Chris Piech
CS 106A

Handout #6
January 10, 2018

Assignment #1: Karel the Robot

Karel problems due: 11:00am on Friday, January 19th

This assignment should be done individually (not in pairs)

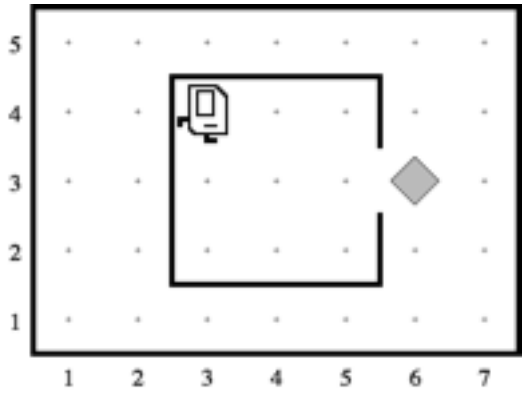
Based on a handout by Eric Roberts, with additional text by Keith Schwarz and Mehran Sahami.

This assignment consists of four Karel programs. There is a starter project including all of these problems on the CS106A web site under the "Assignments" tab. When you want to work on these programs, you need to download that starter project as described in Handout #5 (Using Karel in Eclipse). From there, you need to edit the program files so that the assignment actually does what it's supposed to do, which will involve a cycle of coding, testing, and debugging until everything works. The final step is to submit your assignment using the **Submit Project** entry under the **Stanford Menu**. Remember that you can submit your programs as you finish them and that you can submit more than one version of your project. If you discover an error after you've submitted one of these problems, just fix your program and submit a new copy. **Also, please remember that your Karel programs must limit themselves to the language features described in *Karel the Robot Learns Java* in the Karel and SuperKarel classes. You may not use other features of Java, even though the Eclipse- based version of Karel accepts them.**

The four Karel problems to solve are described on the following pages.

Problem 1

Your first task is to solve a simple story-problem in Karel's world. Suppose that Karel has settled into its house, which is the square area in the center of the following diagram:



Karel starts off in the northwest corner of its house as shown in the diagram. The problem you need to get Karel to solve is to collect the newspaper—represented (as all objects in Karel's world are) by a beeper—from outside the doorway and then to return to its initial position.

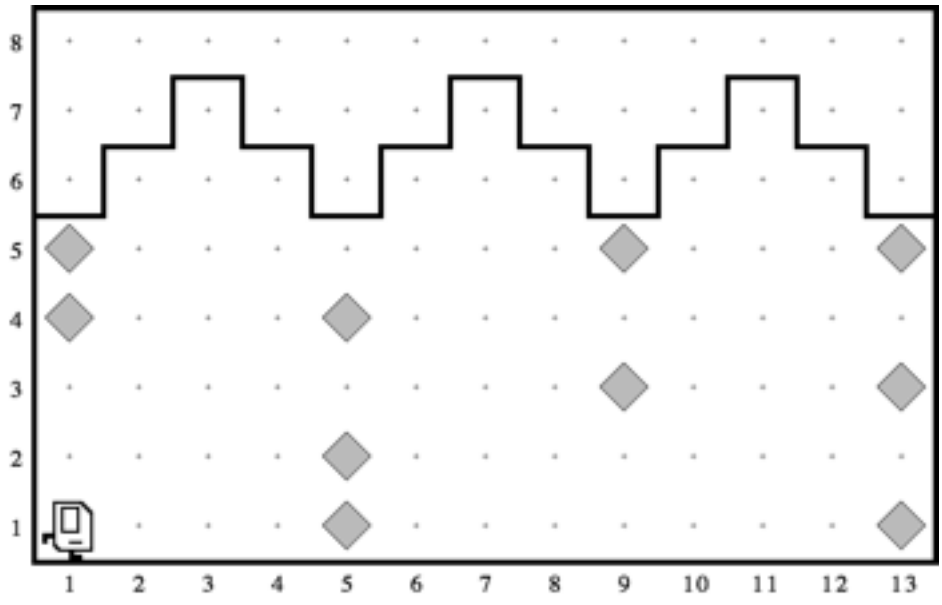
This exercise is extremely simple and exists just to get you started. You can assume that every part of the world looks just as it does in the diagram. The house is exactly this size, the door is always in the position shown, and the beeper is just outside the door. Thus, all you have to do is write the sequence of commands necessary to have Karel

1. Move to the newspaper,
2. Pick it up, and
3. Return to its starting point.

Even though the program is only a few lines, it is still worth getting at least a little practice in decomposition. In your solution, include a private method for each of the steps shown in the outline.

