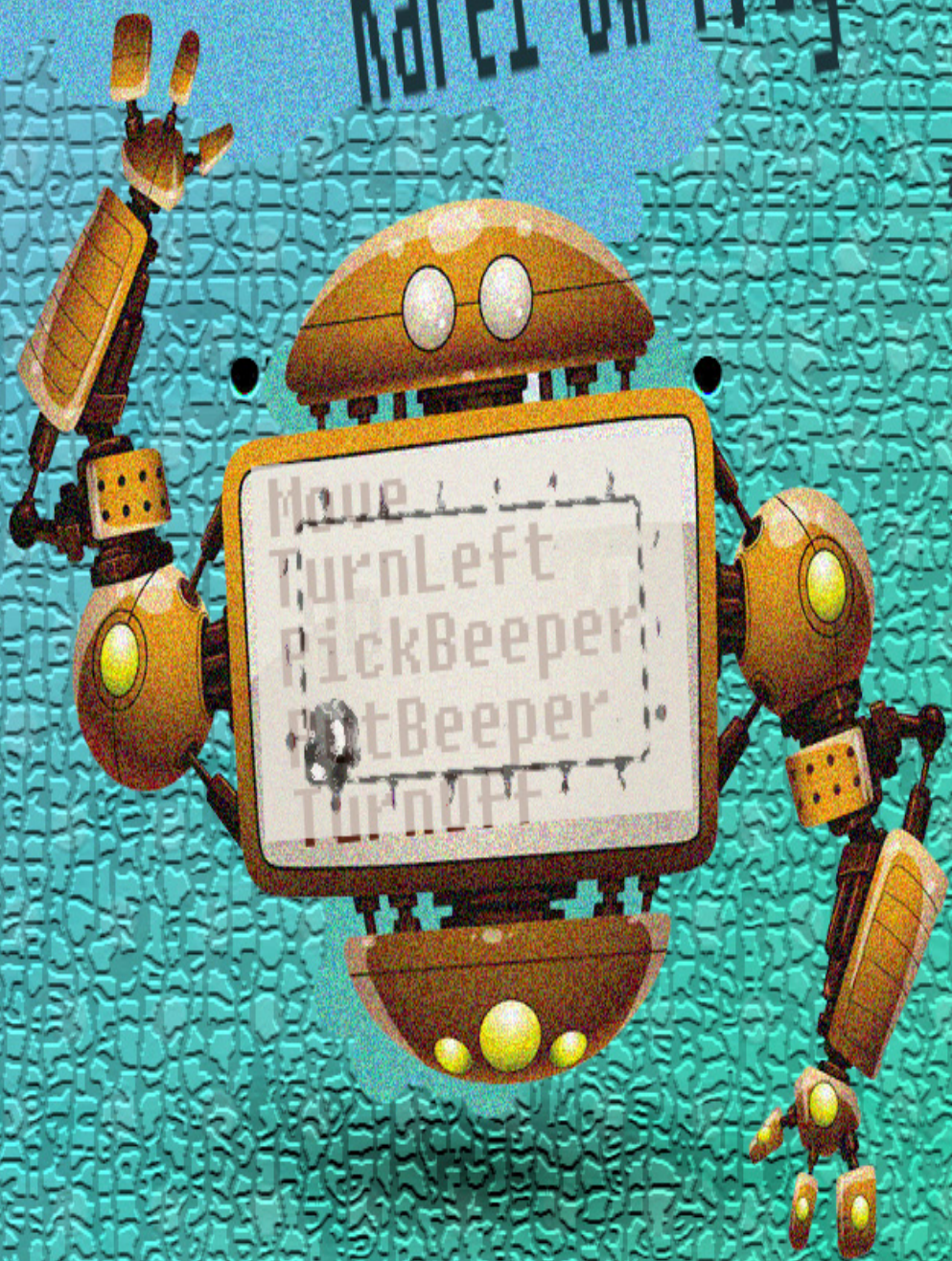




# Karel C# Programming

黃一新  
舒靖 著





# **Karel C#**

## *Programming* λ门

舒喆  
黄一新

编著

Chapter 1-8  
Version 1.0  
Date 20150210

# 目 录

## 1. 欢迎来到 Karel 世界

1.1 准备工作 .....	2
1.2 Karel 程序 .....	2
1.3 Karel 世界 .....	3
1.4 Karel 任务 .....	5

## 2. Karel 基本指令

2.1 第一段 Karel 脚本 .....	7
2.2 Karel 基本指令 .....	9
2.3 错误处理 .....	13
2.4 第一个 Karel 任务 .....	13

## 3. 新方法：扩展 Karel 能力

3.1 如何定义新方法 .....	19
3.2 if .....	24
3.3 逻辑运算 .....	25
3.4 else .....	27
3.5 while .....	28
3.6 Langton 蚂蚁 .....	31

## 4. 表达式和语句

4.1 继续扩展 Karel 指令 .....	38
4.2 for .....	41
4.3 switch case .....	44

## 5. 面向对象设计

5.1 面向对象概念 .....	51
5.2 面向对象原则 .....	54
5.3 如何定义新 Robot 类 .....	62

5.4 扩展 Robot .....	70
5.5 定义 RobotPro 类 .....	80

## 6. 数据结构

6.1 Array 数组 .....	87
6.2 List 列表 .....	91
6.3 Directory 字典 .....	95
6.4 Enumerate 枚举 .....	97
6.5 string 字符串 .....	98
6.6 set 集合 .....	99
6.7 更多的数据结构 .....	99

## 7. 算法

7.1 递归 .....	109
7.2 排序 .....	112
7.3 查找 .....	115
7.4 各种算法 .....	119

## 8. 打造 Karel 世界

8.1 建设场景 .....	128
8.2 场景渲染 .....	136

# 前言

自 1981 年作为 Pascal 语言教学工具的 Karel 机器人诞生以来，计算机的存在方式和运算速度已经今非昔比。计算机编程语言同样在不断发展：从传统的面向过程语言（如 C 语言），继而出现面向对象语言（如 Java 语言），古老的函数式语言（如 Lisp 语言）也在焕发青春。

Karel 机器人在此期间不断的升级换代，陆续作为 Java、Ruby、Python 的语言教学工具，现如今 C#/Scheme 版本的 Karel 机器人也发布了，同时支持的两种语言，分别来自 Microsoft 微软公司和 MIT 麻省理工学院。

C# 语言（下文简称 C#）代表编程语言的现在，这是众多流行桌面软件和网站的主要实现方法。Scheme 语言（一种 Lisp 的简化方言）代表编程语言的过去和未来，过去 MIT 为首一系列高校用其作为的计算机入门语言和算法的学习，而未来 Scheme 所采用的函数式编程思想将大幅度提高软件人员的效率。

本书主要通过 Karel 机器人学习 C#，尽量以简单轻松的文字，讲解 Karel 机器人如何由 C# 来驱动并完成一系列任务，读者无需程序设计基础，可作为学习第一门编程语言之前的开胃菜，亦可将本书作为学习 C# 过程中一个完整的实践环节。

讲解 Scheme 语言的 Karel 机器人，将作为本书的姊妹篇发布。

# 1. 欢迎来到 Karel 世界

## 1.1 准备工作

在我们携手进入 Karel 世界之前，需要将 Karel 程序文件包，下载并解压至我们所拥有的 PC 个人计算机的磁盘上。下载位置可根据个人喜好，建议放在磁盘根目录下或根目录的二级目录中，当然也可以放在桌面上或其他位置，只要我们能够随时找到并且方便使用。

很遗憾，由于我们的 C#/Scheme 的 Karel 程序（下称 Karel 程序）只支持 Windows 系统，所以想在 MacOS 和 Linux 系统中使用 Karel 程序的一部分读者就要失望了，好在 Windows 还是主流操作系统，绝大部分读者都在用。

Karel 程序还需要安装 .NET Framework 4.0 运行环境，这是 Windows 系统上最重要的官方程序运行环境，4.0 并不是最新的版本，Windows 7/8 出厂的时候已经安装好，但使用 Windows XP/Vista 的读者仍然需要在微软官方站点上下载并安装对应（32 位或者 64 位）的 .NET Framework 4.0。要注意的是 x86 和 x64，前者代表古老的 32 位系统，后者则是当前流行的 64 位系统，个人计算机中 32 位至 64 位的变革大概发生在 15 年前，那时 AMD 公司和 Inter 公司分别推出了 64 位处理器。

1. 确定 Windows 操作系统是否安装了 .NET Framework 4.0，没有请先安装
2. 下载并解压 Karel 程序，双击 Lab.Karel.exe

如果顺利打开一个桌面程序窗体，如下图所示，那么就完成了准备工作的第一步——安装 Karel 程序。

接下来我们要选中菜单栏 File -> Language -> C#，C# 选项一旦打钩选中，程序的标题将会出现 Karel-C# 字样，那么就完成了准备工作的第二步——进入 C# 的 Karel 世界。

## 1.2 Karel 程序

容我先描述一下 Karel 程序，也就是准备工作结束时依然打开的程序窗口。由于程序技术的发展，Karel 机器人所在的世界，也有了变化。

在过去，当我们使用其他编程语言操纵 Karel 时，基本上我们需要在专门的代码编辑器中编写好指令，然后编译代码，弹出运行结果就是 Karel 世界。我们所发出的指令和 Karel 世界仿佛跨越了时空。

如今的 Karel 程序，则将代码世界和 Karel 世界巧妙的合而为一，一分为二的存在于程序主体中。左侧写指令，右侧看结果，看上去那么的自然，既省去了安装 C# 集成开发环境的烦恼，也方便了 Karel 程序的传播。

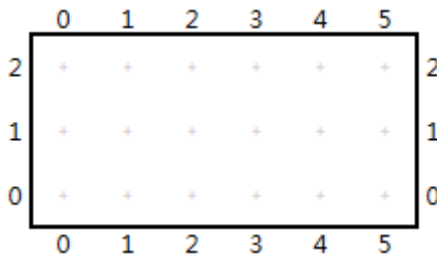
当然有利必有弊，对于 C# 使用者来说，用不到 Visual Studio 这一顶级开发工具，不

能不说是一种缺失。传统的 Karel 开发模式，更贴近于真实开发，故 Karel 程序也提供了基于 Visual Studio 传统开发模式，在本书的最后部分给出具体的操作方法。

## 1.3 Karel 世界

Karel 世界是 Karel 可活动区域的总和，具体表现为 Karel 程序主体右侧的可视部分。Karel 世界的组成元素主要有 Scene、Karel、Street/Avenue/Corner、Wall、Beeper、Mark。

### 1.3.1 Scene



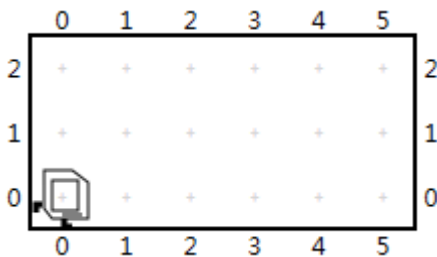
在 Karel 世界中，我们首先能看到的是二维格子空间，称之为 Scene 场景。Scene 由  $n \times n$  的方格组成， $n$  代表正整数。

Scene 有三种生成方法：可由 Karel 程序上菜单栏的 File -> New Scene 按钮生成，可在加载时使用预指令生成，可在运行时使用指令生成。

在 Scene 的四周边缘部分，显示有坐标。坐标分为横纵两类，上下为横，左右为纵，从左下 0 开始计数。故  $n \times n$  的方格，横纵坐标由左至右，由下至上为  $0, 1, 2, \dots, n-2, n-1$ 。

上下左右同样也被可称为上 - 北、下 - 南、左 - 西、右 - 东。

### Karel

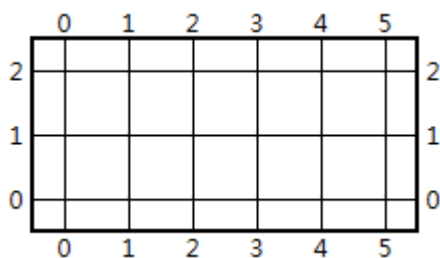


Karel 是 Karel 世界的主角，也是一种基本的 Robot 机器人，能在 Scene 空间内转身移动，并且拥有操控 Beeper 的能力。

Karel 占据一个  $1 \times 1$  格子，有一个正面的朝向，每次移动时，Karel 会沿着这个正面朝向，移动到相邻的格子上。Karel 随身携带了无限多个 Beeper。

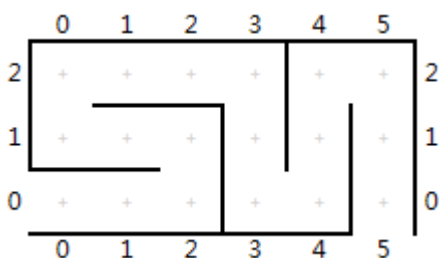
本书中所有的任务和故事，都围绕着 Karel 进行，Karel 将通过一系列的简单指令，完成各种复杂活动。Karel 有着从简单到复杂、近乎无限的扩展能力。我们所要做的就是跟随 Karel 的脚步，不断的去解决更为复杂的问题，在这一过程中领悟推动 Karel 进化的各种编程方法。

### 1.3.2 Street/Avenue/Corner



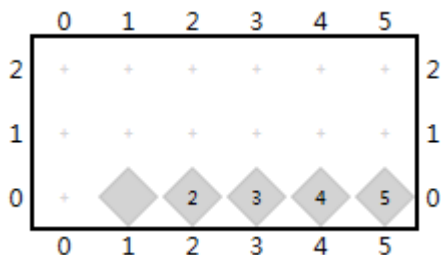
横坐标为 Street 街, 又称为行 Row; 纵坐标为 Avenue 道, 又称为列 Column。横纵街道, 构成了 Karel 的有效活动范围。道和街交叉的地方为 Corner 街角。(x, y) 坐标代表 Karel 所在的街角位置, 我们能够设定 Karel 的初始坐标。为了场景清晰, 默认情况下不显示表示街道的线, 只绘制 “+” 形状的街角。

### 1.3.3 Wall



Wall 墙是空间阻隔, 通常存在于方格的四周, 以阻挡 Karel 的行动。连续的墙可组成迷宫、房间等。通常情况下 Karel 在面向墙时, 无法移动并停留在原地, 而在特定情况下, Karel 可以穿墙而过。默认情况下场景边界处围有一圈墙, 不过即使没有墙 Karel 也不会越过边界。

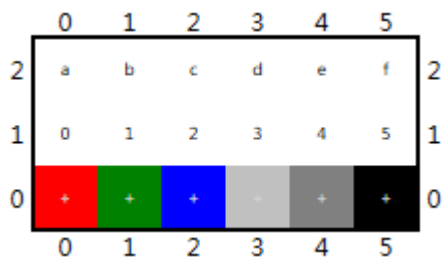
### 1.3.4 Beeper



Beeper 蜂鸣器, 是一种可被 Karel 捡起或放置的能发出声音的装置。Beeper 可在一个位置叠加, 同一方格中 Beeper 个数不同而显示不同数字, 这些不同数字可代表不同物体。默认情况下一个 Beeper 并不显示数字 “1”。

### 1.3.5 Mark





Mark 标记，可在方格中进行设置。Mark 用来标记出一些任务的关键点，支持不同颜色和符号的 Mark。

## 1.4 Karel 任务

根据 Karel 的特征和本书的教学目标，我们在每个章节都设置了一些 Karel 任务，这些任务包括以下几类：

- 寻路任务，如 Karel 离开房间或逃离迷宫；
- Beeper 任务，如改变不同位置 Beeper 的数量；
- 算法任务，如对不同数量 Beeper 进行的排序；
- 综合任务，多种任务的结合。

让我们通过这些任务进入 Karel 世界吧。

## 思考题

- A. 下面哪些方向是 robot 能够朝向的？  
(a) 东北 (b) 东 (c) 北 (d) 164 度 (e) 水平的 (f) 下
- B. Robot 世界中除了 Robot 之外还有什么？
- C. 问题 2 的答案列表中，哪些是 Robot 制造或可更改的？
- D. Robot 世界中的控制点是如何描述 Robot 精确位置的？
- E. 在给定的 Robot 世界中，能有多少 Robot？
- F. 给出下列 Robot 世界中 Robot 的绝对位置，并描述其相对位置。

图 1：

图 2：

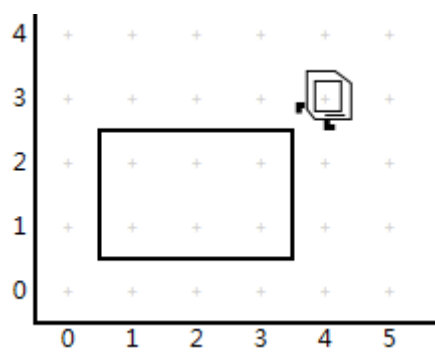


图 3:

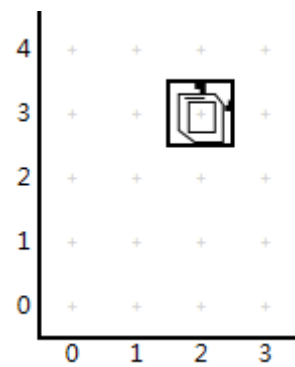
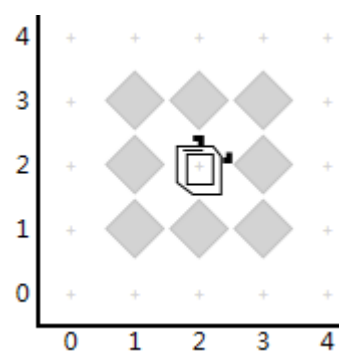


图 4:



## 2. Karel 基本指令

### 2.1 第一段 Karel 脚本

让我们来仔细阅读第一段代码。打开 Karel 程序，选择 C# 模式，我们看到左侧编辑器出现下列代码：

```
using System;
namespace Tech
{
    public class App
    {
        public static void Main()
        {
            // Example:
            Karel.Move();
            Karel.TurnLeft();
            Karel.PutBeeper();
            Karel.PickBeeper();
            Karel.Sleep();
            Karel.Say( "Hello World!" );
        }
    }
}
```

这是一段示例 C#-Karel 脚本（下称 Karel 脚本），我们可以通过修改它来控制右侧的 Karel 做出相应的行为。之所以称为脚本，原因在于这是程序运行之后才编译执行的 C# 代码。

此时程序并未执行，Karel 世界中的 Scene 场景还处在初始状态。我们可以通过工具条上的 K 按钮，在 Scene 中设定 Karel 的起始位置；使用 B 按钮，向方格中左键添加 / 右键删除多个 Beeper；使用四种墙按钮，在方格边框加入阻碍 Karel 活动的墙壁；使用 Mark 按钮，为方格添加 / 删除不同颜色的标记；点击 R 按钮，重置 Scene 场景至最初状态。

► 点击工具条的  按钮，就开始执行 Karel 脚本，在右侧的 Karel 世界中我们观察到 Karel 先朝右侧移动了一格，随后发出了“Hello World”的宣言。在运行时，运行按钮将变为终止按钮，可以点击随时中断执行。



那么这段脚本意味着什么呢，且看下面分析：

首先我们关注的是 1-7 行，以及 15-17 三行括号。这两部分代码合起来是 Karel 脚本执行时必须采用的模板。该模板的涵义为：程序在执行脚本时，调用 Tech 命名空间(namespace) 下的 App 公开类（public class）的 Main 公开静态方法（public static），这也是 C# 的标准语法形式。

在三层大括号之上，第 1 行是 using 指令。

大括号由内向外层层嵌套，如同装着舍利的多重宝函，每个层以 { 开始，以 } 结束，括号内的中间内容均为同一个层级（也称为同一个作用域），每层代表的意义则在 { 的前一行说明。故三组大括号所在的层次（名字空间 > 类 > 方法）依次为：

```
namespace Tech { ... }  
public class App { ... }  
public static void Main() { ... }
```

这几个以小写字母开头的单词 using、public、class、static、void，是 C# 关键字，而大写的 Tech、App、Main 分别是命名空间、类和方法的名称。

C# 关键字是具有特别含义的单词，每个都有其特定用法。学习一门编程语言，需要掌握关键字的用法。让我们先来了解出现的这几个关键字：

### using 指令、命名空间 namespace

using 和 namespace 这两个关键字，提供了一种对 C# 程序和类库分层次进行组织的方式。在 C# 编写的大型程序中，代码分布在不同的源文件中。namespace 封装了一部分代码，using 在代码开始前引用已封装好的库，这样就将一段代码，与其他代码联接起来。在后续的 Karel 任务中，我们将不断的引用各种 C# 标准库，以增强对 Karel 的操控能力。

脚本中的 using System; 引用了 .NET Framework 提供的 System 库文件，分号作为结束符。而 namespace Tech { ... } 则将本段代码封装至 Tech 命名空间。

### 访问修饰符 public (private)

当我们封装了代码后，需要标明哪些可以外部访问，哪些只能内部访问，这就需要 在类、字段或方法名称前，加入访问修饰符。public 代表任何地方都能访问到；private 代表除了与 private 修饰主体在同一作用域或其内部之外，任何地方都无法访问。

除了最常见的 public 和 private 之外，还有 protected、internal、protected internal，这些将在后面章节解释。

### 类 class

类是一种构造，通过使用该构造，我们可以将其他类型的变量、方法和事件组合在一起，从而创建自己的自定义类型。这是 C# 最重要的数据结构，面向对象编程的基石。

类定义了对象的类型，但它不是对象本身。对象是基于类的具体实体，有时称为类的实例。这些我们在后面会详细解释。

脚本中 `public class` 代表公开类，类名称是 `App`。

## 方法和 `void` 返回类型

`void` 代表空类型，在脚本中表示 `Main` 方法的返回值为空。`public static void Main() { ... }`，是我们遇到的第一个方法。方法（也叫做函数或者过程）包括了一段代码。对方法调用，这些代码将会被执行。`Main` 就是一个方法，它会被 Karel 系统调用，这样我们写的脚步才会被执行。

## 静态 `static`

`static` 可以修饰类，也可以修饰方法。使用了静态方法意味着，其操作不需要引用特定的对象。名为 `Main` 的静态方法约定俗成的作为 C# 程序的入口点。我们将在 C# 程序的 `Main` 方法中开始创建对象和调用其他方法，在 Karel 脚本中则开始对 Karel 进行操控。

接下来我们来看脚本的 8-14 行：

## // 注释

双斜杠 `//` 开头的本行内容，均为注释，编译器并不会执行注释内容。通常，注释是为了解释说明接下来代码的意义，毕竟很多代码的实现过于抽象，时间长了连程序员自己都忘记了作用，更别说是其他程序员来阅读这段代码。

9-15 行均以 `Karel` 开头，`Karel` 是已经写好的一个静态类，可以直接调用，用来控制 Karel 机器人。圆点 `.` 是用来访问 `Karel` 静态类中的静态方法，方法名之后的 `()` 表示方法的参数为空。每行代码都是一种指令，执行时右侧 Karel 世界中的 Karel 会显示这些指令所带来的变化，有些是能够被我们捕捉到，有些可能会一晃而过。

我们将在下一个单元中，详细讲解 `Karel` 静态类的这些静态方法，即 `Karel` 的基本指令。

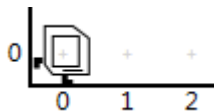
## 2.2 Karel 基本指令

在介绍基本指令前我们再说一说方法。上文提到了 `Main` 方法，这是定义的一种特殊静态方法，用来作为程序的入口点。而一般方法，是为了实现一个特定目的而编写的代码段，在 `class` 类或 `struct` 结构中声明。方法如何声明在下一章还会提及，本节只是使用方法，

执行 Karel 指令就是使用 Karel 静态类中的一系列方法。

## Move 指令

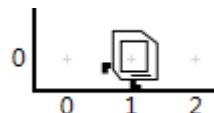
Move 指令使 Karel 沿着前方移动一个街角，例如：



执行代码：

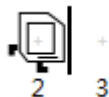
```
Karel.Move();
```

后，变为：



即，Karel 向前方（东方）移动一个街角。

当移动的方向上有一堵墙，如下图：



或者是场景的边界，如下图：



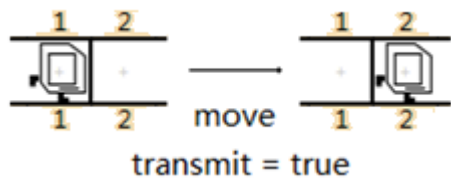
移动都会失败。

当移动成功时 move 指令返回 true，移动失败时返回 false。

move 指令可以附带一个参数：transmit，默认情况下为 false。如果将 transmit 设为 true 则 move 变为“穿越模式”。

```
Karel.Move(true);
```

在穿越模式下，Karel 可以通过前方的墙壁：

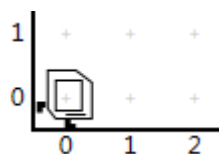


当然，如果前方是场景边界，则不管是不是穿越模式，移动都会失败。

## TurnLeft 左转指令

当 Karel 直立在场景中时，Karel 面向东方，左侧是北方：

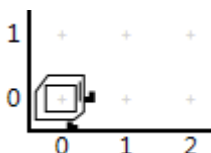




然后执行 turn left 指令：

```
Karel.TurnLeft();
```

这时，前方变为原来左侧指向的方向，即北方：



也就是说，在用户看来，turn left 指令使 Karel 逆时针旋转 90 度：



显然：

$\text{turn back} = \text{turn left} \times 2$

$\text{turn right} = \text{turn left} \times 3$

$\text{turn around} = \text{turn left} \times 4$

## Beeper 指令：PutBeeper/PickBeeper

Karel 可以在当前所处的街角放置或拾取 Beeper 蜂鸣器。

执行指令 put beeper 会在街角处放置一个 Beeper，执行 pick beeper 会从街角处拾取一个 Beeper。

例如：最初街角处没有 Beeper：



然后执行代码：

```
Karel.PutBeeper();
```

Beeper 数变为 1：



继续执行上面的代码，Beeper 数变为 2：



也就是说，每执行一次 put beeper 指令，街角处的 Beeper 会增加 1。

执行 pick beeper 指令，Karel 会从街角上拾取一个 Beeper。

例如：对于上一个场景（街角处有 2 个 Beeper），执行下代码：

```
Karel.PickBeeper();
```

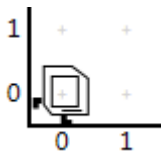
场景会变为上上一个（街角处有 1 个 Beeper），继续执行以上代码，场景会变为最初的样子。这是街角上已经没有了 Beeper，如果再执行 pick beeper 指令，拾取将会失败。

当拾取成功时 pick beeper 指令返回 true，失败时返回 false。

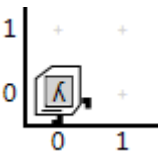
值得注意的是 Karel 口袋中的 Beeper 数是无限的，所以 put beeper 指令总是成功，于是该指令没有返回值。

## TurnOff 指令

Karel 初始时总是处于开机状态（这不由它控制），但是 Karel 可以使自己关机。开机状态的 Karel 总是右侧朝向屏幕外，左侧朝向屏幕里，关机后 Karel 会反转，并且回到直立（面朝西方）方向。



执行代码：Karel.TurnOff(); Karel 变为：



关机指令带有一个 reset 参数，默认是 false，如果设置为 true，关机将变为重新启动，即先关机再启动。

因为关机后 Karel 面朝西方，启动后进行反转，Karel 会面向东方，所以重新启动后 Karel 一定会面向东方。

## Sleep 指令

Karel 执行每个动作都是非常快的，这回导致我们无法看清楚它的动作。sleep 指令会让 Karel 休息若干个毫秒。

例如：执行代码：

```
Karel.Sleep(1000);
```

可以让 Karel 休息 1 秒钟。

Sleep 指令如果不指定时间，默认为休息 100 毫秒，即 0.1 秒。

Say 指令：

还可以命令 Karel “说” 一些话，虽然目前 Karel 并不理解这些话的意思。

例如：执行代码：

```
Karel.Say( "Some Words!" );
```

场景中显示：



介绍完几个基本指令，最后我们来回顾一下上一章节的遗留代码：

```
// Example:  
Karel.Move();  
Karel.TurnLeft();  
Karel.PutBeeper();  
Karel.PickBeeper();  
Karel.Sleep();  
Karel.Say( "Hello World!" );
```

这段代码 Karel 先向正面移动了一格，接着左转身，放下一颗 Beeper，紧接着捡起一颗 Beeper，停了 0.1 秒后，发出了 Hello World 的宣言。由于放下和捡起 Beeper 中间没有停顿，我们看不到 Karel 这组行为。

## 2.3 错误处理

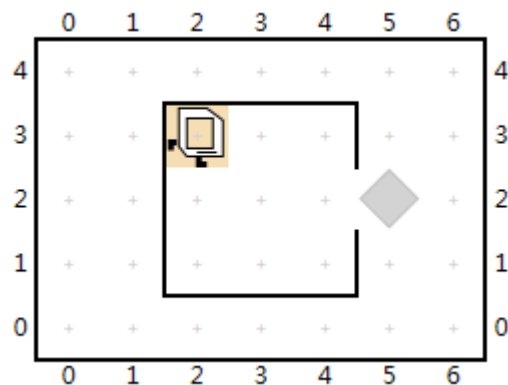
我们在写脚本时难免会出错。最常见的是语法错误，比如脚本中调用不曾定义过的方法或变量，脚本结尾少写了分号或是括号，固定的语法结构并未正确实现，类型转换出现错误等等。在 Karel 脚本运行时，一旦出现语法错误，Karel 的程序底部将显示出错误提示和所在行数。

另一种错误是逻辑错误，脚本可以通过语法错误检查而正常运行，但运行出来并不是我们想要的结果。这就需要反复查看脚本中的 Karel 控制过程，查看是否按预想的步骤进行，最终找到出错位置并修改。往往逻辑错误出现在分支、循环、迭代过程中，或是一些复杂的结构中。

## 2.4 第一个 Karel 任务

任务：捡报纸 (Collect Newspaper)





Karel 在卧室最西北角的床（用小麦色标注）上睡觉，卧室的大小是  $3 \times 3$ 。卧室的门口（5,2）有一张报纸（用 beeper 代替）。Karel 现在想走到门口拿起报纸，然后回到床上看报纸。

