just a minor concern along the way.

当你谋划让卡雷尔执行某项任务的时候，你必需用非常精确的方式写出这些必需的命令，以便这个机器人能够正确的理解你交待它做的事情。（另外）特别（注意的）是，你写的程序必须遵守语法规则，它规定了什么样的命令和语言形式是合法的。二者合在一起，预定义的命令和语法规则（一起）定义了卡雷尔编程语言。卡雷尔编程语言被设计的尽可能类似于 Java 语言，这样便于顺利过渡到（Java）这门你将时刻使用的语言上。卡雷尔程序具有和 Java 程序相同的结构，也涉及到相同的基本元素。最关键的区别是，卡雷尔的编程语言非常的小，从这个意义上讲，它只具有非常少的命令和规则。它非常容易，例如，教授卡雷尔语言只需要几个小时，这也正是我们在 CS106A 中做的。在（课程）结束的时候，你将知道卡雷尔能做的一切事情，以及如何在一个程序中实现它。这些细节是容易掌握的。即便如此，你会发现，（需要）解决的问题可能是极其具挑战性的。解决问题是编程的本质；在这一学习过程中，对规则的关注是次要的。

In sophisticated languages like Java, there are so many details that learning these details often becomes the focus of the course. When that happens, the much more critical issues of problem solving tend to get lost in the shuffle. By starting with Karel, you can concentrate on solving problems from the very beginning. And because Karel encourages imagination and creativity, you can have quite a lot of fun along the way.

在复杂的语言里，如 Java，有许多的细节，这些细节往往成为学习的课程的重点。当这种情况发生时，对解决问题的更关键的东西，往往会在得到一片混乱中失去。通过从卡雷尔入手开始学习，你可以在一开始的时候，就把精力集中在解决问题上面。而且，卡雷尔的学习鼓励想象力和创造力，在学习过程中，你会收获不少乐趣。（这段话的意思是，要把精力集中在比语法细节更重要的--解决问题的能力上。译者注）
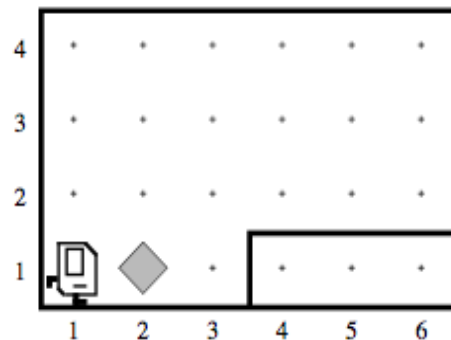
Karel's world
Karel's world is defined by streets running horizontally (east-west) and avenues running vertically (north-south). The intersection of a street and an avenue is called a corner. Karel can only be positioned on

corners and must be facing one of the four standard compass directions (north, south, east, west). A sample Karel world is shown below. Here Karel is located at the corner of 1st Street and 1st Avenue, facing east.

卡雷尔的世界

卡雷尔的世界被定义为水平的街（东西方向），垂直的道（南北方向)。街和道的交点被称为街角。卡雷尔只能定位在街角上，而且只能面对的四个标准罗盘方向（北，南，东，西）。一个简单的卡雷尔世界显示如下。卡雷尔目前位于第一大街和第一大道相交的街角，面朝向东。

Several other components of Karel's world can be seen in this example. The object in front of Karel is a beeper. As described in Rich Pattis's book, beepers are "plastic cones which emit a quiet beeping noise." Karel can only detect a beeper if it is on the same corner. The solid lines in the diagram are walls. Walls serve as barriers within Karel's world. Karel cannot walk through walls and must instead go around them. Karel's world is always bounded by walls along the edges, but the world may have dfferent dimensions depending on the specific problem Karel needs to solve.

在这个例子中，卡雷尔世界的几个其它组件也可以看到。卡雷尔前面的物体是个蜂鸣器，在 Rich Pattis's 的书中描述，蜂鸣器是一个发出轻微的哔哔声的塑料筒。只有当卡雷尔和蜂鸣器位于同一个街角上的时候，卡雷尔才能感知这个蜂鸣器。图中的实线是墙壁。墙是卡雷尔世界的屏障。卡雷尔不能穿过墙壁，而只能在墙的周边行走。卡雷尔的世界总是被作为边界的墙包围起来，但是，随着卡雷尔需要解决不同的具体问题，卡雷尔的世界也有不同的尺寸。

What can Karel do?
When Karel is shipped from the factory, it responds to a very small set of commands:
move() Asks Karel to move forward one block. Karel cannot respond to a
move() command if there is a wall blocking its way.
turnLeft() Asks Karel to rotate 90 degrees to the left (counterclockwise).
pickBeeper() Asks Karel to pick up one beeper from a corner and stores the beeper in its beeper bag, which can hold an infinite number of beepers. Karel cannot respond to a pickBeeper() command unless there is a beeper
on the current corner.
putBeeper() Asks Karel to take a beeper from its beeper bag and put it down on the current corner. Karel cannot respond to a putBeeper() command
unless there are beepers in its beeper bag.

卡雷尔能干点啥？

当卡雷尔出厂的时候，它只能响应非常小的命令集：

move()　　要求卡雷尔向前推进一步。当一堵墙挡在卡雷尔面前的时候，卡雷尔不能响应 move() 这个命令。

turnLeft()　要求卡雷尔向左转 90 度（逆时针转动）。

pickBeeper()　要求卡雷尔捡起街角上的蜂鸣器，把这个蜂鸣器放到它的蜂鸣器收藏包里，这个包可容纳无限多的蜂鸣器。除非这个蜂鸣器恰好在卡雷尔所在的街角上，卡雷尔不能响应这个 pickBeeper()命令。

putBeeper()　要求卡雷尔从蜂鸣器收藏包里拿出一个蜂鸣器，放在卡雷尔所在的街角上。除非卡雷尔的蜂鸣器收藏包里有蜂鸣器，卡雷尔不能响应这个 putBeeper()命令。

The empty pair of parentheses that appears in each of these commands is part of the common syntax shared by Karel and Java and is used to specify the invocation of the command. Eventually, the programs you write will include additional information in the space between the parentheses, but such information is not part of the Karel's primitive world. These parentheses will therefore be empty in standard Karel programs, but you must remember to include them nonetheless.

出现在这些命令后面的一对空括号，是卡雷尔和 Java 相同语法的一部分，通过这个（一对空括号），来指定命令的调用。最终的时候，在你写的（Java）程序里，空括号之间会包含一些额外的内容,但是，这些额外的内容并不是卡雷尔原始世界的一部分。将这些括号之间留空，才是标准的卡雷尔程序，虽然括号里什么也没有，但你一定要记着把它们写在命令里。

It is also important to recognize that several of these commands place restrictions on Karel's activities. If Karel tries to do something illegal, such as moving through a wall or picking up a nonexistent beeper, an error condition occurs. At this point, Karel displays an error message and does not execute any remaining commands.

需要特别注意的是，这几个命令对卡雷尔的行为进行了限制。如果卡雷尔试图做些非法的举动，像穿墙或者捡起一个不存在的蜂鸣器，一个错误就发生了。在这时，卡雷尔会显示一条错误的信息，同时拒绝执行剩余的命令。

Karel's commands, however, cannot be executed on their own. Before Karel can respond to any of these commands, you need to incorporate them into a Karel program.You will have a chance to see a few simple Karel programs in Chapter 2, but before doing so, it is useful to make a few general remarks about the programming philosophy that underlies this particular implementation of the Karel programming language.

卡雷尔的这些命令，不能自己自动执行。在卡雷尔可以执行这些命令之前，你需要先把它们写在一个卡雷尔程序里。在第二章，你将有机会看到一些简单的卡雷尔程序。但是，在运行这些程序之前，作一些执行卡雷尔程序语言必要的基本编程哲学的提醒，还是非常有用的。

Karel and the object-oriented paradigm

When Karel was introduced in the 1970s, the prevailing approach to writing computer programs was the procedural paradigm. To a large extent, procedural programming is the process of decomposing a large programming problem into smaller, more manageable units called procedures that define the necessary

operations. Although the strategy of breaking programs down into smaller units remains a vital part of any style of programming, modern languages like Java emphasize a different approach called the object-oriented paradigm. In object-oriented programming, the programmer's attention shifts away from the procedural specification of operations and focuses instead on modeling the behavior of conceptually integrated units called objects. Objects in a programming language sometimes correspond to physical objects in the real world, but just as often represent more abstract concepts. The central feature of any object—real or abstract—is that it must make sense as a unified whole.

卡雷尔和面向对象模式

在卡雷尔被引入的二十世纪七十年代，采用的计算机编程方法是过程模式。在很大程度上，过程编程是把一个大规模的程序问题分解为小的过程，这些更易于管理的，定义了必要操作的单元称为过程。这种把程序分割成小单元的谋划，是编程风格的重要组成部分，而现代语言例如 Java,强调了不同的编程模式，叫面向对象模式。在面向对象模式下，程序员的注意力被从关注特定操作的过程单元中转移出来，侧重到：为被称为对象的单元，抽象建模的行为上。在编程语言中，"对象"有时和真实世界的物理对象相对应，但更多的时候是抽象的概念。是真实对象的核心特性或代表（事物）整体的一种抽象。

One of the primary advantages of the object-oriented paradigm is that it encourages programmers to recognize the fundamental relationship between the state of an object and its behavior. The state of an object consists of a set of attributes that pertain to that object and might change over time. For example, an object might be characterized by its location in space, its color, its name, and a host of other properties. The behavior of an object refers to the ways in which that object responds to events in its world or commands from other objects. In the language of object-oriented programming, the generic word for anything that triggers a particular behavior in an object is called a message (although it generally seems clearer to use the word command in the context of
Karel). The response to a message typically involves changing the state of an object. For example, if one of the properties defining the state of an object is its color, then it would presumably respond to a setColor(BLUE) message by changing its color to blue.

面向对象模式的主要优势之一是，它鼓励程序员认识到一个对象的状态和其行为的基本关系。一个对象的状态，涉及到一组和该对象相关，并可能随时间而改变的特性。一个对象可以被它在空间的位置，它的颜色，它的名字，其它一些主要特性来特别指定出来。一个对象的行为是指它在它的世界中响应事件或响应来自其它对象的命令的方法。在面向对象编程的语言中，触发指定对象的行为的通用词叫消息--message（虽然从卡雷尔的上下文上看，命令--command 这个词可能更让人明白点）。对一个消息的响应，通常需要改变一个对象的状态。例如，如果定义一个对象状态的特性的之一是它的颜色，它就可以通过响应 setColor(BLUE) 这条消息把它的颜色改变成蓝色。

In many ways, Karel represents an ideal environment for illustrating the object- oriented approach. Although no one has actually built a mechanical implementation of Karel, it is nonetheless easy to imagine Karel as a real-world object. Karel is, after all, a robot, and robots are real-world entities. The properties that define Karel's state are its location in the world, the direction it is facing, and the number of beepers in its beeper
bag. Karel's behavior is defined by the commands to which it responds:move(), turnLeft(), pickBeeper(), and putBeeper(). The move() command changes Karel's location, turnLeft() changes its direction, and the remaining two affect both the number of beepers in Karel's bag and the number of beepers on the current corner.

在许多方面，卡雷尔代表了讲述"面向对象方法"的理想环境。虽然实际上没有人建立一个机器来

执行卡雷尔，但却非常容易把卡雷尔想象为一个真实世界的对象。卡雷尔毕竟是一个机器人，机器人是真实世界存在的实体。定义卡雷尔状态的特性是它在它的世界中的位置，它面对的方向，在它蜂鸣器收藏包里的蜂鸣器数目。定义卡雷尔行为的是它响应的 move(), turnLeft(), pickBeeper(), 和 putBeeper()这些命令。move()命令可以改变卡雷尔的位置，turnLeft()命令可以改变它面朝的方向，剩下的两个命令（指 pickBeeper()和 putBeeper()这两个命令--译者注），会同时影响到收藏包里的蜂鸣器数量和当前街角上的蜂鸣器数量。

The Karel environment also provides a useful framework for defining one of the central concepts of object-oriented programming. In both Karel and Java, it is essential to differentiate the notion of an object from that of a class. The easiest way to understand the distinction is to think about a class as a pattern or template for objects that share a common behavior and collection of state attributes. As you will see in the next chapter, the word Karel in a Karel program represents the entire class of robots that know how to respond to the move(), turnLeft(), pickBeeper(), and putBeeper() commands. Whenever you have an actual robot in the world, that robot is an object that represents a specific instance of the Karel class. Although you won't have occasion to do so in CS 106A, it is possible to have more than one instance of the Karel class running in the same world. Even when there is only a single robot, however, it is important to remember that object and class are different concepts and to keep those ideas straight in your mind.

卡雷尔的环境还提供了面向对象编程的核心概念之一--一个有用的框架。无论卡雷尔还是 Java,一个对象和一个类之间概念的区分是很有必要的。最容易理解这种分别的方法是把类认为是一些对象（共同）的模式或模板，这些对象都有一个共同的行为和状态属性的集合。在你将看到的下一章中，在卡雷尔程序里出现的 "karel"这个词，代表一个完整的机器人的类，这个类知道怎么去响应 move(), turnLeft(), pickBeeper(), 和 putBeeper() 这些命令。但只要你在卡雷尔的世界里有了一个实际的机器人，那么，这个机器人就是一个对象，这个机器人它代表了卡雷尔这个类的特定实例。尽管在 CS106A 课程里，你将不会有机会去做这样一个类，但有一个以上的卡雷尔类的实例，运行在同一个世界里，还是有可能的。即使只有一个单独机器人的时候，在你的脑海里，深刻记得类和对象是不同的概念，还是很重要的。

The importance of practical experience
Programming is very much a learn-by-doing activity. As you will continually discover in your study of computer science, reading about some programming concept is not the same thing as using that concept in a program. Things that seem very clear on the page can be difficult to put into practice.

实际经验的重要性

编程是一项需要大量边学边干的活动。就像你将在你的计算机学习中不断发现的那样，读懂一些编程概念和能在一个程序中使用这个概念不是同一回事。在纸面上看起来似乎很清楚的东西，具体实施起来就变得非常的难。

Given the fact that writing programs on your own and getting them to run on the computer are essential to learning about programming, it may seem surprising to discover that this book does not include much discussion of the hands-on aspects of using Karel on your computer. The reason for that omission is that the steps you need to run a Karel program depend on the environment you're using. Running Karel programs on a Macintosh is somewhat different from running it under Windows. Even though the programming environment you use has a great deal of influence on the nitty-gritty details you need to run programs, it has no influence whatsoever on the general concepts. This book describes the general concepts; the details pertinent to each platform will be distributed as handouts during the course.

在编程学习中，你写出程序和让这些程序能在计算机上运行是同样重要的。你会惊讶的发现，这

本书没有包括多少手把手操作卡雷尔如何在你的计算机运行的讨论。遗漏这些部分的原因是，你运行卡雷尔需要依靠你使用的计算机环境。在苹果机上运行卡雷尔程序和在 Windows 下是不一样的。即使你使用的编程环境对你运行程序的基本细节有很大的影响，但他对常规的（编程）概念没有任何影响。这本书介绍常规概念；（而）每个平台的相关细节，被包含在分发的课程讲义中。

The fact that this book omits the practical details, however, should in no sense be interpreted as minimizing their importance. If you want to understand how programming works—even in an environment as simple as that provided by Karel—it is essential to "get your hands dirty" and start using the computer. Doing so is by far the most effective introduction into the world of programming and the excitement that it holds.

实际上，这本书省略了些实际的细节，但无意减少这些细节的重要性。即使你要弄明白像卡雷尔提供的，这么简单的编程环境下，如何开始编程工作，你也必需"弄脏你的手"--开始实际的计算机操作。
这样做是进入编程世界，享受它带来的激情，目前最有效的方式。

Chapter 2

Programming Karel

<div align="center">

第二章

卡雷尔程序设计

</div>

In its new object-oriented implementation, the simplest style of Karel program consists of a definition of a new Karel class that specifies a sequence of built-in commands that should be executed when the program is run. A very simple Karel program is shown in Figure 1.

