Assignment 1: Email and Karel the Robot

Based on a handout by Eric Roberts and Mehran Sahami

In this first assignment, you'll see just how powerful Karel the Robot can be as you use him to solve a variety of programming problems. But first, Keith and Alisha would really like to meet you! Since we'll be using email as a primary means of communication in this course, we'd like it if you could take the time to send us an email introducing yourself. This assignment consists of two parts:

Introductions!	**Due Sunday, January 18 at 11:59PM**
Karel the Robot	**Due Friday, January 16 at 3:15PM**

Part One: Introductions! (Due Sunday, January 18 at 11:59PM)

We'd love to get to know you in CS106A! As your first task, we'd like you to introduce yourself in an email. Once you've met your section leader, please send an email to Keith (htiek@cs.stanford.edu), head TA Alisha (aadam@stanford.edu), and your section leader that includes

- your name,
- what you like to do for fun, and
- a quick anecdote about something that you feel makes you unique (this can be a cool talent, an interesting experience, or anything else you'd like)!

Note that you won't know your section leader's email address until after your first section, which is why we don't make this part of the assignment due until the Sunday after the rest of the assignment is due.

Please make sure that the subject line of your email says "CS106A Email: <your name>", where "<your name>" is actually filled in with your name. There are about 680 students currently enrolled in CS106A, and having the custom subject line will make it a lot easier for us to group all these emails together.

Part Two (The Real Assignment): Karel the Robot (Due Friday, January 16 at 3:15PM)

The meat of this assignment consists of four Karel programs. There is a starter project including all of these problems on the CS106A web site in the area for Assignment 1. To work on these programs, download that starter folder as described in Handout #05 (Using Karel in Eclipse). From there, you can 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**. You can submit as many times as you'd like; we'll only grade the last version of the assignment that you submit.

We recommend reading through Handout #03 about the Stanford Honor Code before starting this assignment to make sure that you are aware of our Honor Code policies.

A Note about Java

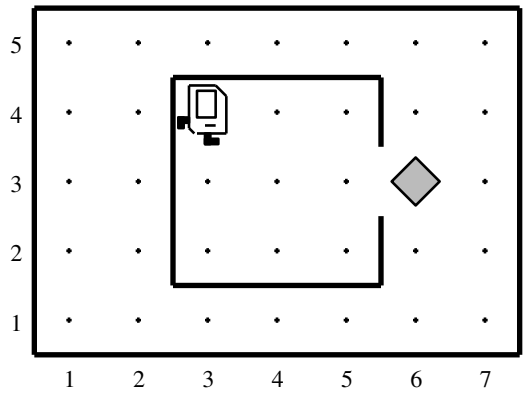
Your Karel programs must limit themselves to the language features described in *Karel the Robot Learns Java* in the `Karel` and `SuperKarel` classes. You may not use other features of Java, even though the Eclipse-based version of Karel accepts them.

In all future assignments, you're welcome to use any features of Java that you'd like, even ones we haven't covered this quarter. This restriction only applies for the Karel assignment.

If you have any questions about what's permitted, please don't hesitate to ask us in person or over email.

Problem One: `CollectNewspaperKarel`

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 always just outside the door. Thus, all you have to do is write the sequence of commands necessary to have Karel