为了使问题简单，我们可以严格按照房间大小来设计指令。

可以按以下步骤考虑：1. 走到门口；2. 拿起报纸；3. 回去。

既然已经熟悉了 Karel 的基本指令，那么就可以编写一系列动作指令来命令 Karel 完成任务。我们先将动作指令系列，用文字描述如下：

移动 -> 移动 -> 右转 -> 移动 -> 左转 -> 移动 -> 拾取 -> 后转 -> 移动 -> 右转 -> 移动 -> 左转 -> 移动 -> 移动 -> 后转

其中：右转 = 左转  $\times 3$ 、后转 = 左转  $\times 2$ ；

并且：我们需要在每次执行完一个动作后，执行休息指令，以便让观众看清整个动作系列各执行过程。

接下来我们根据上一节学习的知识，可以列出每个基本动作指令的代码形式如下：

移动 = Karel.Move();

右转 = Karel.TurnLeft();

拾取 = Karel.PickBeeper();

休息 = Karel.Sleep();

然后我么就可以按照顺序，将动作指令翻译场代码并输入到 Main 方法里，最后代码如下：

```

Karel.Move();
Karel.Sleep();
Karel.Move();
Karel.Sleep();
Karel.TurnLeft();
Karel.TurnLeft();
Karel.TurnLeft();
Karel.Sleep();
Karel.Move();
Karel.Sleep();
Karel.TurnLeft();
Karel.Sleep();
Karel.Move();
Karel.Sleep();
Karel.PickBeeper();
Karel.Sleep();
Karel.TurnLeft();
Karel.TurnLeft();
Karel.Sleep();

Karel.Move();
Karel.Sleep();
Karel.TurnLeft();
Karel.TurnLeft();
Karel.TurnLeft();
Karel.Sleep();
Karel.Move();
Karel.Sleep();
Karel.TurnLeft();
Karel.Sleep();
Karel.TurnLeft();
Karel.Sleep();
Karel.Move();
Karel.Sleep();
Karel.TurnLeft();
Karel.TurnLeft();
Karel.Sleep();

```

最后执行脚本，就会看到 Karel 执行任务的过程。

## 思考题

A. 如下图所示的初始位置，开始并处理如下程序。Karel 的任务是找到 Beeper、捡起、关闭。画出到最终位置的地图并查看是否出错。如果有错，请指出将如何修改程序（本程序无错）。

```

public static void Main()
{
    Karel.Move();
    Karel.TurnLeft();
    Karel.TurnLeft();
    Karel.Move();
    Karel.TurnLeft();
    Karel.Move();
    Karel.TurnLeft();
    Karel.Move();
    Karel.TurnLeft();
    Karel.Move();
    Karel.PickBeeper();
    Karel.TurnOff();
}

```

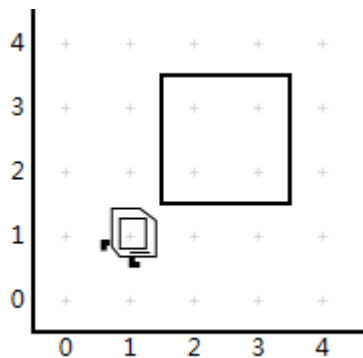
B. 仔细检查下列代码并且纠正所有的词法和语法错误。提示：这里有九个错误，三个

是分号错误，两个语法错误，两个词法错误（是的，还有其他错误）。确认每个词出现在合适的位置上，并且有正确的拼写。你可以使用题目 A 作为词法和语法模板程序。

```
public static void Main()
{
    Karel.Move();
    Karel.Move()
    Karel.PickBeeper();
    Karel.Move();
    Karel.TurnLeft();
    Move();
    Karel.Move();
    Karel.Turnright;
    Karel.PutBeeper();
    Karel.PutBeeper();
    Karel.TurnOff
}
```

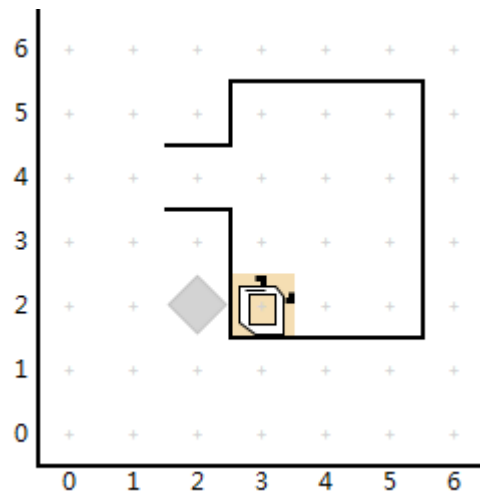
C. 正确 Robot 程序的最小词法和语法是什么？

D. 在大多数城镇中，我们能够按着以下指令重复四次，以便绕着方坛散步：走到最近的转角，左转或右转（每次做同一指令）。当我们正确的返回到最初开始的地方，就结束了。Karel 程序能够绕着方坛散步，你能成功的在以下图为初始位置实现这个程序么？



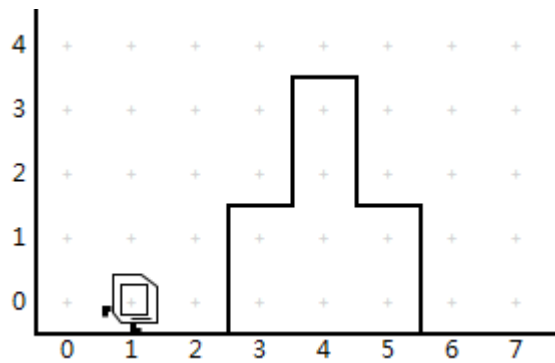
E. Karel 搬到了一个新家，不过每天早上 Karel 起床的时候，报纸仍然（用 Beeper 代表）都会放到房子的前面。请为 Karel 写出取报纸并返回床上的代码，在下图中，Karel 初始就在床上躺着，注意返回后方向一致。



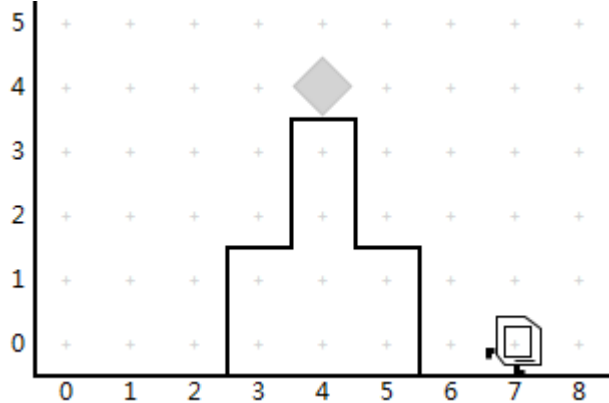


F. 下图中 Wall 代表一座山（上为北），Karel 程序要登山，并且在山顶立一根旗子（用 Beeper 代表），接下来从山的另一边下去（到对称位置结束）。

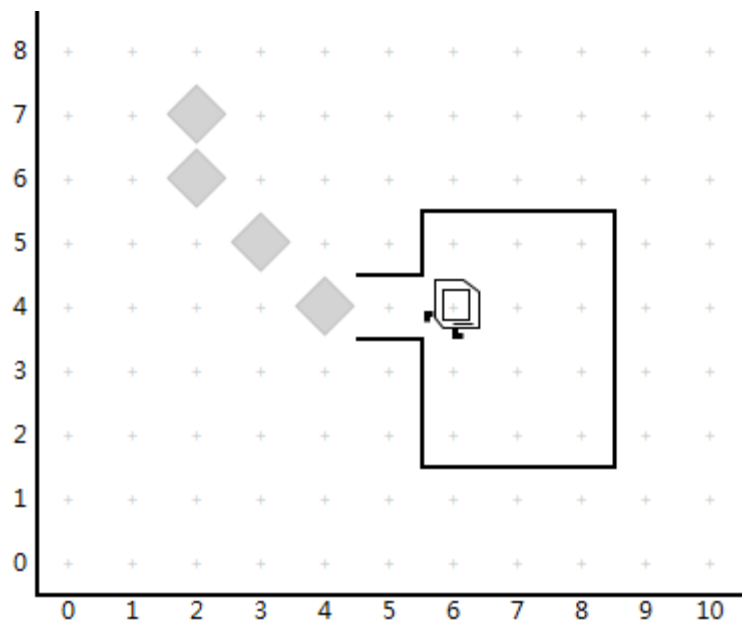
开始时：



结束时：



G. 在从超市回家的路上，Karel 的购物包底部裂了个小口，从中丢失了一些东西。这些杂物用 Beeper 代表。在 Karel 刚发现丢东西时，情况如下图所示。请写出 Karel 捡到所有东西并返回开始位置的程序。



H.Robot 放下无限多的 Beeper 需要多长时间?

I. 如果 Karel 在无限多 Beeper 的街角捡起了一个 Beeper, 街角还剩多少 Beeper, Karel 包里有几个?

## 3. 新方法：扩展 Karel 能力

我们在第 2 章最后完成了 Karel 的第一个任务，不可否认，最终在 Main 中程序的代码有点太长了！

为什么这段代码会冗长呢？那是因为这段程序里，除了“顺序执行”外，我们没有使用任何编程技巧。

这一章我们将介绍新的编程技巧，其中包括：function 函数、parameter 参数、return 返回、if 条件、while 循环。这些技巧的使用将会大大的简化代码长度，并提高代码的复用性。

（注：函数又称为 Procedure 过程，而在 C# 中，函数被叫做 Method 方法，所有在以后行文中：函数、过程、方法是同义词。）

### 3.1 如何定义新方法

通过观察捡报纸的任务代码后，我们发现有两处地方 TurnLeft 被连续 3 次调用。其实我们的目的是要执行 TurnRight 指令，但是由于没有，所以我们不得不 TurnLeft 了 3 次。

我们已经知道，对 Karel 发布指令，其实就是调用的 Static Class 静态类 Karel 上的 Static Method 静态方法。既然 TurnLeft 是静态方法，那么我么也可以定义自己的静态方法 TurnRight。

为了可以通过静态方法对 Karel 进行扩展，我们先定义一个静态类叫：KarelEx，代码如下：

```
public static class KarelEx
{
}
```

关键字 public static class 表明是一个公共的静态类（在这里我们先不要太在意类和方法，有关这部分的内容，我们会在第 5 章中详细讲解）。而在“{}”中，就可以写入我们需要的静态方法了。

定义静态方法 TurnRight 的代码如下：

```
public static void TurnRight()
{
    Karel.TurnLeft();
    Karel.TurnLeft();
    Karel.TurnLeft();
}
```

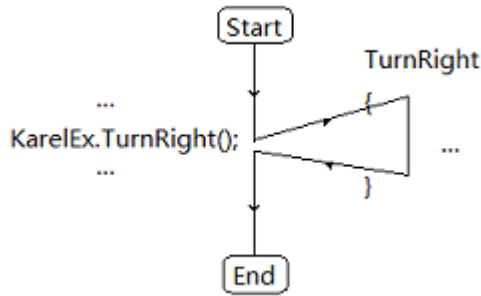
代码中，TurnRight 是方法名称（这里使用：匈牙利命名法），void 表示方法没有返回值（稍后解释），“()”括号里是参数列表（稍后解释），关键字 public static 表明方法是公共的

静态方法，“{}”中是方法体。

当在 KarelEx 类中，定义了 TurnRight 方法后，我们就可以在 Main 中，像 TurnLeft 一样调用它了：

```
KarelEx.TurnRight();
```

当程序执行到以上代码处，系统会跳转到 TurnRight 方法体的开始处，然后执行方法体中的代码直到结束，最后从方法体返回继续向下执行。



依据同样的思路，我们可以定义 TurnBack 方法如下：

```
public static void TurnBack()
{
    Karel.TurnLeft();
    Karel.TurnLeft();
}
```

## return

接下来还发现每个动作后都要执行 Sleep 指令，于是我们可以定义一套带 Sleep 的动作指令。首先，我们将 Sleep 指令加入到上文定义的 TurnRight 和 TurnBack 方法体的代码最后：

```
public static void TurnBack()
{
    ...
    Karel.Sleep();
}

public static void TurnRight()
{
    ...
    Karel.Sleep();
}
```

然后在 KarelEx 类中，定义同名的 TurnLeft 和 PutBeeper 方法：

```
public static void TurnLeft()
```

```
{  
    Karel.TurnLeft();  
    Karel.Sleep();  
}  
public static void PutBeeper()  
{  
    Karel.PutBeeper();  
    Karel.Sleep();  
}
```

接下来需要，扩展 PickBeeper 方法。不同上面所有方法，PickBeeper 具有返回值，所有我们的同名方法也必须具有返回值。定义具有返回值的方法，需要将表示没有返回值的 void 关键字换成，表示返回值类型的标识符。在这里 PickBeeper 的返回值类型是 bool 逻辑类型（也叫布尔类型），于是将 void 换成 bool，方法声明如下：

```
public static bool PickBeeper()  
{  
}
```

所谓返回值，顾名思义就是函数在返回时携带的一个值。那么如何让函数返回一个值呢？这就必须使用 return 语句，其语法格式是：

return 返回值；

说道这里聪明的读者可能已经知道如何实现 PickBeeper 方法了：

```
public static bool PickBeeper()  
{  
    return Karel.PickBeeper();  
}
```

接下来我们需要加入 Sleep 指令。依照以上经验，代码似乎可以这样写：

```
public static bool PickBeeper()  
{  
    return Karel.PickBeeper(); // ?  
    Karel.Sleep();  
}
```

其实这样写是有问题的！你会发现 Sleep 指令永远执行不到。其原因在于，只要执行到 return 语句，不管处于函数体何处都回携带返回值返回。所以，在这里，其后的代码 Karel.Sleep()，将永远无法被执行到。

那么如何解决呢？我们需要定义局部变量临时保存 Karel.PickBeeper() 返回值，然后在



调用完 `Karel.Sleep()` 之后再使用 `return` 返回它，代码如下：

```
bool result = Karel.PickBeeper();
Karel.Sleep();
return result;
```

## variable 变量

变量，计算机程序的最重要部件，由：声明类型 标识符 初始值 定义类型 组成，语法格式为：

声明类型 标识符 = 初始值；

在这里，声明类型：`bool`、标识符：`result`、初始值：`PickBeeper` 的返回值。

(关于变量以及与其相关的表达式，本章稍后还会有详细论述。)

## parameter 参数

接着，扩展最后一个方法 `Move`。`Move` 方法不仅仅有返回值，还有参数。所谓函数的参数就是在函数调用时调用代码向函数中传递的值。返回值的个数一般为 0 或者 1 个，而参数个数可以 0 到多个。(注：函数的参数个数，又叫做函数的 Arity 元数， $n$  个参数的函数叫做也就  $n$  元函数。)

为了区分和使用参数，在函数定义的时候必须定义参数列表，参数列表位于函数名后的 “()” 内，多个参数之间用 “,” 隔开。`Move` 只有一个参数，其声明代码为：

```
public bool Move(bool transmit = false)
```

圆括号内就是参数列表。聪明的读者可能会发现，其形式和变量定义完全一样。其实就参数本质来数就是变量，只不过等号后边的值叫做“默认值”不叫“初始值”。

当对 `Move` 进行调用时，传入的值，赋值给相应参数 `transmit`。比如：

```
Move(true);
```

参数 `transmit` 就会被赋予传入值，即 `true`；

当对 `Move` 进行调研时，没有传入直，例如：

```
Move();
```

参数 `transmit` 就会被赋予默认值，即 `false`；

参数 `transmit` 在声明，也可以不设置默认值，例如：

```
public bool Move(bool transmit)
```

这种情况下，对 `Move` 进行调用时，必须传入值。

在知道函数参数之后，我们就可以扩展 `Move` 指令了，代码如下：

```
public bool Move(bool transmit = false)
{
    bool result = Karel.Move(transmit);
```

```
Karel.Sleep();  
return result;  
}
```

至此，我们就有了一个新的静态类型 KarelEx，在其中定义了六个扩展指令：Move、PickBeeper、PutBeeper、TurnLeft、TurnRight、TurnBack。现在用它们改写“捡报纸”的代码如下：

```
KarelEx.Move();  
KarelEx.Move();  
KarelEx.TurnRight();  
KarelEx.Move();  
KarelEx.TurnLeft();  
KarelEx.Move();  
KarelEx.PickBeeper();  
KarelEx.TurnBack();  
KarelEx.Move();  
KarelEx.TurnRight();  
KarelEx.Move();  
KarelEx.TurnLeft();  
KarelEx.Move();  
KarelEx.Move();  
KarelEx.TurnBack();
```

是不是比原来的代码简单了些！

还能更简单吗？观察代码中，Karel 捡到报纸后的动作：先后转，再移动。其实这相当于后退 MoveBack。于是我们可以扩展出该方法，代码如下：

```
public bool MoveBack(bool transmit = false)  
{  
    Karel.TurnLeft();  
    Karel.TurnLeft();  
    bool result = Karel.Move(transmit);  
    Karel.TurnLeft();  
    Karel.TurnLeft();  
    Karel.Sleep();  
    return result;  
}
```

```
}
```

这里没有使用 `KarelEx.TurnBack` 后转的原因是希望整个动作以一个动作的方式向观众呈现。另外，该方法的参数需要和 `Move` 保持一致。

基于同样的动机，我们可以扩展 `MoveLeft` 和 `MoveRight` 方法，代码类似。

有了这些扩展动作后，我们就可以重新定义动作过程：

前进  $\times 2$  -> 右移 -> 前进 -> 捡拾 -> 后退 -> 左移 -> 后退  $\times 2$

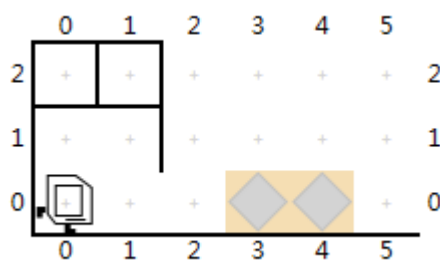
相应代码如下：

```
KarelEx.Move();
KarelEx.Move();
KarelEx.MoveRight();
KarelEx.Move();
KarelEx.PickBeeper();
KarelEx.MoveBack();
KarelEx.MoveLeft();
KarelEx.MoveBack();
KarelEx.MoveBack();
```

是不是又简单了一些！还能更简单吗？当然可以！不过，这里我们先留下一个谜题，读者可以先想一想，我们会在第 4 章中揭晓答案。

## 3.2 if

任务：耕作（Farming）



朋友一家旅游去了，Karel 帮忙照顾他们家的农场。平时 Karel 住在农舍里 (0, 0) 处。农舍外有 2 块地（“小麦色标记处”），Karel 需要在这上面进行耕作。耕作分为种植（放置 Beeper）和收割（拾取 Beeper）。当 Karel 发现地里已经有作物（用 Beeper 表示）了，它就需要收割，发现地空着它就需要种植。Karel 早上出发耕作，晚上回到农舍。

要判断地里是否有作物，我们需要调用 `PickBeeper` 方法。`PickBeeper` 放回一个 `bool` 逻辑值，表示拾取动作是否成功。逻辑类型是最简单的数值类型，它只有两种值：`true` 真、

false 假。真值拥有：是、有、好、成功等正面含义，假值拥有：否、无、坏、失败等负面含义。

如果 PickBeeper 返回 true 表示拾到了 Beeper, 也就说明地上有作物, 并且已经收割成功。如果 PickBeeper 返回 false 表示没有拾到 Beeper, 也就说明地上有作物, 这时候需要种植, 即调用 PutBeeper。

显然是否调用 PutBeeper, 取决于 PickBeeper 的返回值, 也就是说前者的执行是有条件的, 条件是后者的执行结果为 false。在 C# 中, 前者叫做语句体, 后者叫做条件, 两者组成了条件语句: If。

If 语句的格式如下:

```
if( 条件 )
{
    语句体
}
```

如果语句体只有一条语句, “{}” 往往会被省略。

If 语句的含义为:

当 条件是真时 执行 语句体。

套用 If 语句, 一块地的耕作代码, 是乎如下:

```
if (Karel.PickBeeper()) // ?
    Karel.PutBeeper();
```

在仔细观察一下, 发现条件部分有问题。我们的要求是条件为假, 执行语句体, 而 If 的含义恰恰相反。为了符合 If 的逻辑, 必须将返回值进行非运算。在 C# 中非运行符号是:

“!” , 它是一个一元运算符, 放在逻辑值之前:

```
!true = false
!false = true
```

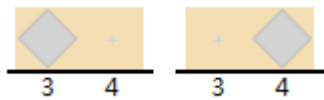
于是上面的代码改为:

```
if (!Karel.PickBeeper())
    Karel.PutBeeper();
```

关键的问题解决后, 耕作任务就非常简单了, 相信你很快就写出任务代码来。

### 3.3 逻辑运算

任务: 耕作 2 (Farming2)



由于两块地种着同种作物，为了种植和销售方便，要求 Karel 保持两块同步耕作。不希望上图中的情况出现，即使出现了也可以改正过来。

在两块地上分别调用 `PickBeeper`，并将返回结果分别保存在逻辑变量 `westHasCrop` 和 `eastHasCrop` 中：

```
bool westHasCrop = Karel.PickBeeper();
Karel.Move();
bool eastHasCrop = Karel.PickBeeper();
```

根据要求，只有当左边没有作物并且右边没有作物，才进行两片地的种植。在 C# 中，并且操作符是：“&&”，它是一个二元运算符，放在两个逻辑值中间：

```
true && true = true
true && false = false
false && true = false
false && false = false
```

使用“并且”运算符“左边没有作物并且右边没有作物”，可以写为：

```
!westHasCrop && !eastHasCrop
```

两地同时种植的代码为：

```
if (!westHasCrop && !eastHasCrop)
{
    Karel.PutBeeper();
    Karel.Move();
    Karel.PutBeeper();
}
```

稍微懂一些逻辑的读者都知道：“左边没有作物并且右边没有作物”等于“非 左边有作物或者右边有作物”。在 C# 中或者操作符是：“||”，它也是一个二元运算符，放在两个逻辑值中间：

```
true || true = true
true || false = true
false || true = true
false || false = false
```



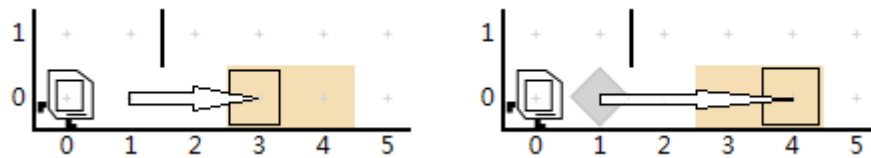
于是，上边的条件也可以写为：

```
if (!(westHasCrop or eastHasCrop))
```

我们将由 逻辑操作符 (!、&&、||) 和逻辑操作数 组成的运算叫做逻辑运算。

### 3.4 else

任务：耕作 3 (Farming3)



眼看 Karel 的朋友就要旅行回来了，可是最近农作物却出现了滞销情况。为了减少产量，而又不会将其中一块地荒废掉，Karel 决定对两块地交替耕作。为了计算天数，卡雷尔在农舍门口 (1,0) 设置了一个标记牌，Karel 每天需要将标志牌反转一次，然后根据标记牌决定耕作那块地。当有标记牌时耕作东边的地，没有时耕作西边的地。

对于标记牌的反转显然和耕作一样，代码如下：（为了关注重点，将移动代码省略。）

```
// Move to (1, 0)
bool hasPlate = KarelEx.PickBeeper();
if (!hasPlate)
    KarelEx.PutBeeper();
```

接下来根据要求写耕地的代码如下：

```
if (hasPlate)
{
    // Move to (4, 0)
    if (!KarelEx.PickBeeper())
        KarelEx.PutBeeper();
}
if (!hasPlate)
{
    // Move to (5, 0)
    if (!KarelEx.PickBeeper())
        KarelEx.PutBeeper();
}
// Move to (0, 0)
```

很显然我们可以毫不费力的写出这段代码。不过这段代码中对同一个 hasPlate 多次使

用 if 进行判断能不能更加简单一些呢？

当然可以，这就需要引入 else 子句。else 是 if 语句的一个可以省略的部分，if 语句的全格式如下：

```
if( 条件 )
{
    then- 语句体
}
else
{
    else- 语句体
}
```

含义：

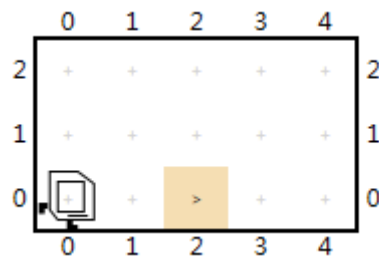
当条件为真时执行 then- 语句体 否则（也就是条件为假）执行 else- 语句体。

这样我们就明白了，其实上面的耕地的代码应当写成：

```
if (hasPate)
{ // Move to (4, 0)
    if (!KarelEx.PickBeeper())
        KarelEx.PutBeeper();
}
else
{
    // Move to (5, 0)
    if (!KarelEx.PickBeeper())
        KarelEx.PutBeeper();
}
// Move to (0, 0)
```

### 3.5 while

任务：找中点（Midpoint Finding）



Karel 的初始位置 (0,0)，此时房间里没有方块。他要走到第 0 行的中间（小麦色标注处）放一个方块。

注意：

1. 在执行任务过程中 Karel 可以在房间的任何位置放方块（如果你愿意）。
2. 房间是矩形，但大小不确定，宽度为奇数。
3. Karel 最后必须要在第 0 行中点（小麦色标注处），面向东（箭头方向）。

解决问题的第一步是确定房间的宽度，方法是：

1. 首先卡雷尔知道自己在 (0,0) 处，这时大道号  $n=0$ ；
2. 让卡雷尔向前走一步，这时候大道号  $n$  加 1；
3. 不断重复第 2 步直到撞到墙壁为止，这时候的大道号  $n$  就是最大大道号，而房间的宽度 = 最大大道号 + 1，所以房间的宽度 =  $n+1$ 。

接下来需要将上面的方法写成代码。

上面的大道号  $n$ ，在执行任务当中一直在变化，在 C# 中叫做变量；而且不管如何变化它的值都必须是整数，于是将整数定义为变量  $n$  的类型。下面是变量声明的代码：

```
int n;
```

其中  $n$  是变量的名称，`int` 标志着变量的类型是整数。

接下来首先要让  $n$  的值变为 0，这个过程叫做对变量赋值，代码如下：

```
n = 0;
```

这里是将整数 0 赋值给  $n$ ，当然，你也可以用上面的形式将任意的整数赋值给  $n$ 。

为了方便，我们还可以将上面的两行代码合并为一行：

```
int n = 0;
```

即，定义一个名字为  $n$  的类型是 `int` 的初始值是 0 的变量。

接下来要找到撞到墙壁的条件。当调用 `Move` 方法时，它会返回一个值，这个值的类型是布尔 (`bool`)。布尔类型的值只有两种结果：`true`、`false`。一般情况下 `Move` 方法返回 `true`，如果返回的值是 `false`，就表示前方有障碍，移动失败！

有了条件后我么就可以，制定代码需要表达逻辑：

当调用 `Move` 返回 `true` 时

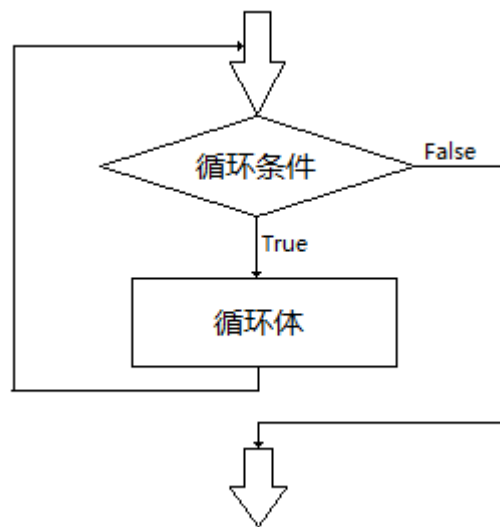
n 加 1

继续 循环

其中“当 ... 时 ...”是一种新的程序技巧，叫做循环，代码形式是：

```
while( 循环条件 )  
{  
    循环体  
}
```

while 语句是一种循环形式：它会执行“()”中的循环条件，然后判断执行结果是否是 true，如果是 true 就执行“{}”内的循环体，然后重新开始新一轮的循环，如果是 false 那么循环结束。整个流程图如下：



有了循环的编程技巧后，就可以实现以上的逻辑了：

```
while(KarelEx.Move())  
{  
    n = n + 1;  
}
```

其中  $n = n + 1$  可以简化为： $n++$ （当然  $n = n - 1$  也可以简化为  $n--$ ）。而且当循环体只有一行语句时“{}”可以省略，所以，以上带代码可以简写为：

```
while(KarelEx.Move())  
    n++;
```

当循环完毕之后，显然房间的宽度  $= n+1$ ，于是可以求出房间的中点  $= (n + 1 + 1) / 2$ （其中任务会保证  $n+1$  等于奇数）。

这时候 Karel 处于房间的最右端，首先需要转身：

```
KarelEx.TurnBack();
```

然后需要向前走  $(n + 1 + 1) / 2 - 1 = n / 2$  步。要完成使 Karel 走  $n/2$  步的动作，我们还要用到 while 循环。

首先，我们将  $n / 2$  保存在 steps 变量中：

```
int steps = n / 2;
```

接下需要执行如下逻辑：

当 steps 不等于 0 时

    移动

    steps 减 1

继续 循环

利用 while 语句，翻译成代码如下：

```
while (steps != 0)
{
    KarelEx.Move();
    steps --;
}
```

最后再向后转，任务完成！

整个任务代码如下：

```
int n = 0;
while(KarelEx.Move())
    n++;
KarelEx.TurnBack();
int steps = n / 2;
while (steps != 0)
{
    KarelEx.Move();
    steps --;
}
KarelEx.TurnBack();
```