在卡雷尔新的面向对象的实现中（这个新的，应该是和七十年代的卡雷尔实现程序相对而言--译者注），最简单的卡雷尔程序包含一个定义的 karel 类，这个类指定了一系列在程序运行的时候，可以被执行的内建命令。卡雷尔一个非常简单的程序如 Figure 1 所示。

Figure 1. Simple Karel example to pick up a single beeper

```
/*
 * File: BeeperPickingKarel.java
 * ----------------------------
 * The BeeperPickingKarel class extends the basic Karel class
 * by defining a "run" method with three commands.  These
 * commands cause Karel to move forward one block, pick up
 * a beeper, and then move ahead to the next corner.
 */
import stanford.karel.*;
public class BeeperPickingKarel extends Karel {
  public void run() {
    move();
    pickBeeper();
    move();
  }
}
```

The program in Figure 1 is composed of several parts. The first part consists of the following lines:

Figure 1 中的程序是由几部分组成的。第一部分由下面的几行组成。

```
/*
 * File: BeeperPickingKarel.java
 * ----------------------------
 * The BeeperPickingKarel class extends the basic Karel class
 * by defining a "run" method with three commands.  These
 * commands cause Karel to move forward one block, pick up
 * a beeper, and then move ahead to the next corner.
 */
```

These lines are an example of a comment, which is simply text designed to explain the operation of the program to human readers. Comments in both Karel and Java begin with the characters /* and end with the characters */. Here, the comment begins on the first line and ends several lines later. The stars on the individual lines that make up the text of the comment are not required, but make it easier for human readers to see the extent of the comment. In a simple program, extensive comments may seem silly because the effect of the program is obvious, but they are extremely important as a means of documenting the design of larger, more complex programs.

这些行是一个注释的例子，注释是指适合人类读者的，用来解释程序操作的，简单的文本。在卡雷尔和 Java 中，注释都是以组合字符 /* 开头，以组合字符 */结尾。在这里，这段注释从第一行开始，在几行之后结束。对注释中各行文本开头的"*",不是必需的，但标记星号可以让人类读者看清注释所占的范围。在一个简单的程序里，广泛的注释似乎是愚蠢的，因为程序的效果是显而易见的，但是，在设计更大，更复杂的程序时，注释是一种非常有意义的记录。

The second part of the program is the line

程序的第二部分是这一行：

import stanford.karel.*;

This line requests the inclusion of all definitions from the stanford.karel library.This library contains the basic definitions necessary for writing Karel programs, such as the definitions of the standard operations move() and pickBeeper(). Because you always
need access to these operations, every Karel program you write will include this import command before you write the actual program.

这一行要求从 stanford.karel 库中调用所有的定义。这个库里包含了编写卡雷尔程序必要的基本定义。例如定义了像 move()和 pickBeeper()这样的标准操作。因为你总是需要访问这些操作，所以，每个你写的卡雷尔程序，在实际的程序语句之前，都要包含这条导入命令。

The final part of the Karel program consists of the following class definition:

这个卡雷尔程序的最后部分包含了下面这个类的定义：

```
public class BeeperPickingKarel extends Karel {
  public void run() {
    move();
    pickBeeper();
    move();
  }
}
```

To understand this definition, it is useful to look more carefully at its structure. The definition of the BeeperPickingKarel class consists of the line beginning with public class and encompasses everything between the curly brace at the end of that line and the corresponding closing brace on the last line of the program. The single line that introduces the new class is called the header of the definition; the code between the braces is called the body.

多仔细观察一下它的结构，对理解这个定义是非常有帮助的。这个 "BeeperPickingKarel" 类 首先包含了以 "public class" 开头的一行，在这一行的结尾，用一个左大括号，和这个类定义最后一行的最后的那个右大括号一起，把中间的部分全部括了进去。声明新类的那一行，被称为（这个类）定义的头部；两个大括号之间的程序代码，被称为（这个类）的主体。

In programming, it is often very useful to think about a particular definition and its body as separable ideas. In this example, the definition of BeeperPickingKarel has the following form, where the entire body of the definition has been replaced by a box that you can put out of your mind for the moment:

在编程中，把特定的定义和它的主体分开考虑是非常有用的，在下图这个例子，BeeperPickingKarel 类定义是如下形式：主体部分被一个方块代替，现在你可以把你的想法放进去：

```
public class BeeperPickingKarel extends Karel {

        body of the class definition

}
```

The header line at the top tells you quite a bit about the BeeperPickingKarel class, even before you have looked to see what the body contains. The key new concept in the class header is embodied in the word extends, which is used in both Karel and Java to indicate that a new class is an extension of an existing one. Here, the class header line indicates
that BeeperPickingKarel is an extension of the standard Karel class imported from the stanford.karel library.

在顶端的头部这一行，在你看到主体里面的内容之前，就会告诉你关于 BeeperPickingKarel 这个类相当多的信息。"extends"这个词，体现出了这个类头部关键的新东西。（"extends"这个词）在卡雷尔和 Java 中，都用来表明 这个新类是一个现存类的扩展。在 BeeperPickingKarel 类的头部这一行表明，它是从 stanford.karel 库中导入的标准的 karel 类的一个扩展。

In object-oriented languages, defining a new class by extension means that the new class (here, BeeperPickingKarel) builds on the facilities provided by the existing class (in this case, Karel). In particular, the fact that it extends Karel guarantees that the new BeeperPickingKarel class will have the following properties：

在面向对象编程语言中，通过扩展来定义一个新类（在这里，就是 BeeperPickingKarel）意味着新类是建立在现有的类（在这个例子里，是 karel 类）提供的功能之上。特别是，在 karel 类之上进行扩展的事实，保证了新的 BeeperPickingKarel 类 将具有下列属性：

1. Any instance of the class BeeperPickingKarel is also an instance of the class Karel.Any instance of the class Karel represents a robot that lives in a world of streets, avenues, beepers, and walls whose state consists of its location, direction, and the number of beepers in its bag. Because BeeperPickingKarel is an extension of Karel, you know that an instance of BeeperPickingKarel will also be a robot that lives in the same type of world and has the same state properties.

1.任何 BeeperPickingKarel 类的实例也是 karel 类的实例。任何 karel 类的实例都被表示成一个机器人，这个机器人生活在一个有街，道，蜂鸣器和墙的世界，它的状态是它的位置，它面朝的方向，它蜂鸣器收藏包里的蜂鸣器数目。因为 BeeperPickingKarel 类是 karel 类的扩展，你也就知道了，

BeeperPickingKarel 类的实例也将是一个机器人，它生活在同样类型世界里，并具有相同的状态属性。

2. Any instance of the BeeperPickingKarel class will automatically respond to the same commands as an instance of the Karel class. Because every robot in the Karel class knows how to respond to the commands move(), turnLeft(), pickBeeper(), and putBeeper(), it follows that a instance of BeeperPickingKarel will understand that same set of commands.

2.任何 BeeperPickingKarel 类的实例将会自动的响应与任何 karel 类的实例相同的命令，因为 每一个 karel 类的机器人都知道如何响应 move(), turnLeft(), pickBeeper(), 和 putBeeper()这些命令，作为它的跟随者，BeeperPickingKarel 类的实例也能明白这同一套命令。

In other words, the new BeeperPickingKarel class automatically acquires the state attributes and the behavior of the Karel class from which it is derived. The process of taking on the structure and behavior of the parent class is called inheritance.

在其它世界里，新的 BeeperPickingKarel 类自动获得 karel 类派生出的状态属性和行为。这个从它的父类取得结构和行为的过程被称为继承。

When a class is defined by extension, the new class is said to be a subclass of the original. In this example, BeeperPickingKarel is therefore a subclass of Karel. Symmetrically, Karel is said to be a superclass of BeeperPickingKarel. Unfortunately, this terminology can be confusing for new programmers, who are likely to make the intuitive inference that a subclass is somehow less powerful that its superclass when in fact the opposite is true. A subclass inherits the behavior of its superclass and can therefore respond to the entire set of commands available to that superclass. A subclass, however, usually defines additional commands that are unavailable to the superclass. Thus, the typical subclass actually has more functionality than the class from which it was derived. This idea is expressed much more clearly by the notion of extension: a subclass extends its superclass and can therefore add new capabilities to it.

当一个类是由扩展而来的，这个新类被称为原始类的子类。在这个例子中，BeeperPickingKarel 因此被称为 karel 类的一个子类。相对的，karel 类被称为 BeeperPickingKarel 类的超级类。
不幸的是，这个称呼可能会引起新程序员的混乱，谁都会直观的认为，一个子类的能力要比它的超级类小，而事实恰恰相反。一个子类继承了超级类的行为，因此可以响应超级类的整套命令。但是，一个子类通常会定义一些额外的命令，这些额外的命令超级类不能执行。因此，一个典型的子类实际上拥有比派生它的原始类更多的功能。这个认识被扩展这个概念表达的更清楚：一个子类扩展了它的超级类，子类因此获得新的能力。

Now that you have some idea about what class extension means, it now makes sense to look at the body of the BeeperPickingKarel class. That body consists of the following lines:

现在你对什么是类扩展的意思有了一些概念，它使 BeeperPickingKarel 类的本体看起来产生了一些意义。这个本体包含下面的这些行：

```
public void run() {
    move();
    pickBeeper();
    move();
}
```

These lines represent the definition of a new method, which specifies the sequence of steps necessary to

respond to a command. As in the case of the BeeperPickingKarel class itself, the method definition consists of two parts that can be considered separately: The first line constitutes the method header and the code between the curly braces is the method body. If you ignore the body for now, the method definition looks like this:

这些行代表了一个新动作的定义，其中阐明了响应一个（新动作）命令必要的步骤序列。在 BeeperPickingKarel 类本身的实例里，这个动作的定义由可以被分开考虑的两部分组成：第一行构成了这个动作的头部，两个括号之间的部分构成了这个动作的本体。如果你现在先忽略本体，这个动作定义看起来是这样的：

```
public void run() {

          body of the method definition

}
```

The first two words in the method header, public and void, are part of Java's syntactic structure, and you should pretty much feel free to ignore them at this point. The next word on the header line specifies the name of the new method, which in this case is the method run. Defining a method means that the new Karel subclass can respond to a new command with that name. The built-in Karel class responds to the commands move(), turnLeft(), pickBeeper(), and putBeeper(); a BeeperPickingKarel responds to that same set of commands plus a new command called run. The run command plays a special role in a Karel program. When you start a Karel program in the Eclipse environment, it creates a new Karel instance of the appropriate subclass, adds that Karel to a world that you specify, and then issues the run command. The effect of issuing that command is defined by the body of the run method, which is a sequence of commands that the robot will execute in order. For example, the body of the run method for the BeeperPickingKarel class is

在这个动作头部的前两个单词，"public"和"void"是 Java 语法结构的一部分，在这里你差不多可以忽视它们。下一个单词指明了这个动作的名字，在这个例子里，这个动作的名字是"run",定义一个新动作，意味着这个新的 karel 类的子类能响应以这个动作命名的新命令。卡雷尔类内建可以响应 move(), turnLeft(), pickBeeper(), 和 putBeeper()这些命令；一个 BeeperPickingKarel 类可以响应这套命令加上这个叫"run"的命令。这个"run"命令在卡雷尔程序中扮演了一个特殊的角色。当你在 Eclipse 环境中打开一个这样的卡雷尔程序，它创造出一个相应的子类的卡雷尔实例，添加了一个你指定的卡雷尔机器人到卡雷尔的世界，它带着你编写的 "run"命令。这个动作的效果是由"run"这个动作的本体来定义的。这个本体是一组命令的序列，机器人将按顺序执行。例如，为 BeeperPickingKarel 类 （编写的）"run" 动作的本体是这样的：

move();
pickBeeper();
move();

Thus, if the initial state of the world matches the example given in Chapter 1, Karel first moves forward into the corner containing the beeper, picks up that beeper, and finally moves forward to the corner just before the wall, as shown in the following before-and- after diagram:

因此，如果一个实例的状态和第一章给出的例子一样，卡雷尔首先向前移动到包含一个蜂鸣器的街角，然后捡起这个蜂鸣器，最后向前移动到墙前面的那个街角。状态像下面前和后（两幅）插图显示的那样：
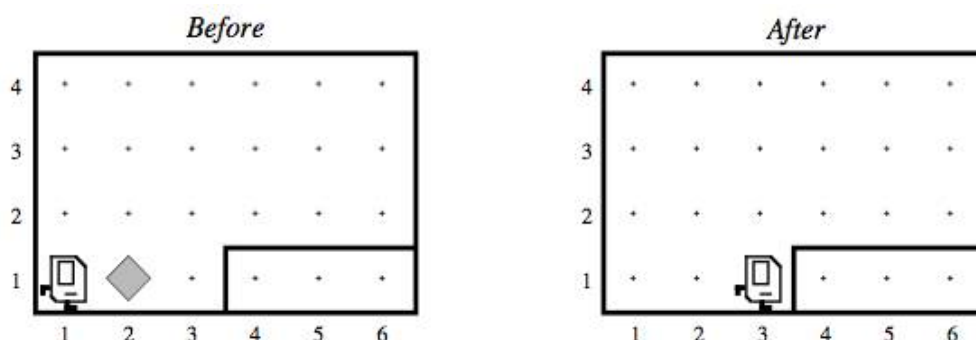
Solving a more interesting problem

The BeeperPickingKarel class defined in Figure 1 doesn't do very much as yet. Let's try to make it a little more interesting. Suppose that the goal is not simply to get Karel to pick up the beeper but to move the beeper from its initial position on 2nd Avenue and 1st Street to the center of the ledge at 5th Avenue and 2nd Street. Thus, your next assignment is to define a new Karel subclass that accomplishes the task illustrated in this diagram:

解决更有趣的问题

在 Figure 1 里定义的 BeeperPickingKarel 类做的动作不多。让我们试着让它更有趣一点。设定目标不是简单的让卡雷尔捡起蜂鸣器，而是要把这个蜂鸣器从它的初始位置，第一大街第二道的交点，移动到第二大街和第五大道的交点。因此，你的下一个任务是定义一个新的 karel 子类，它可以完成下面插图描绘的任务：
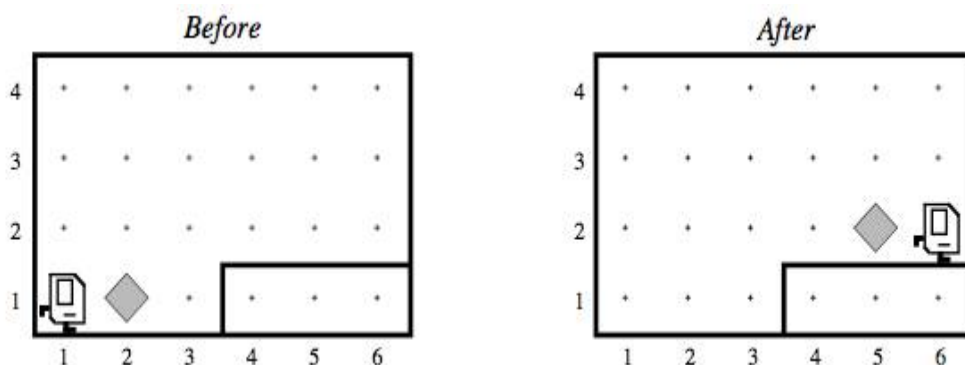
The first three commands in the new program—the ones that move forward, pick up the beeper, and then move up to the ledge—are the same as before:

新程序的前三个命令是,向前移动，捡起蜂鸣器，然后再向前移动到边界前，这个和以前的一样。

move();
pickBeeper();
move();

From here, the next step is to turn left to begin climbing the ledge. That operation is easy, because Karel has a turnLeft command in its standard repertoire. Executing a turnLeft command at the end of the

preceding sequence of commands leaves Karel facing north on the corner of 3rd Avenue and 1st Street. If Karel then executes a move command, it will move north to reach the following position:

从这儿开始，机器人左转，开始攀爬边界。这个操作很容易，因为在它的标准命令库中有一个左转的命令。在卡雷尔到达第一大街和第三大道交汇的街角之后，执行这个命令，会让卡雷尔面向北方。如果这时执行一个移动命令，它将会向北移动，到达随后的位置：
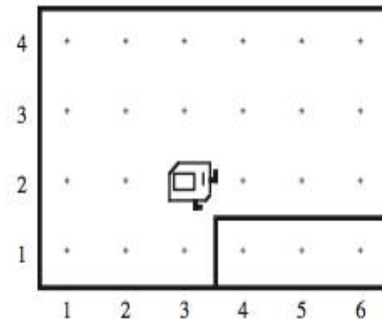
From here, the next thing you need to do is get Karel to turn right so that it is again facing east. While this operation is conceptually just as easy as getting Karel to turn left, there is a slight problem: Karel's language includes a turnLeft command, but no turnRight command. It's as if you bought the economy model and have now discovered that it is missing some important features.

