Colin Kincaid
CS 106A

Assignment #1: Karel the Robot

Due: 11AM PST on Thursday, July 5th

This assignment should be done individually (not in pairs)

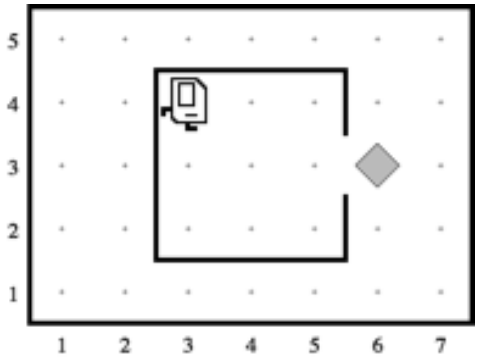
Based on handouts by Eric Roberts, Keith Schwarz, Mehran Sahami and Marty Stepp.

This assignment consists of four Karel programs. There is a starter project including all of these problems on the CS 106A website under the "Assignments" dropdown. First, you need to download and import the starter project as described in the Eclipse instructions in the sidebar on the course website. 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 instructions on the Eclipse page. Remember that you can submit your programs as many times as you like; your section leader will only grade your most recent submission.

NOTE: you must limit yourself to using only the commands and syntax discussed in lecture or in the Karel coursereader in the Karel and SuperKarel classes. Do not use other features of Java, even though Eclipse accepts them. For example, do not use variables of any kind in your program.

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, *facing east*.

This exercise is 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.

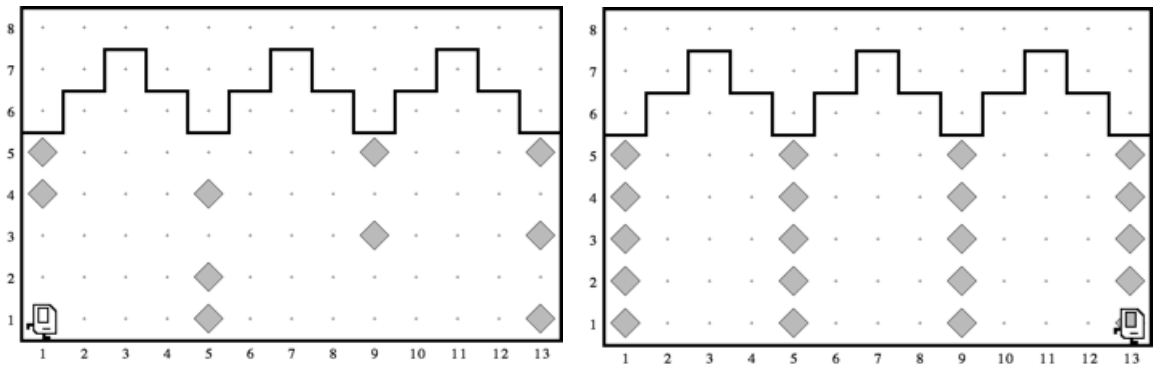
Even though the program is short, it is still worth getting practice with decomposition. In your solution, include a **method** for the first and third steps above.

Note: *once you have finished this problem, you should submit your project! See the Eclipse instructions on the course website for how to do so.* ***DO NOT*** *wait until the last minute to submit, in case you run into any issues submitting for the first time.*

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:

Before: *After:*



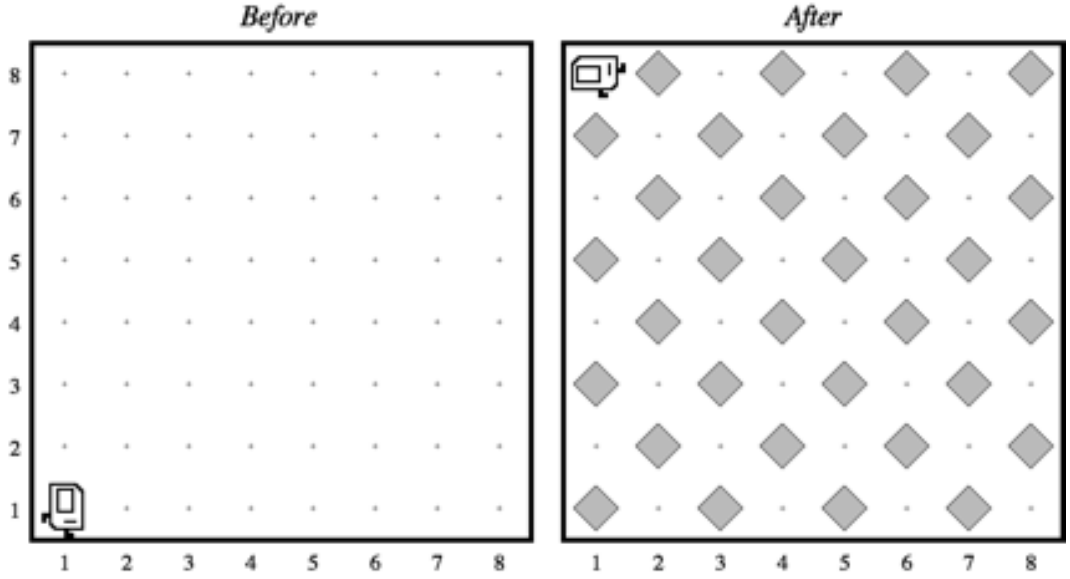
As shown above, when Karel is done, the missing stones in the columns should be replaced by beepers. Karel's final location and the final direction it is facing at end of the run do not matter. Your program should work on the world shown above, but it should be general enough to handle any world that meets the following conditions:

- 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 above 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.