## 3.6 Langton 蚂蚁

最后我们使用本章学到的东西，来完成一个制作游戏个任务，这个游戏叫做 Langton 蚂蚁。



任务：Langton 蚂蚁（Langton Ant）

Karel 需要在场景中扮演一只蚂蚁。蚂蚁的生活习性是：

蚂蚁向前爬行（Move）到达新位置，然后根据新位置的颜色并作相应的动作：

如果新位置是白色（没有 Beeper），则蚂蚁将其变为黑色（有 Beeper），然后向右转。

如果新位置是黑色（有 Beeper），则蚂蚁将其变为白色（没有 Beeper），然后向左转。

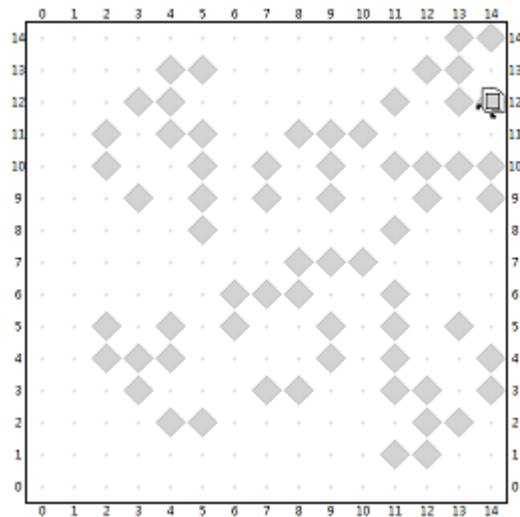
如果在移动时候撞到场景边界，则游戏结束。

游戏玩者，可以任意设置场景大小，蚂蚁的初始位置和方向，以及每个位置的黑白状态。但注意要确保 Beeper 数最多是一个。

任务代码如下：

```
while(KarelEx.Move())
{
    if(KarelEx.PickBeeper())
    {
        KarelEx.TurnRight();
    }
    else
    {
        KarelEx.PutBeeper();
        KarelEx.TurnLeft();
    }
}
```

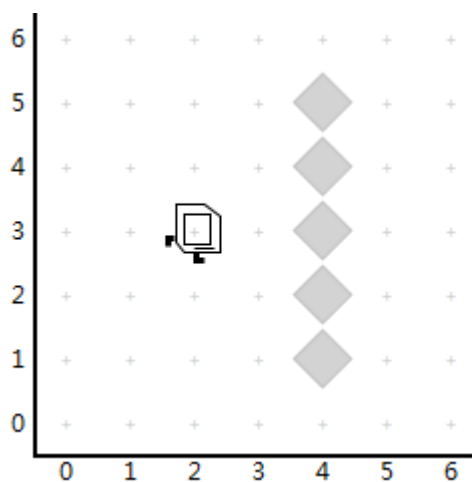
下图是：场景  $15 \times 15$  全白，蚂蚁初始 (7,7) 面向东，的游戏结果：



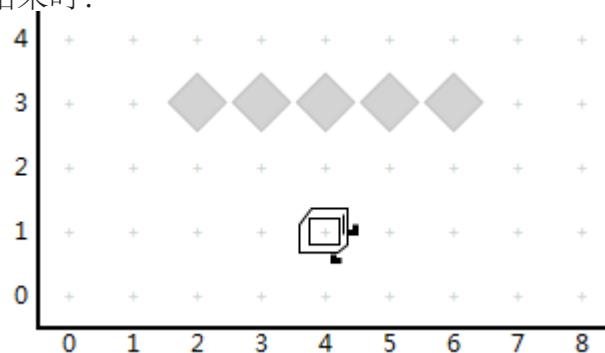
### 思考题

A. 如图所示，写出 Karel 重新排列 Beeper 的程序。

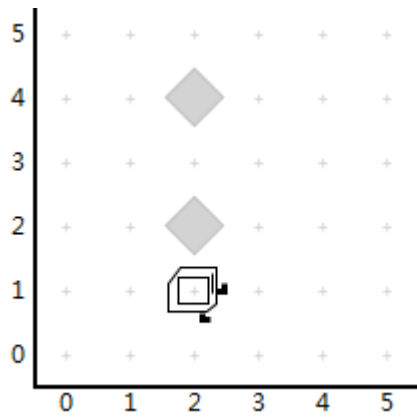
开始时：



结束时：



B. Karel 正在参加 Robot 奥林匹克运动会。其中一项是穿梭竞速，需要 Karel 按数字 8 的模式围绕着两个 Beeper 移动。写出 Karel 尽可能快的程序（最快意味着最少的代码）。结束时，Karel 必须停到起始位置且方向相同。



C. 假定我们让名为 Karel 的 Robot 从原点移动到 100 行 100 列的街区。有多少种写法？

Karel 最少有多少指令？（用目前所学内容作答）

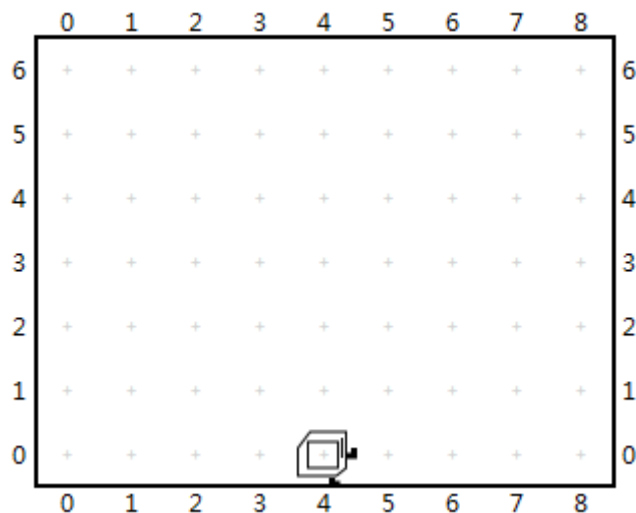
D. 写出符合以下定义的新方法：

- (1).MoveMile, 一个 mile 为 8 个街区长；
- (2).MoveBackward, Karel 向后移动一个街区，但结束时朝向相同；
- (3).MoveKiloMile, Karel 向前移动 1000 miles。

最后一个问题比较困难，但是存在一个相对短的解法。你可以在本问中使用 MoveMile，而不用重新定义。当运行时这些方法会引起错误么？

E.Karel 有时像保龄球比赛中的整瓶器一样工作，按下图所示起始与结束位置，写出该程序。Karel 在任务开始时有 10 个 Beeper。

开始时：



结束时：



## 4. 表达式和语句

到本章之前，我们已经用到了很多编程技巧，为了不至于太过杂乱，这里先总结一下：

### function 函数（method 方法）

程序的组织单位。可以将一段代码组合在一起，构成一个相对独立的单位，以供其他代码调用。

函数在定义里，可以规定参数和返回值，比如：

```
public static bool Move(bool transmit = false) { ... }
```

在函数在调用时，调用者按照函数定义的要求向函数传入零个或多个值，这些值会被赋值给相应参数（一般情况按照参数列表的顺序），函数执行完毕后向调用者返回零个或一个值。比如：

```
bool result = KarelEx.Move(true);
```

### value 值

在使用计算机之前，我们就知道数字值：整数（...-2, -1, 0, 1, 2 ...）、小数（比如：3.14、6.18）。

上一章还使用了逻辑值，逻辑值只有两个：true、false。

另外，还有一种值叫字符串，比如："Hello world!" 就是字符串。字符串其实就是用双引号（"）包括起来的纯文本，就像打开一个记事本在里面书写文章一样。

### type 类型

每个值都有类型，在 C# 中：

整数类型用 int 表示；

小数类型用 double 表示；

逻辑类型用 bool 表示；

字符串类型用 string 表示。

值的类型决定了计算机如何对其进行运算，比如：

可以对逻辑值进行与（&&）或（||）非（!）运算，但是将这些逻辑运算用到其他类型上就没有了意义；反之，可以对整数和小数进行：加（+）减（-）乘（\*）除（/）运算，但这些数字运算不能用于逻辑值。

而且，即便是同一种运算，在不同类型下也表现的不同。比如，加（+）：

$$1 + 1 = 2$$
$$\text{"a"} + \text{"b"} = \text{"ab"}$$

### variable 变量

目前已经使用过两种变量：

声明在函数体内的变量，叫 local 本地变量；

声明在函数参数列表中的变量，叫 parameter 参数。



除此之外，在 C# 中，还可以在类里面声明变量，叫 field 字段，比如：

```
public static class App
{
    public static int n = 5;
}
```

定义在函数中的变量只能被该函数使用，定义在类中大变量可以被多个函数共同使用。

### expression 表达式

将逻辑运算、算数运算等写成代码，这段代码就叫做表达式，比如： $n = n + 1$ 、 $n++$ 、 $1 + 1$ 、 $"a" + "b"$ 。另外对函数进行调用的代码，也叫做表达式，比如：`Karel.Move()`。

表达式相当于数学公式，由操作符、操作数、函数调用和圆括号组成。函数调用我们已经知道了；操作数就是值或变量；操作符就是上文中的运算符： $=$ ， $+$ ， $-$ ， $*$ ， $/$ ， $!$ ， $\&\&$ ， $||$  等。

操作符之间存在先后关系（叫做 priority 优先级），当表达式比较复杂时，这种关系就会体现出来。比如：

$1 + 2 * 3$

就是先做  $2*3$  得到 6，然后  $1+6$  等于 7。

如果想改变这种固有优先级，可以使用圆括号。比如：

$(1 + 2) * 3$

圆括号迫使加法先于乘法而进行。

最后需要说明的是：一个值或变量，也叫做表达式，是最简单的表达式形式。

### statement 语句

最简单的语句叫做表达式语句，即表达式 + “;”。比如：

```
n = n + 1;
Karel.Move();
```

将多个语句很自然的顺序书写，就构成了顺序语句。比如上例就可以看成一个由两个表达式组成的顺序语句。

更为复杂的语句需要语句构件支持，目前我么已经知道两种语句结构：if else 条件、while 循环，格式如下：

if ( 执行条件 )	while ( 循环条件 )
{	{
then- 执行体	循环体
}	}
else	
{	

```

        else- 执行体
    }

```

其中，执行条件和循环条件要求是表达式，而 then- 执行体、else- 执行体、循环体要求是语句，并且当语句只有一个式“{}”可以省略。

这一章我们将介绍一些更酷的表达式和语句，基于它们我们不仅可以使得代码非常简约，而且可以执行完成更加有趣的任务。

## 4.1 继续扩展 Karel 指令

任务 : 到处走动 (go anywhere)

“they come and go and walk away  
but I’ m not going anywhere”

只能一步一步移动的 Karel，试着想要在场景里走动，你能为它做到吗？

为了满足 Karel 的愿望，我们可以定义一个 Go 方法，声明如下：

```
public static int Go(int steps) { ... }
```

参数 steps 是想要向前走几步，返回值是实际上走了几步。

在函数体中，我们使用 while 语句，循环条件是 steps 大于 0，而且移动成功，循环体是将 steps 减去 1。

steps 大于 0 是一种新的运算，叫做比较运算，对应的表达式是：steps > 0。

现在根据分析将循环部分的代码写出来：

```

while (steps > 0 && KarelEx.Move())
    steps --;

```

循环结束后 steps 的值是剩余的步数，而函数要求返回走了多少步，所以需要在循环前将希望走的步数保存在变量（expected）中，这样 expected - steps 就是返回结果：

```

int expected = steps;
...
return expected - steps;

```

最后，为了兼容 Move，我们在 steps 后添加 transmit 参数。另外我们将 steps 的默认值设置为一个足够大的数字，在 C# 中最大的整数为：int.MaxValue。

于是 Go 的最终代码为：

```
public static int Go(int steps = int.MaxValue, bool transmit = false)
```

```

{
    int expected = steps;
    while (steps > 0 && KarelEx.Move(transmit))
        steps--;
    return expected - steps;
}

```

当调用 KarelEx.Go() 时，你会看到 Karel 会一直走到撞墙！

仿照 Go，读者可以写出 GoBack、GoLeft、GoRight。

至此，Karel 终于可以在场景中 “go anywhere” 了。

有了 Go 系列指令后，我们就可以继续简化 Karel 的第一个任务了。简化后代码如下：

```

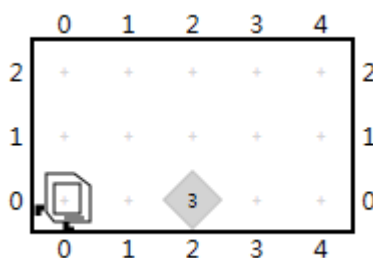
KarelEx.GoRight(1);
KarelEx.Go(3);
KarelEx.PickBeeper();
KarelEx.GoBack();
KarelEx.GoLeft();

```

这段代码 Karel 改变了行走路线，它先向右平移 1 步然后向前走 3 步，然后捡起报纸，然后倒退到墙边，然后向坐平移至墙壁。

还能再简单点吗？可以！只要代码多于一条语句，我们始终都有简化的可能。到底如何继续简化，我们留到第 5 章再揭秘！

任务：方块翻倍 (Double Beeper)



Karel 需要将第一大街（即：0 号街）中点 (2, 0) 处放置的 Beeper 的个数翻倍。

首先使用 Go(2) 指令来到大街中间处，接下来：

尝试着捡起一个方块，如果成功：

向前移动一步，然后放下两方块，然后退回来，继续循环。

如果失败，则说明地上没有方块了，这时前方的方块刚好就是翻倍后的数量，只需要将它搬回原先位置就可以了。搬运的方法和上面的类似，只要每次放一个方块就

可以了。

将上面逻辑翻译成代码如下：

```
KarelEx.Go(2);
while(KarelEx.PickBeeper())
{
    KarelEx.Move();
    KarelEx.PutBeeper();
    KarelEx.PutBeeper();
    KarelEx.MoveBack();
}
KarelEx.Move();
KarelEx.TurnBack();
while(KarelEx.PickBeeper())
{
    KarelEx.Move();
    KarelEx.PutBeeper();
    KarelEx.MoveBack();
}
```

换一种思路：可以一次性将地上的 Beeper 全部捡起了，然后将 Beeper 数加倍，然后再一次性的将加倍后的 Beeper 全部放回去。

要完成这样的想法，需要定义新的扩展指令：PickBeepers 和 PutBeepers，它们的接口形式如下：

```
public static int PickBeepers(int beepers = int.MaxValue)
public static void PutBeepers(int beepers)
```

参数 beepers 是希望捡或放的 beeper 个数，显然 PutBeepers 一定成功，而 PickBeepers 返回实际捡到的 beeper 个数。PickBeepers 中 beepers 默认值是一个足够大的数子，意思是街角上有多少捡多少。

具体实现代码如下：

```
public static int PickBeepers(int beepers = int.MaxValue)
{
    int expected = beepers;
    while (steps > 0 && Karel.PickBeeper())
```

```

        steps--;
        Karel.Sleep();
        return expected - beepers;
    }

    public static void PutBeepers(int beepers)
    {
        while (steps > 0)
        {
            Karel.PutBeeper();
            steps--;
        }
        Karel.Sleep();
    }

```

可以看到 PickBeepers 和 Go 的代码惊人的相似。

好了，有了这两扩展，以上任务，瞬间秒杀：

```

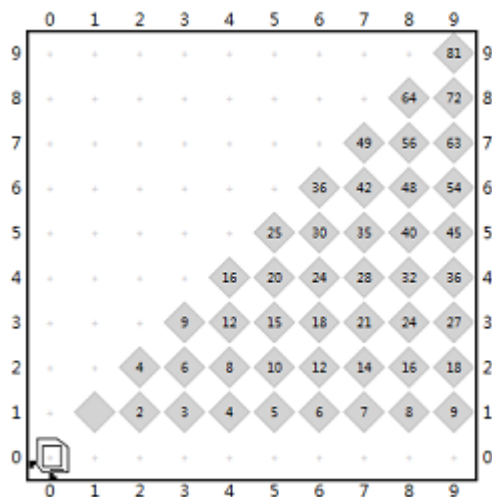
KarelEx.Go(2);
KarelEx.PutBeepers(KarelEx.PickBeepers() * 2);

```

果然是强悍（“强悍地人生不需要解释！”）。

## 4.2 for

任务：九九乘法表 (nine nine)



Karel 需要在  $10 \times 10$  的场景中，摆放 Beeper，要求是街角上的 Beeper 个数等于街号  $\times$  道号。

由于西北方向的数字和东南方向重复，所有只摆放东南方向的街角。

要完成这个任务，显然要使用 while 语句，而且 1 个不够要使用 2 个。

先定义一个循环变量：

```
int street = 0;
```

表示街号。使用 while 语句，让街号从 0 到 9 循环：

```
while(street < 10)
{
    street++;
}
```

然后在这个循环体中，以相似的方法，定义道号变量并使用 while 语句，让它也从 0 到 9 循环。最终我们得到代码如下：

```
int street = 0;
while(street < 10)
{
    int avenue = 0;
    while (avenue < 10)
    {
        avenue ++;
    }
    street ++;
}
```

这个循环结构可以保证 (avenue, street) 覆盖每一个街角。

接下来我们向这个循环结构中加入移动指令，以保证 Karel 位置和 (avenue, street) 同步。

刚开始时 Karel 所处街号和 street 变量都是 0，接着在外层循环中，语句“street ++;”，将街号加 1，所以这时我们需要让 Karel 左移一步，来到正确的大街上，即，需要在这里添加“KarelEx.MoveLeft();”语句。

基于类似的理由，我们也需要在“avenue ++;”语句处添加“KarelEx.Move();”语句。

但这里还需要考虑，每次在内部循环前，“int avenue = 0;”语句道号都被初始化为 0，而如果 Karel 已经经过了一轮内部循环，那么它的道号应当是 9，所以我们必须在这里添加“KarelEx.GoBack();”以保证 Karel 处在 0 号大道上。

同步移动后的代码如下：

```
int street = 0;
while(street < 10)
{
```

```
KarelEx.GoBack();  
int avenue = 0;  
while (avenue < 10)  
{  
    /* ... */  
    KarelEx.Move();  
    avenue ++;  
}  
KarelEx.MoveLeft();  
street ++;  
}
```

有了这样一个循环结构，我们接下只需要在“...”处添加，放置“street \* avenue”个数的 Beeper，就可以了。当然为了保障只放置东南方向，所添加 if 语句，执行条件是“avenue 大于等于 street”，代码如下：

```
if (avenue >= street)  
    KarelEx.PutBeepers(street * avenue);
```

其中，“>=”表示大于等于，是一种新的比较运算符号。

代码完成，运行脚本，会看到 Karel 生成九九乘法表的过程。

显然循环结构是这段代码的重点，我们使用了两个嵌套在一起的 while 语句，它们的结构是相似的，我们可以这样表示其格式：

```
声明循环变量  
while( 循环条件判断 )  
{  
    循环体  
    改变循环变量  
}
```

这种循环结构在 C# 中会经常使用。不过以上面的格式书写即不方便也不够严谨，特别是“循环体”和“改变循环变量”常常混杂一起。既然经常使用，所以为了有清晰的语法 C# 提供了另外一种循环语句：for 语句，来支持这种循环结构。for 语句格式为：

```
for( 声明循环变量 ; 循环条件判断 ; 改变循环变量 )  
{  
    循环体  
}
```



有了 for 语句后，我们也可以将任务代码用 for 语句改写：

```
for(int street = 0; street < 10; street ++)  
{  
    KarelEx.GoBack();  
    for (int avenue = 0; avenue < 10; avenue ++)  
    {  
        if (avenue >= street)  
            KarelEx.PutBeepers(street * avenue);  
        KarelEx.Move();  
    }  
    KarelEx.MoveLeft();  
}
```

这样以来是不是清晰多了！

代码中 Karel 会遍历整个街道，其实由于西北方不摆放 Beeper 所 Karel 没有必要去。

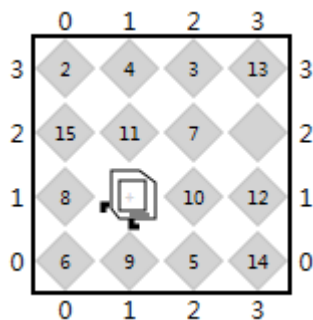
我们可以让内部循环变量 avenue 每次初始化为 street。相应地将 “KarelEx.GoBack();” 改为 “KarelEx.GoBack(9 - street);”

这样以来 if 条件自然成立，所以可以删去。最终代码为：

```
for(int street = 0; street < 10; street ++)  
{  
    KarelEx.GoBack(9 - street);  
    for (int avenue = street; avenue < 10; avenue ++)  
    {  
        KarelEx.PutBeepers(street * avenue);  
        KarelEx.Move();  
    }  
    KarelEx.MoveLeft();  
}
```

## 4.3 switch case

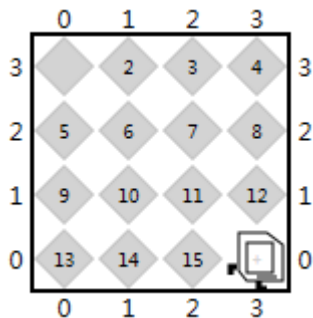
任务：拼图游戏 (Puzzle)



在  $4 \times 4$  的场景中摆放着打乱顺序的图片（用 1 到 15 的 beepers 表示），Karel 处在唯一的空位上。

游戏者操作“方向键”，控制 Karel 移动。Karel 移动后会将新位置的 Beepers 交换到原有空位上，而使得新位置变成空位。

游戏的目的是将打乱顺序的图片，按顺排列。最后恢复成下图的样子：




首先需要知道玩家按键状态。这里我们使用 Karel.Listen 方法，执行监听指令，具体代码如下：

```
string key = Karel.Listen() as string;
```

Listen 可以监听多种信息，所以放回值为 object 类型的值。不过在监听键盘信息的时候，Listen 指令总是返回字符串，所以我们需要通过 as 操作符将返回值转换为 string 字符串类型。转换后的值就可以保存在字符串变量 key 中了。

Listen 指令可以监听玩家向场景中发出的信号。玩家向场景中发送键盘信息的方法如下：

找到位于应用程序由下角的键盘信号发射器：。

用鼠标点击它，使其被激活：。

然后用户在键盘上的敲击动作就会发送到场景中，从而被 listen 指令会监听到。不过，为了方便：在脚本执行时，会自动激活，在执行完毕时会自动失效。

用户按下四个方向键，则 key 值分别是 "left"、"right"、"up" 或 "down"。

接下来，karel 需要根据 key 值进行相应的动作。这里使用 if 可以将程序框架搭建出来：

```
if (key == "right" ) { ... }  
if (key == "up" ) { ... }  
if (key == "left" ) { ... }  
if (key == "down" ) { ... }
```

这里要特别注意：在 C# 中比较两个值是否相等的运算符是 “==”，而不是 “=”，因为后者是赋值符号。

这个框架显然有点低效率。因为当判定 “key 等于 "right"” 后，之后的判定显然都失败不许要判定，只有在 “key 不等于 "right"” 时后面的代码才有意义。所以可以改为：

```
if (key == "right" ) { ... }  
else  
{  
    if (key == "up" ) { ... }  
    else  
    {  
        if (key == "left" ) { ... }  
        else  
        {  
            if (key == "down" ) { ... }  
        }  
    }  
}
```

由于 else- 语句体其实只有一条语句，即 if 语句，所有 else 后的 “{}” 可以省略，最后的代码框架改为：

```
if (key == "right" ) { ... }  
else if (key == "up" ) { ... }  
else if (key == "left" ) { ... }  
else if (key == "down" ) { ... }
```

其实这——一个经典的代码结构，C# 专门提供了一个 switch 语句来支持这种结构。

使用 switch 语句，上面的框架变为：

```
switch (key)  
{
```

```
case "right" :  
    ...  
    break;  
case "up" :  
    ...  
    break;  
case "left" :  
    ...  
    break;  
case "down" :  
    ...  
    break;  
}
```

看到这里，你可能会说：“怎么 switch 语句搞的比 if 的嵌套形式还复杂？”。你说对了！switch 语句的引入不是为了使得代码更简单，而是使代码运行效率更高！为了使得运行效率高，switch 语句还规定 case 后的比较值必须是数值而不能是变量。至于为什么这样运行效率会高，这牵扯到编译原理的内容，比较复杂，不宜在本书中解释，有兴趣的读者可以在本书的姊妹篇中找到答案。

有了框架后，我们就可以向其中填入代码了。这里以 "right" 为例，代码如下：

```
case "right" :  
    if (KarelEx.Move())  
    {  
        int beepers = KarelEx.PickBeepers();  
        KarelEx.MoveBack();  
        KarelEx.PutBeepers(beepers);  
        KarelEx.Move();  
    }  
    break;
```

其他情况代码类似。

最后我们将整个框架放在一个循环中，这个循环判定用户按下 “esc” 键来结束循环：

```
string key = null;  
while (key != "esc" )
```

```
{  
    key = Karel.Listen() as string;  
    /* swith 框架 */  
}
```

代码完成，运行脚本，我们就可以玩拼图游戏了。

不过，这个游戏目前有四个缺点：

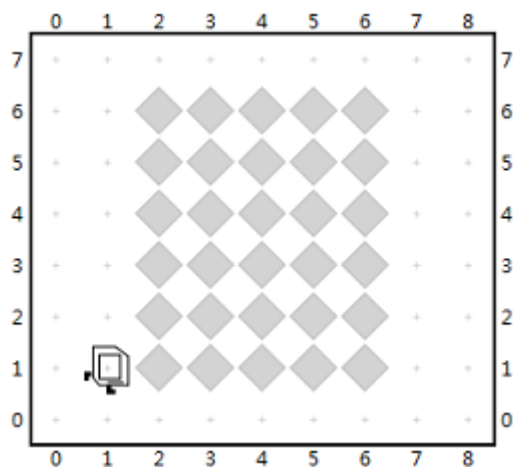
1. Karel 移动时玩家会明显看到，它来回移动了几次；
2. 没有自动生成游戏；
3. 没有检查玩家游戏是否成功；
4. Beeper 始终不能代替真正的图片。

那么如何弥补这些缺点呢？我们会在后文一一给出改进方案。

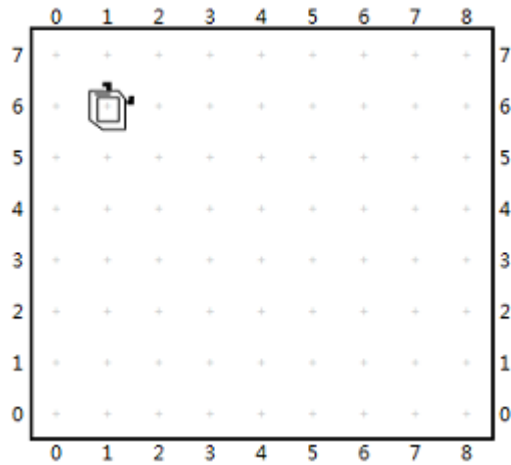
## 思考题

A. 大型农村的麦子成熟了，Karel 负责对一块麦田进行收割，它将如何完成任务呢？

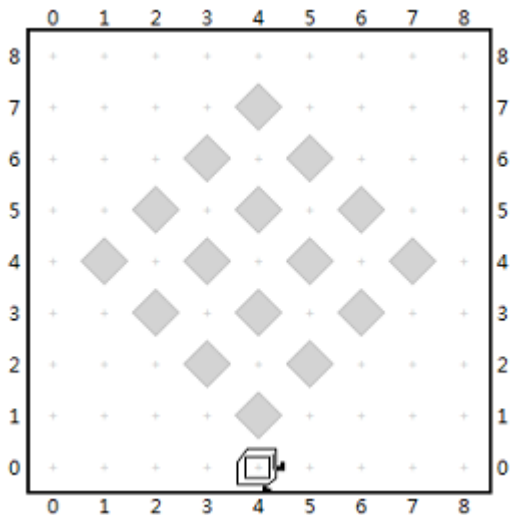
开始时：



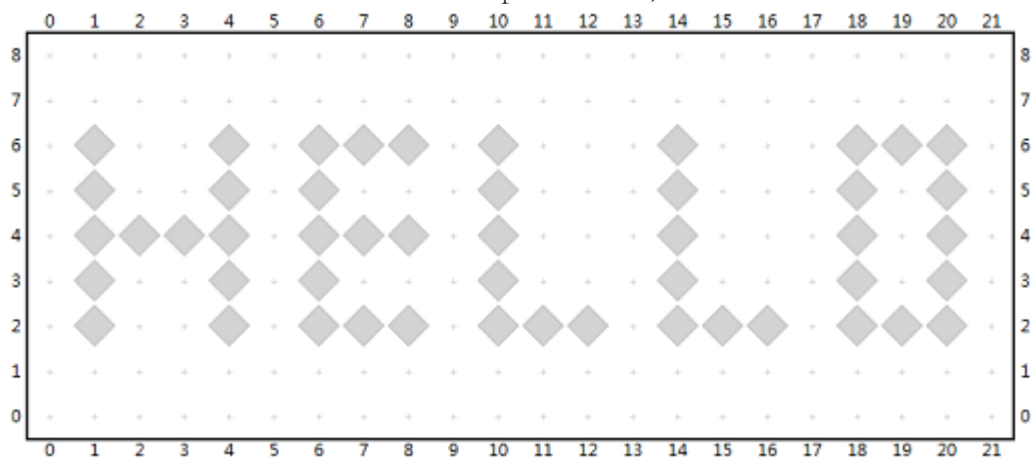
结束时：



B. 下图描述的是 Karel 在棒球赛后的一夜间种下了一堆 Beeper。写一段程序收获这些 Beeper。注意：这个任务和上一题的丰收例子相差并不大。如果你能看到两个任务间的一致性，你就能按上例的程序写出本题程序。



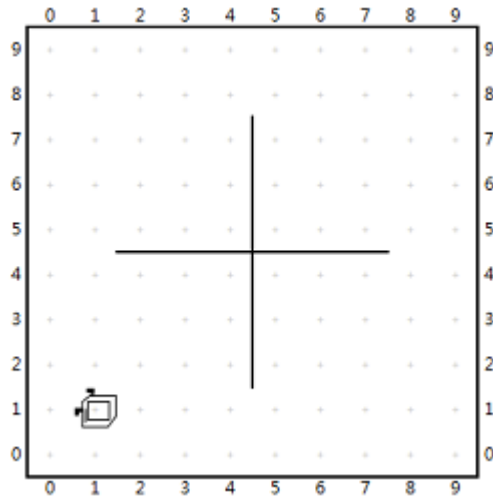
C. Karel 想向宇宙的其他居民发送问候，于是 Karel 需要种下一堆 Beeper，将其广播至外星球。请按下列图示写出 Karel 种 Beeper 的程序，你可以自定 Karel 的初始位置。