从这儿开始，你需要做的下一件事情是让卡雷尔向右转，以便它再一次面对东方。虽然这个操作是和卡雷尔左转一样容易的概念，但这里有一个轻微的问题：卡雷尔的语言包括了一个 turnLeft 命令，但是没有 turnRight 命令。这就好像你买了个便宜的模型玩具，现在发现它缺少一些重要的功能。

At this point, you have your first opportunity to begin thinking like a programmer. You have one set of commands, but not exactly the set you need. What can you do? Can you accomplish the effect of a turnRight command using only the capabilities you have? The answer, of course, is yes. You can accomplish the effect of turning right by turning left three times. After three left turns, Karel will be facing in the desired direction. From here, all you need to do is program Karel to move over to the center of the ledge, drop the beeper and then move forward to the final position. A complete implementation of a BeeperTotingKarel class that accomplishes the entire task is shown in Figure 2.

在这一点上，你有了第一个，像一个程序员一样，开始思考的机会。你有了一套命令，但是恰好没有你需要的命令。你能做什么？你能仅仅使用你现在拥有的能力实现 turnRight 的效果吗？答案当然是：Yes 。你可以通过三次左转来实现右转的效果。三次左转之后，卡雷尔将面对需要的方向。从这儿开始，所有你需要在卡雷尔程序里做的就是移动到所需的那个中心点，放下蜂鸣器，然后移动到最终的位置。一个可以实现整个任务的，完整的 BeeperTotingKarel 类的实现（后面的这个实现是名词，也就是程序代码的意思--译者注）被显示在 Figure 2 。

Figure 2. Program to carry a beeper to the top of a ledge

```
/*
 * File: BeeperTotingKarel.java
 * ----------------------------
 * The BeeperTotingKarel class extends the basic Karel class
```

```
 * so that Karel picks up a beeper from 1st Street and then
 * carries that beeper to the center of a ledge on 2nd Street.
 */
import stanford.karel.*;
public class BeeperTotingKarel extends Karel {
  public void run() {
    move();
    pickBeeper();
    move();
    turnLeft();
    move();
    turnLeft();
    turnLeft();
    turnLeft();
    move();
    move();
    putBeeper();
    move();
  }
}
```

Defining new methods
Even though the BeeperTotingKarel class in Figure 2 demonstrates that it is possible to perform the
turnRight operation using only Karel's built-in commands, the resulting program is not particularly clear
conceptually. In your mental design of the program, Karel turns right when it reaches the top of the ledge.
The fact that you have to use three turnLeft commands to do so is annoying. It would be much simpler if
you could simply say turnRight and have Karel understand this command. The resulting program would
not only be shorter and easier to write, but also significantly easier to read.

定义些新动作

尽管通过在 Figure 2 中的 BeeperTotingKarel 类展示了，仅仅通过卡雷尔内建命令实现 "turnRight"
操作是可能的， 但程序结果不是一个特别清楚的概念。你在这个程序里的设计思路是，当卡雷尔
到达了所需的那个点时，卡雷尔向右转。你不得不使用三次左转命令来做这个（向右转）的事实
让人很恼火。如果你能简单的说 "turnRight",卡雷尔就能明白这个命令的话，那将简单多了。效果
是程序不但更短了，更容易写了，而且阅读起来明显更容易了。

Fortunately, the Karel programming language makes it possible to define new commands simply by
including new method definitions. Whenever you have a sequence of Karel commands that performs
some useful task—such as turning right—you can define a new method that executes that sequence of
commands. The format for defining a new Karel method has much the same as the definition of run in the
preceding examples, which is a method definition in its own right. A typical method definition looks like
this:

幸运地是，卡雷尔编程语言使得通过简单的包括一个新动作，来定义新命令成为可能。当你有了
一个可以执行一些有用任务的卡雷尔命令序列，你可以把这个序列定义成可以执行命令序列的新
动作。定义一个新卡雷尔动作的格式，有很多和上面定义"run"的例子相同的地方。这就是一个定
义它自己右转的办法。一个典型的动作定义看起来像这样：

```
private void name() {
    commands that make up the body of the method
}
```

In this pattern, name represents the name you have chosen for the new method. To complete the definition, all you have to do is provide the sequence of commands in the lines between the curly braces. For example, you can define turnRight as follows:

在这个格式里，"name"代表你给新动作选定的名字。接下来完成这个（动作）定义，所有你要做的就是提供占两个括号中间几行的一套命令序列。例如，你可以按照下面的方式定义 "turnRight":

```
private void turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
}
```

Similarly, you could define a new turnAround method like this:

类似的，你可以像这样定义一个新 "turnAround"动作：

```
private void turnAround() {
    turnLeft();
    turnLeft();
}
```

You can use the name of a new method just like any of Karel's built-in commands. For example, once you have defined turnRight, you could replace the three turnLeft commands in the BeeperTotingKarel class with a single call to the turnRight method. A revised implementation of the program that uses turnRight is shown in Figure 3.

你可以直接调用新动作的名字，好像这个新动作是卡雷尔的内建动作一样。例如，一旦你定义了 "turnRight",在 BeeperTotingKarel 类里，你就可以调用一个 "turnRight" 来代替三个 "turnLeft"命令。一个使用 "turnRight" 执行的，订正过的程序被显示在 Figure 3 。

There is, of course, one obvious difference between the definitions of the run and turnRight methods shown in Figure 3: the run method is marked as public in contrast to turnRight, which is marked as private. The difference between these two designations is that public methods can be invoked from outside the class, while private methods cannot. The run method needs to be public because the Karel environment needs to be able to issue a run command to get things going. By contrast, turnRight is used only inside the other code appearing within this class. That definition, therefore, can be private, and it is generally good programming practice to keep definitions private whenever possible. The reasons for this rule are difficult to appreciate until you have had a chance to work with larger programs, but the basic idea is that classes should try as much as possible to encapsulate information, which means not only to gather it together but also to restrict access to that information if possible. Large programs quickly become very complex in terms of the volume of detail that they encompass. If a class is well designed, it will seek to reduce that complexity by hiding as much extraneous detail as it can. This property is known

as information hiding and is a cornerstone of the object- oriented philosophy.

当然了，在这里，定义 "run" 和定义 "turnRight" 动作之间一个明显的不同被显示在 Figure 3: 对比被标记为 "private"的"turnRight","run"动作被标记为 "public",这两种设定间的不同是："public"动作可以被外部的类调用，而"private"类不可以。"run"动作被标记为"public",是因为卡雷尔编程环境需要一个向前走，拿东西的"run"命令。相比之下，"turnRight" 仅仅被出现在这个类里面的其它代码使用。因此，这个定义是（这个类）私用的。尽可能的保持定义私有化，总体来说是一个好的编程习惯。在你有机会为大项目工作之前，这么做的理由，你很难体会到。但是，基础的思想是，这些类应该试着尽可能多的封装信息，这意味着这些类不仅可以聚集在一起工作，而且尽可能的限制（这些类之间互相的）访问信息。大型程序在它们包含的项目的细节数量上很快变得非常复杂。如果一个类被设计的很好，它将通过隐藏尽可能多的额外信息来寻求降低系统的复杂性。这种属性被称为信息隐藏，是面向对象哲学的一块基石。

Figure 3. Revised implementation of BeeperTotingKarel that includes a turnRight method
----
```
/*
 * File: BeeperTotingKarel.java
 * ----------------------------
 * The BeeperTotingKarel class extends the basic Karel class
 * so that Karel picks up a beeper from 1st Street and then
 * carries that beeper to the center of a ledge on 2nd Street.
 */
import stanford.karel.*;
public class BeeperTotingKarel extends Karel {
  public void run() {
    move();
    pickBeeper();
    move();
    turnLeft();
    move();
    turnRight();
    move();
    move();
    putBeeper();
    move();
  }
/**
 * Turns Karel 90 degrees to the right.
 */
  private void turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
  }
}
```

At this point in your study of programming, you are not likely to find these arguments about encapsulation particularly convincing. Defining turnRight and turnAround in every program is certainly a bit of a pain, particularly in light of the fact that they are so useful. The difficulty, however, in making them more easily available lies in figuring out

where to put those definitions. Declaring turnRight to be public in the definition of BeeperTotingKarel would not be much help. In an object-oriented language, the methods that specify the behavior of a class are encapsulated within that class. The turnRight method that appears within that class knows how to turn an instance of BeeperTotingKarel 90 degrees to the right, but that method cannot be applied to an instance of the Karel class or any its subclasses.

在你学习编程的这一点上，你很难找出关于封装特别有说服力的证据。在每个程序中都定义 "turnRight"和"turnAround"确实有点痛苦，鉴于事实上，他们非常有用，（每次都)编写它们比指出它们在哪里定义过更容易使用是个谎言（上面这句话的本义是：从一般理解上讲，turnRight 用的很频繁，不应该定义为私有，在每个类里都编写一遍，而应该定义为 public，编写一次，到处引用。--译者注）。在 BeeperTotingKarel 类的定义中，把 "turnright"声明成"public"，也不会有太多帮助。在面向对象语言中，作为指定一个类的行为的方法，是要被封装到这个类里的。这个让 BeeperTotingKarel 类的实例知道如何向右转 90 度的"turnRight"动作，不能被应用在 karel 类和它的任何其它子类的实例里。

In some sense, what you really want to do is add turnRight and turnAround to the Karel class so that all subclasses will be able to use these undeniably useful commands. The problem with that strategy is that you don't necessarily have the access to the Karel class necessary to make such a change. The Karel class is part of the Stanford library, which is used by all students in this CS106A. If you were to go and make changes to it, you might end up breaking someone else's program, which would not endear you to the other students. Similarly, if you at some point later in the quarter decide that you really want to add something to one of the standard Java classes, you won't actually be able to change that class because it is under the control of Sun Microsystems. What you can do,
however, is define a new class that includes your new features as extensions. Thus, if you want to use turnRight and turnAround in several different Karel programs, you could define a new class that included these method definitions and then create your programs by extending that class. This technique is illustrated in Figure 4, which consists of two program files. The first contains a class definition called NewImprovedKarel that
includes the turnRight and turnAround definitions as public methods so that other classes can use them. The second is yet another implementation of BeeperTotingKarel that extends NewImprovedKarel, thereby giving itself access to these methods.

从某种意义上说，你真的想把 "turnRight"和"turnAround"这些不可否认，很有用的命令加入到 karel 类里，这样以来，所有（karel 类）的子类都能够使用。一个战略意义上的问题是，你没有必要访问 karel 类来做这个所谓必要的改变。karel 类 是 Stanford 库的一部分，（这个库）被这个 CS106A 的所有学生使用。如果你去对这个类做了更改，你可能会破坏别人的程序，你不会得到其它学生的拥戴。在这以后的某一刻，你真的想加些东西到一个 Java 标准类中，你实际上也不能改变这个类（这里指标准的 Java 类，应该不包括 karel 类--译者注），因为这些标准类是在 Sun 公司控制之下的。你能做的，就是定义一个包含你的新特性的新类，作为扩展。因此，如果你想在几个不同的卡雷尔程序里都使用 "turnRight"和"turnAround",你可以定义一个包含这些动作定义的新类，通过扩展这个类来创作你的程序。这种技术的说明在 Figure 4 里，它包含了两个程序文件。第一个里包含了一个类定义叫 NewImprovedKarel，在 NewImprovedKarel 类里包含的 "turnRight" 和"turnAround"被定义为 "public"动作，这样其它类也可以使用它们。第二个里面，BeeperTotingKarel 类是 NewImprovedKarel 类的扩展，（BeeperTotingKarel 类）它自己从而可以访问这些动作。

Figure 4. Defining a NewImprovedKarel class that understands turnRight and turnAround
----
/*

```
 * File: NewImprovedKarel.java
 * -------------------------------------------
 * The NewImprovedKarel class extends the basic Karel class
 * so that any subclasses have access to the turnRight and
 * turnAround methods.  It does not define any run method
 * of its own.
 */
import stanford.karel.*;
public class NewImprovedKarel extends Karel {
/**
 * Turns Karel 90 degrees to the right.
 */
  public void turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
  }
/**
 * Turns Karel around 180 degrees.
 */
  public void turnAround() {
    turnLeft();
    turnLeft();
  }
}
/*
 * File: BeeperTotingKarel.java
 * ----------------------------
 * The BeeperTotingKarel class extends the basic Karel class
 * so that Karel picks up a beeper from 1st Street and then
 * carries that beeper to the center of a ledge on 2nd Street.
 */
import stanford.karel.*;
public class BeeperTotingKarel extends NewImprovedKarel {
  public void run() {
    move();
    pickBeeper();
    move();
    turnLeft();
    move();
    turnRight();
    move();
    move();
    putBeeper();
    move();
  }
}
```

The stanford.karel package does not include the NewImprovedKarel class as it appears here but does
include a SuperKarel class that includes the methods turnRight and turnAround along with several other
extensions that will make it possible for you to write much more exciting programs. The examples that

follow extend SuperKarel to ensure that these methods are available. The other extensions are described in Chapter 6.

stanford.karel 包里不包括这个在这儿出现的 NewImprovedKarel 类，但是它包括一个叫 SuperKarel 的类，这个 SuperKarel 类包括"turnRight"和"turnAround"动作以及其它几个扩展，它使得你写出更多精彩的程序成为可能。接下来的例子是 SuperKarel 类的扩展，这确保了这些动作都是可用的。（Superkarel 类）其它的扩展将在第六章被描述。

Decomposition
As a way of illustrating more of the power that comes with being able to define new methods, it's useful to have Karel do something a little more practical than move a beeper from one place to another. The roadways around Palo Alto often seem to be in need of repair, and it might be fun to see if Karel can fill potholes in its abstract world. For example, imagine that Karel is standing on the "road" shown in the left-hand figure, one corner to the left of a pothole in the road. Karel's job is to fill the hole with a beeper and proceed to the next corner. The diagram on the right illustrates how the world should look after the program execution.

分解

作为说明定义新动作是带来更多能力的一种途径，让卡雷尔做些比，从一个地方移动一个蜂鸣器到另外一个地方更实际些的东西，是很有用的。围绕在 Palo Alto 周围的道路经常好像需要修理，在它的抽象世界里，如果卡雷尔能够填坑，应该是比较有趣的。例如，设想卡雷尔正站在左侧插图显示的路上，左侧公路上一个街角有个坑，卡雷尔的工作是用蜂鸣器把这个洞填上，继续推进到下一个街角。右侧的插图描绘了这个程序执行以后，这个世界看起来的样子。
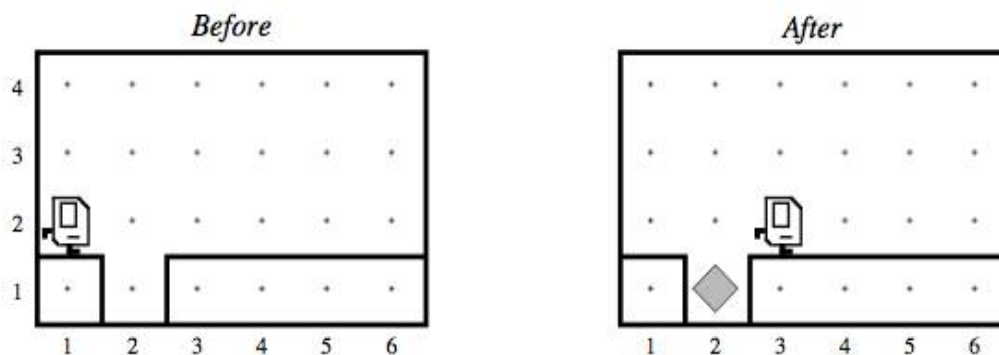
If you are limited to the four predefined commands, the run method to solve this problem would look like this:

如果你受到那四个预定义命令的限制，解决这个问题的 "run" 的代码看起来像这样：

```
public void run() {
  move();
  turnLeft();
  turnLeft();
  turnLeft();
  move();
  putBeeper();
  turnLeft();
  turnLeft();
```

```
  move();
  turnLeft();
  turnLeft();
  turnLeft();
  move();
}
```

You can, however, make the main program easier to read by extending SuperKarel and then making use of the turnAround and turnRight methods. This version of the program appears in Figure 5.