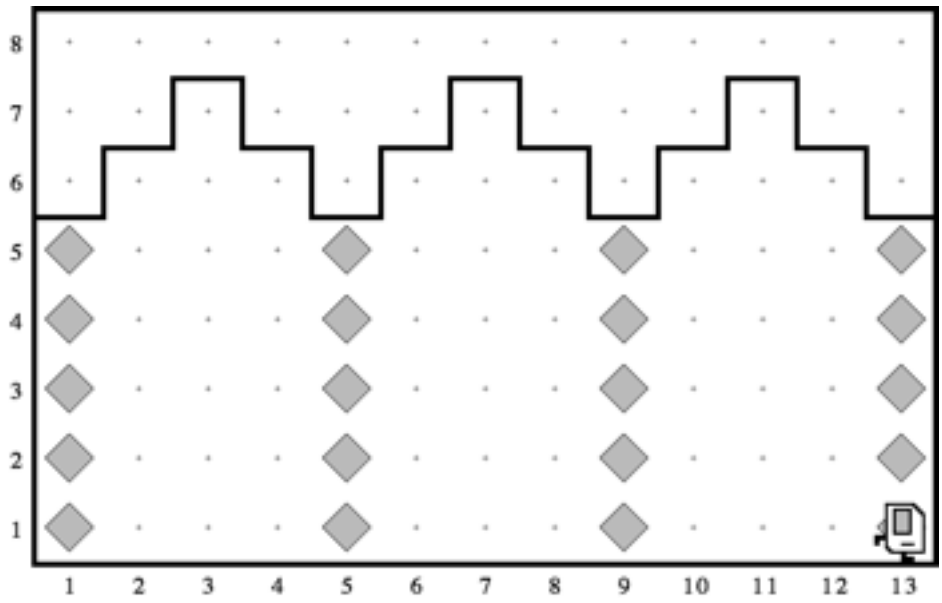
Problem 2

Karel has been hired to repair the damage done to the Quad in the 1989 earthquake. In particular, Karel is to repair a set of arches where some of the stones (represented by beepers, of course) are missing from the columns supporting the arches, as follows:



Your program should work on the world shown above, but it should be general enough to handle any world that meets certain basic conditions as outlined at the end of this problem. There are several example worlds in the starter folder, and your program should work correctly with all of them.

When Karel is done, the missing stones in the columns should be replaced by beepers, so that the final picture resulting from the world shown above would look like this:

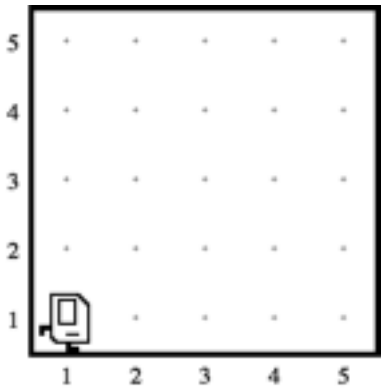


Karel's final location and the final direction he is facing at end of the run do not matter. Karel may count on the following facts about the world, listed below:

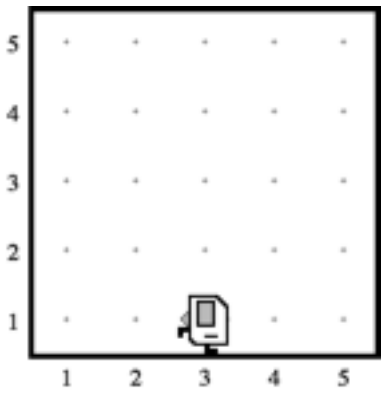
- Karel starts at 1st Avenue and 1st Street, facing east, with an infinite number of beepers in Karel's beeper bag.
- The columns are exactly four units apart, on 1st, 5th, 9th Avenue, and so forth.
- The end of the columns is marked by a wall immediately after the final column. This wall section appears after 13th Avenue in the example, but your program should work for any number of columns.
- The top of the column is marked by a wall, but Karel cannot assume that columns are always five units high, or even that all columns are the same height.
- Some of the corners in the column may already contain beepers representing stones that are still in place. Your program should not put a second beeper on these corners.