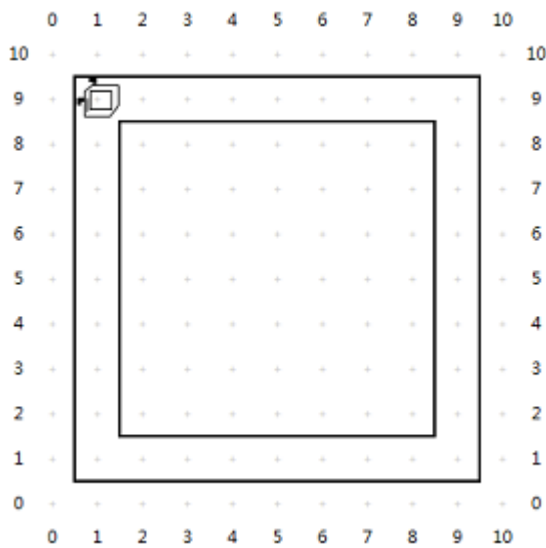
D. Karel 接收到宇航局的合同，在宇航员到达 Karel 世界轨道中显示当前的时间。时间必须以数字形式，并且在  $23 \times 10$  的街区中显示。你可以自定义大小和数字形状。程序必

须能快速的改变时间，这样 Karel 可以迅速更新显示。在本例中，显示 10:52。

E.Karel 找了份花匠的兼职工作。Karel 的特长是种植 Beeper。Karel 当前任务是围绕着“+”形状墙的边缘，每个街角种一个 Beeper，开始位置如下图所示。



F.Karel 在花园干活已经烦了，于是它决定换个兼职工作。Karel 现在要给 Karel 世界的建筑安装地毯（由 Beeper 制作）。写一段程序安装地毯，起始状态如下图所示。地毯必须没有凸起，所以确保 Karel 每走一步只放一个 Beeper，尤其是在转角处。





## 5. 面向对象设计

到目前为止，我们已经知道了很多编程技巧：函数、表达式、变量、条件语句、循环语句等。在前几章中，我们在编写卡雷尔程序时，经常使用函数将一组代码组合在一起，形成一个相对独立的可以被调用的程序执行过程，这种编程风格称为：面向过程编程。从这一章开始，我们将讲解一种新的编程风格：面向对象编程。

面向对象编程方式更接近人的正常思维，因此而被广泛的使用，因此你可能已经听说过她的鼎鼎大名。面向对象程序的可读性、健壮性、复用性都很强，写出的任务程序清晰而优美。用面向对象的方法式编写程序的感觉是非常酷的！说到这里你可能已经迫不及待要开始卡雷尔任务之旅了，不过在这之前让我先澄清一些概念以便减少以后行文的歧义。

### 5.1 面向对象概念

前几章中，我们以类的静态方法作为函数的实现方式。其实 Class 类和 Method 方法就是面向对象的概念，相关的概念还有：Object 对象、Property 属性、Event 事件、Field 字段、Construct 构造 /Destruct 析构、Abstract 抽象 /Interface 接口、Inherit 继承、Encapsulate 封装、Polymorphic 多态性、Attribute 特性 /Reflection 反射。

#### Object 对象

在日常生活中，我们感觉到的、想到的事物，都称为对象。比如：你和我，正在阅读的这本书、卡雷尔等都是对象。在面向对象的世界里，万物皆是对象。

#### Property 属性

凭经验，我们知道对象具有一些性质，比如：你和我的身高、体重，这本书的厚度，这些特性称为属性。

#### Method 方法 /Event 事件

另外，我们还观察到对象可以做某些动作，比如：卡雷尔可以移动、你正可以阅读。对于对象的动作有两件事情值得关注：

1. 动作何时发生，比如：你正在阅读，所以阅读动作正在发生、卡雷尔还没有执行任务，所有移动没有发生。对象的动作发生了称为对象产生了一个事件。
2. 动作如何执行，比如：你如何阅读，卡雷尔如何移动。事物可以执行某个动作称为其掌握了执行这个动作的方法。

#### Field 字段

从对象构成来看，属性、方法、事件都是对象的组成部分，所以为了方便将它们统称为对象的 Member 成员。另外 C# 还允许在对象里定义变量，就像函数里的变量一样。这种变量也是对象的组成部分，属于对象的成员，我们将这种成员变量叫作字段。

#### Class 类

凭借日常经验，人们发现很多对象是相似的，即：它们的某些成员是一样的，我们将

这些成员的总体叫作类。比如：你我他都具有生命、智慧，于是将生命和智慧放在一起就是人类。如果一个对象的成员包含了某个类所含有的所有成员，那么这个对象就属于这个类。比如：如果人类只包含：生命和智慧，那么由于卡雷尔具有生命（TurnOff 可视为结束生命）和智慧（可以执行指令），所以卡雷尔就属于人类。

类是一个抽象出来的 Concept 概念，在现实世界中并不存在，它只存在于大脑和计算机中。类规定了一组成员，用来辨别那些对象属于这个类，这些成员叫作类的 Connotation 内涵。一个属于类的对象叫作这个类的一个 Instance 实例，所有实例的集合叫作类的 Denotation 外延。

内涵和外延其实都来自概念，类由于也是概念，所有自然具有这两个性质。但是 C# 类和一般概念的内涵还是有一定的区别：C# 类的内涵只定义成员个数，而概念的内涵不仅定义成员个数还可以定义成员的值。例如：人类的内涵包括年龄，而成年人概念的内涵不仅包括年龄还规定年龄值必须  $\geq 18$  岁。

类的外延还可为空，在 C# 中类可以没有对象实例，这样的类叫做 static class 静态类。

在 C# 中，名为 Object 的类，是外延最大的类，所有的 C# 对象都属于 Object 类的实例。Object 类中定义了所有 C# 对象共有的成员，它们分别用于：判断两个对象是否相等、将对象变成数字、将对象变成文字。

### Construct 构造 /Destruct 析构

在现实中，概念的内涵，随着时间科技发展、地域文化环境的不同，往往有所改变。偷偷的改变概念内涵，是诡辩学上的常用手段，称作偷换概念。而在现实中，对象的成员往往经常改变，比如：你本来不会开车后来学会，于是你就增加了驾驶这个方法，于是你从非司机类变为属于司机类。

在 C# 中，定义一个类，其实就是定义类的内涵包含的成员。C# 规定类的内涵成员和对象的成员定义是不能改变的，而且还规定任何对象的成员集合必须和一个类的内涵相同，这个类叫做该对象的 Type 类型类（也叫做类型类）。

对象的类型类负责该对象的构造和析构，对象的构造过程叫做 Initialize 初始化、析构过程叫做 Finalize 结束化，参与初始化的方法称为 Initializer、参与结束化过程的方法称为 Finalizer。

（数据的类型是面向过程的概念，它规定了该数据的操作范围和方式。在 C# 中数据等同于对象，类型等同于类，操作等同于方法。）

（一些更加灵活的语言，比如 Python、JavaScript、Ruby 等，支持类和对象成员定义的改变。）

### Abstract 抽象 /Interface 接口

对象必然有类型类，相反过来不一定成立，不能构造对象的类称为抽象类。不能产生对象的原因是：

非字段成员（方法、属性、事件）的定义分为两部分 Declare 声明和 Implement 实现，声明定义了如何使用，实现定义了执行代码。有些非字段成员的定义只有声明没有实现，称为抽象方法（或属性、事件）。含有抽象成员的类就是抽象类。抽象类由于某些成员的定义不完整，属于半成品，所以没有办法创建实例对象。

（变量和字段的定义包括：声明和赋初始值，为了方便，有时当和成员声明字眼同时出现时，成员定义特指实现或赋初始值）

接口是只含有抽象成员的类。接口可以将类的使用和实现完全分开。

## Inherit 继承

现在我们已经知道了类是由类的内涵定义的，而类的内涵就是一组成员构成的集合。于是，拿任意两个类 A 和 B 进行比较，如果发现 A 的内涵包含 B 的内涵的所有成员，那么我们称：A 是 B 的 Subclass 子类，B 是 A 的 Baseclass 基类（其他语言也叫作 Superclass 超类），A（或 A 的对象）继承 B（或 B 的对象）。

继承给程序设计带来了复用性，如果在定义 A 的时候 B 已经存在了，那么 A 就可以从 B 继承，从而 A 中只用定义比 B 多出来的成员即可。

一个类或成员可规定自己不能被子类继承，称为：Sealed 封存的。子类也可以不要基类中的成员，而完全重新定义，称为：new 新的。

最有趣的是子类可以在基类允许的情况下 override 重写成员的全部或部分实现。基类中允许重写的成员，称作：virtual 虚的。另外 C# 规定，所有抽象成员都是虚成员，都可以被冲重写（因为本身就没有实现，所以也必须被重写）。

## Encapsulate 封装

在定义类的时候，一组成员被封装在一起。封装的结果使得类（或对象）的内部和外部形成了两个不同的区域。于是我们可以规定成员的可见性：只对自己内部可见叫 private 私有的。

另外由于类（或对象）具有继承关系，所以将只对自己内部和子类内部可见叫 protected 保护的。

最后我们所写的 C# 程序都被封装在一个个 Assembly 程序集中，只对自己的程序集内部可见叫 internal 内部的，对所有程序集可见叫 public 公开的。

## Polymorphic 多态性

在面向过程设计中，我们将变量的类型，叫做变量的声明类型，变量值的类型叫做变量的定义类型。一般情况下：定义类型必须等于声明类型。

在面向对象中，变量的声明类型是声明类，值就是对象，变量的定义类型是值对象的类，即定义类。定义类可以是声明类，也可以是声明类的子类。也就是说：变量可将多种类型的对象作为其值，只要这些对象的类是声明类或其子类就可以了。更神奇的是通过该变量使用虚成员（或抽象成员）时，实际调用都是当前值对象的类型类的重写实现（如果有重

写)。这样就使“得对成员的使用”这段不变的代码，根据变量的具体赋值对象不同，而具有不同的执行方式，所以叫做多态性。

### Attribute 特性 /Reflection 反射

万物皆是对象，类虽然是一个抽象的事物但也是事物，所以类也是对象。

类作为对象，本身也可拥有自己的成员。为了区分，将类自己的成员叫作 static 静态成员（也叫类成员），类的内涵定义成员叫作对象成员。

另外，类或成员在定义时，我们可以向其添加特性，比如：成员的可见性、是否是抽象的、虚的等，而且除了这些 C# 语言规定的特性外，还可自己定义新的特性。所有这些特性也是对象。

甚至于成员本身也是对象。

类对象、特性对象、成员对象，其实就是代码本身的面向对象化。使用它们的代码就像在照镜子一样看着自己，所有这些对象叫做反射。

特性和反射由于比较复杂，不适合本书“入门”的称号，所以不再本书的讲解范围内。有兴趣的读者可以在本书的姊妹篇：《Karel C# Programming 应用》中找到详细的讲解。

## 5.2 面向对象原则

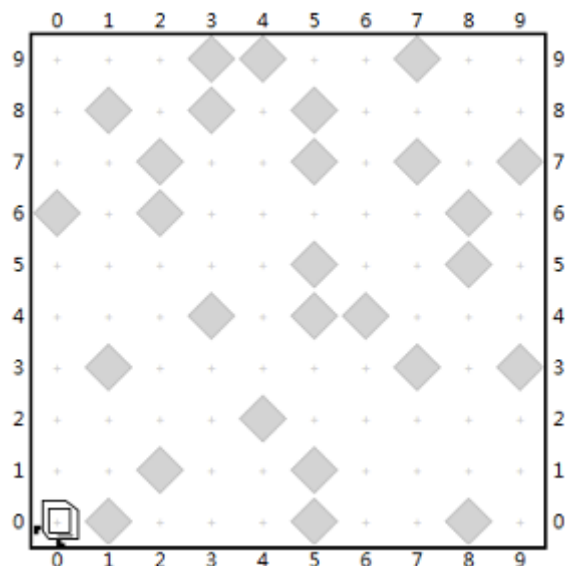
我们一般称面向对象的卡雷尔系统为：Karel++，程序集 Tech.Robots 就是对 Lab.Karel 的面向对象扩展。

请打开命令行工具，cd 到 Lab.Karel 的安装目录下，输入：

```
Lab.Karel.exe "Tech.Robots"
```

启动卡雷尔系统后，切换到 C# 脚本模式。然后我们就可以以面向对象的方式执行卡雷尔任务了。

任务：清洁工 (Cleaner Karel)



Karel 发现房间里到处都是垃圾（用一个 beeper 表示），于是他准备打扫房间。

注意：Karel 初始位置为 (0, 0) 面向东，打扫完后，回到初始位置。

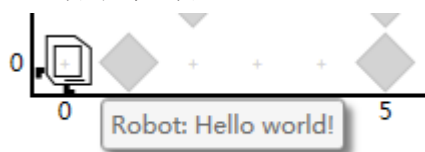
首先将 Main 中的代码删除干净，然后我们创建 Karel 对象，Karel 显然属于 Robot 机器人。所有 Karel 是 Robot 类的对象，在 C# 中用 new 关键字来创建类的对象实例，代码如下：

```
Robot karel = new Robot();
```

代码中我们将创建出来的 Robot 对象赋予 karel 变量，以方便对它进行操作。我们需要做的唯一事情是让 Karel 执行任务，即调用 Robot 对象的 Run 方法：

```
karel.Run();
```

好了，脚本完成，执行脚步。场景中出现：



Robot 类并不知道将要执行什么任务，所有 Run 方法只是简单的说了这一句话！

因为 Robot 类的对象不具备执行清扫任务的能力，所以我们必须定义一个新的 CleanerRobot 类。CleanerRobot 类是 Robot 类的子类，C# 中使用 class 关键字定义，代码如下：

```
public class CleanerRobot : Robot
{
}
```

在 CleanerRobot 类中重写 Robot 类的 Run 方法：

```
public override void Run()
{
```

```

        Say( "CleanRobot: Hello world!" );
    }

```

最后将创建 CleanerRobot 对象赋予 karel 变量，所以将 Main 中的代码改为：

```

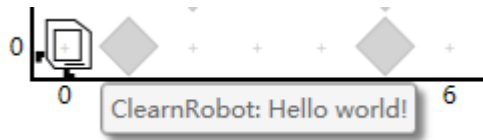
    Robot karel = new CleanerRobot();

    karel.Run();

```

根据多态性，虽然 karel 的声明类是 Robot，但是由于 Run 是 virtual 方法，所有 karel.Run 实际上调用的是定义类 CleanerRobot 中的重写方法。

执行脚本，场景中显示：



既然名为 CleanerRobot，当然要会打扫卫生才行，所有我们要开在 Run 方法里添加代码。任务要求清扫整个场景，我们先写清扫一条街的代码：

```

    Sensor sensor = new Sensor(this);

    while (sensor.FrontIsClear())
    {
        Move();

        PickBeeper();
    }

    while (sensor.BackIsClear())

        MoveBack();

    PickBeeper();

```

代码中先要创建一个 Sensor 传感器对象。在创建对象时，要求指定传感器给那个机器人用，这里我们需要将当前正在调用 Run 方法的 CleanerRobot 对象指定给传感器对象，而 this 关键字总是代表当前对象，所以正是我们要的。

有了传感器后，我们可以通过它判断该机器人前 - 后 - 左 - 右的道路是否畅通。

代码 2-6 行的循环，命令 Karel 一边走一边捡拾地上的垃圾。

代码 7-8 行的循环，命令 Karel 退回到大街的最西端。

最后一行代码，命令 Karel 捡拾起点（大街最西端）处垃圾。

执行脚本，可以看到清扫当前大街的过程。

接下来写清扫所有大街的代码。

我们定义一个 CleanStreet 方法，将 Run 方法中带代码转移过去：

```

    public void CleanStreet()
    {

```



```
}
```

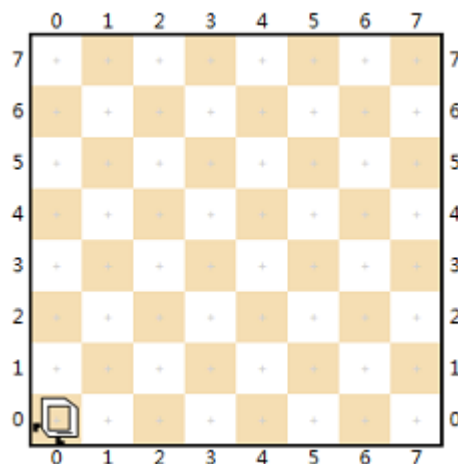
然后，在 Run 中输入：

```
Sensor sensor = new Sensor(this);
while (sensor.LeftIsClear())
{
    MoveLeft();
    CleanStreet();
}
while (sensor.RightIsClear())
    MoveRight();
CleanStreet();
```

这段代码和上一段，惊人的相似。代码 2-6 行：一边向左走（到达一条街）一边清扫一条街的垃圾。代码 7-8 行：向右走回到起点。9 行：清扫最开始的街。

执行脚本，我们可以看到整清扫过程。

任务：摆棋盘 (Checkerboard)



在一个空房间里摆棋盘。

场景中标注为“小麦色”的 Corner 是需要摆放棋子的地方。

注意：棋盘的大小是不确定的。比如可能是  $5 \times 3$  的棋盘。

根据上一个任务的设计经验，需要先定义 CheckerbaordRobot 类，它继承 Robot 类，并重新 Run 方法，然后在 Main 方法中创建该类的对象并执行 Run 方法。

我们向将 Mian 中代码该改正确：

```
Robot karel = new CheckerbaordRobot();
```



```
karel.Run();
```

然后我们需要重新开定义 CheckerbaordRobot 类吗？显然不是，这不符合面向对象原则。既然我们已经定了 CleanerRobot 类，我们就要复用其中的代码。我们让 CheckerbaordRobot 类改从 CleanerRobot 继承，然后，只需要这样实现：

```
public class CheckerbaordRobot : CleanerRobot
{
    private Compass compass = null;
    public override void PickBeeper()
    {
        if (compass == null)
            compass = new Compass(this);
        if ((compass.Avenue + compass.Street) % 2 == 0)
            PutBeeper();
    }
}
```

代码完成，执行脚本，神奇的事情发生了！Karel 会沿着清扫大街的路线前进，但是做着摆放棋子的事情！下面让我来解释一下到底发生了什么。

CleanerRobot 每到一个新的街角都会调用一次 PickBeeper 方法，用于拾取垃圾。而 CheckerbaordRobot 重写了 PickBeeper 方法，将其变成摆放棋子。而根据多态性以 CheckerbaordRobot 对象执行 Run 方法时，PickBeeper 调用执行的是重写的 PickBeeper 方法，即摆放棋子。于是神奇的事情就发生了。

在摆放棋子时，要判断是否需要摆放，根据要求，显然“街号和道号之和是偶数”是摆放棋子的判断条件。于是我么需要创建 Compass 罗盘对象对 Karel 的位置进行定位。罗盘创建时需要第一次初始定位，这是一件非常费时的工作，所有我们不希望每次进入 PickBeeper 都创建新的罗盘，于是我们将罗盘定义为私有字段，然后只有在第一次使用罗盘时才创建它就可以了。

虽然很神奇，但是，这段代码给我们一个反面教材，它违反了面向对象原则：重写不能改变概念，只能改变实现方式。也就是说既然 PickBeeper 方法定义为“取”，就不能实现为“放”。

接下来我们重新实现 CheckerbaordRobot 类，这一次让它从 RobotPro 类继承：

```
public class CheckerbaordRobot : RobotPro
{
}
}
```

RobotPro 类是 Tech.Robots 提供的增强型机器人类，它默认就带有 Sensor、Compass

和一个装有无限多个 Beeper 的 Bag。

RobotPro 类提供了 Go 系列方法，对其进行调用可以使 Karel 连续移动，而且在移动过程中会产生 Going 事件。我们将利用它们对整个棋盘进行“扫街”，方法如下：

1. 先定义 Going 事件处理方法，并和该事件链接。
2. 调用 GoLeft() 向左连续移动。
3. 在 Going 事件处理方法中，当行走方向是 Left 时，用 Sensor 判断前方是否畅通：
  - a) 如果前方畅通：调用 Go() 向前连续移动。
  - b) 如果前方不畅通：调用 GoBack() 向后连续移动。

具体代码如下：

```
public override void Run()
{
    Going += new EventHandler<GoEventArgs>(Process_Going);
    GoLeft();
}
void Process_Going(object sender, GoEventArgs e)
{
    if (e.Direction == Around.Left && !e.IsAfter)
    {
        if (Sensor.FrontIsClear())
            Go();
        else
            GoBack();
    }
}
```

其中，“+=”是一种新的运算符，负责挂接事件处理器。事件处理器是代理类（这里是 EventHandler<GoEventArgs> 类）的对象，它在构造是需要一个事件处理方法（这里是 Process\_Going）作为参数。当事件发生时通过事件处理器，事件处理方法会被调用。

Going 事件会在出发前和完成后各发生一次，这时事件参数的 IsBefore 和 IsAfter 属性分别为 true，然后，每走一步也发生一次，这时上面两个属性都是 false。代码中，if 条件里的 !e.IsAfter 是防止最后一条大街被扫两次。

执行脚本，可以看到 Karel 呈“之字形”扫街。

接下来完成摆放动作，这里我们接收 Moved 事件，在每次移动后根据条件摆放棋子，具体代码如下：

在 Run 方法中 this.GoLeft(); 之前添加：

```

        Moved += new EventHandler<PositionChangeEventArgs>(Process_Moved);
        PutBeeper();
    然后实现消息处理方法 Process_Moved:
    void Process_Moved(object sender, PositionChangeEventArgs e)
    {
        if ((Compass.Avenue + Compass.Street) % 2 == 0)
        {
            PutBeeper();
        }
    }

```

其中 PutBeeper() 为在起始位置摆放棋子，因为起始位置没有移动事件发生。

执行脚本，看到效果。

如果能让 Karel 在摆放完棋盘回到起点。为了不重复摆放棋子，需要将 Going 和 Moved 事件处理链接断开，具体代码如下：

在 Run 方法中 this.GoLeft(); 之后添加：

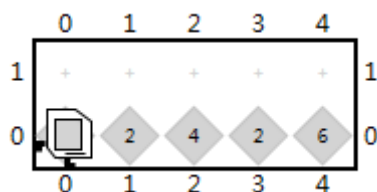
```

        Going -= new EventHandler<GoEventArgs>(Process_Going);
        Moved -= new EventHandler<PositionChangeEventArgs>(Process_Moved);
        GoBack();
        GoRight();
    
```

其中 “-” 操作符号和 “+” 刚好相反，用于断开事件链接。

另外，这段代码中只要将 Process\_Moved 方法的实现改为 PickPeeper 就可以完成“清洁工”任务。

任务：方块翻倍 2(DoubleBeeper2)



这是第四程序 DoubleBeeper 的升级版，所以叫 DoubleBeeper2。场景是初始状态：1、2、4、2、6，加倍后的状态应为：2、4、8、4、12。编程实现。

注意：房间的宽度是任意的，不一定是 5。

脚本模板同上，新类叫做 DoubleBeeper2Robot，从 RobotPro 类继承。定义 Going 消息

处理方法，重写 Run 方法，在其中连接 Going 事件，并调用 Go()：

```
public class DoubleBeeper2Robot : RobotPro
{
    public override void Run()
    {
        Going += new EventHandler<GoEventArgs>(Process_Going);
        Go();
    }
    void Process_Going(object sender, GoEventArgs e)
    {
    }
}
```

接下来是关键，RobotPro 默认定义了一个有无限个 beeper 的 Bag，为了可以计算并翻倍 Beeper，我们需要定义 DoubleBeeperBag 类，它从 Bag 类继承，代码如下：

```
public class DoubleBeeperBag : Bag
{
    private int _beepers = 0;
    public override bool BeepersInBag(int beepers)
    {
        return _beepers >= beepers;
    }
    public override void PutIntoBeepers(int beepers)
    {
        _beepers += beepers * 2;
        base.PutBeepers(beepers)
    }
    public override void TakeOutBeepers(int beepers)
    {
        _beepers -= beepers;
        base.PutBeepers(beepers)
    }
}
```

程序使用 \_beepers 作为计算变量，当其值大于等于检测数量时，表明 Bag 中 Beepers 数量足够。通过 PutBeeper 方法在每次放入时 \_beepers 个数加 2 倍的放入数，TakeBeeper

方法在每次取出时 `_beepers` 只减 1 倍的取出数，这样就达到了翻倍的效果。

为了使替换掉 `RobotPro` 中 `Bag` 的默认对象，我们必须定义 `DoubleBeeper2Robot` 类的初始化方法，在 C# 中初始化方法为和类同名的无返回值方法，具体代码如下：

```
public DoubleBeeper2Robot()
{
    Bag = new DoubleBeeperBag();
}
```

初始化方法，在对象创建时自动被调用，所以我们就有机会在对象创建时将 `Bag` 赋值正确。

最后我们需要在 `Going` 事件处理函数里添加如下代码：

```
if (e.Direction == Around.Front && !e.IsAfter)
{
    PickBeepers();
    PutBeepers(int.MaxValue);
}
```

`PickBeepers` 和 `PutBeepers` 为 `RobotPro` 提供的方法用于取放出多个 `Beepers`，`int.MaxValue` 表示放多个直到直到 `Bag` 里没有为止。

执行脚本，成功！

## 5.3 如何定义新 Robot 类

从这一节开始，我们将仅从 `move`、`turnleft`、`pickbeeper`、`putbeeper`、`turnoff` 这五条指令出发来构造自己的 `Robot` 类。

在此之前我们需要先介绍以下新的多机器人指令集合。多机器人指令和定义在 `Karel` 类中的基本指令几乎一模一样，只不过它们定义在 `Karel.Robots` 类中，而且每个指令都多个了一个叫做 `Id` 的参数。比如：

```
public static bool Move(string id = null, bool transmit = false);
public static void PutBeeper(string id = null);
```

在 `Karel` 的世界里可以存在多个机器人，每个机器人都有一个唯一的 `Id`。`Karel` 这个世界上诞生的第一个机器人，它的 `Id` 为 `null`。其他的机器人都是 `Karel` 的仿造品，并且 `Id` 不能为空。字符串值 `Id` 是机器人的身份象征，也代表了机器人的某些特点。如果 `id` 是颜色，则机器人将会呈现这种颜色的皮肤。比如：“`wheat`”的机器人将呈现“小麦色”。



而且其他 id 会在机器人左上角显示出来。比如：“Rose”。



有了这些新的指令后，我们就可以开始打造自己的机器人系列类了。

## Engine 类

“引擎”，作为机器人的最重要部件，它是操控机器人动作的执行部件。Engine 类声明如下：

```
public class Engine { }
```

在 C# 中如果在声明类的时候没有指定基类，那么编译器会默认以 Object 类为基类，也就是说上面的声明等于：

```
public class Engine : Object { }
```

由于，每个机器人都要配备一个 Engine，因此 Engine 必须区分不同的机器人，这就要求 Engine 知道了不同机器人的 Id。所以 Id 是 Engine 的一个属性，表明该 Engine 被安置在那个机器人上。而且 Engine 对象在初始化时必须给定 Id 值。关于 Id 的定义如下：

```
public string Id { get; private set; }  
public Engine(string id = null)  
{  
    Id = id;  
}
```

C# 的属性其实是有两个方法组成的，它们分别是 get 方法和 set 方法。get 方法用于得到属性值，set 方法用于设置属性值。一般情况下我们不不用实现这连个方法，C# 会自动为我们实现。

属性 Id 的 set 方法被定义成 private，表明 Engine 对象一旦创建就和给定 id 绑定在一起了，外部代码无权改变。

有了 Id 属性后，Engine 中就可以添加基本指令了：

```
public bool Move(bool transmit = false)  
{  
    return Karel.Robots.Move(Id, transmit);  
}
```

显然，这些基本指令，就是对 Karel.Robots 中的指令的浅 wrapper 包装。以上只以 Move 为例，用类似的方法，还可以定义：TurnLeft、PickBeeper、PutBeeper、TurnOff 以及 Sleep 和 Say 方法。

接下来，我们对这些基本指令进行一些扩展，定义 TurnRight、TurnBack、MoveLeft、MoveRight 和 MoveBack 方法。这些方法的具体实现，本书前面几章已经作了详细的讲解，这里不再重复。

最后，我们需要添加事件。在这里我们添加三个事件：

PositionChanged 位置变化：

当 Robot 在场景中的位置改变后触发该事件。事件定义为：

```
public EventHandler<PositionChangeEventArgs> PositionChanged;
```

FacingChanged 方向变化：

当 Robot 在方向发生改变后触发该事件。事件定义为：

```
public EventHandler<FacingChangeEventArgs> FacingChanged;
```

其中 FacingChangeEventArgs 为事件参数，它有两属性：IsOff 和 IsReset。

IsOff 是 true：表示 Robot 的方向发生改变是关机引起的。

IsReset 是 true：表示 Robot 的方向发生改变是冲启动引起的。

IsOff 和 IsReset 都是 false：表示 Robot 的方向发生改变是左转引起的。

BeeperChanged 蜂鸣器变化：

当 Beeper 个数改变后触发该事件。事件定义为：

```
public EventHandler<BeeperChangeEventArgs> BeeperChanged;
```

其中 BeeperChangeEventArgs 的属性：Beepers 为整数类型，表示 Robot 拥有的 Beeper 个数的改变。显然在 PickBeeper 成功时 Beepers == 1，而在 PutBeeper 时 Beepers == -1。

定义了事件后就是触发事件了，我们以 FacingChanged 为例子，说明如何触发事件。显然“方向改变”有两个触发源：TurnLeft 和 TurnOff 方法。触发源一般不会直接触发事件，而是发出消息给事件触发函数。事件触发函数一般是：保护的名为 OnXXX 的虚方法，其中 XXX 为事件名称。FacingChanged 的事件触发函数定义为：

```
protected virtual void OnFacingChanged(FacingChangeEventArgs args)
{
    if (FacingChanged)
        FacingChanged(this, args);
}
```