但是，你可以使用带有 "turnAround"和"turnRight"动作的 SuperKarel 扩展类使主程序读起来更容易。这个版本的程序显示在 Figure 5 里。

Figure 5. Karel program to fill a single pothole

```
/*
 * File: PotholeFillingKarel.java
 * ------------------------------
 * The PotholeFillingKarel class puts a beeper into a pothole
 * on 2nd Avenue.  This version of the program uses no
 * decomposition other than turnRight and turnAround,
 * which are inherited from SuperKarel.
 */
import stanford.karel.*;
public class PotholeFillingKarel extends SuperKarel {
  public void run() {
    move();
    turnRight();
    move();
    putBeeper();
    turnAround();
    move();
    turnRight();
    move();
  }
}
```

The initial motivation for defining the turnRight method was that it was cumbersome to keep repeating three turnLeft commands to accomplish a right turn. Defining new methods has another important purpose beyond allowing you to avoid repeating the same command sequences every time you want to perform a particular task. The power to define methods unlocks the most important strategy in programming—the process of breaking a large problem down into smaller pieces that are easier to solve. The process of breaking a program down into smaller pieces is called decomposition, and the component parts of a large problem are called subproblems.

定义一个"右转"动作的初始动机是：老是重复三个左转命令来实现一个右转是累赘的。定义一个新动作有另外一个超越了 "允许你在特定任务里避免每次都重复相同命令序列"这个事情更重要的目的。这个定义新动作的能力解锁了编程过程中最重要的战略思想，那就是把一个大的问题，分

解成一些更容易解决的小一些的部分。这个把一个程序分成一些小部分的过程被称为分解，一个大问题的这些小的组成部分被称为子问题。

As an example, the problem of filling the hole in the roadway can be decomposed into the following subproblems:

1. Move up to the hole
2. Fill the hole by dropping a beeper into it
3. Move on to the next corner

作为一个例子，这个在路上填坑的问题可以被分解为下列子问题：

1.移动到坑的上方
2.丢下一个蜂鸣器填这个坑儿
3.移动到下一个街角

If you think about the problem in this way, you can use method definitions to create a program that reflects your conception of the program structure. The run method would look like this:

如果你用这种方式思考这个问题，你可以使用动作定义来创造一个"程序"，这个"程序"从概念上，反映了你的真实程序的结构。这个 "run"动作看起来像这样:

```
public void run() {
  move();
  fillPothole();
  move();
}
```


The correspondence with the outline is immediately clear, and everything would be great if only you could get Karel to understand what you mean by fillPothole. Given the power to define methods, implementing fillPothole is extremely simple. All you have to do is define a fillPothole method whose body consists of the commands you have already written to do the job, like this:

对应的大纲立刻非常清楚了，只要你让卡雷尔明白你的意思是填坑，所有事情将变得很棒。通过定义些动作的能力，实施填坑是非常简单的。所有你要做的是定义一个填坑的动作，这个动作的主体包含着你上面已经写过的，做这个工作的那些命令，像这个：

```
private void fillPothole() {
  turnRight();
  move();
  putBeeper();
  turnAround();
  move();
  turnRight();
}
```

The complete program is shown in Figure 6.

完整的程序显示在 Figure 6。

Figure 6. Program to fill a single pothole using a fillPothole method for decomposition
----

```
/*
 * File: PotholeFillingKarel.java
 * -----------------------------
 * The PotholeFillingKarel class puts a beeper into a pothole
 * on 2nd Avenue.  This version of the program decomposes
 * the problem so that it makes use of a fillPothole method.
 */
import stanford.karel.*;
public class PotholeFillingKarel extends SuperKarel {
   public void run() {
      move();
      fillPothole();
      move();
   }
/**
 * Fills the pothole beneath Karel's current position by
 * placing a beeper on that corner.  For this method to
 * work correctly, Karel must be facing east immediately
 * above the pothole.  When execution is complete, Karel
 * will have returned to the same square and will again
 * be facing east.
 */
   private void fillPothole() {
      turnRight();
      move();
      putBeeper();
      turnAround();
      move();
      turnRight();
   }
}
```

Choosing the correct decomposition
There are, however, other decomposition strategies you might have tried. For example, you could have
written the program as

选择正确的分解

但是，这里也有其它你可能试图使用的分解策略。例如，你可以像下面这样写程序：

```
public void run() {
   approachAndFillPothole();
   move();
}
```

where the approachAndFillPothole method is simply

这里的 approachAndFillPothole 动作也简单：

```java
private void approachAndFillPothole() {
    move();
    turnRight();
    move();
    putBeeper();
    turnAround();
    move();
    turnRight();
}
```

作为另外一种选择，你也可以像这样写程序：

```java
public void run() {
    move();
    turnRight();
    move();
    fillPotholeYouAreStandingIn();
    turnAround();
    move();
    turnRight();
    move();
}
```

where the body of fillPotholeYouAreStandingIn consists of a single putBeeper command. Each of these programs represents a possible decomposition. Each program correctly solves the problem. Given that all three versions of this program work, what makes one choice of breaking up the problem better than another?

这里的 fillPotholeYouAreStandingIn 类的本体包含了单独的一条 "putBeeper"命令。这些程序每一个都代表了一个可能的分解。每个程序都能正确的解决问题。给出这个程序工作的所有三种版本，哪一种分解的选择比其它的好？

In general, deciding how to decompose a program is not easy. In fact, as the problems become more complex, choosing an appropriate decomposition will turn out to be one of the more difficult aspects of programming. You can, however, rely to some extent on the following guidelines:

一般来说，决定如何分解一个程序不容易。选择一个适当的分解，将变成编程中比较困难的一个方面。但是，你能在一定程度上依赖以下原则：

1. Each subproblem should perform a conceptually simple task. The solution of a subproblem may require many commands and may be quite complex in terms of its internal operation. Even so, it should end up accomplishing some conceptual task that is itself easy to describe. A good indication of whether you have succeeded in identifying a reasonable task comes from the name you give to the method. If you can accurately define its effect with a simple descriptive name, you have probably chosen a good decomposition. On the other hand, if you end up with complex names such as approachAndFillPothole, the decomposition does not seem as promising.

1.每一个子问题应该执行一个在概念上是很简单的任务。一个子问题的解决方案也许需要许多条命令，也许在内部操作条件方面相当复杂。即便如此，它最终应该完成一些在概念上容易描述的任务。如果你给出的动作名字能成功的概括出一个合理的任务，是一个好迹象。如果你可以通过一个简单的描述性名称，准确的定义（这个动作）它的作用，你可能已经选择了一个很好的分解方案。另一方面，如果你最终使用了像 approachAndFillPothole 这样的复杂名称，这个分解方案不像有前途。

2. Each subproblem should perform a task that is as general as possible, so that it can be used in several different situations. If one decomposition results in a program that is only useful in the exact situation at hand and another would work equally well in a variety of related situations, you should probably choose the more general one.

每个子问题应该执行尽可能具有通用性的任务，这样，这个（子问题）就可以在几个不同的场景下调用。如果一个分解方案的结果只能在一个特定的场景中使用，而手上的另一个在几个不同的相关场景中都能工作的同样好，你应该选择更通用的那个。

The technique of defining new methods—as useful as it is—does not actually enable Karel to solve any new problems. Because each method name is merely a shorthand for a specific set of commands, it is always possible to expand a program written as a series of method calls into a single main program that accomplishes the same task, although the resulting program is likely to be long and difficult to read. The commands—no matter whether they are written as a single program or broken down into a set of methods—are still executed in a fixed order that does not depend on the state of Karel's world. Before you can solve more interesting problems, you need to discover how to write programs in which this strictly linear, step-by-step order of operations does not apply. In particular,you need to learn several new features of the Karel programming language that make it possible for Karel to examine its world and change its execution pattern accordingly.

定义新动作的技巧，虽然实际上并没有让卡雷尔解决任何新问题，但一样有用。因为每个动作的名字不过是一组指定命令的速记，在一个单独的主程序中，总是可能把调用来完成相同任务的，一系列动作扩展成一个程序，尽管这样（扩展出）的程序很可能是很长而且难以阅读的。这些命令，无论是写成了一个单一的程序或被分解写成一组方法，，仍然会不依赖于卡雷尔的世界的状态，按一个固定的顺序执行。在你解决更有趣的问题之前，你需要发现如何写出不再按照一步步的操作顺序，而是严格的线性的程序。特别是，你需要学习一些卡雷尔编程语言的新功能，这些功能使得卡雷尔可以探测它的世界，（根据探测结果）改变它的执行模式成为可能。

Statements that affect the order in which a program executes commands are called control statements. Control statements generally fall into the following two classes:

（在程序代码中--译者注）那些可以影响一个程序里执行命令次序的语句被称为控制语句。控制语句一般分为以下两类：

1. Conditional statements. Conditional statements specify that certain statements in a program should be executed only if a particular condition holds. In Karel, you specify conditional execution using an if statement.

2. Iterative statements. Iterative statements specify that certain statements in a program should be executed repeatedly, forming what programmers call a loop. Karel supports two different iterative statements: a for statement that is useful when you want to repeat a set of commands a predetermined number of times and a while statement that is useful when you want to repeat an operation as long as some condition holds.

1.条件语句。条件语句是指在程序里这样一些语句，只有当特定的条件成立了，才会被执行。在卡雷尔编程里，你可以使用 "if"来指定条件语句。

2.迭代语句。迭代语句是指在一个程序里，某些语句需要被反复执行，程序员称这个为"循环"。卡雷尔支持两种不同的迭代语句。"for"语句用于当你想按预定的次数重复执行一组命令的时候，"while"语句用于当你想在某些条件满足时，重复执行一组命令的时候

This chapter introduces each of these control statement forms in the context of Karel problems that illustrate the need for each statement type.

这一章介绍了在每一种语句类型具体需求所对应的，卡雷尔环境问题上，这些控制语句的每一种形式。

Conditional statements

To get a sense of where conditional statements might come in handy, let's go back to the fillPothole program presented at the end of Chapter 2. Before filling the pothole in the fillPothole method, there are a few conditions that Karel might want to check. For example, Karel might want to check to see if some other repair crew has already filled the hole, which means that there is already a beeper on that corner. If so, Karel does not need to put down a second one. To represent such checks in the context of a program, you need to use the if statement, which ordinarily appears in the following form:

if (conditional test) {
    statements to be executed only if the condition is true
}

条件语句

为了明白条件语句可以派上用场的情况，让我们回到在第二章末尾提出的 "填坑" 程序。在 "fillPothole"动作之前，有几个卡雷尔想检查的条件。例如，卡雷尔想检查一下，看看是否有其它人员已经把这个坑儿填了，这就意味着，那个街角上已经有一个蜂鸣器了。如果是这样的，卡雷尔就不需要放第二个蜂鸣器了。为了在程序里表达在这种情况下的这种检查，你需要使用"if"条件语句，它通常以下面的形式出现：

if (条件检测) {
    只有当条件满足时才会执行的语句
}

The conditional test shown in the first line of this pattern must be replaced by one of the tests Karel can perform on its environment. The result of that conditional test is either true or false. If the test is true, Karel executes the statements enclosed in braces; if the test is false, Karel does nothing.

显示在这个模式第一行里的"条件检测"，必需指代一种卡雷尔可以在它的环境里执行的一种检测。这种条件检测返回的结果是"真"或"假"。如果这个测试结果是"真"，卡雷尔执行括在大括号里的语句；如果测试结果是"假"，卡雷尔什么也不做。

The tests that Karel can perform are listed in Table 1. Note that each test includes an empty set of parentheses, which is used as a syntactic marker in Karel's programming language to show that the test is being applied. Note also that every condition in the list has a corresponding opposite. For example, you can use the frontIsClear condition to check whether the path ahead of Karel is clear or the frontIsBlocked condition to see if there is a wall blocking the way. The frontIsClear condition is true whenever frontIsBlocked is false and vice versa. Choosing the right condition to use in a program requires you to think about the logic of the problem and see which condition is easiest to apply.

Table 1. Conditions that Karel can test

Test	Opposite	What it checks
frontIsClear()	frontIsBlocked()	Is there a wall in front of Karel?
leftIsClear()	leftIsBlocked()	Is there a wall to Karel's left?
rightIsClear()	rightIsBlocked()	Is there a wall to Karel's right?
beepersPresent()	noBeepersPresent()	Are there beepers on this corner?
beepersInBag()	noBeepersInBag()	Any there beepers in Karel's bag?
facingNorth()	notFacingNorth()	Is Karel facing north?
facingEast()	notFacingEast()	Is Karel facing east?
facingSouth()	notFacingSouth()	Is Karel facing south?
facingWest()	notFacingWest()	Is Karel facing west?

卡雷尔可以执行的这些检测被列在表1里。注意每个测试格式里表括一对空括号，在卡雷尔的编程语言中，这个被用来作为一个语法标记，表明这是个正在使用的测试。还要注意，列表中的每个状态检测都有一个对应的反面。例如，你可以使用 "frontIsClear" 这个条件来检测卡雷尔前面的路是通的，或者用"frontIsBlocked"条件来检测卡雷尔的面前是否有一堵墙。当"frontIsBlocked"的返回值是"假"的时候，"frontIsClear"的返回值就是"真"，反之亦然。在程序中，选择使用一个正确的条件，需要你思考程序的逻辑并且看看哪个条件是最容易应用的。

卡雷尔可以判断的环境条件（一共九组十八个判断方法，两两相对）

测试条件	相反的测试条件	判断的内容
frontIsClear() 面前无墙返回肯定值	frontIsBlocked() 面前被挡返回肯定值	面前是否有墙
leftIsClear() 左面无墙返回肯定值	leftIsBlocked() 左面被挡返回肯定值	左面是否有墙
rightIsClear() 右面无墙返回肯定值	rightIsBlocked() 右面被挡返回肯定值	右面是否有墙
beepersPresent() 所在位置有方块返回肯定值	noBeepersPresent() 所在位置无方块返回肯定值	所在位置是否有方块
beepersInBag() 包里目前有方块返回肯定值	noBeepersInBag() 包里目前无方块返回肯定值	包里是否有方块
facingNorth() 面朝北返回肯定值	notFacingNorth() 没有面朝北返回肯定值	是否面朝向北
facingEast() 面朝东返回肯定值	noFacingEast() 没有面朝东返回肯定值	是否面朝向东
facingSouth() 面朝南返回肯定值	noFacingSouth() 没有面朝南返回肯定值	是否面朝向南
facingWest() 面朝西返回肯定值	noFacingWest() 没有面朝西返回肯定值	是否面朝向西

You can use the if statement to modify the definition of the fillPothole method so that Karel puts down a beeper only if there is not already a beeper on that corner. To do so, the conditional test you need is noBeepersPresent. If there are no beepers present on the corner, Karel should put down a new one. If there is already a beeper there, Karel should do nothing. The new definition of fillPothole, which checks to make sure that there is not already a beeper in the hole, looks like this:

你可以使用"if"语句来修改"fillPothole" 动作的定义，以便让卡雷尔只有在街角没有蜂鸣器的时候，才会放下一个蜂鸣器。为了做这个，你需要的条件测试是"noBeepersPresent"。如果没有蜂鸣器在这个街角存在，卡雷尔就放一个新的。如果那个街角已经有一个蜂鸣器了，卡雷尔啥也不做。这个确认坑里是否有蜂鸣器的检测，在新的"fillPothole"定义里，看起来像这样：

```
private void fillPothole() {
  turnRight();
  move();
  if (noBeepersPresent()) {
    putBeeper();
  }
  turnAround();
```

```
    move();
    turnRight();
}
```

The if statement in this example illustrates several features common to all control statements in Karel. The control statement begins with a header, which indicates the type of control statement along with any additional information to control the program flow.In this case, the header is

if (noBeepersPresent())

which shows that the statements enclosed within the braces should be executed only if the noBeepersPresent test is true. The statements enclosed in braces represent the body of the control statement.