Problem 3

As an exercise in solving algorithmic problems, program Karel to place a single beeper at the center of 1st Street. For example, if Karel starts in the world



it should end with Karel standing on a beeper in the following position:



Note that the final configuration of the world should have only a single beeper at the midpoint of 1st Street. Along the way, Karel is allowed to place additional beepers wherever it wants to, but must pick them all up again before it finishes.

In solving this problem, you may count on the following facts about the world:

- Karel starts at 1st Avenue and 1st Street, facing east, with an infinite number of beepers in its bag.
- The initial state of the world includes no interior walls or beepers.
- The world need not be square, but you may assume that it is at least as tall as it is wide.

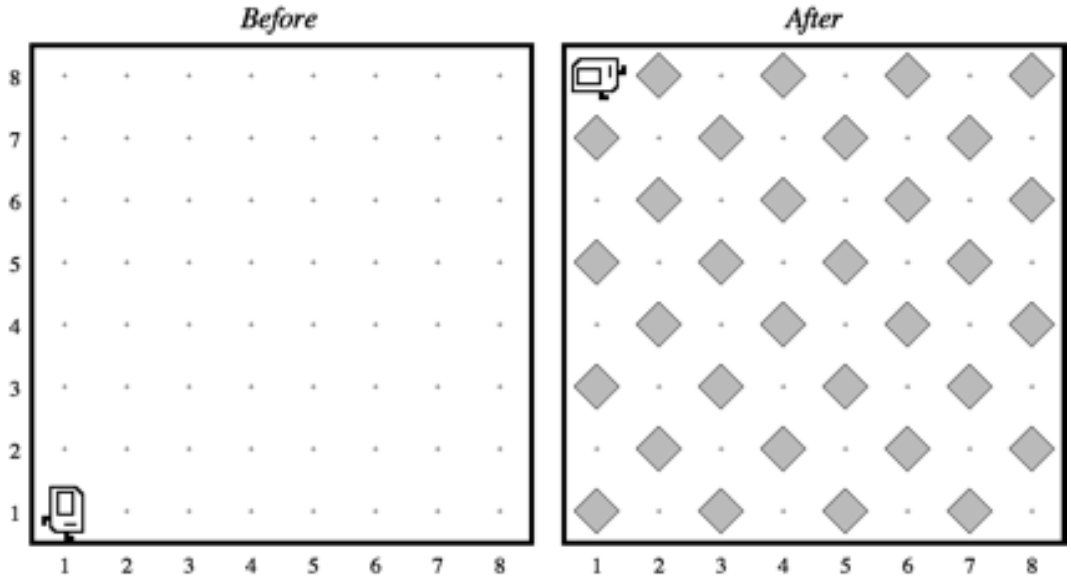
Your program, moreover, can assume the following simplifications:

- If the width of the world is odd, Karel must put the beeper in the center square. If the width is even, Karel may drop the beeper on either of the two center squares.
- It does not matter which direction Karel is facing at the end of the run.

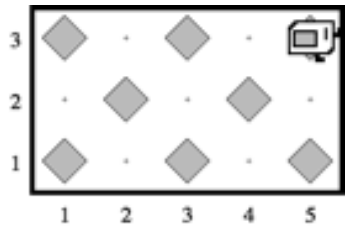
There are many different algorithms you can use to solve this problem. The interesting part of this assignment is to come up with a strategy that works.

Problem 4

In this exercise, your job is to get Karel to create a checkerboard pattern of beepers inside an empty rectangular world, as illustrated in the following before-and-after diagram. (Karel's final location and the final direction it is facing at end of the run do not matter.)