事件触发函数的主要工作就是调用事件代理，从而触发事件。

而事件源，则调用事件触发函数：

```
public void TurnLeft()
{
```



```

        Karel.Robots.TurnLeft(Id);
        OnFacingChanged(new FacingChangeEventArgs(false, false));
    }

    public void TurnOff(bool reset = false)
    {
        Karel.Robots.TuTurnOffrnLeft(Id, reset);
        OnFacingChanged(new FacingChangeEventArgs(true, reset));
    }

```

以上就是定义和触发事件的标准写法。其他两个事件类似，篇幅有限不再重复讲解。

讲到这里读者可能会问：“为什么会这么复杂呢？事件源直接触发事件不就可以了吗，为什么要事件触发函数呢？”要回答这些问题还不是很容易，这牵涉到了“设计模式”的问题。在入门阶段，我们可以这样理解：事件触发函数，给了子类一个在事件发送前后，对事件进行处理的机会。

以上就是 Engine 类的所有定义。还记得前两章定义的扩展类 KarelEx 吗？显然其中还定义了非常有用的方法：PickBeepers、PutBeepers、Go、GoLeft、GoBack、GoRight。为使得它们可以在面向对象方式下被使用，我们同样定义 EngineEx 类从 Engine 继承：

```

public class EngineEx : Engine
{
    public EngineEx(string id = null) : base(id) { }
}

```

由于 Engine 类的构造函数参数个数不为零，所以作为其子例子，我们也必须定义初始化函数，而且要使用 base 将 id 传给 Engine 的初始化函数。上面是标准的写法。

接下来就可以仿照 KarelEx 的实现方法，实现这些扩展指令了。

至此对于“引擎”的打造就此完成。当然我们还可以按照 EngineEx 的办法无限制的扩展 Engine。

## Robot 类

Robot 类首先要定义当然 Id 属性，定义方法和 Engine 的类似。

接着 Robot 类就需要负责维护 Id 所代表的机器人，对机器人的维护有两种：

第一种，是在场景设计的时候就已经由任务设计师创建好了，这里最典型的的就是 Karel 了。对于这种机器人，Robot 对象在构造时，只是负责挂接到对应的机器人身上，

所以初始化函数定义如下：

```
public Robot(string id = null)
{
    Id = id;
}
```

而且在 Robot 对象结束化时并不销毁该机器人！

第二种，是由于任务需要由任务代码创建的机器人。对于这类机器，Robot 对象在构造时要负责创建它，这就需要设定它的初始位置和方向，代码如下：

```
public Robot(string id, int street, int avenue, Orientation facing = Orientation.East)
{
    if (CreatedRobotIds.Contains(id))
        throw new ArgumentException(string.Format( "The robot {0} has been created!" ,
id));

    CreatedRobotIds.Add(id);
    Id = id;
    Karel.Robots.CreateRobot(id, street, avenue, facing.ToString());
}
```

而且在 Robot 对象结束化时，负责销毁该机器人代码如下：

```
~Robot()
{
    this.Dispose();
}
public void Dispose()
{
    if (CreatedRobotIds.Contains(Id))
    {
        Karel.Robots.DestroyRobot(Id);
        CreatedRobotIds.Remove(Id);
    }
}
```

这里需要解释三点：

1. Karel.Robots 类的 CreateRobot 和 DestoryRobot 类负责创建和销毁机器人。
2. CreatedRobotIds 是一个集合，定义如下：

```
private static HashSet<string> CreatedRobotIds = new HashSet<string>();
```

用来避免一个机器人被创建或销毁多次。这是一个有关数据结构的使用方法，有关的详细讲解见下一章。

3. 结束化方法 ~Robot 中没有直接销毁机器人，而是交给名为 Dispose 的方法进行。而其实后者是接口 IDisposable 的实现。看看 Robot 类的声明你就会明白了：

```
public class Robot : IDisposable { }
```

实现了这个接口的好处是，我们可以使用下面的 C# 语法糖：

```
using(Robot rose = new Robot( "rose" , 1, 1))
{
    ...
}
```

语句 using，会保证在 “}” 处，自动调用 rose.Dispose()。

这样可以保证在 rose 使用完后立即销毁，而不是等到 GC 回收。

接下来是要构造 Engine 对象了。首先定义一个名为 Engine 的属性，用于连接 Engine 对象（也叫做聚合）：

```
public Engine Engine { get; private set; }
```

接下来需要在初始化函数中创建 Engine 对象。不过这里有一点小麻烦，Robot 类并不知道创建 Engine 对象还是 EngineEx 对象，其实就 Robot 类本身来说 Engine 就已经足够强悍了，但是这不能保证子类就足够使用。解决这个矛盾的方法是定义 Engine 的工厂方法：

```
protected virtual Engine CreateEngine()
{
    return new Engine(Id);
}
```

Engine 工厂方法的默认实现时创建 Engine 对象，而子类可以 override 该工厂方法创建别的 Engine 对象，只要它愿意！

最后我么需要在两个初始化函数中添加如下代码：

```
Engine = CreateEngine();
```

至此 Engine 的构造完毕！

还记得 KarelEx 中的所有方法在最后都必须调用 Sleep 指令让观众可以看清 Karel 动作吗？显然 Robot 类也要解决这个问题。这里定义第一个名为 Daley 的方法，和一个名为 DelayTime 的属性：

```
public int DelayTime { get; set; }
protected void Delay()
```

```

{
    Engine.Sleep(DelayTime);
}

```

这样的设计显然比 KarelEx 更高档一些，这里允许改变每个机器人的动作快满。其中 DelayTime 属性在初始化方法中被设置为 100，使得默认时和 KarelEx 的动作快慢保持一致。另外显然，Delay 的实现需要使用 Engine。

接着我们就可以在 Robot 上定义基本指令：Move、TurnLeft、PutBeeper、PickBeeper、TurnOff 和 Say，以及相关的事件 Moved 和 Turned。

这里需要说明的是：

1. 这些指令都没有返回值，也就是说我们不从这些指令处获得该动作是否成功的信息，而是另有来源。

2. Moved 事件和 Engine 的 PositionChanged 事件类似，而 Turned 的事件和 Engine 的 FacingChanged 事件类似。这里没有 BeeperChanged 事件是因为，Robot 并不负责直接管理 Beepers，而是另有其他部件。

特殊说明的是：

1. 这里 Move 指令没有参数，默认实现就是 transmit 模式。具体代码如下：

```

public virtual void Move()
{
    if (Engine.Move(true))
    {
        OnMoved(new PositionChangeEventArgs());
        Delay();
    }
}

```

显然 Robot 是一个可以自由地在整个场景中移动的机器人。

2. Say 指令的等待时间，会根据参数 words 中的文字多少动态计算，但是保证不小于 DelayTime，不大于 5 秒。如果 DelayTime 大于 5 秒，以 DelayTime 为准。

接下来我们需要定义扩展指令：TurnRight、TurnBack、MoveLeft、MoveBack 和 MoveRight。以 TurnBack 为例，基于 KarelEx 的经验，代码如下：

```

public virtual void TurnBack()
{
    TurnLeft();
}

```

```
        TurnLeft();  
        Delay();  
    }
```

这样的定义，其实存在一个问题：TurnLeft 方法会调用 Delay 停顿，所以在场景中，观众会看到 Robot 向左转动了两次，而不是一次转向后方。为了解决这个不足我们需要提供将多个动作变成一个动作的子动作的方法，这就需要增加两方法：BeginAct 表示一个动作的开始，而 EndAct 表示一个动作的结束，并修改 Delay 中的部分实现，具体代码如下：

```
private int _noDelayCount = 0;  
public void BeginAct()  
{  
    _noDelayCount++;  
}  
public void EndAct(bool doDelay = true)  
{  
    _noDelayCount--;  
    if (doDelay)  
        Delay();  
}  
protected void Delay()  
{  
    if (_noDelayCount == 0)  
        Engine.Sleep(DelayTime);  
}
```

有了 BeginAct 和 EndAct 后，TurnBack 的最后定义如下：

```
public virtual void TurnBack()  
{  
    BeginAct();  
    TurnLeft();  
    TurnLeft();  
    EndAct();  
}
```

有关 BeginAct 和 EndAct 的实现，这是一个经典的实现技巧：利用一个计数器 \_noDelayCount，来表示当前是否进入 Act 定义阶段，在这个阶段中 Delay 将不起作用。

读者可能会：“问为什么使用的是 int 类型的计数器，而不是 bool 类型的标志呢？”

其原因是：由一组子动作组成的动作可能是另外的动作的子动作，这就导致 BeginAct - EndAct 组合发生嵌套。比如 TurnRight 扩展指令的实现：

```
public virtual void TurnRight()
{
    BeingAct();
    TurnBack();
    TurnLeft();
    EndAct();
}
```

显然在调用 TurnBack 时发生了上述的嵌套情况！我们当然可以保证 TurnRight 用三个 TurnLeft 来实现，但是我们不能保证以后的使用中不发生这种情况。

最后 Robot 类需要定义名为 Run 的方法，它表明 Robot 可以执行任务，当然对于 Robot 类来说，不知道完成什么任务，所以默认实现是：

```
public virtual void Run()
{
    Say( "Robot: Hello world!" );
}
```

这表明：“我说自我在！”

## 5.4 扩展 Robot

从对象的角度出发，扩展 Robot 有两个方向，一是定义更多的类似于 Engine 的 Robot 部件，二是从 Robot 继承定义新的 Robot 子类。这一节使用第一种方式来扩展 Robot，下一节使用第二种方式。

### Sensor 类

刚才在定义 Robot 时，所有的基本动作都没有返回值，所以我们不能从基本动作获得 Robot 周围的基本环境信息，于是我们需要定义一个感知部件来感知这些信息，这就是 Sensor 类。

Sensor 类既然是 Robot 的部件所以首先要有一个属性表示它所属的 Robot，并在创建对象时初始化正确。基于这个要求，我们可以初步定义 Sensor 类如下：

```
public class Sensor
{
    public Robot Robot { get; private set; }
```

```
public Sensor(Robot robot)
{
    Robot = robot;
}
}
```

接下, 需要向其中添加方法。根据需求, 可以知道, Sensor 需要让 Robot 看到: 周围的墙, 听到: 地上是否有 Beeper 发出声音, 于是我们可以列出 Sensor 需要定义的最基本方法:

FrontIsClear: 返回一个逻辑值表示前方街道是否畅通。要实现这个方法, 我们可以使用 Robot 的 Engine 中的 Move 方法, 让 Robot 向前移动。如果移动失败那么前方一定有墙或是边境, 道路不畅通; 如果成功那么一定道路畅通, 不过这时我们需要让 Robot 退回来, 就好像它没有移动一样! 具体代码如下:

```
public virtual bool FrontIsClear()
{
    if (!Robot.Engine.Move())
        return false;
    Robot.Engine.MoveBack();
    return true;
}
```

BeepersInPresent: 返回一个逻辑值表示当前街角上是否有 beeper 存在。实现方法和 FrontIsClear 类似:

```
public virtual bool BeepersPresent()
{
    if (Robot.Engine.PickBeeper())
    {
        Robot.Engine.PutBeeper();
        return true;
    }
    return false;
}
```

给予基本方法 FrontIsClear, 可以定义 LeftIsClear 方法:

```
public virtual bool LeftIsClear()
{
    Robot.Engine.TurnLeft();
    bool r = FrontIsClear();
}
```

```
Robot.Engine.TurnRight();  
return r;  
}
```

这里需要注意的是 LeftIsClear 中调用 FrontIsClear 来实现，而不是直接用 Robot.Engine.LeftMove 实现。这是因为：这样做可以方便子类改变策略时只用重写 FrontIsClear 一个方法。

类似的可以定义 RightIsClear 和 BackIsClear 方法。

## Bag 类

Robot 将从街角建起来的 Beeper 放在一个无限大的包中，而有的时候我们需要一个有限个数的包，于是我们定义 Bag 类。和 Sensor 不同 Bag 类并不知道谁在使用他，他可以被一个 Robot 使用也可以被多个 Robot 使用。所以 Bag 类的定义如下：

```
public class Bag { }
```

下面定义方法和事件。

和 Robot 从街角捡拾 Beeper 一样，这里定义 TakeBeepers、PutBeepers 和 BeepersChanged 事件，代码如下：

```
public virtual void TakeOutBeepers(int beepers)  
{  
    OnBeepersChanged(new BeeperChangeEventArgs(beepers));  
}  
public virtual void PutIntoBeepers(int beepers)  
{  
    OnBeepersChanged(new BeeperChangeEventArgs(-beepers));  
}  
public event EventHandler<BeeperChangeEventArgs> BeepersChanged;  
protected virtual void OnBeepersChanged(BeeperChangeEventArgs args)  
{  
    if (BeepersChanged != null)  
        BeepersChanged(this, args);  
}
```

对于 Bag 来说，显然需要允许一次取或放多个 Beepers。

最后定义 BeepersInBag 方法：

```
public virtual bool BeepersInBag(int beepers)  
{  
    return true;
```



```
}
```

从 BeepersInBag 的实现可以看出 Bag 类的实现也是一个无限包，它只是给使用者一个机会定义其他的各种包包。在第 2 节中，我们就继承 Bag 定义了一个翻倍 beeper 的包包。

“无为而无所不为”或“less is more”是基类设计的精髓！

为了方便我们从 Bag 出发定义一个可以计数 Beeper 的包包：

```
public class BeeperBag : Bag
{
    public const int Infinity = int.MaxValue;
    public BeeperBag(int beepers = 0)
    {
        Beepers = beepers;
    }
    public int Beepers { get; set; }
    public override bool BeepersInBag(int beepers)
    {
        return Beepers >= beepers;
    }
    public override void TakeOutBeepers(int beepers)
    {
        if (beepers != Infinity)
            Beepers -= beepers;
        base.TakeOutBeepers(beepers);
    }
    public override void PutIntoBeepers(int beepers)
    {
        if (beepers != Infinity)
            Beepers += beepers;
        base.PutIntoBeepers(beepers);
    }
}
```

这里的关键是定义了一个常量 Infinity 表示无限（因为 int 类型没有无限值，所以我们用最大值表示无限）。在 C# 中常量定义需要加关键字 const，另外常量一定是静态的所有

不需要 static 关键字。常量顾名思义，就是定义后就不能改变。常量可以防止程序员的失误修改！

## Compass 类

“罗盘”用于让 Robot 在场景中的定位。和 Sensor 类一样 Compass 需要和 Robot 建立联系，类的定义如下：

```
public class Compass : IDisposable
{
    public Robot Robot { get; private set; }
    public Compass(Robot robot)
    {
        Robot = robot;
    }
}
```

罗盘信息包括：场景大小、机器人位置和朝向。将它们都定义成属性：

```
public int Street { get; private set; }
public int Avenue { get; private set; }
public Orientation Facing { get; private set; }
public int Streets { get; private set; }
public int Avenues { get; private set; }
```

接下来是如何将这些信息设置正确的问题的问题，基本思路如下方法如下：

1. 在创建 Compass 对象时，获得场景大小，和当前 Robot 的定位信息。（获得初始信息的方法我们稍候讲解）

2. 在初始化方法里连接 Robot.Engine 类的 PositionChanged 和 FacingChanged 事件：

```
robot.Engine.PositionChanged += new EventHandler<PositionChangeEventArgs>(Robot_PrimitivelyMoved);
```

```
robot.Engine.FacingChanged += new EventHandler<FacingChangeEventArgs>(Robot_PrimitivelyTurned);
```

当然需要在结束化时，断掉这种连接：

```
~ Compass()
{
    Dispose();
}

public void Dispose()
```

```
{
    if (this.Robot == null)
        return;

    Robot.Engine.PositionChanged -= new EventHandler<PositionChangeEventArgs>(Robot_PrimitivelyMoved);

    Robot.Engine.FacingChanged -= new EventHandler<FacingChangeEventArgs>(Robot_PrimitivelyTurned);

    Robot = null;
}
```

并在事件处理函数中，同步 Robot 的定位信息：

```
void Robot_PrimitivelyTurned(object sender, FacingChangeEventArgs e)
```

```
{
    if (e.Off)
    {
        if (e.Reset)
            Facing = Orientation.East;
        else
            Facing = Orientation.West;
    }
    else
    {
        switch (Facing)
        {
            case Orientation.East:
                Facing = Orientation.North;
                break;
            case Orientation.North:
                Facing = Orientation.West;
                break;
            case Orientation.West:
                Facing = Orientation.South;
                break;
            case Orientation.South:
                Facing = Orientation.East;
```

```
        break;
    }
}
}
void Robot_PrimitivelyMoved(object sender, PositionChangeEventArgs e)
{
    switch (Facing)
    {
        case Orientation.East:
            Street += 1;
            break;
        case Orientation.North:
            Avenue += 1;
            break;
        case Orientation.West:
            Street -= 1;
            break;
        case Orientation.South:
            Avenue -= 1;
            break;
    }
}
```

接下来重点讲解如何初始定位。

首先不管 Robot 现在处于场景中何处方向如何，我们都可以让他一直向前穿越到边界，然后向后转，再一直向前穿越到边界，然后向右转，再一直向前穿越到边界，然后向后转，再一直向前穿越到边界。我们将四次穿越的步数 +1 保存在四个变量: fx, width, fy, height 中。相应的代码如下：

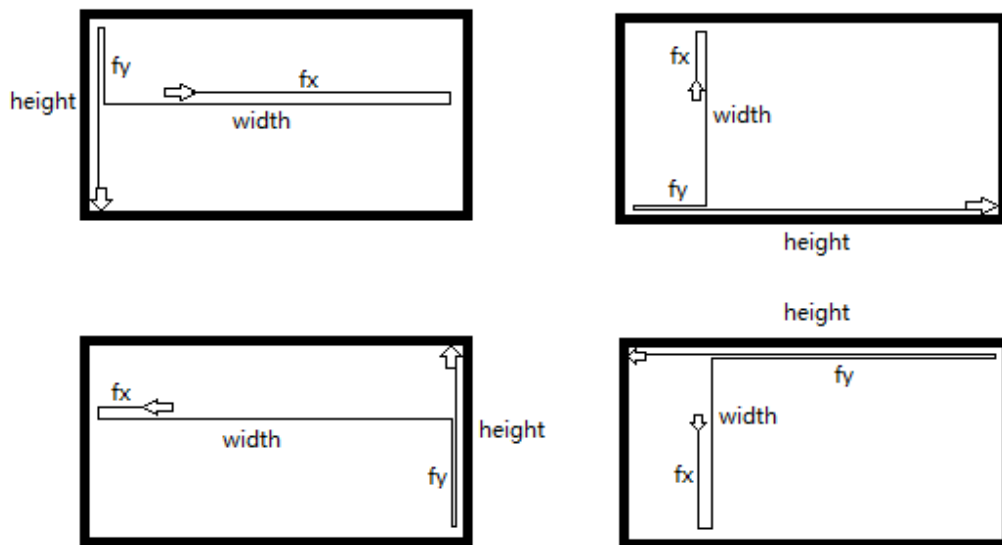
```
int fx = 1;
int width = 1;
int fy = 1;
int height = 1;
while (robot.Engine.Move(true))
    fx++;
robot.Engine.TurnBack();
```

```

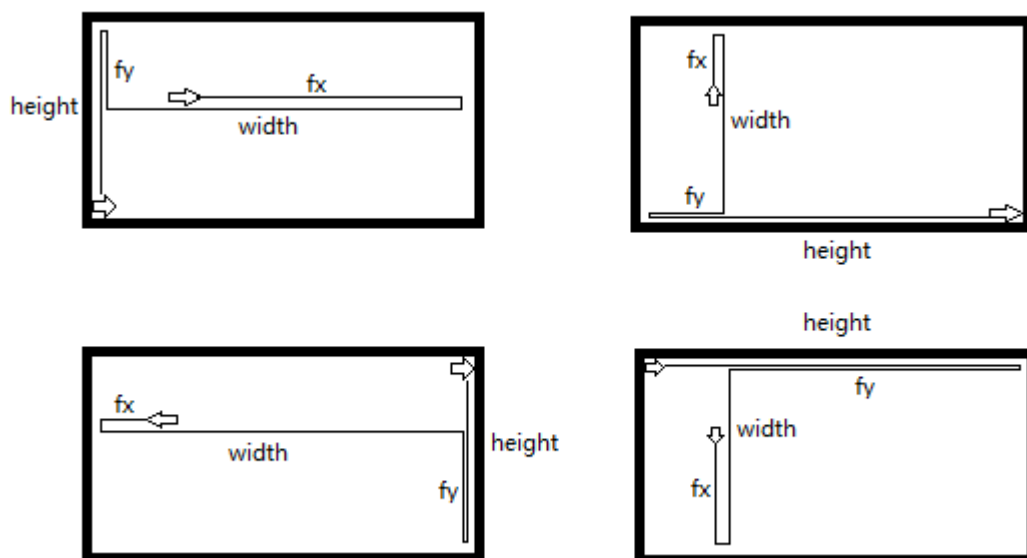
while (robot.Engine.Move(true))
    width++;
robot.Engine.TurnRight();
while (robot.Engine.Move(true))
    fy++;
robot.Engine.TurnBack();
while (robot.Engine.Move(true))
    height++;

```

为了进一步分析方便, 我们将 Robot 分别向东、北、西、南的穿越路线图画出来, 如下:



进行穿越后不管哪种情况, 都是: 前方和右方是边界。这时我们 Reset。四种情况的方向变为:



显然这时根据两面墙的位置, 我们知道当前位置, 并可以反推出来 Robot 的起始朝向。

具体代码如下：

```
Sensor sensor = new Sensor(robot);
Orientation facing;
if (sensor.FrontIsClear())
{
    if (sensor.LeftIsClear())
        facing = Orientation.East;
    else
        facing = Orientation.South;
}
else
{
    if (sensor.LeftIsClear())
        facing = Orientation.North;
    else
        facing = Orientation.West;
}
```

有了起始朝向，我们就可以决定是哪种情况，于是就可以根据四个变量求出 streets 场景的大街数，avenues 场景的大道数，street 起始街号，avenue 起始道号。并且还可以退回到原来的位置。具体代码如下：

```
int x = 0, y = 0;
bool needSwap = false;
switch (facing)
{
    case Orientation.East:
        x = width - fx;
        y = height - fy;
        break;
    case Orientation.South:
        x = fx - 1;
        y = height - fy;
        robot.Engine.TurnRight();
        needSwap = false;
        break;
```

```
case Orientation.North:
    x = width - fx;
    y = fy - 1;
    robot.Engine.TurnLeft();
    needSwap = true;
    break;
case Orientation.West:
    x = fx - 1;
    y = fy - 1;
    robot.Engine.TurnBack();
    break;
}
fx = width - fx;
fy = height - fy;
while (fx-- != 0)
    robot.Engine.Move(true);
robot.Engine.TurnLeft();
while (fy-- != 0)
    robot.Engine.Move(true);
robot.Engine.TurnRight();
if (needSwap)
{
    Street = y;
    Avenue = x;
    Streets = height;
    Avenues = width;
}
else
{
    Street = x;
    Avenue = y;
    Streets = width;
    Avenues = height;
}
```

```
Facing = facing;
```

以上就是初始定位的代码，这些代码都写在类的初始化函数里。显然初始定位很耗费时间，所以 Compass 对象创建时只初始化一次，以后到靠事件同步。

## 5.5 定义 RobotPro 类

RobotPro 类从 Robot 继承，将使用上面定义的三个部件，并且扩展取放多个 Beepers 和走多步的方法，并将 Engine 升级到 EngineEx。

### 使用 Compass

RobotPro 类并不确定，用户是否会使用 Compass 类，而且 Compass 对象创建时候会耗时，解决的方法是使用时才创建，具体代码如下：

```
private Compass _compass = null;
public virtual Compass Compass
{
    get
    {
        if (_compass == null)
            _compass = new Compass(this);

        return _compass;
    }
    set
    {
        if (value != null && !object.ReferenceEquals(value.Robot, this))
            throw new ArgumentException( "This compass does not belong to me!" );

        if (object.ReferenceEquals(_compass, value))
            return;

        if (_compass != null)
            _compass.Dispose();

        _compass = value;
    }
}
```



```
}
```

另外我们要允许用户设置新的 Compass 对象，也许他找到了更快速的 Compass 实现也不一定！但这里必须判断该 Compass 属于本 Robot。

## 使用 Sensor

属性的定义和 Compass 类似，这里不写出它的实现。但需要注意一点是，RobotPro 类必须遵照 Sensor 的指导先进和捡拾地上的 Beepers 当不能前进或地上已经没有 Beepers 时，需要停止动作。这就需要重写 Move 和 PickBeeper 方法。

```
public override void Move()
{
    if (Sensor.FrontIsClear())
        base.Move();
}

public override void PickBeeper()
{
    if (Sensor.BeepersPresent())
    {
        base.PickBeeper();
    }
}
```

## 使用 Bag

属性定义和上面类似，主要是使用方法。

首先在 PickBeeper 中 base.PickBeeper(); 语句下添加下面的代码：

```
    Bag.PutBeepers(1);
```

然后重写 PutBeeper 方法：

```
public override void PutBeeper()
{
    if (Bag.BeepersInBag(1))
    {
        Bag.TakeBeepers(1);
        base.PutBeeper();
    }
}
```

## 升级 Engine

根据 Robot 的设计，我们显然需要重新 CreateEngine 方法，来升级 Engine：

```
protected override Engine CreateEngine()
{
    return new EngineEx(this.Id);
}
```

另外，需要重新定义 Engine 属性，将其类型变为 EngineEx：

```
public new EngineEx Engine {
    get { return base.Engine as EngineEx; } }
```

“as” 是一个运算符（虽然看起来不像符号），它将左边的对象看成是右边的类型。

## 扩展 Go/GoLeft/GoRight/GoBack

这里已 Go 为例，其他相似。

和 KarelEx 中的 Go 类似，只不过这里需要定义事件，代码如下：

```
public virtual void Go(int steps = int.MaxValue)
{
    int expected = steps;
    OnGoing(new GoEventArgs(Around.Front, expected, expected - steps, true, false));
    while (steps != 0 && Sensor.FrontIsClear())
    {
        steps--;
        Move();
        OnGoing(new GoEventArgs(Around.Front, expected, expected - steps, false, false));
    }
    OnGoing(new GoEventArgs(Around.Front, expected - steps, 0, false, true));
}

public event EventHandler<GoEventArgs> Going;
public virtual void OnGoing(GoEventArgs args)
{
    if (Going != null)
        Going(this, args);
}
```

最后是扩展 PickBeepers 和 PutBeepers 以及重写 Run 中招呼语为："RobotPro: Hello

world!", 篇幅有限这里就不写出代码了!