在这个例子里的"if"语句描绘出所有卡雷尔控制语句的共同特点。控制语句以一个头部开始，它表明随着哪种控制语句的附加信息，来控制程序流。在这个例子里，头部是：

if (noBeepersPresent())

这个表明，只有当"noBeepersPresent"检测的值是"真"的时候，大括号里的语句才会被执行。在大括号里的语句代表这个控制语句的"本体"。

By convention, the body of any control statement is indented with respect to the statements that surround it. The indentation makes it much easier to see exactly which statements will be affected by the control statement. Such indentation is particularly important when the body of a control statement contains other control statements. For example, you might want to make an additional test to see whether Karel had any beepers before trying to put one down. To do so, all you would need to do is add a new if statement inside the existing one, like this:

按照惯例，任何控制语句的本体，和（不属于）它的前后语句相比，是缩进的。这种缩进很容易的表明哪些语句是被这个控制语句影响的。这个缩进在一个控制语句里包含另一些控制语句的时候，是非常重要的。例如，你可能想做个额外的检测，在卡雷尔试图放下一个蜂鸣器之前，先看看它是否有蜂鸣器。为了做这个，所有你需要做的是向现在的控制语句里再加一条新的"if"语句。像这样：

```
if (noBeepersPresent()) {
    if (beepersInBag()) {
        putBeeper();
    }
}
```

In this example, the putBeeper command is executed only if there is no beeper on the corner and there are beepers in Karel's bag. Control statements that occur inside other control statements are said to be nested.

在这个例子里，"putBeeper"命令只有在这个街角没有蜂鸣器，（同时）卡雷尔的收藏包里有蜂鸣器的时候才会被执行。这个控制语句出现在其它控制语句里的情形叫嵌套。

The outcome of a decision in a program is not always a matter of whether to do nothing or perform some

set of operations. In some cases, you need to choose between two alternative courses of action. For these cases, the if statement in Karel has an extended form that looks like this:

```
if (conditional test) {
   statements to be executed if the condition is true
} else {
   statements to be executed if the condition is false
}
```

在一个程序里，输出的决定也不总是要么什么都不做，要么执行一些设置的操作。在一些情况下，你需要在两个行动方针上二选一。为了这种情况，在卡雷尔里，"if"语句有个扩展形式，看起来像这样：

```
if(条件检测) {
   只有当条件满足时才会执行的语句
} else {
   只有当条件不满足时才会执行的语句
}
```

This form of the if statement is illustrated by the method invertBeeperState, which picks up a beeper if there is one and puts down a beeper if the corner is empty.

这个 "if"语句的形式，在 "invertBeeperState" 动作里被描述了，这个动作的意思是，如果这个街角有一个蜂鸣器，就捡起来，如果没有，就放下一个。

```
private void invertBeeperState() {
   if (beepersPresent()) {
     pickBeeper();
   } else {
     putBeeper();
   }
}
```

Iterative statements
In solving Karel problems, you will often find that repetition is a necessary part of your solution. If you were really going to program a robot to fill potholes, it would hardly be worthwhile to have it fill just one. The value of having a robot perform such a task comes from the fact that the robot could repeatedly execute its program to fill one pothole after another.

迭代语句

在解决卡雷尔问题的时候，重复将是你的解决方案的必要组成部分。如果你真的组织一个机器人去填坑，就填一个坑是不值的，事实上，用这一个机器人执行像这样的任务的价值在于，机器人可以重复执行它的程序，一个接一个的填坑。

To see how repetition can be used in the context of a programming problem, consider the following stylized roadway in which the potholes are evenly spaced along 1st Street at every even-numbered

avenue:

为了了解在一个程序问题的背景下如何重复使用，考虑下面的程式化的道路坑洼，在第一大街上，和偶数大道的交点的街角，均匀地分布着坑儿。
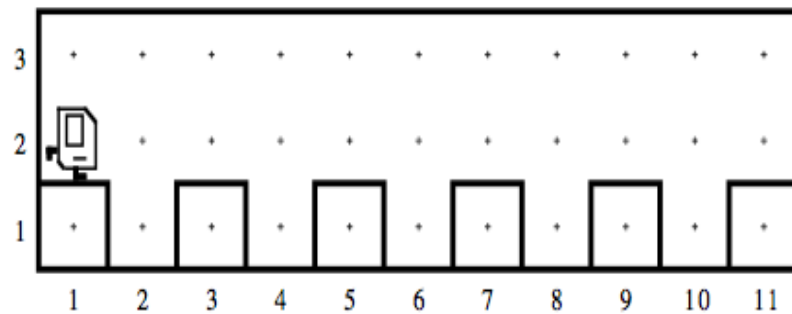
Your mission is to write a program that instructs Karel to fill all the holes in this road. Note that the road reaches a dead end after 11th Avenue, which means that you have exactly five holes to fill.

你的使命是写一个程序，指导卡雷尔把这条路上所有的坑填上。注意路在第十一大道到达一个死胡同，这就意味着，你恰好有 5 个坑要填。

Since you know from this example that there are exactly five holes to fill, the control statement that you need is a for statement, which specifies that you want to repeat some operation a predetermined number of times. The structure of the for statement appears complicated primarily because it is actually much more powerful than anything Karel needs. The only version of the for syntax that Karel uses is

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

where count is an integer indicating the number of repetitions. For example, if you want to change the fillPothole program so that it solves the more complex problem of filling five evenly-spaced holes, all you have to do is change the run method as follows:

既然你知道，这个例子里恰好有 5 个坑儿需要填，你需要的是"for"语句，你可以用这个语句指定按预先设置的次数，重复执行一些操作。"for"语句的结构显得复杂，主要是因为它实际上比任何卡雷尔需要的东西都强大。卡雷尔使用的"for"（语句）语法只有一种版本：

```
for (int i = 0; i < count; i++) {
    重复执行的语句
}
```

这里的 "count"是一个整数，指需要重复的次数。例如，如果你想改变这个填坑的程序，让它解决比填 5 个均匀的坑更复杂的问题，所有你需要做的是像下面这样修改 "run"动作：

```
public void run() {
    for (int i = 0; i < 5; i++) {
        move();
        fillPothole();
        move();
```

```
    }
}
```

The for statement is useful only when you know in advance the number of repetitions you need to perform. In most applications, the number of repetitions is controlled by the specific nature of the problem. For example, it seems unlikely that a pothole-filling robot could always count on there being exactly five potholes. It would be much better if Karel could continue to fill holes until it encountered some condition that caused it to stop, such as reaching the end of the street. Such a program would be more general in its application and would work correctly in either of the following worlds as well as any other world in which the potholes were evenly spaced exactly two corners apart:

只有当你事先知道，你需要重复的执行次数，"for"语句才是有效的。在大多数应用里，重复的次数是被问题里具体的自然数控制的。例如，一个填坑机器人总指望那儿恰好是 5 个坑儿，似乎不太可能。

这样更好些：卡雷尔能持续的填坑，直到遇到一些可以让它停下来的条件，比如，到达街的终点。像这样的程序应用上更具有普遍性，它可以正常工作在这个世界（这里指有 1 个坑的世界--译者注），以及另外一个坑是在每两个街角均匀分布的世界（这里指有 4 个坑的世界--译者注）。

To write a general program that works with any of these worlds, you need to use a while statement. In Karel, a while statement has the following general form:

```
while (test) {
    statements to be repeated
}
```

The test in the header line is chosen from the set of conditions listed in Table 1 earlier in this chapter. In this case, Karel needs to check whether the path in front is clear by invoking the condition frontIsClear. If you use the frontIsClear condition in a while loop, Karel will repeatedly execute the loop until it hits a wall. The while statement therefore makes it possible to solve the more general problem of repairing a roadway, as long as the potholes appear at every even-numbered corner and the end of the roadway is marked by a wall. The RoadRepairKarel class that accomplishes this task is shown in Figure 7.

为了写一个可以在这些世界（指有 1，4，5...个坑的世界--译者注）工作的通用程序，你需要使用"while"语句。在卡雷尔编程语言里，一个"while"语句有如下的一般形式：

```
while (test) {
    重复执行的语句
}
```

在头部行里的"test"，是指在这一章前面表 1 里列出的一套检测条件。在这个例子里，卡雷尔需要调用"frontIsClear"这个检测条件来检测它前面的道路是不是通的。如果你在一个"while"循环里，

使用"frontIsClear"检测条件，卡雷尔将重复执行循环语句，直到它前面遇到墙。"while"语句使我们能够解决更具普遍性的修路问题，成为可能。只要这些坑出现在偶数街角，路最终被一堵墙挡住。

完成这个任务的"RoadRepairKarel"类被显示在 Figure 7.

Figure 7. Program to fill regularly spaced potholes in a roadway
----
```
/*
/*
 * File: RoadRepairKarel.java
 * --------------------------
 * The RoadRepairKarel class fills a series of regularly
 * spaced potholes until it reaches the end of the roadway.
 */
import stanford.karel.*;
public class RoadRepairKarel extends SuperKarel {
  public void run() {
    while (frontIsClear()) {
      move();
      fillPothole();
      move();
    }
  }
/**
 * Fills the hole beneath Karel's current position by
 * placing a beeper in the hole.  For this method to
 * work correctly, Karel must be facing east immediately
 * above the hole.  When execution is complete, Karel
 * will have returned to the same square and will again
 * be facing east.  This version of fillPothole checks to
 * see if there is already a beeper present before putting
 * a new one down.
 */
  private void fillPothole() {
    turnRight();
    move();
    if (noBeepersPresent()) {
      putBeeper();
    }
    turnAround();
    move();
    turnRight();
  }
}
```

Solving general problems
So far, the various pothole-filling programs have not been very realistic, because they rely on specific conditions—such as evenly spaced potholes—that are unlikely to be true in the real world. If you want to write a more general program to fill potholes, it should be able to work with fewer constraints. In particular,

解决普遍性问题

到目前为止，各种填坑程序并不很实用，因为他们依靠特定条件，如均匀分布的坑洞，在真实世界里，这个是不现实的。如果你想写一个更通用的填坑程序，它应该可以工作在更少的限制下。特别是，

• The program should be able to work with roads of arbitrary length. It does not make sense to design a program that works only for roads with a predetermined number of corners. Instead, you want to make the same program work for roads of any length. Such programs, however, do need to know when they have come to the end of the road, so it makes sense to require that the end of the roadway is marked by a wall.
• The potholes may occur at any position in the roadway. There should be no limits on the number of potholes or any restrictions on their spacing. A pothole is identified simply by an opening in the wall representing the road surface.
• Existing potholes may already have been repaired. Any of the potholes may already contain a beeper left by a previous repair crew. In such cases, Karel should not put down an additional beeper.

• 这个程序应该能在任意长度的路上工作。设计一个只能在预定好街角数目的路上工作，是没有意义的。相反，你想要同一个程序工作在不同长度的路上。但是，像这样的程序，当它们已经到达路的尽头时，能够检测到。因此路的尽头被标记为一堵墙，是很有意义的要求。

• 坑可以出现在路的任何位置。坑的数目和间距应该不受任何限制。一个坑简单的定义为，被当作路的墙的表面的一个开口。（从概念上说，卡雷尔的世界里没有路，只有墙，所以，所谓的路，从概念上讲，实际上就是墙--译者注）

• 现有的坑可能已经被填充。任何坑儿可能被以前的维修人员留下的蜂鸣器填充了。在这种情况下，卡雷尔应该不会放下一个额外的蜂鸣器。

To change the program so that it solves this more general problem requires you to think about the overall strategy in a different way. Instead of having the loop in the main program cycle through each pothole, you need to have it check each corner as it goes. If there is an opening at that corner, Karel needs to try and fill the pothole. If there is a wall, Karel can simply move ahead to the next corner.

为了修改程序来解决这些更具普遍性的问题，需要你用不同的方式思考整体的战略思路。通过的每一个坑都要循环，而不是在主程序中的一个循环。（卡雷尔每向前一步--译者注）如果街角有一个开口，卡雷尔需要试着填坑儿，如果那儿有一堵墙，卡雷尔能简单的向前移动到下一个街角。

This strategic analysis suggests that the solution to the general problem may require nothing more than making the following change to the run method from Figure 7:

对这一战略的分析表明，对一般问题的解决方案可能就只是，对 Figure 7 里的 "run" 动作，进行如下更改：

```
public void run() {
  while (frontIsClear()) {
    if (rightIsClear()) {
      fillPothole();
    }
    move();
  }
}
```

However, as the bug symbol off to the side suggests, this program is not quite right. It contains a logical flaw—the sort of error that programmers call a bug. On the other hand, the particular bug in this example is relatively subtle and would be easy to miss, even if you tested the program. For example, the program works correctly on the following world, as shown in the following before-and-after diagrams:

但是，就像旁边的错误符号表示的那样，这个程序不是很对。它包含一个逻辑漏洞，这种错误，程序员称为"bug"。在另外一方面，另一方面，在这个例子中的特定错误是比较微妙，甚至在你测试程序的时候，会很容易错过。例如，这个程序在下面的世界里工作的很好，像下面前后两幅插图显示的：
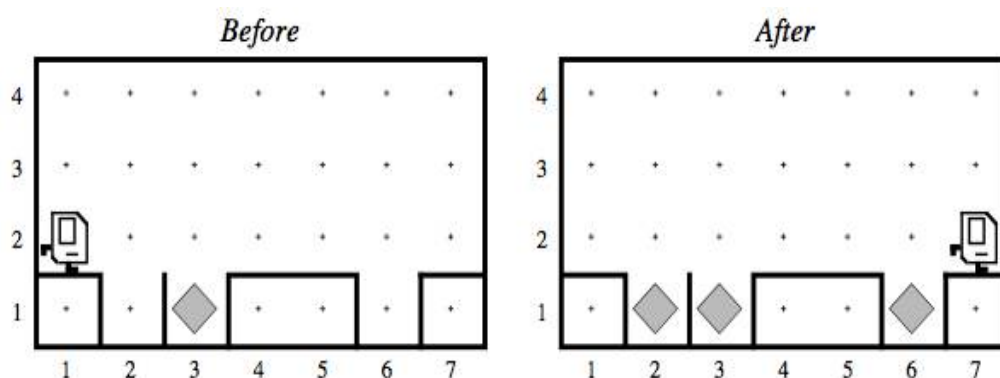
From this example, things look pretty good. If you ended your testing here, however, you would never notice that the program fails if you change the world so that there is a pothole on 7th Avenue. In this case, the before-and-after pictures look like this:

从这个例子上看，事情进行的很好。但是，如果你就在这儿结束了你的测试，你将永远不会注意到，如果你改变这个世界，在第 7 大道设置一个坑，这个程序将运行失败。在这种情况下，前后画面看起来像这样：
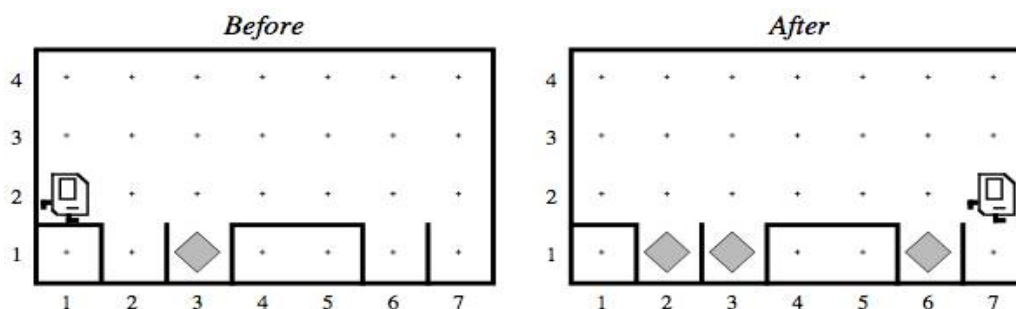
Karel stops without filling the last pothole. In fact, if you watch the execution carefully, Karel never even goes down into that last pothole to check whether it needs filling.What's the problem here?

卡雷尔停下了，最后一个坑没有填。事实上，如果你仔细看执行过程，卡雷尔没有下到最后的坑里去检查这里是否需要填充。这里的问题是什么？

If you follow through the logic of the program carefully, you'll discover that the bug lies in the loop within the run method, which looks like this:

如果你仔细的追随这个程序的逻辑，你将会发现，错误在于"run"动作的循环里，这个循环看起来是这样：

```
public void run() {
    while (frontIsClear()) {
        if (rightIsClear()) {
            fillPothole();
        }
        move();
    }
}
```

As soon as Karel finishes filling the pothole on 6th Avenue, it executes the move command and returns to the top of the while loop. At that point, Karel is standing at the corner of 7th Avenue and 2nd street, where it is blocked by the boundary wall. Because the frontIsClear test now fails, the while loop exits without checking the last segment of the roadway.