For examples, there are several sample worlds in the starter project, and your program should work correctly with all of them.

Problem 3

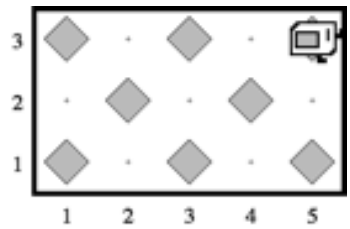
In this exercise, your job is to get Karel to create a checkerboard pattern of beepers inside an empty rectangular world, as illustrated in the following before-and-after diagram. (Karel's final location and the final direction it is facing at end of the run do not matter.)



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.

Your program should put the beepers in *exactly* the squares shown and start checkering by putting a beeper down on (1, 1). For example, in the picture above, Karel has placed beepers at (1, 1), (3, 1), … (2, 2), (4, 2), etc. If you were to put beepers on the opposite squares, such as (2, 1), (4, 1), …, (1, 2), (3, 2), etc., this would also be a checkerboard pattern, but it would not exactly match the expectations for this problem. In a 1x1 world, Karel should place a beeper on (1, 1).

As you think about how to 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. As an example, odd-sized checkerboards are tricky, and you should make sure that your program generates the following pattern in a 5x3 world:



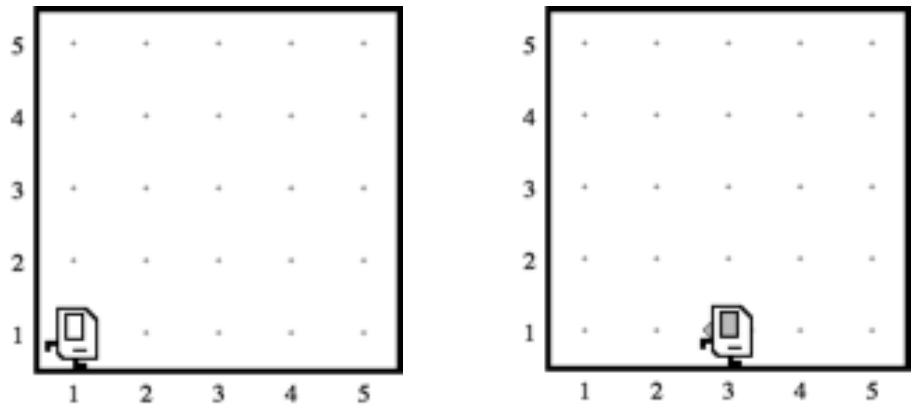
Another special case you need to consider is that of a world which is only one column wide or one row high. The starter folder contains several sample worlds that test these special cases, and you should make sure that your program works for each of them.

The checkerboard problem can be tricky, so we suggest developing it in stages using the technique of "stepwise refinement" as described in the Karel coursereader. For example, write an initial version that just checkers a single row, or a single column, or some other chunk of the overall work. Test it until you're satisfied that it works in a variety of different world sizes. Then modify your code to implement the rest of the required behavior.

Problem 4

Program Karel to place a single beeper at the center of 1st Street and **finish on top** of that beeper, as illustrated in the following before-and-after diagram.

Before: *After:*



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 things 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 it is at least as tall as it is wide.
- 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, but the program must finish with Karel standing on the midpoint on top of the beeper it has placed there.

The reason this problem is trickier than it seems is that Karel cannot count, so you can't simply walk to the edge of the world and count your steps as a way of finding the midpoint. The interesting part of this problem is coming up with a strategy that works in all valid worlds. There are *many* different algorithms you can use to solve this problem (we have seen at least 10 different approaches), so be creative and have fun coming up with one. Remember that you must solve this problem using only the syntax shown in the Karel coursereader; you may **not** use Java variables.

Development Strategy

When working on a programming assignment, it helps to have a step-by-step process, or **development strategy.** We suggest working on each problem one at a time, starting from Problem 1 and going forward in the natural order. Working on a simpler problem can help you be ready to solve the next harder problem.

A common mistake students make is to try to write an entire program without running or testing it. Instead, we suggest an **incremental approach**: try solving a small part of the problem, running it to verify that what you wrote works properly, and then continuing. As much as possible, also try to use the **top-down-design** techniques we developed in class. Break the task down into smaller pieces until each sub-task is something that you know how to do using the basic Karel commands and control statements. These Karel problems are tricky, but appropriate use of top-down design can greatly simplify them.

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. Each quarter, many students submit Karel programs that work well in the default worlds but which fail in some of the other test worlds. Before you submit, ***be sure to test your programs 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. Handout #3 contains instructions on how to create your own Karel worlds.

Grading

Functionality: We will grade your program's behavior to give you a **Functionality grade**. You can run your program in various worlds to help check your behavior for various test cases. Your code should compile without any errors or warnings in Eclipse, and should work with the existing support code and input files as given. All programs should terminate gracefully; that is, the bottom Karel status bar should say, "Finished running" when your code is finished. This means Karel has finished executing commands.

Style: As mentioned in class, it is just as important to write clean, readable code as it is to write correct, functional code. Therefore, we will also grade your program's code quality to give you a **Style grade**. There are many general style guidelines you should follow, such as with naming, indentation, commenting, avoiding redundancy, etc.; for a list, please see the **Style Guide** linked to from the "Assignments" dropdown on the course website.

The Honor Code

Please remember to follow the Honor Code when working on this assignment. Submit your own work and do not look at other people's code or share your code. Cite any help you receive. Limit discussions with others to high-level problem discussions. If you need help, please stop by the LaIR or office hours!