还记得上一章的拼图游戏吗? 有了 RobotPro 类, 我们可以轻松完成任务, 并且还可以解决第一个缺点:

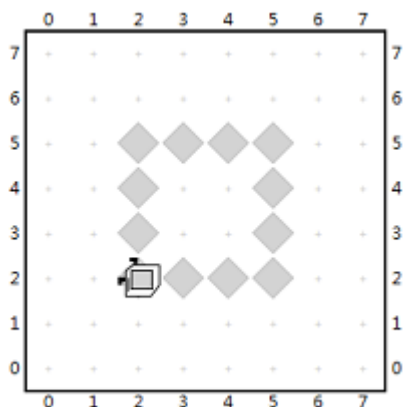
```
public class PuzzleRobot : RobotPro
{
    public PuzzleRobot()
    {
        Bag = new BeeperBag();
    }
    public override void Run()
    {
        while(true)
        {
            switch (Karel.Listen() as string)
            {
                case "esc" :
                    return;
                case "right" :
                    if (Sensor.FrontIsClear())
                    {
                        BeginAct();
                        Move();
                        PickBeepers();
                        MoveBack();
                        PutBeepers();
                        Move();
                        EndAct();
                    }
                    break;
                ...
            }
        }
    }
}
```

```
}
```

## 思考题

A.Karel 在起点面朝北，包里有 1 个 Beeper，东边三个街区有另一个名为 Carl 的 Robot，面朝东没有 Beeper。世界中没有 Wall。Karel 移动到 Carl 将 Beeper 交给它。Carl 再向北移动两个街区后放下 Beeper。两个 Robot 随后回到起点并呈初始朝向。

B.Karel 最终状态如下图所示，Karel 程序排列 Beeper。Karel 初始带有 12 个 Beeper，可在任意一个转角开始。



C. 给 class 中添加特性：

- (1). 新 Robot 拥有 Robot 类中机器人的所有特性；
- (2). 能够执行方法 PickAPair，能够一次性的从当前街角捡起两个 Beeper。

新 class 命名为 PickAPairRobot。写出程序。

D. 假定我们在同一个 Robot class 中定义了一个方法：

```
public void move()
{
    base.Move();
    Move();
}
```

如果发送给我们的 Robot 这样一个消息，会发生什么？如果你要测试这个，将你的 Robot 开始时候面朝西或者南。使用这个方法的移动指令意味着什么？

E. 重新阅读第 3 章的思考题 E，有没有能力写出解决这些任务的更短代码？

F. 为什么易读的程序很重要？

G. 我们如何能够写出易读程序？

H. 重做第 4 章思考题 C（发送问候），分别按照以下要求完成：

- (1). 这次使用 5 个 Robot，每个写一个字母。你可以定义每个 Robot 起始位置。

- (2). 使用 5 个 Robot, 每个一行。
- (3). 使用 16 个 Robot, 每个一列。
- I. 重做第 4 章思考题 D (显示时间) , 分别按照以下要求完成:
  - (1). 使用 5 个 Robot, 每个负责一个数字或符号。
  - (2). 使用 16 个 Robot, 每个一列。
  - (3). 使用 5 个 Robot, 每个负责一个数字或符号。
- J. 重做第 4 章思考题 E (种植 Beeper) , 分别按照以下要求完成:
  - (1). 使用 4 个 Robot。
  - (2). 使用 8 个 Robot。
  - (3). 使用一个 Choreographer 和它的三个 Helper。部分实现如下:

```
public class Choreographer : RobotPro
{
    // 三个 Helper, 在 Choreographer 工作时也同时工作
    private RobotPro lisa = new RobotPro( "Lisa" ,
        4, 2, Orientation.East);
    private RobotPro tony = new UrRobot( "Tony" ,
        6, 2, Orientation.East);
    private RobotPro tom = new UrRobot( "Tom" ,
        8, 2, Orientation.East);

    public void Harvest(){...}
    public void HarvestARow(){...}
    public void HarvestCorner(){...}
    public override void Move()
    {
        base.Move();
        lisa.Move();
        tony.Move();
        tom.Move();
    }

    public override void PickBeeper(){...}
    public override void TurnLeft(){...}
    public override void TurnOff(){...}
    public Choreographer(...){...}
}
```

```
public static void Main()
{
    Choreographer karel = new Choreographer(null,
        2, 2, Orientation.East);
    karel.Harvest();
    karel.TurnOff();
}
```

K. 重做第 4 章思考题 A（丰收），使用一队 6 个 Robot，每个收获一行。

## 6. 数据结构

数据这个词，基本上是相对于代码而言的，指不能执行的但是可以被处理（分析、计算等）的数值。（注意：这个定义只对 C# 等一部分语言有效，而对于向 Lisp 等语言代码也是数据的一种）

前面章节中，我们已经接触过的一些数据：int 整数、double 小数、bool 逻辑值，这些数据因为不能被再次细分，所以叫做原子数据；而另一种数据与此相反，它们是由更小的数据按照按照一定的结构关系组合而成，这种数据成为复合数据。在 C# 中，比较特殊的是 string 字符串，它本身是复合数据，但是往往表现出原子殊绝的某些特点，所以它叫做半原子类型。

在 C# 中，所有东西都是对象，对象本身都是数据和代码的合体，显然站在对象的角度讲，C# 中没有数据。因此在 C# 中，所讲的数据是指这个对象所表达的概念。比如：int 对象代表整数，所有它就是数据，而 Robot 对象代表机器人，所有它不是数据。

学习原子数据，主要是熟知与它相关的运算（或操作），而学习复合数据，最主要的是学习结构关系。数据结构就是专门研究数据的结构关系的学问。

### 6.1 Array 数组

作为以图灵机为原理的，建构在冯·诺依曼体系结构上的 C# 语言，数组就是最基本的数据结构。

数组就是一组连续的类型相同的元素，它是连续随机存储器的抽象。在 C# 中定义数组变量的格式是：

```
type[] variable = new type[size];
```

其中，type 是数组元素的类型；而在元素名字后加一个 [] 就是数组的类型了；size 是一个非负整数，指定创建的数组的元素个数。

显然，一个数组一旦创建完毕，其长度（元素个数）就不能改变。Length 属性可以得到数组的长度。

如果数组在初始化时，其中的每个元素就已经定义好了可以使用下面的格式：

```
type[] variable = new type[]{element0, element1, ...};
```

“{}” 内是数组的元素。

数组元素可以通过索引进行访问：

```
variable[index] = element;
```

```
type element = variable[index];
```

数组的索引通常从“0”开始，直到“数组的长度减 1”；如果索引超出这个范围，系统会抛出异常。

在 C# 中，数组还可定义为多维的，以下是二维数组的定义格式：

```
type[,] variable = new type[size0, size1];
type[,] variable = new type[,] {
    {element00, element01, ...},
    {element10, element11, ...},
    ...};
```

二维数组其实就是数学上的 matrix 矩阵（一维数组是 vector 向量），它有这广泛的用途。Karel 所处的场景就可以用二维数组表示，有了它我们就可以解决拼图游戏的第 2，3 个缺点了。

在 PuzzleRobot 类里定义一个二维数组的字段用于保存每个街角对应图片序号：

```
private int[,] _pictures = { {1, 2, 3, 4},
                             {5, 6, 7, 8},
                             {9, 10, 11, 12},
                             {13, 14, 15, 0}}
```

首先我们需要定义一个函数，它可以根据二维数组将场景中的图片（即 beeper）摆放正确。这里我们将利用场景建设指令“beeper”，它可以在指定位置摆放指定个数的 beepers。场景建设指令属于非基本指令，所有必须使用 Karel.Run 方法来执行，具体格式如下：

```
Karel.Run(“(beeper {0} {1} {2})”, avenue, street, beepers) ;
```

这里需要注意：如过街角位置是 (avenue, street)，则它对应的二维数组的索引是 [3 - street, avenue]，因为在纵向上数组索引和场景坐标刚好相反。

另外，当街角上没有 Beeper 时，根据游戏要求，我们需要将 Karel 的放置在该街角处。这是我们需要使用机器人设置指令“robot”，它也属于特殊指令，所以也必须使用 Karel.Run 方法来执行，具体格式如下：

```
Karel.Run(“(robot {0} {1} e)”, avenue, street, facing);
```

“robot”指令的第 3 个参数表示 Karel 面向方向，这里是 e 等于 east 表示面向东方。根据上面介绍的新指令和我们已经掌握的编成技巧，该函数的代码编写如下：

```
private void RenderPictures()
{
    for (int street = 0; street < 4; street++)
        for (int avenue = 0; avenue < 4; avenue++)
        {
            Karel.Run(“(beeper {0} {1} {2})”, avenue, street, _pictures[3 - street,
            avenue]);
```



```

        if (_pictures[3 - street, avenue] == 0)
            Karel.Run( "(robot {0} {1} e)" , avenue, street);
    }
}

```

这样写看似没有问题，其实存在性能问题：Karel.Run 方法会在每次执行完成指令后重新绘制整个场景，但是在循环中场景处于创建过程，所以并不需要绘制场景。解决这个问题方法是，不要每次都执行指令，而是将所有的指令收集起来，最后再一起执行。这里我们使用 StringBuilder, 它相当于一个记事本，我们可以将指令字符输入其中，最有再一次性度出。具体代码如下：

```

StringBuilder insts = new StringBuilder();
for (int street = 0; street < 4; street++)
    for (int avenue = 0; avenue < 4; avenue++)
    {
        insts.AppendFormat( "(beeper {0} {1} {2})" , avenue, street, _pictures[3
        - street, avenue]);
        if (_pictures[3 - street, avenue] == 0)
            insts.AppendFormat( "(robot {0} {1} e)" , avenue, street);
    }
Karel.Run(insts.ToString());

```

然后，在初始化时图片顺序是正确的。我们需要定义一个函数将图片顺序弄乱。思路是我们使用 Random 对象，随机生成两个二维数组的索引，然后将它们对应的元素交换位置。一次交换显然不算乱，所以我们交换 100 次。

该函数的代码如下：

```

private Random _random = new Random();
private void ShufflePictures()
{
    int count = 100;
    while (count-- > 0)
    {
        int x1 = _random.Next(4);
        int y1 = _random.Next(4);
        int x2 = _random.Next(4);
        int y2 = _random.Next(4);
    }
}

```

```
        int temp = _pictures[x1, y1];
        _pictures[x1, y1] = _pictures[x2, y2];
        _pictures[x2, y2] = temp;
    }
}
```

最后，我们还需要准备一个判断玩家是否完成游戏的函数。基本想法是我们浏览整个场景的所有街角，得到街角上的 Beeper 个数，然后判断这些 Beeper 是否按照顺序摆放。这里需要使用“check-beeper”指令，它可以直接度处街角上的 beeper 数（注意：不是拾取 beeper），该指令也不是常规指令，所以需要 Karel.Run 执行。

该函数的具体代码如下：

```
private bool IsSuccessful()
{
    for (int street = 0; street < 4; street++)
        for (int avenue = 0; avenue < 4; avenue++)
        {
            int beepers = (int)Karel.Run(“(check-beeper {0} {1})” , avenue, street);
            if (beepers != 0 && beepers != street * 4 + avenue)
                return false;
        }
    return true;
}
```

有了这些函数后，我们就可以改写上改任务的 Run 方法了：

```
public override void Run()
{
    ShufflePictures();
    RenderPictures();
    while(true)
    {
        switch (Karel.Listen() as string)
        {
            ...
        }
    }
}
```

```
        if (IsSuccessful())
        {
            Say( "You are successful!" );
            return;
        }
    }
}
```

Run 方法的源代码基本保留，只是在循环之前，添加创建游戏场景的语句，以及在每次执行完用户操作后判断，游戏是否成功。

## 6.2 List 列表

数组一旦创建就不能改变大小，为了克服这个弱点，C# 引入了列表。列表是可以改变大小的数组。

列表变量的定义格式如下：

```
List<type> list = new List<type>();
```

List<type> 是列表类型，这种类型带有一个参数 type，叫做广泛类型，简称泛型。泛型 List 的参数 type 表示列表元素的类型。

列表不能直接初始化，但是我们可以利用数组来间接初始化，格式如下：

```
List<type> list = new List<type>(new type[] {element0, element1, ...});
```

使用 variable.Count 属性可以得到列表当前元素个数。

列表元素的访问和数组一样：

```
list[index] = element;
type element = list[index];
```

列表的方法 Add、Remove 分别添加和删除元素：

```
list.Add(element);
list.Remove(element);
```

列表的方法 Insert、RemoveAt 分别在指定索引位置添加和删除元素：

```
list.Insert(index, element);
list.RemoveAt(index);
```

最后，和数组不同，列表只能是一维的，不能多维。要实现多维的效果，只能是嵌套列表，即列表的元素是子列表。比如：

```
List<List<int>> _pictures = new List<List<int>> {
```

```

        new List<int>(new int[] {1, 2, 3, 4}),
        new List<int>(new int[] {5, 6, 7, 8}),
        new List<int>(new int[] {9, 10, 11, 12}),
        new List<int>(new int[] {13, 14, 15, 0}),
    };

```

嵌套列表访问时用连续的 “[ ]”，比如：\_pictures[0][0]。

## Stack 栈

对于 List 来说，我们对其中元素的访问是非常自由的，可以取存任何一个索引对应的元素，可以在任何一个地方添加（插入）或删除元素。

可是，有的时候，我们并不需要如何丰富的操作。其中，有一种情况，我们只对列表的尾部进行操作；于是我们就可以专门针对这种情况，提供只能进行尾部操作的特殊列表，这就是所谓的栈。栈的定义格式如下：

```
Stack<type> stack = new Stack<type>();
```

type 是栈中每个元素的类型。

栈的操作有：

入栈站 Push，即将一个元素添加到栈顶（也就是列表尾部，相对于 Add）。

窥视 Peep，即得到栈顶元素（相当于 list[list.Count - 1]）。

出栈 Pop，即得到栈顶元素后将它从栈定删除。

空否？IsEmpty，判断栈是否为空。在 C# 中我们使用 Count 属性是否为零来判断。

满否？IsFull，判断栈是否已经装满。在 C# 中栈是无限大的，所以不需要此方法。

Stack 是非常有用的一种数据结构，利用它我们可以制造一种 Rollback 机器人，它可以将走过的路线记录下来，并可以原路返回。该机器人的类定义如下：

```
public class RollbackRobot : RobotPro { }
```

既然要使用 Stack 就要确定栈中的元素是什么类型？为了达到可以原路返回的效果，我们需要执行放回的 Action 动作，所以 Stack 元素是 Action 类型：

```
private Stack<Action> _actions = new Stack<Action>();
```

接下来，我们需要重写 OnMoved、OnTurned 方法，将相应动作的返回动作记录下来，下面是 OnMoved 的重写代码（OnTurned 类似，篇幅有限这里就不罗列了）：

```

protected override void OnMoved(PositionChangeEventArgs e)
{
    _actions.Push(() =>
    {
        MoveBack();
    });
}

```

```
base.OnMoved(e);
}
```

代码里有一段非常特别的代码，按照我们以前掌握的知识，既然 `_actions` 的元素是 `Action` 那么 `Push` 方法的参数应当是 `new Action()` 才对。其实 `Action` 是一个非常特殊的类型，它代表的是一段可以执行的代码，有点像没有名字的函数，而且可以定义在真正的函数里，叫做  $\lambda$  表达式。 $\lambda$  表达式的定义格式是：

```
( 参数列表 ) => { 代码段 }
```

可见上面 `Push` 中的代码，就是定义了一个没有参数的  $\lambda$  表达式。注意这里只是定义了  $\lambda$  表达式，并没有执行  $\lambda$  表达式中的代码。执行的动作在 `Rollback` 函数中。

不过这里有一个问题，那就是在执行回滚时，同样会触发 `Moved` 事件，这时候应当不要向 `_actions` 中压入回滚动作，所以这里需要加一个标志：

```
private bool _isRolling = false;
并在上述代码外加一个条件：
if (!_isRolling)
{
    ...
}
```

接下来就是实现 `Rollback` 函数了：

```
public void Rollback(int steps = int.MaxValue)
{
    _isRolling = true;
    while (steps > 0 && _actions.Count != 0)
    {
        Action action = _actions.Pop();

        action();
        steps--;
    }
    _isRolling = false;
}
```

这段代码很简单，这里就不解释了。

有了 `RollbackRobot` 我们显然可以更加简化捡报纸的任务：

```
RollbackRobot karel = new RollbackRobot();
karel.MoveRight();
```

```
karel.Go(3);
karel.PickBeeper();
karel.Rollback();
```

显然，使用 Rollback 不论道路多复杂，Karel 都可以原路返回！

执行代码有个不完美的地方，放回动作和被打散了，为了解决这个问题，我们可以重写 BegineAct 和 EndAct 方法，并在 BegineAct 中记录 EndAct 的动作，在 EndAct 中记录 BegineAct 的动作。修改后 Rollback 动作就会连贯起了。

## queue 队列

如果将 statck 的操作改为，在尾部（栈顶）添加元素，在首部（栈底）取出元素，这就变成了队列。队列定义格式如下：

```
Queue<type> queue = new Queue<type>();
```

type 是队列中每个元素的类型。

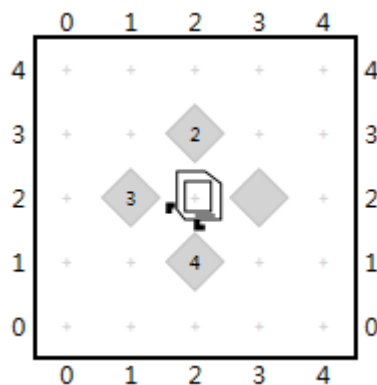
队列的操作如下：

Enqueue 入队：在尾部添加元素。

Dequeue 出队：在首部取出元素。

IsEmpty/IsFull：同 Stack。

任务：旋转木马 (Carrousel)



节日到来，Karel 负责开动游乐场的旋转木马。旋转木马只有 4 个座位，可是当天却来得 9 位游客（用 1 到 9 的 Beepers 表示）。为了公平起见，他们将轮流乘坐木马，旋转一圈后等待下一轮。Karel 将如何调配游客呢？

这是一个典型的队列问题，我们需要两个队列，一个保存正在乘坐木马的 4 位游客，一个保存正在正待的 5 位游客。每一次循环都从等待队列中取出一位游客放入乘坐队列，然后从乘坐队列中取出一位游客放入等待队列，最后根据乘坐队列将 Beepers 摆放正确。

完整代码如下：

```
public class CarrouselRobot : RobotPro
{
    private Queue<int> _riding = new Queue<int>(new int[] { 1, 2, 3, 4 });
    private Queue<int> _waiting = new Queue<int>(new int[] { 5, 6, 7, 8, 9 });
    public override void Run()
    {
        while (true)
        {
            BeginAct();
            _waiting.Enqueue(_riding.Dequeue());
            _riding.Enqueue(_waiting.Dequeue());
            foreach (int tourist in _riding)
            {
                TurnLeft();
                Move();
                PickBeepers();
                PutBeepers(tourist);
                MoveBack();
            }
            EndAct();
        }
    }
}
```

以上代码在摆放 Beepers 时使用了 foreach 语句，该语句从一个 sequence 序列（在这里是 riding）中，一次枚举一个元素，赋值给循环变量（在这里是 tourist），然后执行循环体，直到序列中元素枚举完结束循环。这里需要注意的是：对于队列中元素的访问不能使用 for 循环，因为队列不支持索引访问！

## 6.3 Directory 字典

数组和列表都是以整数作为元素索引的，有时候我们希望使用任意对象为元素索引，这就需要使用字典。列表变量的定义格式如下：

```
Directory<key-type, value-type> variable = new Directory<key-type, value-type>();
```

在字典中作为索引的对象叫做 Key 键，而被索引的元素叫做 Value 值。key-type 就是键的类型，而且 value-type 就是值的类型。

如果键的类型是整数，那么我们就可以向数组和列表一样访问字典。比如：

```
Directory<int, object> variable = new Directory<int, object>();
variable[0] = zero; zero = variable[0];
```

但是一般用的最多的是字符串。比如：

```
Directory<string, object> variable = ...;
variable[ "a" ] = ...;
... = variable[ "a" ];
```

使用字典可以实现简单的指令系统，从而可以使得捡报纸的任务代码简化到一行代码。所谓简单指令是指我们可以使用简单的符号来表示机器人的指令，比如 m 表示 Move、ml 表示 MoveLeft 等。

接下来，我们仍然改写 Rollback 类，给它加上指令系统。

首先定义 key 为字符串，value 为 Action 的字典，存放简单指令：

```
Dictionary<string, Action> _instructions = new Dictionary<string, Action>();
```

在初始化函数中定义所有的简单指令：

```
public RollbackRobot()
{
    _instructions.Add( "pk" , () => PickBeeper());
    _instructions.Add( "pt" , () => PutBeeper());
    _instructions.Add( "tl" , () => TurnLeft());
    ...
    _instructions.Add( "m" , () => Move());
    ...
    _instructions.Add( "g" , () => Go());
    ...
    _instructions.Add( "r" , () => Rollback());
}
```

接下来定义处理简单指令的方法，我们重载 Run 方法：

```
public void Run(string insts)
{
    string[] instss = insts.Split( ' ' );
    foreach (string inst in instss)
    {
        _instructions[inst]();
    }
}
```



```

    }
}

```

这里使用了字符串处理函数 `Split`，它用来将连续指令分割成单字的指令。

有了简单指令系统，捡报纸的任务代码最终简化为：

```
(new RollbackRobot()).Run( "g mr m pk r" );
```

## 6.4 Enumerate 枚举

所有的数据结构，抛开具体的结构形式不说，其本质都是一堆元素的聚合。枚举就是将这堆聚合在一起的元素一个一个列举出来。在 C# 中所有支持枚举的数据结构，都实现了 `IEnumerable` 接口：

```

public interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}

```

方法 `GetEnumerator` 是这个接口的关键，我们可以使用该方法得到一个枚举器 `IEnumerator`：

```

public interface IEnumerator<T>
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}

```

这里面，最关键是 `MoveNext` 方法和 `Current` 属性，每当我们调用一次 `MoveNext`，就可以从 `Current` 得到当前列举的元素，直到 `MoveNext` 返回 `false` 表示所有元素已经列举过了。

在旋转木马的任务中，`riding` 就具有枚举能力，我们可以这样使用 `riding` 的枚举器：

```

IEnumerator<int> enumerator =
    (_riding as IEnumerable<int>).GetEnumerator();
while(enumerator.MoveNext())
{
    int tourist = enumerator.Current;
    ...
}

```

不可否认，这样的枚举使用起来非常不方便，所以 C# 提供了 `foreach` 语句。`foreach` 语句就是对上面代码的包装。所以 `foreach` 语句要求关键字 `in` 后面的数据结构必须实现

IEnumerable<T> 接口。

当枚举器使用完后，如果还想使用，可以调用 Reset 方法重置。

枚举是微软的专有名字（最早出现在 COM 中），而在其他非微软系语言中叫做 iterate 迭代。一般 iterator 迭代器没有 Reset 方法，这里有两个原因：

1. 如果迭代器来自于一个数据结构，迭代器使用完后，如果想重新使用，我们从该数据结构重新得到一个就可以了。
2. 迭代器也可以是随机序列，这样的序列不仅是无限的而且，也无法支持 Reset。

在字典中，键和值具有索引和被索引关系，所以可以组成 KeyValuePair 键值对。字典其实就是键值对组成的集合。所以我们可以枚举键值对。

比如，枚举所有简单动作指令：

```
foreach(KeyValuePair<string, Action> inst in _instructions) { ... }
```

## 6.5 string 字符串

字符串非常特殊而且又十分重要的数据结构，所以我们不得不拿来单独讲解。字符串实际上是由 char 字符组成的一维数组。而字符是和数字、逻辑并列的第三类最基本的原子数据类型。字符变量定义格式如下：

```
char variable = 'A' ;
```

在 C# 中用单引号包括起来的符号叫做字符。字符最终会存储成数字，字符和数字的映射关系定义成字符集。我们常常在计算机中发现乱码，这就是使用的字符集不一致导致的。

C# 支持 Unicode 标准字符集，所以字符可以是汉字，比如：

```
char tao = '道' ;
```

有些字符属于控制字符：比如：换行、回车、制表。这些字符无法直接输入，所以 C# 支持转义，即在“\”符号后的字符表示字符的意义而不是本身，比如：'\n' = 换行、'\r' = 回车、'\t' = 制表。

将多个字符连接起起来然后改用双引号，就表示字符串了，比如：

```
string varibale = "ABC" ;
```

字符串当然也支持转意，比如：

```
string variable = "\n\r\t" ;
```

不过当字符串以“@”为前缀时，将不支持转意，这时双引号中输入什么字符就是什么字符。比如：

```
string variable = @" (lambda (x)
                        (x x))" ;
```

字符串和一般的数组不同，它是 ReadOnly 只读数组。也就是说我们可以通过索引得到字符串中的字符，比如：

```
char variable = "ABC" [0];
```

但是我们不能通过索引修改字符串中的字符，比如：

```
"ABC"[0] = a;
```

字符串的只读特性让它如同原子类型一样，一旦创建就是一个整体。

我们可以使用 “+” 运算，将两个字符拼接成一个字符串：

```
“A” + “B” + “C” → “ABC”
```

Split 方法也可以将一个字符串，分割成一组字符串：

```
“A B C” .Split( ‘ ’ ) → { “A” , “B” , “C” }
```

```
“A,B,C” .Split( ‘,’ ) → { “A” , “B” , “C” }
```

不过我们有时需要，可以反复修改的字符串，C# 为了弥补这个缺陷，提供了 StringBuilder 类。上面的任务中我们已经使用过了，它就像一个记事本，使用非常方便，接口也和 List<char> 类似。

## 6.6 set 集合

集合就是一堆元素的聚集，C# 中，集合的类型是 HashSet<T> 类。

集合对于代数学来说至关重要，根据 ZFC 公理系统中的外延公理，可以推导出：

$\{x, x\}$  等于  $\{x\}$

也就是说，集合中相同元素只当作一个。这是集合的最大用处，使用集合可以消除数据结构中的重复元素。比如：

```
HashSet<int> set = new HashSet<int>(new int[] {1, 1, 1, 2, 3});
```

```
int c = set.Count;
```

这里 c 的值是 3，而不是 5，set 中只有三个元素 {1, 2, 3}。

上一章中，Robot 类，就是利用集合的这种特性，来记录所有已经创建的机器人，以便在重复创建时抛出异常！

还有一点需要注意，集合中的元素没有顺序，也就是说： $\{1, 2\} == \{2, 1\}$ 。

## 6.7 更多的数据结构

这一章我们介绍了几个最重要也是最实用的数据结构。由于本书以“入门”为主，所

以我们将主要精力放在数据结构的使用上，而避免谈及数据结构的实现。由于，C# 并没有提供：Tree 树和 Graph 图的支持，而有时候我们可能需要用到它们，这时就要求我们自己实现，所以继续学习数据结构的实现是非常必要的。关于这部分内容，我们将放在《Karel C# programming 应用》一书中详细讲解。

## 思考题

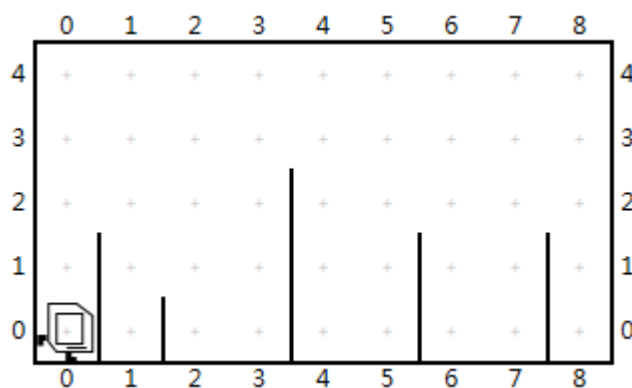
A. 写一个 LeftIsBlocked 方法，判断是否 Karel 的左侧有墙。注意在结束时，Karel 的朝向需要和起始朝向一致。

B. 假定 Robot 在当前街角携带有一个或两个 Beeper。写一个新的方法，名称为 FindNextDirection，来指定 Robot 是朝北转向还是朝南转向，前者朝北有一个 Beeper，后者朝南有两个 Beeper。。

C. 另写一个 FindNextDirection 方法（参见上题描述），这个方法中，Robot 在转向后，无论之前携带多少 Beeper，最终 Robot 将不携带任何 Beeper。

D. 写一个关闭 Robot 的方法，在 Robot 完全被墙包围并无法向四周移动的时候关闭 Robot。如果未被墙包围，仍然保持开启状态且方向位置与之前一致。方法名为 TurnOffIfSurrounded。注意：需要使用 turnOff 方法。

E. 为 Robot 跑障碍赛写段程序，有点像真实的跨栏，跑道为 8 个格子 = 1 mile，但是这里的障碍赛场有三种障碍墙，分别有一、二、三块高。下图示意了一个障碍赛例子。Robot 的起始和结束都是向东，需要贴着障碍和地面跑。新类命名为 Steeplechaser，可继承自 Racer 父类，并重写父类中的行为指令（Racer 只能跨一块高的障碍墙，新类可跨多种）。



Racer 类部分代码：

```
public class Racer : RobotPro
{
    public void RaceStride()
    {
```

```
        if (FrontIsClear())
        {
            Move();
        }
    else
    {
        JumpHurdle();
    }
}

public void JumpHurdle()
{
    JumpUp();
    Move();
    GlideDown();
}

public void JumpUp()
{
    TurnLeft()
    Move();
    TurnRight();
}

public void GlideDown()
{
    TurnRight();
    Move();
    TurnLeft();
}
}
```

F. 重写并检查以下代码，注意语法是否正确。这段代码使用了 IF 语句，使 Robot 面朝东方；并通过模拟来验证是否正确。

```
public void FaceEast()
{
    if (!FacingEast())
    {
```

```
        if (FacingWest())
        {
            TurnLeft();
            TurnLeft();
        }
        else
        {
            if (FacingNorth())
            {
                TurnRight();
            }
            else
            {
                turnLeft();
            }
        }
    }
}
```

G. 当前版本的 `mysteryInstruction` 代码语法正确，但是不易阅读。请简化。

```
public void MysteryInstruction ()
{
    if(FacingWest())
    {
        Move();
        TurnRight();
        if (FacingNorth())
        {
            Move();
        }
        TurnAround();
    }
    else
    {
        Move();
    }
}
```

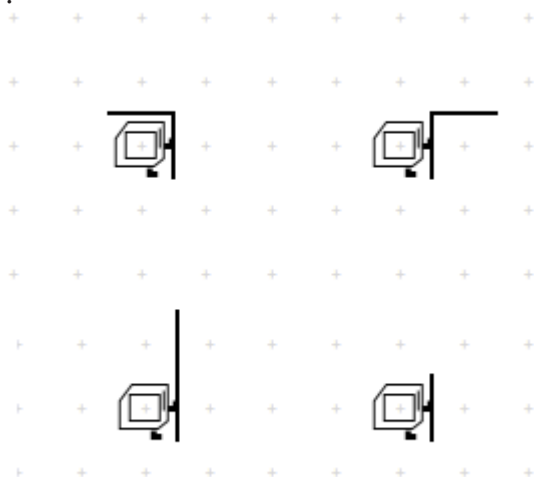
```

    TurnLeft();
    Move();
    TurnAround();
}
}

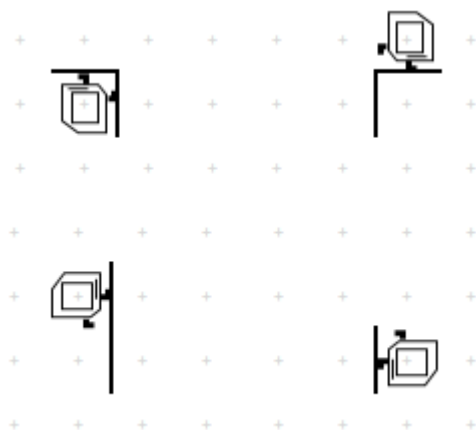
```

H. 为 MazeWalker 类写名为 followWallRight 的指令，假定了无论何时 Robot 在遇到墙的时候，都会向右转。下图展示了四种 Robot 在不同的位置的改变情况。该指令是迷宫脱逃问题的一段关键程序。

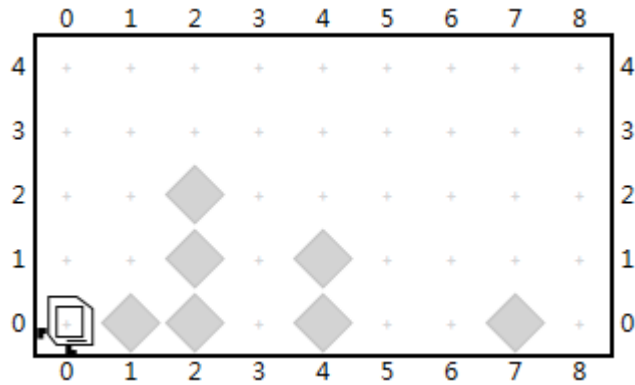
开始时：



结束时：

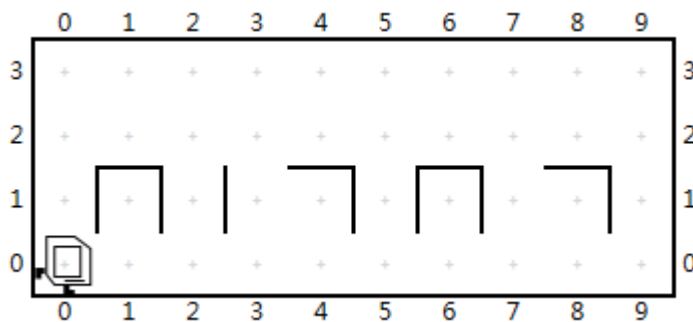


I. 写一段 Robot 程序跑一段障碍赛，跑道为 1 个 mile 也就是八个格子，本次的障碍由 Beeper 而非墙构成。Robot 必须以捡起当前列所有 Beeper 的形式来跳过这些障碍。每个障碍可能由当前列的一、二、三个 Beeper 组成，普通跑道没有 Beeper。下图为一种示例：