一旦卡雷尔完成了第六大道的填坑，它执行"move"命令，返回到"while"循环的顶端。在这一点上，卡雷尔正站在第二大街和第七大道的相交的街角。这里被墙的边界封锁了。因为 "frontIsClear"检测现在返回的值是"假"，这时（程序从）"while"循环退出了，没有检查路的最后一个坑。

The bug in this program is an example of a programming problem called a fencepost error. The name comes from the fact that it takes one more fence post that you might think to fence off a particular distance. How many fence posts, for example, do you need to build a 100-foot fence if the posts are always positioned 10 feet apart? The answer is 11, as illustrated by the following diagram:

在这个程序中的这个错误，是编程问题中"fencepost error"（栅栏柱错误--译者注）的一个例子。这个名字来源于下面的事实，如果你想用栅栏围出一个特定的距离，可能多需要一个栅栏柱。例如，如果你想建 100 英尺长的栅栏，每隔 10 英尺放一个栅栏柱，你需要多少栅栏柱？答案是 11 个，就像下面插图描绘的那样：

100 feet, 11 fenceposts
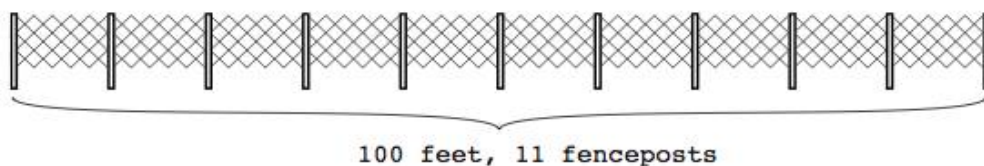
The situation in Karel's world has much the same structure. In order to fill potholes in a street that is seven corners long, Karel has to check for seven potholes but only has to move six times. Because Karel starts and finishes at an end of the roadway, it needs to execute one fewer move command than the number of corners it has to check.

在卡雷尔的世界里形式上有许多相同的结构。为了在有 7 个街角长的街上填坑，卡雷尔不得不检查 7 次坑儿，但只需要移动六次。因为卡雷尔开始和结束时都站在路上，它需要执行的"移动"命令的次数比它需要检查的街角次数就少一次。

Once you discover it, fixing this bug is actually quite easy. Before Karel stops at the end of the roadway, all that the program has to do is make a special check for a pothole at the final intersection, as shown in the program in Figure 8.

一旦你发现了这一点，修改这个错误实际上非常容易。在卡雷尔在路上停下之前，程序需要做的是在最后的交点上，为填坑做一个特定的检查。Figure 8 里显示了这个程序。

Figure 8. Program to fill irregularly spaced potholes

```java
/*
/*
 * File: RoadRepairKarel.java
 * --------------------------
 * This version of the RoadRepairKarel class fills an
 * arbitrary sequence of potholes in a roadway.
 */
import stanford.karel.*;
public class RoadRepairKarel extends SuperKarel {
   public void run() {
      while (frontIsClear()) {
         checkForPothole();
         move();
      }
      checkForPothole();
   }
/**
 * Checks for a pothole immediately beneath Karel's current
 * looking for a wall to the right.  If a pothole exists,
 * Karel calls fillPothole to repair it.
 */
   private void checkForPothole() {
      if (rightIsClear()) {
         fillPothole();
      }
   }
/**
 * Fills the pothole beneath Karel's current position by
 * placing a beeper on that corner.  For this method to
 * work correctly, Karel must be facing east immediately
 * above the pothole.  When execution is complete, Karel
 * will have returned to the same square and will again
 * be facing east.  This version of fillPothole checks to
 * see if there is already a beeper present before putting
 * a new one down.
 */
   private void fillPothole() {
      turnRight();
      move();
      if (noBeepersPresent()) {
         putBeeper();
      }
      turnAround();
      move();
      turnRight();
   }
}
```

Chapter 4
Stepwise Refinement

第四章 逐步求精

（译者重要说明："method"这个单词，是指在程序中定义的 private void ...{}这样一段代码，在流行的书籍和人人影视的视频中，这个词被翻译为"方法"。在前三章中，这个概念是通过介绍为卡雷尔增加新动作来讲解的。为了便于理解，这个词在前三章被翻译为"动作"，例如"defining the turnRight method"被翻译为"定义右转的动作"。通过前三章的学习和第一次的习题，读者应该已经完全理解了"method"所表达的含义。所以，从这一章开始，按照约定俗成的翻译，"method"这个词被翻译为"方法"）

To a large extent, programming is the science of solving problems by computer. Because problems are often difficult, solutions—and the programs that implement those solutions—can be difficult as well. In order to make it easier for you to develop those solutions, you need to adopt a methodology and discipline that reduces the level of that complexity to a manageable scale.

在很大程度上，编程是通过计算机解决问题的一门学问。因为问题通常很难，解决方案以及执行这些解决方案的程序也变得很难。为了让你开发这些方案变得容易些，你需要采取一个方法和规律让问题复杂性的水平降低到可管理的程度。

In the early years of programming, the concept of computing as a science was more or less an experiment in wishful thinking. No one knew much about programming in those days, and few thought of it as an engineering discipline in the conventional sense. As programming matured, however, such a discipline began to emerge. The cornerstone of that discipline is the understanding that programming is done in a social environment in
which programmers must work together. If you go into industry, you will almost certainly be one of many programmers working to develop a large program. That program, moreover, is almost certain to live on and require maintenance beyond its originally intended application. Someone will want the program to include some new feature or work in some different way. When that occurs, a new team of programmers must go in and make the necessary changes in the programs. If programs are written in an individual style with little or no commonality, getting everyone to work together productively is extremely difficult.

在编程的早期，计算的概念作为一门学问或多或少是一厢情愿的试验。在那个年代，没有人对编程了解很多，很少有人想到它能作为传统意义上的一门工程学科。但是，象征编程成熟，这样一门学科开始出现。该学科的基石是，要明白编程是程序员们在必须共同努力的一个社会环境中进行的。如果你进入这个行业，你几乎肯定会是和很多程序员一起，合作开发大型项目。而且，程序的存活和维护需求几乎可以肯定会超出原始的预期应用。有人会希望程序，增加一些新的功能或以一些不同的方式工作。当这种情况发生时，一个新的程序员团队必需进驻，对程序进行各种必要的修改。如果程序是以个人风格写成的，很少或没有通用性，让大家一起有效工作是非常困难的。

To combat this problem, programmers began to develop a set of programming methodologies that are collectively called software engineering. Using good software engineering skills not only makes it easier for other programmers to read and understand your programs, but also makes it

easier for you to write those programs in the first place. One of the most important methodological advances to come out of software engineering
is the strategy of top-down design or stepwise refinement, which consists of solving problems by starting with the problem as a whole. You break the whole problem down into pieces, and then solve each piece, breaking those down further if necessary.

为了解决这一问题，程序员开始开发出一系列的编程方法，它们被统称为软件工程。运用好的软件工程技巧，不仅可以使其它程序员更容易阅读和理解你的程序，而且让你在开始写这些程序时更容易。把起初的一个问题，分解为许多要解决的小问题，这叫自上而下设计或逐步求精，它是出自软件工程的，最重要的方法论发展。你把整个问题分解成小块，然后解决每一块，如果有必要，那些分解的块还需要进一步的分解。

An exercise in stepwise refinement
To illustrate the concept of stepwise refinement, let's teach Karel to solve a new problem. Imagine that Karel is now living in a world that looks something like this:

一个逐步求精的练习

为了描述逐步求精的概念，让我们教卡雷尔解决一个新问题。设想卡雷尔生活在一个看起来像这样的世界：
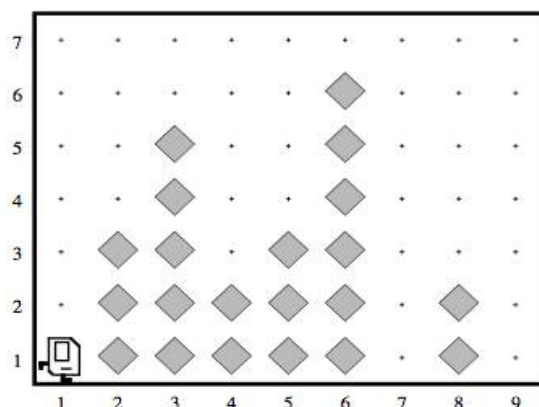
On each of the avenues, there is a tower of beepers of an unknown height, although some avenues (such as 1st, 7th, and 9th in the sample world) may be empty. Karel's job is to collect all the beepers in each of these towers, put them back down on the easternmost corner of 1st Street, and then return to its starting position. Thus, when Karel finishes its work in the example above, all 21 beepers currently in the towers should be
stacked on the corner of 9th Avenue and 1st Street, as follows:

在每一条大道（大道是垂直方向的--译者注）上，都有一座由蜂鸣器组成的未知高度的塔，尽管在一些大道上（比如例子中的第一，第七和第九）可能是空的。卡雷尔的工作是收集组成这些塔的蜂鸣器，把它们放到第一大街最东面的角上，然后（卡雷尔）返回到它开始的位置。因此，在上面的例子里，当卡雷尔完成了它的工作，所有的组成塔的 21 个蜂鸣器都应该被叠落在第一大街和第九大道相交的街角上，像下面这样：

（请注意：在这个问题中，题目假设在一开始，卡雷尔的蜂鸣器收藏包里蜂鸣器的数量为零，没有任何蜂鸣器--译者注。）
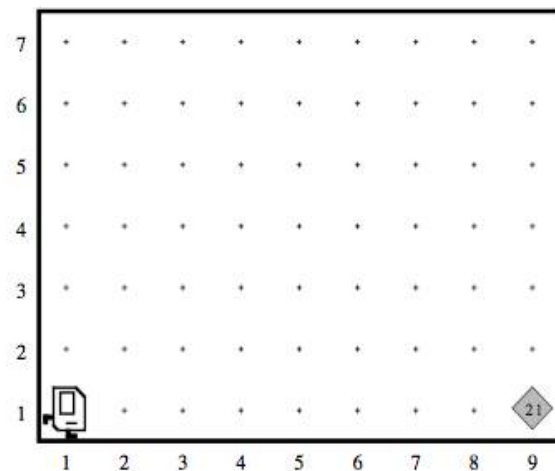
The key to solving this problem is to decompose the program in the right way. This task is more complex than the others you have seen, which makes choosing appropriate subproblems more important to obtaining a successful solution.

（请注意：在这个问题中，题目假设在一开始，卡雷尔的蜂鸣器收藏包里蜂鸣器的数量为零，没有任何蜂鸣器--译者注。）

正确的分解方案是解决这个问题的关键。这个任务比你曾经见过的更复杂，选择合适的子问题比获得一个成功解决方案更重要。

The principle of top-down design
The key idea in stepwise refinement is that you should start the design of your program from the top, which refers to the level of the program that is conceptually highest and most abstract. At this level, the beeper tower problem is clearly divided into three independent phases. First, Karel has to collect all the beepers. Second, Karel has to deposit them on the last intersection. Third, Karel has to return to its home position. This
conceptual decomposition of the problem suggests that the run method for this program will have the following structure:

自上而下的设计原则

逐步求精的关键思想是，你应该从顶部开始你程序的设计，顶部是程序最概念化，最抽象的层次。在这个层次上，蜂鸣器塔问题被清楚地分为三个独立的阶段。第一步，卡雷尔收集所有的蜂鸣器，第二步，卡雷尔把它们存到最后一个交叉街角，第三步，卡雷尔回到它家的位置。这个问题的概念分解表明，此程序的 run 方法（从此处开始，"method" 将被翻译为"方法"--译者注）将具有以下结构：

```
public void run() {
    collectAllBeepers();
    dropAllBeepers();
    returnHome();
}
```

At this level, the problem is easy to understand. Of course, there are a few details left over in the form of

methods that you have not yet written. Even so, it is important to look at each level of the decomposition and convince yourself that, as long as you believe that
the methods you are about to write will solve the subproblems correctly, you will then have a solution to the problem as a whole.

在这个层次上，问题非常容易理解。当然，在上面的形式里，遗留了些你没有写的细节。即便如此，重要的是要查看每一个层次的分解，并说服自己，只要你相信你写下的方法可以正确的解决子问题，你就有了一个问题的整体解决方案。

Refining the first subproblem
Now that you have defined the structure for the program as a whole, it is time to move on to the first subproblem, which consists of collecting all the beepers. This task is itself more complicated than the simple problems from the preceding chapters. Collecting all the beepers means that you have to pick up the beepers in every tower until you get to the final corner. The fact that you need to repeat an operation for each tower suggests that you need a while loop here.

精解第一个子问题

现在你已经定义了一个程序的整体结构，到了把时间转到第一个子问题的时候了，这个子问题是收集所有的蜂鸣器。这个任务本身就比前几章简单的问题更复杂。收集所有的蜂鸣器意味着，你不得不捡起每个塔的蜂鸣器，直到你到达最后的街角。实际上，在这儿你需要一个 "while" 循环，来表达你需要在每个塔做的重复操作。

But what does this while loop look like? First of all, you should think about the conditional test. You want Karel to stop when it hits the wall at the end of the row. Thus, you want Karel to keep going as long as the space in front is clear. Thus, you know that the collectAllBeepers method will include a while loop that uses the frontIsClear test. At each position, you want Karel to collect all the beepers in the tower beginning on that corner. If you give that operation a name, which might be something like collectOneTower, you can go ahead and write a definition for the collectAllBeepers method even though you haven't yet filled in the details.

但是，这个"while"循环看起来是什么样的？首先，你要思考（while 循环）检测的条件。你想让卡雷尔在到达行的终点，碰到墙的时候停下来。因此，你想让卡雷尔在它面前无墙时保持前进。因此，你认为 "collectAllBeepers" 这个方法应该包含一个用 "frontIsClear"作为检测条件的 "while"循环。在塔开始的每一个街角位置，你想让卡雷尔收集这个塔所有的蜂鸣器。如果你想给这个操作一个名字，像"collectOneTower"这样的。你能走上前去，写出"collectAllBeepers"方法的定义，即使你还没有填补这个方法的细节。

You do, however, have to be careful. The code for collectAllBeepers does not look like this:

你这样做了，不过，要小心。"collectAllBeepers"的代码不要看起来像这样：

```
private void collectAllBeepers {
  while (frontIsClear()) {
    collectOneTower();
    move();
  }
}
```

This implementation is buggy for exactly the same reason that the first version of the general RoadRepairKarel from the previous chapter failed to do its job. There is a fencepost error in this version of the code, because Karel needs to test for the presence of a beeper tower on the last avenue. The correct implementation is

这个实现是错误的，恰恰和前面章节里通用的"RoadRepairKarel"第一版工作失败是相同的原因（这里指第34页中后半页举的例子的错误--译者注）。在这一版的代码里有一个栅栏柱错误，因为卡雷尔需要检查最后一个街角存在的蜂鸣器塔。正确的实现是：

```
private void collectAllBeepers {
   while (frontIsClear()) {
     collectOneTower();
     move();
   }
   collectOneTower();
}
```

Note that this method has precisely the same structure as the main program from the RoadRepairKarel program presented at the end of the last chapter. The only difference is that this program calls collectOneTower where the other called checkForPothole. These two programs are each examples of a general strategy that looks like this:

```
while (frontIsClear()) {
   Perform some operation.
   move();
}
Perform the same operation for the final corner.
```

注意，这个方法恰好和在上一章的末尾提交的，"RoadRepairKarel"的主程序有相同的结构。唯一的不同是在这个程序里叫"collectOneTower"，另一个程序里叫"checkForPothole"。这两个程序里每个例子的整体思路看起来像这样：

```
while (frontIsClear()) {
   执行一些操作.
   move();
}
在最后一个街角执行同样的操作.
```

You can use this strategy whenever you need to perform an operation on every corner as you move along a path that ends at a wall. If you remember the general structure of this strategy, you can use it whenever you encounter a problem that requires such an operation. Reusable strategies of this sort come up frequently in programming and are referred to as programming idioms or patterns. The more patterns you know, the easier
it will be for you to find one that fits a particular type of problem.