This problem has a nice decomposition structure along with some interesting algorithmic issues. As you think about how you will solve the problem, you should make sure that your solution works with checkerboards that are different in size from the standard 8x8 checkerboard shown in the example above. Some examples of such cases are discussed below. Odd-sized checkerboards are tricky, and you should make sure that your program generates the following pattern in a 5x3 world:



Another special case you need to consider is that of a world which is only one column wide or one row high. The starter folder contains several sample worlds that test these special cases, and you should make sure that your program works for each of them.

This problem is hard: Try simplifying the problem. Can you checker a single row or column? Make it work for different row widths or column heights? Okay, now can you make it do two rows? Three? All rows? Etc. Incrementally developing your program in stages by making it solve a smaller simpler problem is a wise strategy for attacking hard programming problems.

Optional Extension

If you finish early you may **optionally** choose to write a Karel Extension. Modify ExtensionKarel.java to solve any problem of your choosing. Extensions are a great chance for practice, and, if your extension is substantial enough, for a + score. Make sure

to write comments to explain what your program is doing and change ExtensionKarel.w to be an appropriate world for your program.

Advice, Tips, and Tricks

All of the Karel problems you will solve, except for `CollectNewspaperKarel`, should be able to work in a variety of different worlds that match the problem specifications. When you first run your Karel programs, you will be presented with a sample world in which you can get started writing and testing your solution. However, we will test your solutions to each of the Karel programs, except for `CollectNewspaperKarel`, in a variety of test worlds. Unfortunately, each quarter, many students submit Karel programs that work brilliantly in the default worlds but which fail catastrophically in some of the other test worlds. Before you submit your Karel programs, ***be sure to test them out in as many different worlds as you can.*** We've provided several test worlds in which you can experiment, but you should also develop your own worlds for testing.

As mentioned in class, it is just as important to write clean and readable code as it is to write correct and functional code. A portion of your grade on this assignment (and the assignments that follow) will be based on how well-styled your code is. Before you submit your assignment, take a minute to review your code to check for stylistic issues like those mentioned on the next page.

Have you added comments to your methods? To make your program easier to read, you can add comments before and inside your methods to make your intention clearer. Good comments give the reader a clue about what a method does and, in some cases, how it works. Did you add comments to your methods to indicate what they do?

Not-so-Good Code	Good Code

`public void fillRowWithBeepers() {`
`  while (frontIsClear()) {`
`  putBeeper();`
`  move();`
`  }`
`  putBeeper();`
`}`
	`/* Makes Karel move to the end of`
	` * the row, dropping a beeper`
	` * before each step he takes.`
	` *`
	` * Precondition: None`
	` * Postcondition: Karel is facing`
	` * the same direction as before,`
	` * and every step between Karel's`
	` * old position and new position`
	` * has had a beeper added to it.`
	` */`
	`public void fillRowWithBeepers() {`
	`  while (frontIsClear()) {`
	`  putBeeper();`
	`  move();`
	`  }`
	`  putBeeper();`
	`}`

Is your code indented properly? In Java, each line of code can be indented by any amount. Does your indentation help show how the different pieces of code are related to one another? For example:

Not-so-Good Code	Good Code

`public void run() {`
`  move();`
`while (frontIsClear()) {`
`  move();`
`  turnRight();`
`if (beepersPresent()) {`
`  pickBeeper();`
`}`
`}`
`}`
	`public void run() {`
	`  move();`
	`  while (frontIsClear()) {`
	`  move();`
	`  turnRight();`
	`  if (beepersPresent()) {`
	`  pickBeeper();`
	`  }`
	`  }`
	`}`

Did you decompose the problem? There are many ways to break these Karel problems down into smaller, more manageable pieces. Decomposing the problem elegantly into smaller sub-problems will result in a small number of easy-to-read methods, each of which performs just one small task. Decomposing the problem in other ways may result in methods that are trickier to understand and test. Look over your code and check to see whether you've decomposed the problem into smaller pieces. Does your code consist of a few enormous methods (not so good), or many smaller methods (good)?

This is not an exhaustive list of stylistic conventions, but it should help you get started. As always, if you have any questions on what constitutes good style, feel free to stop by to see the course helpers in Old Union with questions, come visit us during office hours, or email your section leader with questions!