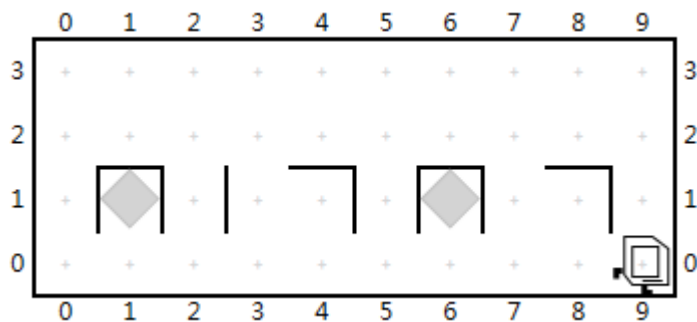


J.Karel 被雇佣来在一些“小房子”内安装地毯，这个任务世界长 1 mile 也就是 8 个格子。一个小房子是由东、西、北三面墙组成的空间，门开在南边。Karel 需要在这个房间内放置一个 Beeper，其他非房子的空间则不放置。下图为一个示例：

开始时：



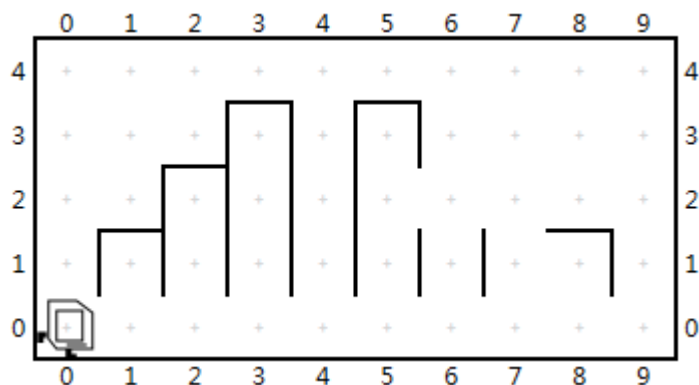
结束时：



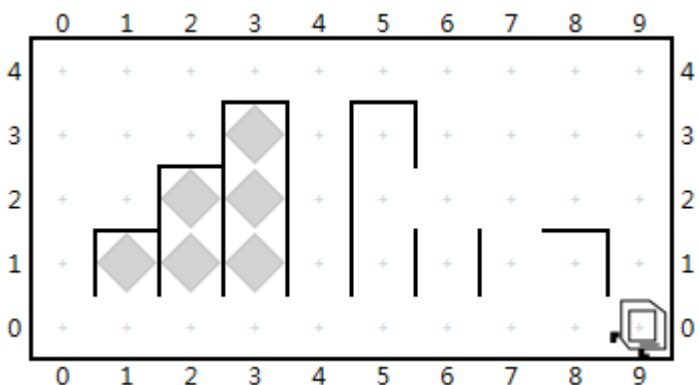
K. 由于 Karel 在上一个任务中（铺地毯）表现的极为出色，于是又有人雇佣 Karel 来完成更复杂的铺毯任务。这次任务世界依然长 8 块，房间则分别有一、二、三块长。房间必须由东西北连续的墙组成。如果任一块墙缺失了，则这个空间不铺毯。同时，Karel 不能重复使用 Beeper，这意味着一旦一个 Beeper 放下去，将不能取出。下图所示为一个例子：

开始时：





结束时:



L. 写一个断言方法, 如果 Robot 临近一个 Beeper (移动到下一个位置有) 并且左侧有墙阻挡, 则返回 true;

另写一个断言方法, 如果 Robot 临近一个 Beeper (移动到下一个位置有) 或者左侧有墙阻挡, 则返回 true。

从这个练习你能得到什么结论。

M. 写一段断言方法, 仅当 Robot 包里有两个 Beeper 时, 返回 true。我们能写另一个断言, 当 Robot 在当前街角, 有两个其他 Robot 时, 返回 true 么? 为什么可或不可? 写下另一段方法, 当 Robot 携带最多两个 Beeper 时, 返回 true。

N. 我们用下列代码来确认 Robot 并没有陷入困局, 例如:

```
if(Karel.Sensor.FrontIsClear())
{
    Karel.Move();
}
```

但是有一种情况下, Robot 程序并不安全。例如如果我们重写 Move 方法, 假定 Move 是去做一些事情而不是移动。那么我们如何再次在类中写上述的安全代码。

O. 重写 ExactlyTwoBeepers, 使其更加简明易读。代码如下:

```
public boolean ExactlyTwoBeepers()
{
```

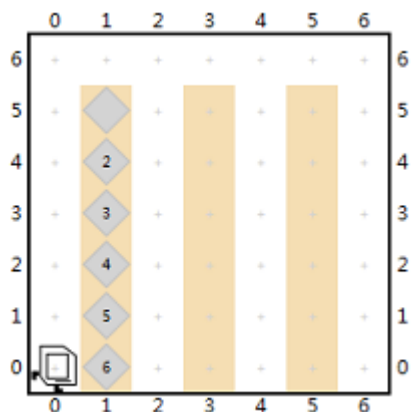
```
if(NextToABeeper()) // one or more beepers
{
    PickBeeper();
    if(NextToABeeper()) //two or more beepers
    {
        PickBeeper();
        if(NextToABeeper()) // more than two
        {
            PutBeeper();
            PutBeeper();
            return false;
        }
        else // exactly two beepers
        {
            PutBeeper();
            PutBeeper();
            return true;
        }
    }
    else //only one beeper
    {
        PutBeeper();
        return false;
    }
}
else //no beepers
{
    return false;
}
```

## 7. 算法

算法在编程当中至关重要，它是解决问题、完成任务的关键。相比于其他的编程技巧只能使得代码更漂亮一些（最多更快一些），算法的正确与否却决定任务的成败。

编写算法也是很有意思的一件事情，这一章，我们将介绍几个经典的算法，它们无疑都散发着理性的光辉！这些算法的产生展示了人类的智慧。

任务：汉诺塔（Hanoi Tower）



Karel 面前有三根柱子（标记为“小麦色”），分别位于 1、3、5 号大道上。在 1 号柱子上从下往上放着，从大到小的盘子（用 beeper 表示，数大小代表盘子大小）。

要求 Karel 将所有盘子从 1 号柱挪动到 5 号柱子，在挪动中有如下要求：

- (1).Karel 每次只能挪动一个盘子。
- (2). 盘子只能放在 3 根柱子上。
- (3). 在任何情况下，都要求保证下方的盘子比上方的大（除非下方是地面），盘子下方也不能为空。

首先，机器人类定义为 HanoiTowerRobot，从 RobotPro 继承，并且在初始化方法中使用 BeeperBag：

```
public class HanoiTowerRobot : RobotPro
{
    public HanoiTowerRobot() { Bag = new BeeperBag(); }
}
```

通过分析，不管具体挪动盘子的循序，Karel 都是一直在重复“将一根柱子顶端的一个盘子挪动到另外一根柱子的顶端上”的动作。

因此我们需要先定义挪动一个盘子的方法：

```
private void MoveAPlate(int from, int to)
{
```

```
GoRight();
GoBack();
Go(from);
while (Sensor.BeepersPresent())
    MoveLeft();
MoveRight();
PickBeepers();
GoRight();
GoBack();
Go(to);
while (Sensor.BeepersPresent())
    MoveLeft();
PutBeepers();
}
```

这个方法把一个盘子从 from 号柱子顶端搬动到 to 号柱子顶端。方法默认 Karel 一直面向东方。不管 Karel 处于任何地方 GoRight+GoBack 都会退回原点 (0, 0) 处。然后 Go(from) 就可以来到 from 号柱子底部。接下来的循环找到最顶端的盘子，然后将其拾起来。接下来同同样的方式来到 to 号柱子顶端，然后放下刚才拾起来的盘子。

有了搬动一个盘子的方法，接下来就需要定义搬动多个盘子的方法。

这里先定义该方法接口如下：

```
private void MovePlates(int plates, int from, int by, int to);
```

意义是将位于 from 号柱子上的 plates 个盘子，经由 by 号柱子，最终搬到 to 号柱子上。

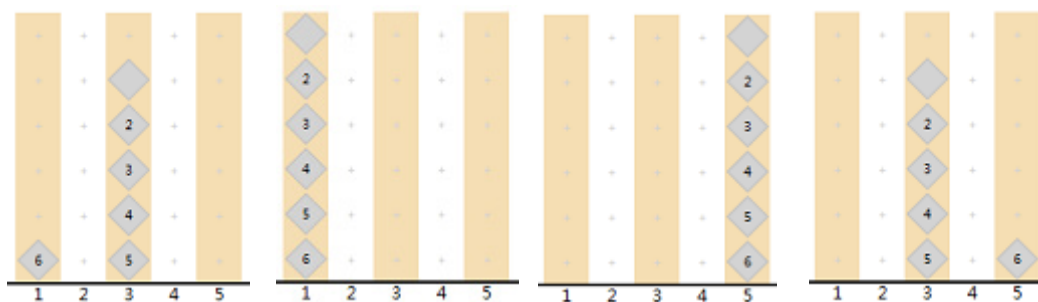
如何实现该方法呢？我们一步一步分析：

首先当 plates 是 1 的时候，就变成了搬动一个盘子的情况，这时候我们只需要调用 MoveAPlate 方法就可以了。代码如下：

```
if (plates == 1)
{
    MoveAPlate(from, to);
}
```

这里直接将盘子从 from 号柱子搬动到 to 号柱子，by 号柱子对我们来说无用。

如果 plates 大于 1，我们可以将搬动分成三个步骤：



第 1 步：将 plates-1 个盘子从 from 号柱子经由 to 号柱子搬动到 by 号柱子。

第 2 步：将 from 号柱子剩下的一个盘子搬动到 to 号柱子。

第 3 步：将 plates-1 个盘子从 by 号柱子经由 from 号柱子搬动到 to 号柱子。

接下来我们需要用代码实现这三个步骤，显然于第 2 步来说，我们可以直接调用 MoveAPlate 方法，所以关键是如何实现第 1, 3 步。

仔细分析第 1, 3 步的需求，抛开具体位置不说，都是：“将多个盘子，从 ... 号柱子，经由 ... 号柱子，搬动到 ... 号柱子”。我们会发现，这不就是 MovePlates 方法的意义吗？，我们是否可以直接调用 MovePlates 方法呢？既然有了想法，不如试一试，代码如下：

```
else
{
    MovePlates(plates - 1, from, to, by);
    MoveAPlate(from, to);
    MovePlates(plates - 1, by, from, to);
}
```

为了可以运行，我们重写 Run 方法如下：

```
public override void Run()
{
    MovePlates(6, 1, 3, 5);
}
```

即，将 6 个盘子从 1 号柱子，经过 3 号柱子，搬到 5 号柱子。

在 Main 方法中创建 HanoiTowerRobot 对象，然后调用它的 Run 方法。

执行脚本，我们会惊奇的发现，Karel 完美的完成了任务！

## 7.1 递归

在汉诺塔任务中，实现 MovePlates 方法时，有两处语句调用了 MovePlates 方法自己，这种在函数体内调用函数自己的程序叫做递归，相应的函数叫做递归函数。

关于递归的一个更为经典的例子是：求  $n$  的阶乘：

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

分析这两个递归函数，我们可以得出以下的递归函数特性：

(1). 从原理上他们都是自我迭代的。

汉诺塔的搬运步骤中表明：对于  $plates$  个盘子的搬动，是建立在对  $plates-1$  个盘子的搬动的基础上的。

阶乘更明显： $n! = n * (n - 1)!$ 。

找到迭代关系，是编写递归算法的必要条件。

(2). 都有迭代终止点。

汉诺塔：判断  $plates$  是否为 1，如果是则不再迭代。

阶乘：根据数学上的定义 0 的阶乘是 1，而负数没有阶乘，所以 0 的阶乘不能通过迭代得到而是直接返回 1。

递归如果没有迭代终止点，就会一直递归下去，直到堆栈耗尽。所以找到迭代终止点，也是递归算法的必要条件。

递归算法更接近人类的思维方式，所以可以写出非常漂亮简约的算法。不过递归算法比较耗费时间，在性能要求苛刻的程序里，我们一般要将递归算法转换为响应的循环算法。比如， $n$  阶乘对应的循环算法如下：

```
public static int Factorial(int n)
{
    int fac = 1;
    while (n > 0)
    {
        f = f * n;
        n
        -
        -
        ;
    }
    return fac;
}
```

显然，循环算法更为复杂一些，但是它更接近机器的思维方式，所以运行效率更高。

考虑将汉诺塔的递归算法也可以改为循环算法，方法如下：

每次递归调用 MovePlates 时，就是发布了一个挪动的要求，如果我们将这些要求保存在 Stack 中就可以避免递归调用。这里使用 Tuple 来记录每一个挪动要求。写成代码如下：

```
Stack<Tuple<int, int, int, int>> acts = new Stack<Tuple<int, int, int, int>> ();
```

首先我们将 MovePlates 的参数压入栈，作为第一个挪动要求：

```
acts.Push(new Tuple(plates, from, by, to));
```

然后进入循环：

```
while(acts.Count != 0)
{
}
```

循环以栈为空为结束。这很合理，因为栈中的挪动要求都执行完了那么整个挪动任务也就执行完了。

在循环中，我们首先从栈定弹出需要执行的请求对他进行处理：

```
Tuple<int, int, int, int> act = acts.Pop();
```

为了清楚，我们将 act 赋值到 plates, from, by, to 四个参数中：

```
plates = act.Item1, from = act.Item2, by = act.Item3, to = act.Item4;
```

接下来的代码就和递归比较相似了：

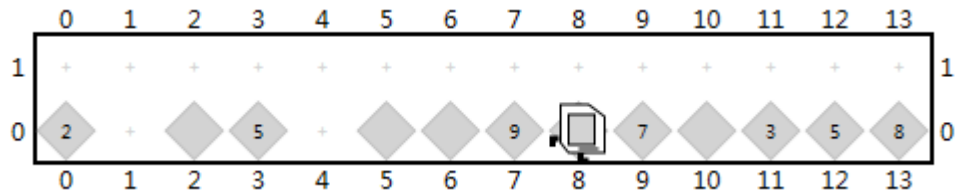
```
if (plates == 1)
{
    MoveAPlate(from, to);
}
else
{
    act.Push(plates - 1, by, from, to);
    act.Push(1, from, by, to);
    act.Push(plates - 1, from, to, by);
}
```

因为，栈是先进后出的，所以要以反顺序将三个步骤对应的挪动要求，压入栈中。

从原理上来说，循环都可以转换为递归，而递归不一定都可以转换为循环。（循环仅和尾递归等价）

## 7.2 排序

排序任务：(Sort)



卡雷尔位于一条长街 ( $n \times 2$ ) 上，面向东方。长街的地面上（南端）放满了不同编号的物品（用 Beeper 的数量表示物品编号）。任务要求卡雷尔，根据编号对这些物品进行排序。排序后，必须保证西边的物品编号不大于东边。

注意：

1. 没有 Beeper 表示编号为 0 的物品。
2. 物品编号可以重复。

3. 卡雷尔初始时位置为  $(x, 0)$ ，其中  $x$  任意，面向东方（正立）；排序完成后，卡雷尔可以处于任何地方，任何方向。

### 利用 List 进行排序

将一组元素按照某种顺序排列，是一个非常基本的需求。所以 `List<T>` 类提供了排列方法 `Sort`，我们可以利用 `Sort` 方法来完成任务：

```
public static void Main()
{
    KarelEx.GoBack();
    List<int> nos = new List<int>();
    do
    {
        nos.Add(KarelEx.PickBeepers());
    }
    while(KarelEx.Move());
    nos.Sort();
    KarelEx.GoBack();
    foreach(int no in nos)
    {
```



```

        KarelEx.PutBeepers(no);
        KarelEx.Move();
    }
}

```

方法中 Karel 从西向东将地面上的编号物品全部收进一个 List 中，然后调用 Sort 方法对 List 排序，最后将 List 中排序后的结果由西向东摆放在大街上。

这段代码中使用了 do while 语句它是 while 语句的变种，它是先执行再判断再循环。do while 的另外一个有用的技巧是可以模拟 switch 的 break 语句：

```

do
{
    ...
    if(...)
    {
        ...
        break;
    }
    .
} while(false);

```

### Bubble Sort 冒泡排序：

如果不借助 List 帮助，那么冒泡排序就是最简单的排序算法。冒泡排序算法的运作如下：

1. 比较相邻的两个物品的编号。如果西边的比东边的那个大的，就交换它们两个。
2. 对每一对相邻物品做同样的工作，从西边第一对到东边的最后一对。在这一点，最后的物品编号应该会是最大的数。
3. 然后针对所有的物品重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的物品重复上面的步骤，直到没有任何一对物品需要排序。

根据算法我么首先要写出排列两个物品的方法：

```
public static bool SortTwo() {}
```

先前进，然后后退，途中捡起前后两个物品：

```

    if (!KarelEx.Move())
        return false;

    int front = KarelEx.PickBeepers();
    KarelEx.MoveBack();

```

```
int back = KarelEx.PickBeepers();
```

对两个物品进行排序：

```
if (back > front)
{
    int temp = front;
    front = back;
    back = temp;
}
```

将排序后的两个物品摆放回街道上：

```
KarelEx.PutBeepers(back);
KarelEx.Move();
KarelEx.PutBeepers(front);
return true;
```

接着我们需要一个标记表示街道上那些物品已经排好了顺序：

```
public static void PutBeeperAbove() { ... }
public static bool PickBeeperAbove() { ... }
```

这两方法分别在 Karel 头顶上放置或拾取一个 Beeper，实现方法很简单，这里省略。

有了这些准备就可写排序算法了：

```
public static void Main()
{
    KarelEx.GoBack();
    for (;;)
    {
        while(!PickBeeperAbove() && SortTwo());
        if (!KarelEx.MoveBack())
            break;
        PutBeeperAbove();
        KarelEx.GoBack();
    }
}
```

代码中，for(;;) { ... } 等价于 while(true) { ... }。

while 语句根据是否有标志不断的调用 SortTwo。

If 条件判断是否已经是大街的最西端了，这时候要跳出循环。

接着打标志，表示从该点起向东的物品都排列正确了，然后回到大街最西端，重新开始新一轮两两比较交换。

## 7.3 查找

除了排序外，另外一个算法大类就是查找。所谓查找，就是在各种集合中搜寻所和目标对象有关系的元素的位置。

在一个普通的列表或一维数组中搜寻元素的算法是最简单的查找。这里假设有一个列表 `list`，给定一个目标元素 `element`，现在需要在 `list` 中查找 `element` 的索引。

算法就是从 `list` 的第一个元素开始到最后一个元素结束一个一个和目标元素比较，直到找到相等的元素位置：

```
int index = -1;
for(int i = 0; i < list.Count; i++)
    if (list[i] == element)
    {
        index = i;
        break;
    }
```

`List` 类其实已经在 `IndexOf` 方法中提供了上面的算法，我们可以直接使用：

```
int index = list.IndexOf(element);
```

### 折半查找

以上的算法是针对无序列表的，如果是排序列表，那么我们有更快速的查找算法。具体算法如下：

将目标对象和列表中间的元素比较：

如果对象等于元素那么显然就找到了；

如果对象比元素小那么我们只需要在列表前半部分继续查找；

如果对象比元素大那么也只需要在列表前半部分继续查找。

这种查找因为不断的将列表分为一半，所以叫做折半查找。

下面是折半查找的一个例子：

```
public static int BinarySearch(List<int> list,
                                int startIndex, int endIndex, int element)
{
    if (startIndex < endIndex)
        return -1;
```

```
int middleIndex = (startIndex + endIndex) ? 2;
if (element == list[middleIndex])
    return middleIndex;
if (element < list[middleIndex])
    return BinarySearch(list, startIndex, middleIndex - 1, element);
else
    return BinarySearch(list, middleIndex + 1, endIndex, element);
}
```

折半查找是非常简单而又高效的算法，我们在将来的实际编程中会经常使用。

### 字符串搜索

以上的查找都是搜索单一的对象，而字符串查找的目标是一串字符。比如：

在 "abcdefg" 中查找 "def" 的开始位置。

可以先定义子串比较的方法：

```
public static bool Compare(string str, int start, string substr)
{
    if (str.Length - start < substr.Length)
        return false;
    for (int j = 0; j < substr.Length; j++)
        if (substr[j] != str[j + start])
            return false;
    return true;
}
```

然后从字符串开始一个一个位置进行子串比较：

```
int index = -1;
for (int i = 0; i < str.Length - substr.Length; i++)
    if (Compare(str, i, substr))
    {
        index = i;
        break;
    }
```

这个算法虽然简单可行，但是效率并不高！

当 str = "ababc", substr = "abc" 时，下面是比较过程：

```

          a b a b c
i = 0    a b c
i = 1    a b c
i = 2    a b c

```

当  $i = 0$  时 Compare 比较失败时  $j = 2$ , 这时其实没有必要比较  $i = 2$  的情况。因为这时已经确定  $i = 1$  的字符是 b 而子串第一个字符是 a 所以肯定失败! 于是可以直接跳到  $i = 2$  开始比较。

基于子串自身的特点, 我们可以根据比较失败时  $j$  的值, 确定  $i$  向后移动  $k$  字符。就拿子串 "abc" 为例子, 通过分析我们可以得出以下结果:

$j = 0 \quad k = 1;$

$j = 1 \quad k = 1;$

$j = 2 \quad k = 2;$

$j = 3 \quad k = 0;$  0 表示匹配成功

接下来我们需要写一个函数, 可以分析出来任意给定子串的  $k$  值表, 代码如下:

```

public static int[] Analyse(string substr)
{
    int[] ks = new int[substr.Length];
    ks[0] = 1;
    for (int i = 1; i < substr.Length; i++)
    {
        int k = 1;
        while (k < i)
        {
            bool matched = true;
            for (int j = 0; j < i - k; j++)
                if (str[j] != str[k + j])
                {
                    matched = false;
                    break;
                }
            if (matched)
                break;
            k++;
        }
    }
}

```

```
        }  
        ks[i] = k;  
    }  
    return ks;  
}
```

有了这个 k 值表我们就可以改进字符串匹配算法了：

```
public static int Compare(string str, int start, string substr)  
{  
    if (str.Length - start < substr.Length)  
        return 0;  
    for (int j = 0; j < substr.Length; j++)  
        if (substr[j] != str[j + start])  
            return j;  
    return -1;  
}  
  
int[] ks = Analyse(substr);  
int index = -1;  
for (int i = 0; i < str.Length - substr.Length; i++)  
{  
    int j = Compare(str, i, substr);  
    if (j == -1)  
    {  
        index = i;  
        break;  
    }  
    else  
    {  
        i += ks[j] - 1;  
    }  
}
```

其实，string 类的 IndexOf 方法已经提供了以上的子串查找算法，我们可以直接使用：

```
int index = "ababc".IndexOf("abc");
```

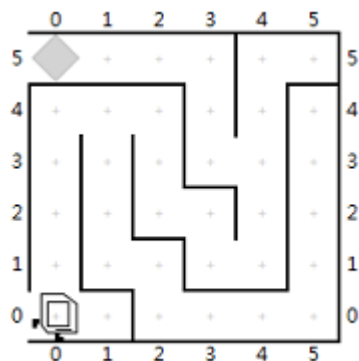
## 7.4 各种算法

以上的所有算法，都是结构性的算法，而对于数值本身来说也是有相当经典的算法的。比如：求一个函数在某个区间内的最大值、解多元一次方程组、一元高次方程等。这些又是算法的一大类。

而结构性和数值性算法相结合，会产生更加有用的算法，比如：正则表达式、有限自动机、数据挖掘等。

关于算法的学习和研究是永无止境，本书只是为读者开启算法世界的一个大门，希望读者可以进入并且在算法的世界中遨游！最后我们以一个有趣的任务来结束本章。

任务：走迷宫（Maze）



Karel 处在迷宫入口处，Beeper 表示迷宫出口。Karel 如何走出迷宫呢？

走迷宫的秘诀是：一直向着正确（right）的方向走，前方遇到无法克服的困难，我们要及时回头。

根据秘诀可以写出任务代码：

```
public static void Main()
{
    RobotPro karel = new RobotPro();
    while (!karel.Sensor.BeepersPresent())
    {
        if (karel.Sensor.RightIsClear())
            karel.TurnRight();
        if (karel.Sensor.FrontIsClear())
            karel.Move();
        else
            karel.TurnBack();
    }
}
```

```
}
```

算法和人脑常常是相反的，在人类看来非常困难的问题，对于算法来说却非常简单。走迷宫就是一个最好的佐证。

## 思考题

A. 写一个名为 EmptyBeeperBag 的新方法。在 Robot 执行该方法之后，它的 Beeper 包清空为零。

B. 写一个名为 GoToOrigin 的新方法，使得 Robot 回到 (0, 0) 街区并且面朝东。不考虑初始位置和初始方向。假定空间中没有墙。注意：使用西、南边界墙作为引导。

C. 分别学习下列的代码片段。请指出每段程序代表什么？是否有更简单的实现方法？如果有请写出，并解释。

代码段 1:

```
while (!NextToABeeper())
{
    Move();
}
if (NextToABeeper())
{
    PickBeeper();
}
else
{
    Move();
}
```

代码段 2:

```
while (!NextToABeeper())
{
    if (NextToABeeper())
    {
        PickBeeper();
    }
    else
    {
        Move();
    }
}
```



```

    }
}

```

描述一下两段程序的区别：

第一段：

```

while (Sensor.FrontIsClear())
{
    Move();
}

```

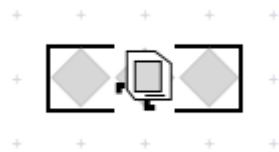
第二段

```

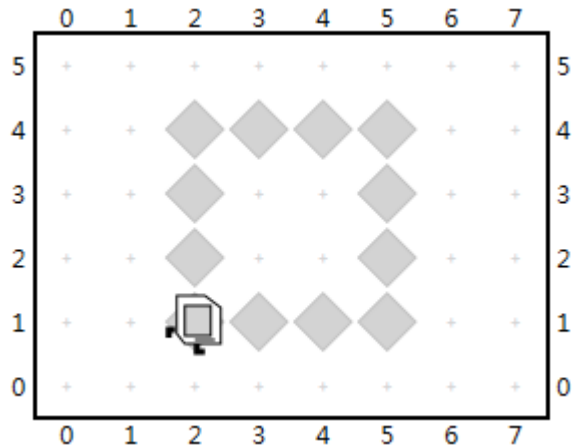
if (Sensor.FrontIsClear())
{
    Move();
}

```

D. 在 Karel 世界中有一种恐惧——一堆无限多的 Beeper。是的，听上去不可能但是却在这个世界发生了。如果 Karel 试图去捡一堆无限多的 Beeper，Karel 将注定失败，无法捡起这一堆 Beeper。Karel 现在就处于这种极度危险的状态中，它站在东、西两个房间外：只有一个房间是 Karel 可以捡起来的这一堆 Beeper，另一个房间是无法捡起的无限 Beeper 堆。Karel 必须决定哪个房间是安全房间，走进去并且取出安全房间中的所有 Beeper。为了帮助 Karel 选出安全房间，有第三堆 Beeper 放置在当前 Karel 所在的街角。如果第三堆有偶数 Beeper，安全房间就在东边。如果奇数，则在西边。第三堆最少有一个 Beeper。请写出捡起安全房间中 Beeper 堆的程序。



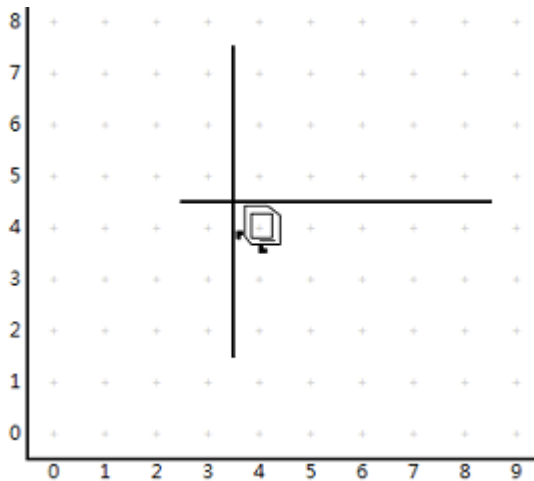
E. 请按下图排列出 Beeper 方阵，开始时世界中并无 Beeper，而 Karel 起始于 Beeper 方阵的左下角位置。



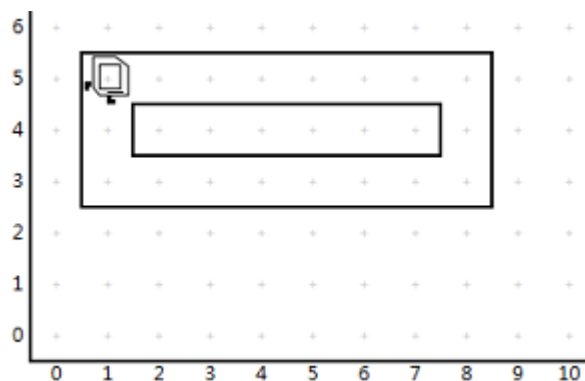
F. 写一段指令，让 Robot 逃离任一长方形房间，打开的房间门总是长一个格子。当 Robot 逃出房间时，程序必须让 Robot 关闭。