当你沿着路走到墙跟，需要在每一个街角执行这个操作时，你就能使用这个结构。如果你还记得这个程序思路的总体结构，当你遇到问题，需要这样的操作时，你就可以用到它。这种在编程中经常出现的，可重复使用的策略，被称为"编程的习惯用法"或"模式"。你知道的模式越多，你找到一个适合特定问题的模式就越容易。

Coding the next level
Even though the code for the collectAllBeepers method is itself complete, you can't actually execute the program until you solve the collectOneTower subproblem. When collectOneTower is called, Karel is either standing at the base of a tower of beepers or standing on an empty corner. In the former case, you need to collect the beepers in the tower. In the latter, you can simply move on. This situation sounds like an application for
the if statement, in which you would write something like this:

下一层编码

尽管"collectAllBeepers"这个方法的自身代码完成了，但你不能实际执行这个程序，直到你解决了"collectOneTower" 这个子问题。当 "collectOneTower"被调用时，卡雷尔或是站在一个蜂鸣器塔的底部，或是站在一个空街角上。在前者的情况下，你需要收集这个塔里的蜂鸣器。在后者的情况下，你就简单的向前移动一下。这种情况听起来像是一个"if"语句的应用，你就写下一些像这样的东西：

```
if (beepersPresent()) {
    collectActualTower();
}
```

Before you add such a statement to the code, you should think about whether you need to make this test. Often, programs can be made much simpler by observing that cases that at first seem to be special can be treated in precisely the same way as the more general situation. In the current problem, what happens if you decide that there is a tower of
beepers on every avenue but that some of those towers are zero beepers high? Making use of this insight simplifies the program because you no longer have to test whether there is a tower on a particular avenue.

在你加入一条这样的语句到代码里之前，你应该想想你是否需要做检测。通常，通过将"开始时的状况"这一特定状况作为普遍状况对待，程序可以被做的更简单些。在当前问题里，如果你认为在每个街角都有一座蜂鸣器塔，但有些塔的高度是零，这时会发生什么？运用这种检测，简化了程序，因为你不再需要在每个实际的街角都去（从最下面到最上面）检测是否存在一座塔（这段话的意思是，按照程序要求，在每个街角都要检查是否有塔，现在用检测塔的最底层是否存在蜂鸣器来检测这个街角是否存在蜂鸣器塔。因为塔没有第一层也就不可能有第二层。这样就简化了检测程序--译者注）。

The collectOneTower method is still complex enough that an additional level of decomposition is in order. To collect all the beepers in a tower, Karel needs to undertake the following steps:

1. Turn left to face the beepers in the tower.
2. Collect all the beepers in the tower, stopping when no more beepers are found.
3. Turn around to face back toward the bottom of the world.
4. Return to the wall that represents the ground.
5. Turn left to be ready to move to the next corner.

"collectOneTower" 这个方法还是足够复杂，额外的分层是必要的。为了收集一座塔里所有的蜂鸣器，卡雷尔需要采取以下步骤：

1.向左转，面向这座蜂鸣器塔。

2.收集塔里所有的蜂鸣器，直到塔里没有蜂鸣器了，才停下来。
3.向后掉头，面向卡雷尔世界的底部。
4.返回到构成地面的墙前。
5.向左转，准备移动到下一个街角。

Once again, this outline provides a model for the collectOneTower method, which looks like this:

```
private void collectOneTower() {
   turnLeft();
   collectLineOfBeepers();
   turnAround();
   moveToWall();
   turnLeft();
}
```

再一次，这个纲要提供了一个 "collectOneTower" 方法的模型，它看起来像这样：

```
private void collectOneTower() {
   turnLeft();
   collectLineOfBeepers();
   turnAround();
   moveToWall();
   turnLeft();
}
```

Preconditions and postconditions
The turnLeft commands at the beginning and end of the collectOneTower method are both critical to the correctness of this program. When collectOneTower is called, Karel is always somewhere on 1st Street facing east. When it completes its operation, the program as a whole will work correctly only if Karel is again facing east at that same corner. Conditions that must be true before a method is called are referred to as
preconditions; conditions that must apply after the method finishes are known as postconditions.

前置条件和后置条件

在"collectOneTower"这个方法中，在开头和结尾的"turnLeft" 命令对这个方案的正确性至关重要。
当 "collectOneTower" 被调用的时候，卡雷尔总是面向东在第一大街的某个点上。当它（这里指
"collectOneTower"--译者注）完成它的操作后，只有当卡雷尔再次面向东，站在同一个街角上，这
个程序作为一个整体才是运行正确的。那些在一个方法被调用前，必需满足的条件被称为"前置条
件"；那些在一个方法完成后，必需达成的条件被称为"后置条件"。

When you define a method, you will get into far less trouble if you write down exactly what the pre- and postconditions are. Once you have done so, you then need to make sure that the code you write always leaves the postconditions satisfied, assuming that the preconditions were satisfied to begin with. For example, think about what happens if you call collectOneTower when Karel is on 1st Street facing east. The first turnLeft command leaves Karel facing north, which means that Karel is properly aligned with the column of beepers representing the tower. The collectLineOfBeepers method—which has yet to be written but nonetheless performs a task that you understand conceptually— simply moves without

turning. Thus, at the end of the call to collectLineOfBeepers,
Karel will still be facing north. The turnAround call therefore leaves Karel facing south. Like
collectLineOfBeepers, the moveToWall method does not involve any turns but instead simply moves until
it hits the boundary wall. Because Karel is facing south, this boundary wall will be the one at the bottom
of the screen, just below 1st Street. The final turnLeft command therefore leaves Karel on 1st Street
facing east, which satisfies the postcondition.

当你定义一个方法，如果你准确的写下什么是前置和后置条件，你将惹上很大的麻烦。一旦你这
样做了，你需要确认：你写下的代码（运行完毕）总能满足后置条件，假设开始时总能满足前置
条件。例如，思考一下当卡雷尔面向东，站在第一大街上时，如果你调用"collectOneTower"会发
生什么。首先，"turnLeft"命令让卡雷尔面向北，这意味着卡雷尔恰好面对着代表塔的蜂鸣器柱子。
"collectLineOfBeepers"方法--也是你写的，只管执行一个你理解的概念上的任务--简单的移动不转
动。因此，在调用"collectLineOfBeepers"结束时，卡雷尔仍然面向北。因此"turnAround"调用让卡
雷尔面向南。类似"collectLineOfBeepers"，"moveToWall"方法不调用任何转向命令，而是简单的移
动，直到它碰到墙。因为卡雷尔是面向南的，这面边界墙将在屏幕的底部的某一处，第一大街的
下面。最后的"turnLeft"命令让卡雷尔面向东站在第一大街上。这就满足了后置条件。

Finishing up
Although the hard work has been done, there are still several loose ends that need to be resolved. The
main program calls two methods—dropAllBeepers and returnHome— that are as yet unwritten. Similarly,
collectOneTower calls collectLineOfBeepers and moveToWall. Fortunately, all four of these methods are
simple enough to code without any further decomposition, particularly if you use moveToWall in the
definition of
returnHome. The complete implementation appears in Figure 9.

即将完成

尽管已经做了艰苦的工作，还有一些细枝末节需要被解决。主程序调用的两个方
法--"dropAllBeepers" 和"returnHome"还没有写。同样，"collectOneTower"调用的
"collectLineOfBeepers"和"moveToWall"（也没有写）。幸好，这四个方法足够简单，不需要进行任
何进一步分解，特别是如果你在定义"returnHome"时使用"moveToWall"。完整的实现显示在
Figure 9。

Figure 9. Program to solve the collect towers of beepers

```
/*
 * File: BeeperCollectingKarel.java
 * -------------------------------
 * The BeeperCollectingKarel class collects all the beepers
 * in a series of vertical towers and deposits them at the
 * rightmost corner on 1st Street.
 */
import stanford.karel.*;
public class BeeperCollectingKarel extends SuperKarel {
/**
 * Specifies the program entry point.
 */
  public void run() {
    collectAllBeepers();
    dropAllBeepers();
```

```
      returnHome();
   }
/**
 * Collects the beepers from every tower by moving along 1st
 * Street, calling collectOneTower at every corner.  The
 * postcondition for this method is that Karel is in the
 * easternmost corner of 1st Street facing east.
 */
   private void collectAllBeepers() {
      while (frontIsClear()) {
         collectOneTower();
         move();
      }
      collectOneTower();
   }
/**
 * Collects the beepers in a single tower. When collectOneTower
 * is called, Karel must be on 1st Street facing east.  The
 * postcondition for collectOneTower is that Karel must again
 * be facing east on that same corner.
 */
   private void collectOneTower() {
      turnLeft();
      collectLineOfBeepers();
      turnAround();
      moveToWall();
      turnLeft();
   }
/**
 * Collects a consecutive line of beepers. The end of the beeper
 * line is indicated by a corner that contains no beepers.
 */
   private void collectLineOfBeepers() {
      while (beepersPresent()) {
         pickBeeper();
         if (frontIsClear()) {
            move();
         }
      }
   }
/**
 * Drops all the beepers on the current corner.
 */
   private void dropAllBeepers() {
      while (beepersInBag()) {
         putBeeper();
      }
   }
/**
 * Returns Karel to its initial position at the corner of 1st
 * Avenue and 1st Street, facing east.  The precondition for this
 * method is that Karel must be facing east somewhere on 1st
```

```
 * Street, which is true at the conclusion of collectAllBeepers.
 */
  private void returnHome() {
    turnAround();
    moveToWall();
    turnAround();
  }
/**
 * Moves Karel forward until it is blocked by a wall.
 */
  private void moveToWall()
  {
    while (frontIsClear()) {
      move();
    }
  }
}
```

Chapter 5
Algorithms

Although top-down design is a critical strategy for programming, it cannot be applied mechanically without thinking about problem-solving strategies. Figuring out how to solve a particular problem by computer generally requires considerable creativity. The process of designing a solution strategy is traditionally called algorithmic design.

第五章

算法

尽管自上而下设计是编程的一个关键思想，但没有解决问题的实际思路，它不可能被生搬硬套。弄清楚如何运用电脑解决特定问题，通常需要相当大的创造力。设计一个解决（问题）思路的过程，通常被称为"算法设计"。

The word algorithm comes from the name of a ninth-century Persian mathematician, Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî, who wrote an influential treatise on mathematics. Today, the notion of an algorithm has been formalized so that it refers to a solution strategy that meets the following conditions:

• The strategy is expressed in a form that is clear and unambiguous.
• The steps in the strategy can be carried out.
• The strategy always terminates after a finite number of steps.

算法这个词来自于九世纪波斯数学家的名字,Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî,他写了一篇有影响的数学论文。今天，算法的概念已经被正式确定，一个解决思路必需满足下面的条件：

该思路采用一个清晰的，毫不含糊的表达形式。
该思路的每一步都是可行的。
在有限的步骤之后，该思路总是会终止的。（不能是死循环--译者注）

You will learn much more about algorithms as you continue your study of programming, but it is useful to look at a few simple algorithms in Karel's world.

当你继续你的编程学习时，关于算法你将学到更多的东西，但是，看看几个在卡雷尔的世界里的简单算法是有用的。

Solving a maze
As an example of algorithmic design, suppose that you wanted to teach Karel to escape from a maze. In Karel's world, a maze might look like this:

解决迷宫

作为算法设计的一个例子，假设你想教卡雷尔从迷宫里逃出来。在卡雷尔的世界里，一个迷宫可能看起来像这样：
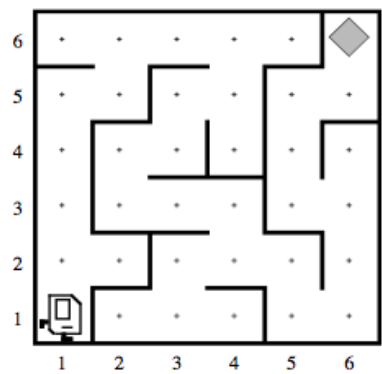
The exit to the maze is marked by a beeper, so that Karel's job is to navigate the corridors of the maze until it finds the beeper indicating the exit. The program, however, must be general enough to solve any maze, and not just the one pictured here.

迷宫的出口被一个蜂鸣器标记出来，所以卡雷尔的工作是穿过这些迷宫的走廊，直到它找到代表出口的蜂鸣器。（被编写的走迷宫）程序必需足够通用解决任何的迷宫，而不仅仅是图上这个。

There are several strategies you could use for solving such a maze. When Theseus needed to escape from the Labyrinth of Crete, he adopted—at the suggestion of King Minos's daughter Ariadne, whom Theseus promptly abandoned on the next island he reached—the strategy of unwinding a ball of string as he explored the maze. You could devise a similar strategy for Karel, in which beepers serve the same function.

这里有几个思路你可以用来解决像迷宫这样的问题。当特修斯需要逃离克里特迷宫时，他采用米诺斯国王的女儿阿里阿德涅的建议，（阿里阿德涅）后来被特修斯遗弃在他登陆的下一个岛上，他探索迷宫时，一路退下的线团的线（相关神话背景 http://wenwen.soso.com/z/q117565047.htm--译者注）（西方有句成语"阿里阿德涅的线"，用来比喻解决问题的方法--译者注）。你可以为卡雷尔设计一个类似的策略，它的蜂鸣器可以提供同样的功能。

For most mazes, however, you can use a simpler strategy called the right-hand rule, in which you begin by putting your right hand on the adjacent wall and then go through the maze without ever taking your hand off the wall. Another way to express this strategy is to proceed through the maze one step at a time, always taking the rightmost available path.

但是，对于大多数的迷宫，你可以使用一个被称为"右手规则"的简单策略，在入口处你把右手放在旁边的墙上，通过保持右手始终在墙上，你就能通过迷宫。这个策略的另外一种表达就是通过迷宫的时候，每次一步，总是靠最右侧的路走。

Figure 10. Program to solve a maze （解决迷宫的程序）

/*
/*

```
 * File: MazeRunningKarel.java
 * --------------------------
 * This program extends Karel so that it can solve a maze
 * using the right-hand rule.
 */
import stanford.karel.*;
public class MazeRunningKarel extends SuperKarel {
  public void run() {
    while (noBeepersPresent()) {
      turnRight();
      while (frontIsBlocked()) {
        turnLeft();
      }
      move();
    }
  }
}
```

You can easily write a Karel program to apply the right-hand rule. The program in Figure 10, for example, expresses the algorithm for the right-hand rule in a particularly compact form. You should work through the logic of this algorithm and convince yourself that it indeed accomplishes the task. It is important to note that the code that implements an algorithm may not be very complicated. Indeed, coming up with the right
algorithm often leads to extremely simple code.

你可以容易的写出一个卡雷尔程序来应用右手规则。在 Figure 10 里的程序，例如，用一种特别紧凑的形式表达了实现右手规则的算法。你应该利用这个算法的逻辑编写程序，你自己确保它能实际完成任务。重要的是要注意，实现一个算法的代码可能不是非常复杂。事实上，伴随着正确的算法，经常带来非常简单的代码。

Doubling the number of beepers
Another programming task that leads to interesting algorithmic choices is the problem of getting Karel to double the number of beepers on a corner. For example, suppose that Karel starts out in the world

加倍蜂鸣器

另外一个带来有趣的算法选择的编程任务是，让卡雷尔把一个街角上的蜂鸣器数量加倍的问题。假设卡雷尔开始在这样一个世界，

where there are some number of beepers—in this case four—on the corner of 1st Street and 2nd Avenue and an infinite number of beepers in Karel's beeper bag. The goal in this problem is to write a method doubleBeepers that doubles the number of beepers on the current square. Thus, if you execute the method on the world shown in the preceding diagram, the final state of the world should look like this:

那里有一定数目的蜂鸣器-在这个例子里是四个-在第一大街和第二大道的交点上，而且有无限的蜂鸣器在卡雷尔的包里。这个问题的目标是写一个叫"doubleBeepers"的方法，它能把当前广场上蜂鸣器的数目翻倍。因此，如果你在前面的图上所示的世界里执行这个方法，卡雷尔世界的最终状态看起来应该像这样：

```
public void run() {
  move();
  doubleBeepers();
  move();
}
```
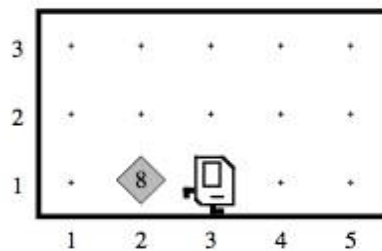

The program should be general enough to work for any number of beepers. For example, if there had originally been 21 beepers on the corner of 1st Street and 2nd Avenue, the program should end with 42 beepers on that corner.