G. 写一段程序，如果能找到出口，Robot 逃离长方形房间。如果没有门的话，Robot 关闭。我们不要使用上一题写好的程序，因为在没有门的房子中，运行那段程序会造成绕圈停不下来的结果。提示：有一种略微复杂的方法解决这个问题，而不需要重排列 Beeper。你可以用上面这个方法，也可以假定 Robot 中有一个 Beeper 在包里，你可以用它来记住是否已经绕行这个房间一圈。这个程序还需要分开 turnOff 指令，包括找到门和没有门。

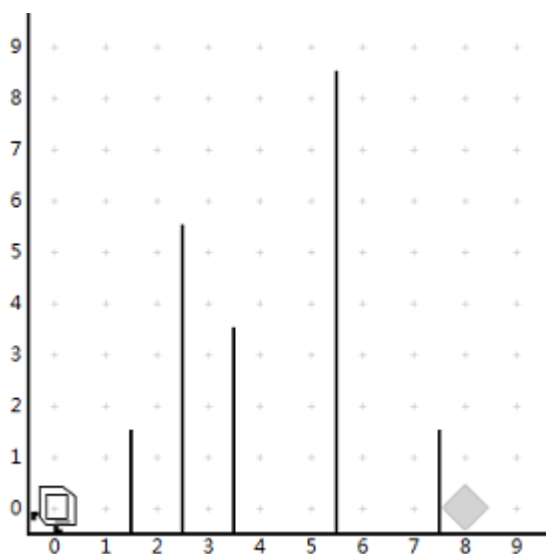
H. Karel 再次做了花匠的工作。Karel 必须在如下图所示的墙边用 Beeper 环绕，一个 Beeper 只种在一个紧邻墙的街角。你可以假定 Karel 总是开始于同样的相对位置上，且有足够的 Beeper 来完成任务。可以写一个简单的变量来记录 Karel 设置的 Beeper 数目。



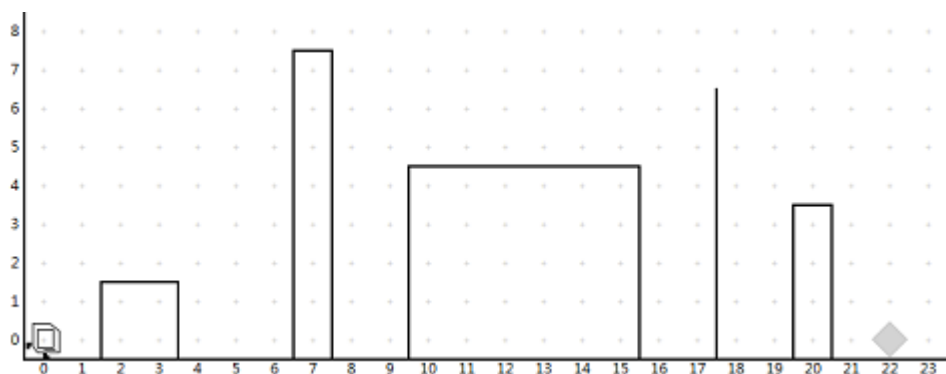
I. Karel 种下的 Beeper 作物失败了，于是它返回了给走廊铺地毯的工作。走廊总是如下图所示的形状，并且宽度为 1 个格子。可以确定的是地毯并没有凸起，每个街角只能铺一个毯。Karel 有足够的 Beeper 完成这个工作，并且总是开始于相同的相对位置。你能写出一段程序同时解决 H 和 I 两道题么？



J. 写一段程序 Robot 可以跑超级障碍赛。赛道上跨栏非常高并且没有固定的终点。每个跑道的终点都由一个 Beeper 标记，捡起后关闭自己视为结束。下图展示了一种可能。其他赛道可能更长更高。



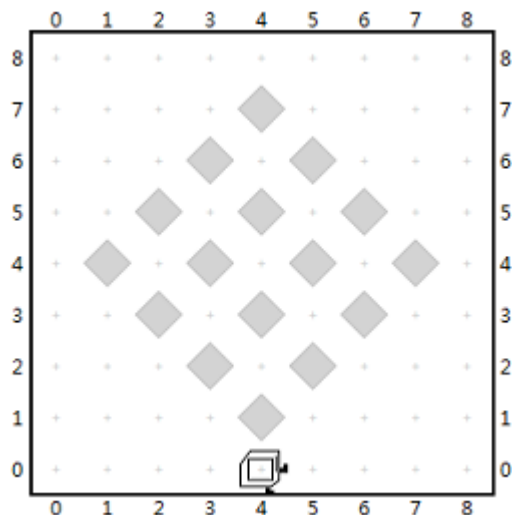
K. 写一段 Robot 能跑更强的超级障碍赛程序。在赛道上，跨栏将很高且很宽。下图显示了一种可能，最好的方法是按上题的思路，从已有的 Robot 类中建立一个继承类。



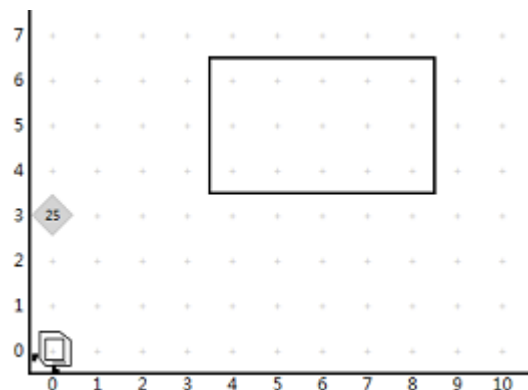
L. 写一段指令收割者可以收割任意大小的矩形麦田。麦田的边缘由无 Beeper 的街角组成。假定麦田内每个街角都填充了 Beeper。Robot 开始于麦田左下方并面朝东。我们能否派生自上述的 Harvester 类？

M.Karel 返回了方形 Beeper 麦田继续收割。写一个收割 Beeper 的新程序。麦田的总是

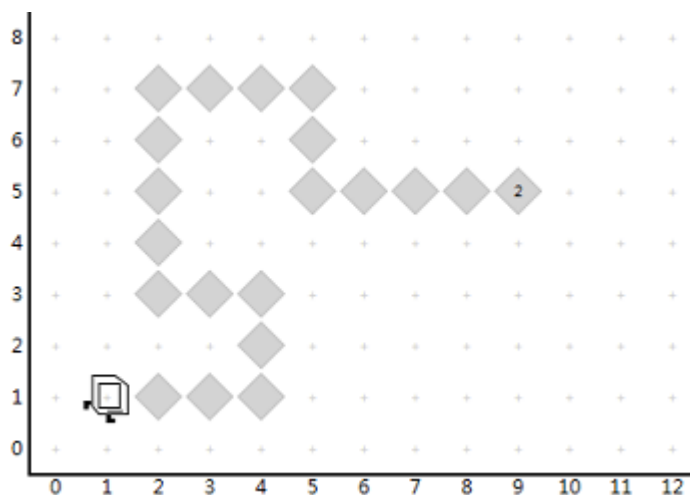
有相同的边，如下图所示：



N. 名为 Karel 的 Robot 正在修栅栏。栅栏由 Beeper 组成，并且围绕着长方形的墙来建造。墙的大小是未知的。Karel 开始时面朝未知方向。栅栏的原料 Beeper 放置在墙的西边，这些材料足以修建栅栏。Beeper 堆与栅栏的南边在同一条直线上，如图所示。Beeper 堆和墙的距离也是不得而知。

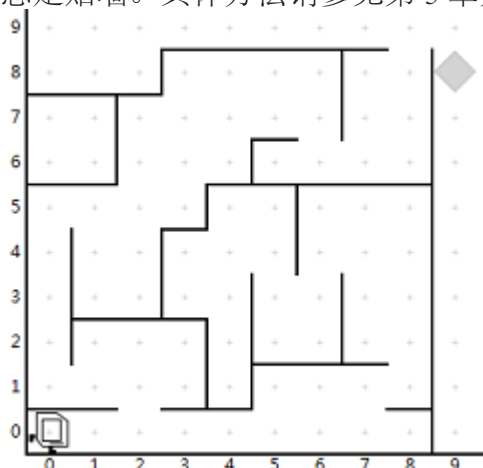


O. Karel 喜欢在这个世界的森林中漫步，即便它有自己内置的罗盘，Karel 有时候依然不能找到回家的路。为了缓解这个问题，在散步之前 Karel 给包里装了 Beeper，且留下了 Beeper 足迹。写一个程序 Karel 可以沿着路径返回家。关于这个我们可以有很多疑问。在本题中我们忽略墙的阻挡，假定终点是由两个 Beeper 组成，路径不会交叉，一个例子参见下图。提示：Karel 必须能够找到路径上的下一个街角，并且能捡起路径上的 Beeper；否则可能会引起前进和后退的无限循环。写另外一个类似的路径程序，如果我们允许一个 Beeper 在路径中缺失，但一行内不会有两个 Beeper 缺失，有多么困难？



P. 假定 Robot 在一个完全闭合的长方形房间中，房间里有一个 Beeper。写一个程序 Robot 找到这个 Beeper，捡起，关闭自己。

Q. 给 Karel 写段程序，逃离迷宫（不包括复杂回路）。迷宫出口被一个 Beeper 标记，位置在迷宫外靠近右墙的第一个街角。这个任务能够通过 Karel 的移动指令完成，移动后 Karel 右边总是贴墙。具体方法请参见第 5 章第 9 题中各种移动类型。下图是一个例子：



有一个简单的方法并不需要用第 5 章第 9 题的这些指令。试试写一个简单版本的迷宫脱逃程序。提示：Program karel to make the least amount of progress toward its goal at each corner in the maze。最后，对比迷宫逃离和本章第 11 题超级障碍赛，你看出他们之间的相似点了么，房间逃离问题呢？什么 class 是用来解决问题的，什么是父类？

R. 这个问题受到对 WHILE 循环进行讨论时的启发。Robot 所在的街角有 0, 1, 2, 3, 7 个 Beeper 时，执行下列代码：

```
void WillThisClearCornerOfBeeoers()
{
    for (int i = 0; i < 10; i++)
    {
        if (NextToABeeper())
```

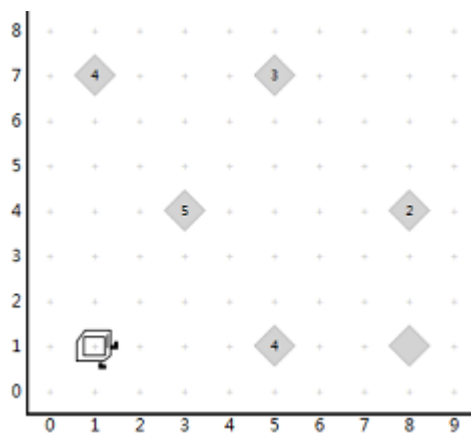
```

    {
        PickBeeper();
    }
}

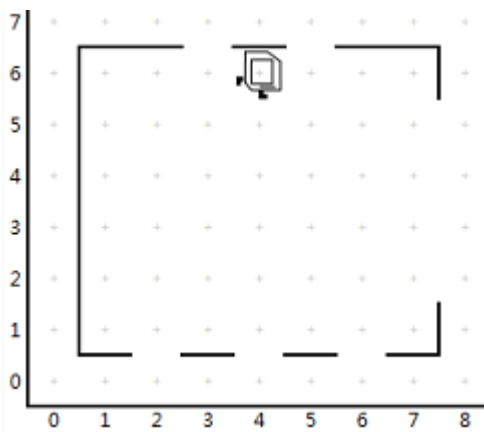
```

在上述初始条件下，这段代码可以正确工作。在其他状态下呢？

S. 写一段程序让 Karel 可以寻宝。宝藏标记有 5 个 Beeper 的街角处。其余街角（包括 Karel 开始的位置）包含有线索，每一种线索代表 Karel 将要往下走的方向。线索包括：1 个 Beeper 向北，2 个向西，3 个向南，4 个向东。Karel 应该跟随线索，直到最终找到宝藏，关闭 Karel。下图是一种可能：



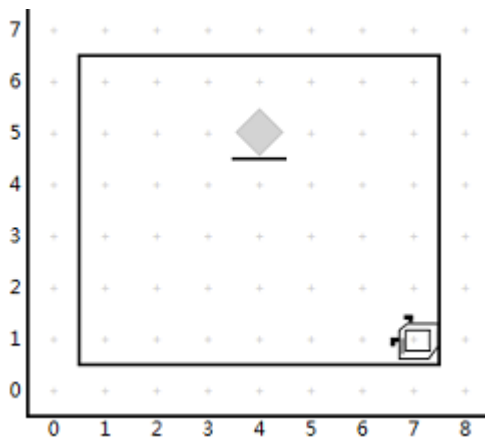
T. Karel 在一间开了很多窗户的房间里。房间内没有窗户。Karel 贴着北墙面朝东。写一段程序通过在窗户前放置 Beeper 来关闭窗口，Karel 身上带有足够多的 Beeper。



U. Karel 在坐标 (0, 0) 面朝东。在 Karel 面前的街道，有一行 Beeper（一个 Beeper 占一个街角，一行至少有一个 Beeper）。这行 Beeper 的长度未知，但是 Beeper 间没有间断。Karel 必须捡起这些 Beeper，并且移动这些 Beeper 到北边的行中，行号与 Beeper 数量相关。举个例子，如果在第 0 行有 5 个 Beeper，需要将这些 Beeper 移动到第 5 行。如果第一个 Beeper 出现在第 3 列，当程序结束时，Karel 也需要停在第 3 列。

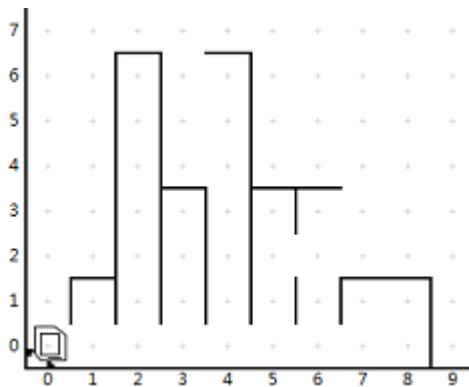
V. Karel 在完全闭合没有门窗的房间里，起始位置在东南街角面朝南。房间里有一堵

长为 1 的墙，封锁了墙南北街角的通行。墙的一侧有一个 Beeper（哪一侧并不知道）。为 Karel 写一段程序，将 Beeper 从墙的一侧捡起，放置在另一侧。

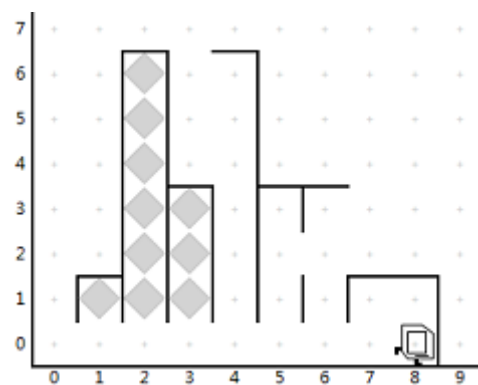


W.Karel 再次接到了铺地毯的工作。在铺毯之前，Karel 必须确保房间的东西北三个方向有连续的墙。所有的房间宽度为 1，总是在北墙终止。Karel 的任务是当在第 0 行遇到封路墙，则结束。下图示意了一种可能：

开始时：



结束时：



X. 执行以下代码会产生什么效果？

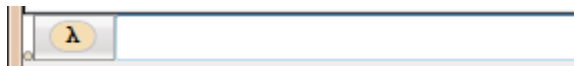
```
while (FacingNorth())
{
    TurnLeft();
}
```

## 8. 打造 Karel 世界

除了 Karel 动作指令外，Lab.Karel 系统还提供了一套场景建设指令。这就使得我们可以用程序的方式建造 Karel 世界。用 C# 执行场景建设指令的唯一办法是调用 Karel.Run 方法。该方法执行以参数方式传入的场景建设指令代码，然后将执行结果返回。

Karel.Run 指令使用的脚本语言是 Scheme 语言（一种 Lisp 方言），她是世界上最优雅的语言，拥有诗歌一般简约的语法形式。

除了用 C# 执行 Scheme 脚本外，这些指令也可以直接输入到处于界面下方的命令行中执行。



当然将系统的语言设定为 Scheme 后，执行的也是同样的脚本。

### 8.1 建设场景

场景建设指令分为场景设置和场景信息检测，它们基本上是一一对应的，信息检测一般以 “check-” 为前缀。

(scene avenues streets)

创建新的场景。streets 和 avenues 分别为场景中大街和大道的条数（街道数必须大于等于 0）。

比如，下面代码创建  $8 \times 8$  的场景：

```
Karel.Run(@" (scene 8 8) ");
```

当新场景创建的时候 Karel 也被跟着一起创建出来，不管场景的大小如何，Karel 总是位于 (0,0) 面向东方。

这里需要注意的是：街道数可以为 0，虽然 0 没有意义。

另外新场景创建时，会自动在边境上围一圈墙壁。

(check-scene)

将得到一个有两个元素的整数数组，数组中的两个元素分别是大街和大道的条数。比如：

```
int[] size = Karel.Run(@" (check-scene) ") as int[];
```

```
int avenues = size[0];
```

```
int streets = size[1];
```

(robot avenue street facing)/(robot id avenue street facing)

创建或设置机器人。id 表示机器人的标志，没有 id 表示 Karel。(avenue street) 为机器人位置，facing 是机器人面向，可以是：e 或 east、n 或 north、w 或 west、s 或 south。

这里需要注意的是：机器人的位置可以超出场景，处于 Karel 世界的任何地方。所以，



将机器人位置设置到距离场景足够远的地方，可以达到让机器人隐身的效果。

比如，执行下面代码，Karel 隐身，

```
var show = Karel.Run(@"
    (let ([k (check-robot)])
      (robot -100 0 e)
      (lambda () (robot (vector-ref k 0) (vector-ref k 1) (vector-ref k 2))))" );
```

然后执行下面代码，Karel 显身，

```
Karel.Run( "{0}" , show);
```

(destroy-robot id)

销毁指定id的机器人。一般情况下不会销毁 Karel,但是确实需要时,可以执行如下指令:

```
Karel.Run(@" (destroy-robot ())" );
```

(check-robot) / (check-robot id)

得到机器人的位置和面向。比如:

```
object[] info = Karel.Run(@" (check-robot)" ) as object[];
int avenue = (int)info[0];
int stree = (int)info[1];
string facing = (string)info[3];
```

(wall avenue street <orientations>)

设置 (avenue,street) 处街角周围四面墙壁。<orientations> 和 facing 一样但是那个可以有多个，表示那面有墙；另外 none 和 all 分别表示四面没有墙和四面都墙。

比如，下面的代码分别设置街角 (1, 1) 周围的墙壁：

```
Karel.Run(@" (wall 1 1 e n)" ); // 东边和北边有墙壁，西边和南边没有墙壁。
Karel.Run(@" (wall 1 1 none)" ); // 周围没有墙壁。
Karel.Run(@" (wall 1 1 all)" ); // 周围四面皆有墙壁。
```

(build-wall avenue street <orientations>)

语法和 wall 相似，建设指定的墙壁。

(destroy-wall avenue street <orientations>)

语法和 wall 相似，销毁指定的墙壁。

(walls (avenue street <orientations>) ...)

参数 “()” 内的语法和 wall 相似，负责设置场景中所有的墙壁。当 walls 没有参数时表示场景中没有墙壁。

(check-wall avenue street)

返回一个数组表示 (avenue,street) 周围那个方向有墙壁。

(beeper avenue street count)

Karel 在负数个 beepers 的街角上 PickBeeper 将失败，但是 PutBeeper 将成功这时 beepers 数加 1。数个 beepers 可以模拟深井，在井没有填满时，Karel 放在街角的 beeper 总会掉入井中，消失不见。

检测 (avenue,street) 处街角上有多少个 beepers。

设置 (avenue,street) 处街角地面上的标记。str 是字符串，可以是如果是系统可以识别的颜色则系统将该地面涂上这种颜色，否则系统显示该字符串。为了不影响到其他街角，系统会对超出街角的字符串进行裁减。

检测 (avenue,street) 处街角地面上的标记。

### 任务：推箱子 (Push Man)

推箱子是一个经典游戏，要求将 Karel 改造成推箱子游戏。

定义PushMan类,它从RobotPro继承,并在静态初始化函数中,定义创建管卡的新指令:

[illegible]

```

(lambda (m)
  (case m
    [(+) #f]
    [(x) (when (or (= x 0) (not (eq? (car left) 'x)))
      (build-wall x y w))
      (unless (eq? (car top) 'x)
        (build-wall x y n))
      (when (= y 0)
        (build-wall x y s))
      (mark x y "wheat" " "))]
    [(o) (beeper x y 1)
      (mark x y "green" " "))]
    [($) (beeper x y 2)]
    [(@) (beeper x y 3)
      (mark x y "green" " "))]
    [(!) (robot x y east)]
    [else (mark x y m)]]
  (unless (eq? m 'x)
    (when (and (not (= x 0)) (eq? (car left) 'x))
      (build-wall x y w))
    (when (eq? (car top) 'x)
      (build-wall x y n)))
  (set! x (+ x 1))
  (when (= x width)
    (when (eq? m 'x)
      (build-wall (- x 1) y e))
    (set! y (- y 1))
    (set! x 0))
  (set! left (cdr left))
  (set! top (cdr top)))
  maps)))

```

```

(define-syntax build (syntax-rules ()
  [(_ width height maps) (house width height 'maps)]))

```

```

        ");
    }
}

```

这段代码使用 Scheme 语言定义了新指令 “build” 使用它可以很方便的定义关卡。

每一个关卡我们都定义成一个函数，比如，第 s 关定义如下：

```

public void SectionS()
{
    Karel.Run(@" (build 8 8 (
        s x x x x s s
        s x + ! + x x x
        x x + x $ + + x
        x + @ o + o + x
        x + + $ $ + x x
        x x x + x o x s
        s s x + + + x s
        s s x x x x x s
    )
    )
    "
    );
}

```

其中：! 代表 PushMan 的初始位置、x 代表墙壁、o 代表正确摆放位置、\$ 代表没有到位的箱子、@ 表示到位的箱子、+ 仓库内空位、其他仓库外空位。

接着需要定义 push、push-left、push-right、push-back 动作指令，我们仍然在静态初始化函数中用 Scheme 定义：

```

Karel.Run(@"
    (define (turn-back)
      (turn-left)
      (turn-left))

    (define (turn-right)
      (turn-back)
      (turn-left))

    (define (pick-beepers)
      (let loop ([count 0])

```

```
(define (push)
  (if (move)
      (let ([beepers (pick-beepers)])
        (case beepers
          [(0 1) (put-beepers beepers) #t]
          [else
           (put-beepers (- beepers 2))
           (set! beepers 2)
           (if (move)
               (let ([beepers2 (pick-beepers)])
                 (case beepers2
                   [(0 1)
                    (put-beepers (+ beepers beepers2))
                    (turn-back)
                    (move)
                    (turn-back)
                    #t]
                   [else
                    (put-beepers beepers2)
                    (turn-back)
                    (move)
                    (put-beepers beepers)
                    (move)
                    (turn-back)

```

```

        #f]))
    (begin
      (put-beepers beepers)
      (turn-back)
      (move)
      (turn-back)
      #f)))))
  #f))

```

```

(define (push-back)
  (turn-back)
  (let ([r (push)])
    (turn-back)
    r))

```

```

(define (push-left)
  (turn-left)
  (let ([r (push)])
    (turn-right)
    r))

```

```

(define (push-right)
  (turn-right)
  (let ([r (push)])
    (turn-left)
    r))

```

“);

为了方便起见我们用, r、u、l、d (即: r = push、u = push-left、l = push-back d = push-right) 分别简化这些动作, 并添加延时:

```
Karel.Run("@
```

```

  (define (delay) (sleep 100))

```

```

(define-syntax act (syntax-rules ()
  [(_ args ...) (begin (args) ...)]))

```

```
(define (u) (push-left) (delay))  
(define (r) (push) (delay))  
(define (l) (push-back) (delay))  
(define (d) (push-right) (delay))  
“);
```

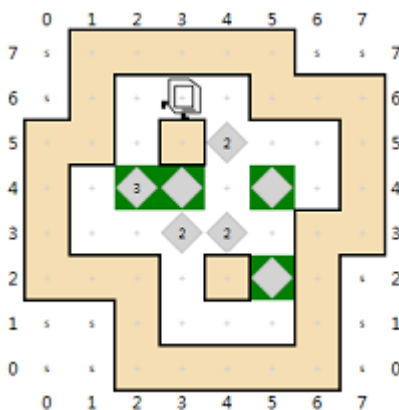
接下来重写 Run 方法，关键键盘上的操作键：

```
public override void Run()  
{  
    Karel.Run(@”  
        (let loop ([key (listen)])  
          (case key  
            [( “” left” ” ) (act l)]  
            [( “” right” ” ) (act r)]  
            [( “” up” ” ) (act u)]  
            [( “” down” ” ) (act d)])  
          (sleep)  
          (loop (listen)))  
        “);  
}
```

最后，就可以在 Main 方法中输入如下代码：

```
PushMan pushMan = new PushMan();  
pushMan.SectionS();  
pushMan.Run();
```

运行脚本，场景中程序如下界面：



游戏利用不同数量的 beeper 表示不同的事物：

1 个 beeper 表示箱子正确摆放位置；

2 个 beeper 表示箱子；

显然处于正确位置的箱子，箱子 + 正确 = 3 个 beeper。

现在可以使用方向键试一试，这可使最难的一关呀！

推箱子游戏中，我们大量使用了 Scheme 语言，因为本书不是介绍 Scheme 的书，所以这里不解释 Scheme 的语法，有兴趣的读者可以参考《Karel Scheme Programming 入门》，它是本书的姊妹篇，详细讲述如何使用 Scheme 语言执行 Karel 任务。

另外，读者也可以在：[www.r6rs.org](http://www.r6rs.org) 阅读该语言的最新标准：算法语言 Scheme 修订<sup>6</sup>报告。

最后本书附录里附有《Karel Scheme Programming 简言》，它对于有兴趣学习 Scheme 语言的读者来说，也是一个不错的帮助。

## 8.2 场景渲染

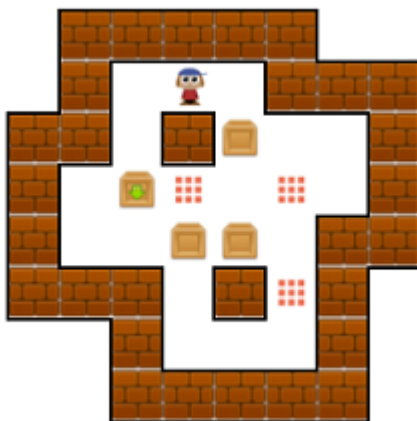
不可否认，这个界面实在太抽象了，如果能让场景变得像推箱子游戏一样，就要定义自己的场景渲染器。

Tech.Robots 提供了，推箱子的场景渲染器，我们只需要在，Main 方法中添加如下代码就可以了：

```
SceneManager.Using(new PushManSceneRender());
```

运行脚本，界面就变为：



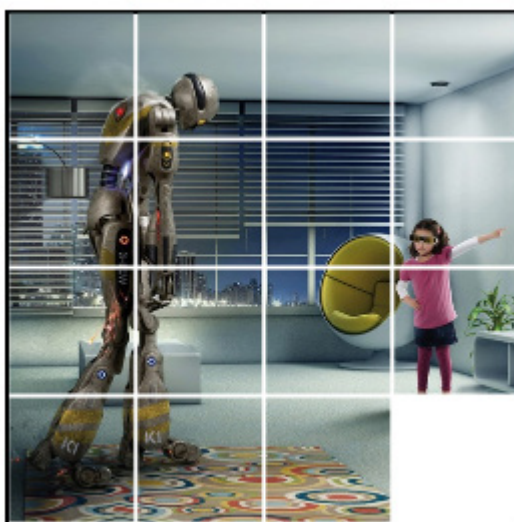


是不是漂亮多了？

另外，也可以通过改变场景渲染器来弥补 Puzzle 游戏的最后一个缺陷。Tech.Robots 提供了，Puzzle 场景渲染器的一个范例，我们可以直接使用

```
SceneRenderManager.Using(new PuzzleSceneRender());
```

运行脚本，界面就变为：

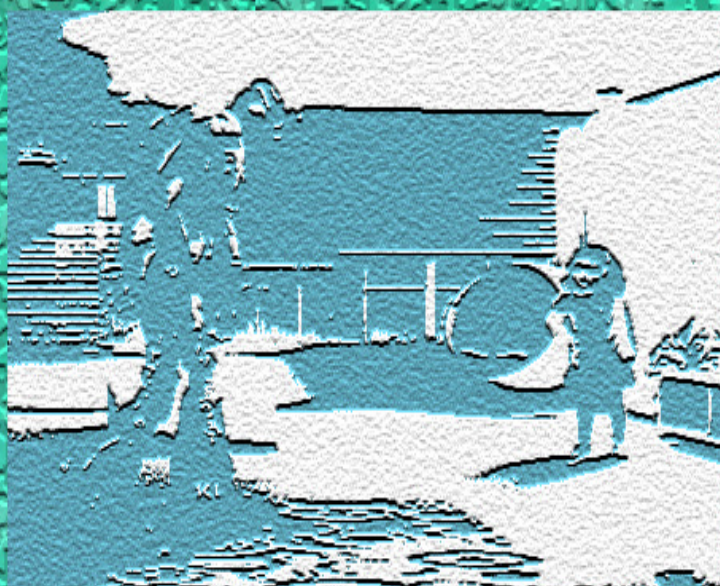


显然 Robot 在 Puzzle 中其实没有显示的意义，所以渲染器不绘制它。另外渲染器也可以指定街角的大小，这里的大下是  $64 \times 64$  个像素，而并非默认的  $32 \times 32$  个像素。

关于如何实现场景渲染器，牵扯到了 WPF 的内容，这部分我们留在《Karel C# Programming 应用》一书中详细讲解。







内部书刊