这个应该足够通用于任何数目的蜂鸣器。例如，如果在第一大街和第二大道交点的街角上有 21 个蜂鸣器，程序最终应让这个街角有 42 个蜂鸣器。

Writing the doubleBeepers method is harder than it initially appears. Your first step is to devise an algorithmic strategy to solve the problem. You can't start by picking up all the beepers on the corner, because you would then have no way of telling how many beepers to put down. As with most algorithms in Karel's world, you need to process the beepers one at a time. You can pick one up from the corner, but you then have to keep
track somehow of the fact that you have to add two beepers to the result.

写 "doubleBeepers"方法比它开始看起来要困难些。你的第一步是制定一个解决问题的算法。你不能从把这个街角所有的蜂鸣器都捡起来开始，因为你没有办法说出要放下多少蜂鸣器。与大多数在卡雷尔世界的算法一样，你需要一次处理这些蜂鸣器中的一个。你能从这个街角捡起一个，但你就必须保持跟踪，这种你必须添加两个蜂鸣器的结果的事实。

The easiest strategy to devise involves the use of a temporary storehouse on some corner that is initially empty, such as the corner at 1st Street and 3rd Avenue. If every time you pick up a beeper from the original pile on 2nd Avenue you put down two beepers in the storehouse on 3rd Avenue, you will have twice the original number of beepers when the first pile is exhausted. Thus, you can create the correct value in the storehouse by calling the following method:

```
private void doubleIntoStorehouse() {
  while (beepersPresent()) {
```

```
      pickBeeper();
      move();
      putBeeper();
      putBeeper();
      turnAround();
      move();
      turnAround();
   }
}
```

最容易的思路是制定某个街角作为一个临时仓库使用，它在开始时是空的，例如在第一街和第 3 大道交点的街角。如果每次你从原来的第二大道拿起一个蜂鸣器，你会放下两个蜂鸣器在第三大道的仓库，当第一堆空了，你就会有两倍数目的蜂鸣器（在仓库里）。因此，你能通过调用下面的方法在仓库里创造出正确的数目。

```
private void doubleIntoStorehouse() {
   while (beepersPresent()) {
      pickBeeper();
      move();
      putBeeper();
      putBeeper();
      turnAround();
      move();
      turnAround();
   }
}
```

The precondition for this method is that Karel is standing on a corner containing a pile of N beepers facing a corner with no beepers. The postcondition is that Karel winds up in its original position with no beepers on that corner but 2N beepers on the corner Karel is facing.

这个方法的前置条件是卡雷尔站在有一摞 N 个蜂鸣器的街角上，面向一个没有蜂鸣器的街角。后置条件是卡雷尔拉风的站在它的原始位置，但没有蜂鸣器在这个街角上，但是它面对的街角有 2N 的蜂鸣器。

This method does the interesting algorithmic work, but does not entirely satisfy the constraints of the problem as stated because the final pile of beepers is not on the original square. To get it there, you need to implement a similar method that simply transfers the pile back to the adjacent square. This method has almost exactly the same structure,
except that it deposits only one beeper for each one it collects. If you design a transferBeepersBack method to work with the same precondition as that used for doubleIntoStorehouse, it will look like this:

```
private void transferBeepersBack() {
   while (beepersPresent()) {
      pickBeeper();
      move();
      putBeeper();
      turnAround();
      move();
      turnAround();
```

```
    }
}
```

这个方法做了有趣的算法工作，但并没有完全满足所述问题的条件，因为最后这摞蜂鸣器不在原始的街角。为了完成它，你需要执行一个简单的方法，这个方法简单的把这摞蜂鸣器移回相邻的街角。此方法（和 doubleIntoStorehouse 方法）具有几乎完全相同的结构，除了每次它捡起一个蜂鸣器后只放下一个蜂鸣器。如果你为这个工作设计了一个 "transferBeepersBack"方法，和 "doubleIntoStorehouse"使用相同的前置条件，它看起来像这样：

```
private void transferBeepersBack() {
    while (beepersPresent()) {
        pickBeeper();
        move();
        putBeeper();
        turnAround();
        move();
        turnAround();
    }
}
```

The doubleBeepers method itself then consists of the following code:

"doubleBeepers "方法本身包含下面的代码：

```
private void doubleBeepers() {
    doubleIntoStorehouse();
    move();
    turnAround();
    transferBeepersBack();
    move();
    turnAround();
}
```

This strategy, however, is not the only one you might use. In many cases, there are algorithms that work much better than the obvious ones, although they are often difficult to discover. Many such algorithms depend on sophisticated programming techniques that you will encounter later in your study of computer science. For example, the doubleBeepers problem can be solved quite easily if you use a technique called recursion, which is simply the process of having a method call itself. The following implementation of doubleBeepers gets the job done without needing a storehouse or any moving around:

```
private void doubleBeepers() {
    if (beepersPresent()) {
        pickBeeper();
        doubleBeepers();
        putBeeper();
        putBeeper();
    }
}
```

然而，这个思路，不仅在这个例子里你可以使用。在许多情况下，有些算法让工作有明显的改善，尽管通常都难以发现。许多这样的算法依赖于先进的编程技术，在你完成计算机科学的学习后，您会领教到这些算法。例如，如果你使用了一个叫"递归"的技术，"doubleBeepers"问题可以被相当容易的解决，它是一个简单的让方法调用它本身的过程。用以下代码执行"doubleBeepers"的工作，不需要一个仓库或来回移动。

```
private void doubleBeepers() {
  if (beepersPresent()) {
    pickBeeper();
    doubleBeepers();
    putBeeper();
    putBeeper();
  }
}
```

Although it is fun to try and figure out what this program is doing, you shouldn't worry at this point if you find it hard to understand. The point of showing this solution is simply to demonstrate that there are many different algorithms for solving problems, some of which can be very compact and efficient. As you study computer science, you will learn a great deal more about algorithmic techniques and gain the skills you need to write this type of program on your own.

尽管尝试找出这个程序怎么运行是有趣的，但如果你很难理解这个，你不需要焦虑。在这里给出这个解决方法只是为了简单的展示，解决问题有许多不同的算法，其中一些可能会非常紧凑，高效。当你学习了计算机科学，你会学到很多关于算法的技术，获得更多的，当你需要自己写这种类型的程序时的技能。

第六章

超级卡雷尔

As it comes from the factory, Karel is a bit on the boring side. The world—reflecting the nature of hardware at the time Karel was invented—is entirely black and white. Moreover, Karel always behaves in a strictly deterministic fashion. Each of these limitations makes it harder to program Karel to perform exciting tasks. To make it easier to write more interesting Karel programs, the stanford.karel package includes a
SuperKarel that includes several new features. Those features are described in the sections that follow.

当它从工厂里出来的时候，卡雷尔枯燥了点。卡雷尔的世界反映了当卡雷尔被发明时，硬件的性质，完全是黑色和白色。此外，卡雷尔总是遵从严格确定的举止。这些限制很难让卡雷尔程序执行激动人心的任务。为了让写更有趣的卡雷尔程序变得容易些，"stanford.karel"包里包含了具有几个新特性的"SuperKarel"。这些特性在接下来的这章里描述。

The turnRight and turnAround methods
As you already know from Chapter 2, the SuperKarel class includes definitions for turnRight and turnAround. Although it certainly easy to define these methods on your own, it is inconvenient to do so in every Karel program. Moreover, the SuperKarel implementations of these methods tie into the internal workings of Karel to implement these operations more efficiently. If you use turnRight in a SuperKarel subclass, your
program will make an immediate right turn and not go through the process of turning left three times.

"turnRight"和"turnAround"方法

从第二章已经知道了，"SuperKarel"类包含"turnRight"和"turnAround"的定义。尽管这些方法你自己也很容易定义,但在每个程序里都做这个不太顺手。此外，"SuperKarel"是在卡雷尔内部运作这些方法，执行起来更有效率。如果你在"SuperKarel"子类里使用"turnRight"，你的程序（里的卡雷尔）将马上做出右转，不经过三次左转的过程。

Using color
The SuperKarel class allows Karel to paint the corner on which it is standing using the instruction

paintCorner(color);

"SuperKarel"类允许卡雷尔使用这个指令给它站立的街角画上颜色 ：

paintCorner(color);

The value enclosed in the parentheses—which is called an argument in the terminology of programming languages—may be any of the following:
BLACK
BLUE
CYAN
DARK_GRAY
GRAY

GREEN
LIGHT_GRAY
MAGENTA
ORANGE
PINK
RED
WHITE
YELLOW
null

在封闭括号里的值，它在编程语言术语里被称为参数，可以是下面的这些：

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

The null color value indicates that a corner has no color, which means that the little cross in the center shows through. When you create a new Karel world, all corners initially have null as their color.

"null"颜色值表示这个街角没有颜色，这意味着中心的小十字会显露出来。当你创造了一个新的卡雷尔的世界，所有的街角初始用"null"作为它们的颜色。

SuperKarel also understands a new condition cornerColorIs(color), which is true if the current corner has been painted with the specified color. For example, you could paint the current corner red by executing the instruction paintCorner(RED);

"SuperKarel"也明白一个新的判断语句"condition cornerColorIs(color)"，如果当前的街角被画上指定的颜色，它会返回一个真值。例如，你可以通过执行"paintCorner(RED)"指令在当前街角画上红色。

paintCorner(RED);

and later perform operations only on red corners by using the following if statement:

if (cornerColorIs(RED)) {
    Statements to execute only if the corner is painted red
}

事后可以通过使用"if"语句仅仅对红色的街角执行一些操作。

```
if (cornerColorIs(RED)) {
   仅仅当街角被画上红色，才会执行的一些操作
}
```

Random behavior
SuperKarel also defines a new condition random, which is true half the time, but in an unpredictable way.
For example, if you execute the statements

```
if (random()) {
   paintCorner(YELLOW);
} else {
   paintCorner(MAGENTA);
}
```

Karel will paint the current corner yellow or magenta, each with equal probability.

随机行为

超级卡雷尔也有定义个新的"随机" 判断条件，它有一半的时间会返回真值，但是没有方法可以预测。例如，如果你执行了下面的语句:

```
if (random()) {
   paintCorner(YELLOW);
} else {
   paintCorner(MAGENTA);
}
```

卡雷尔将把当前的街角刷上黄色或品红色，每种颜色都有相等的可能性。

If you need greater control over how often Karel executes a random event, the random condition takes an argument, which is the a number specifying the probability of the condition returning true. As in statistics, probabilities are numbers between 0.0 and 1.0, where 0.0 indicates that the condition will always be false and 1.0 indicating that it will
always be true. If, for example, you wanted to have Karel put down a beeper 25 percent of the time, you could use the following code:

```
if (random(0.25)) {
   putBeeper();
}
```

如果你需要更大程度上，经常性的控制卡雷尔执行一个随机事件，"随机"条件有个参数，这个参数是一个数字，它指明了条件返回真值的概率。按照统计学，概率是些在 0.0 到 1.0 的数字，这里 0.0 表示条件测试将总是返回假值，1.0 表示条件测试将永远返回真值。假设，例如，你想让卡雷尔在 25%的时间里放下一个蜂鸣器，你可以使用下面的代码：

```
if (random(0.25)) {
   putBeeper();
}
```

Logical operations
As you write more sophisticated Karel programs, you will discover that it is sometimes difficult to
```

express certain conditional tests whose English equivalents include conjunctions like and and or. As an example, try to write a Karel while statement that moves Karel forward until it is either blocked by a wall or encounters a beeper. To make it easier to write interesting programs, the Karel language allows you to use the following
logical operators, which are actually part of Java and not a SuperKarel extension:

&& Equivalent to the English word and.
|| Equivalent to the English word or (in the formal sense of either or both).
! Equivalent to the English word not.

逻辑操作

当你编写更复杂的卡雷尔程序，你将发现，有时表达某种条件测试，这种测试的英文等值包括像连词 and 和 or ，是困难的。作为一个例子，试图写一个卡雷尔语句，让卡雷尔向前移动，直到它被墙挡住或遇到一个蜂鸣器。为了让写这种有趣的程序变得更容易，卡雷尔语言允许你使用下面的逻辑操作，它实际是 Java 的一部分，不是"SuperKarel"扩展。

&& 相当于英文单词"与"
|| 相当于英文单词"或"（在正式意义上的一方或双方）
! 相当于英文单词"非"

With these operators, it is easy to write the while statement suggested earlier in this section, because you can combine the conditions into a single test:

```
while (frontIsClear() && noBeepersPresent()) {
  move();
}
```

用这些操作，写下这章上面建议的"while"表达很容易，因为你能综合多个条件在一个单独的测试语句里。

```
while (frontIsClear() && noBeepersPresent()) {
  move();
}
```

The fact that these operators work in Karel programs reveals a notable fact about the way such programs are implemented. The Karel programs that you write turn out to be simply Java programs in disguise. There is no separate Karel language; everything that you've seen in Karel is actually just part of standard Java or implemented using standard
Java as part of one of the classes in the stanford.karel package. While this strategy makes the Karel simulator much easier to implement and means that you will be using the same tools that you will use throughout the quarter, it does have a downside. The logical operators &&, ||, and ! are not the only pieces of standard Java that you might incorporate into a Karel program. Given the way Karel is implemented, you could include
anything from standard Java in a Karel program, and the Java compiler would not complain at all. Doing so, however, defeats the purpose of Karel, which is intended to provide a simple platform for learning programming. So even though the Java compiler won't complain if you use more advanced Java structures, your section leader will. Acceptable Karel programs must limit themselves to the features described in this book.

实际上，这些在卡雷尔程序里的逻辑操作工作，揭示了这些程序的实施方式是值得注意的事实。你写的卡雷尔程序，简单的变成是 Java 程序的变相。没有单独的卡雷尔语言；你在卡雷尔里看到的一切，实际是斯坦福 Java 或在 stanford.karel 包里使用标准的 Java 作为类的一部分，这一策略使卡雷尔模拟器更容易实现，它也意味着在整个学期，你将使用相同的工具，这是它的缺点。（这句话译者没有明白--译者注）。逻辑操作符"&&", "||", 和"!"不仅是标准 Java 的一部分，你还可以在卡雷尔程序里引入。鉴于卡雷尔的这种执行方法，你可以引入任何标准 Java 的东西到卡雷尔程序里，Java 编译器都不会抱怨。然而这样做，违背了卡雷尔是为了提供一个简单的平台，学习编程的目的。因此，如果你使用更高级的 Java 结构，即使 Java 编译器不会抱怨，你的老师也会的。可接受的卡雷尔方案必须限制在（仅使用）它自己在这本书中描述的功能。

Appendix A

Karel Reference Card
This appendix defines the structure of the Karel programming language on a single page.

卡雷尔参考卡片

这个附录把卡雷尔编程语言的结构定义列在单独一页上。

Built-in Karel commands:

```
move();
turnLeft();
putBeeper();
pickBeeper();
```
Conditional statements:

Conditional statements:

```
if (condition) {
     statements executed if condition is true
}
if (condition) {
     statements executed if condition is true
} else {
     statements executed if condition is false
}
```

Karel program structure:

```
/*
 *  Comments may be included anywhere in
 *  the program between a slash-star and
 *  the corresponding star-slash characters.
 */
import stanford.karel.*;
/* Definition of the new class */
public class name extends Karel {
    public void run() {
        statements in the body of the method
    }
    definitions of private methods
}
```

Iterative statements:

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

Method definition:

```
private void name () {
     statements in the method body
}
```

Karel condition names:

```
frontIsClear()    frontIsBlocked()
leftIsClear()     leftIsBlocked()
rightIsClear()    rightIsBlocked()
beepersPresent()  noBeepersPresent()
beepersInBag()    noBeepersInBag()
facingNorth()     notFacingNorth()
facingEast()      notFacingEast()
facingSouth()     notFacingSouth()
facingWest()      notFacingWest()
```

New commands in the SuperKarel class:

```
turnRight();
turnAround();
paintCorner(color);
```

New conditions in the SuperKarel class:

```
random()
random(p)
cornerColorIs(color)
```