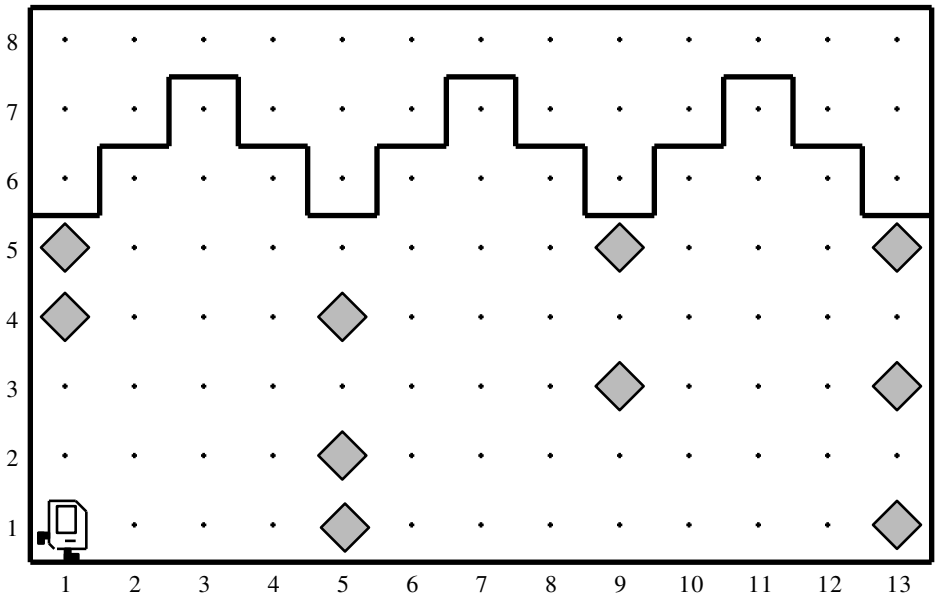
1. Move to the newspaper,
2. Pick it up, and
3. Return to its starting point.

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.

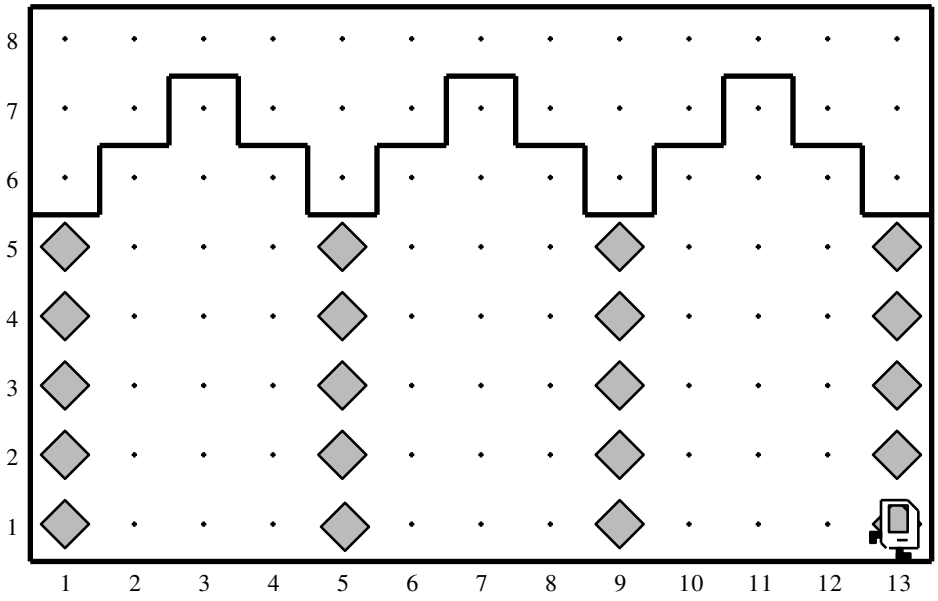
We recommend that you complete this part of the assignment as early as possible. That way, if you run into any trouble getting Eclipse set up, you have plenty of time to ask for help.

Problem Two: `StoneMasonKarel`

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:



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:



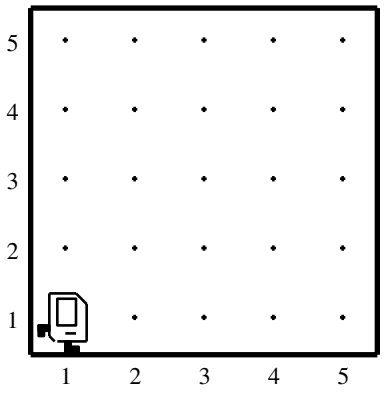
Your program should work on the world shown above, but it should be general enough to handle any world that meets certain basic conditions as outlined at the end of this problem. There are several example worlds in the starter folder, and your program should work correctly with all of them.

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:

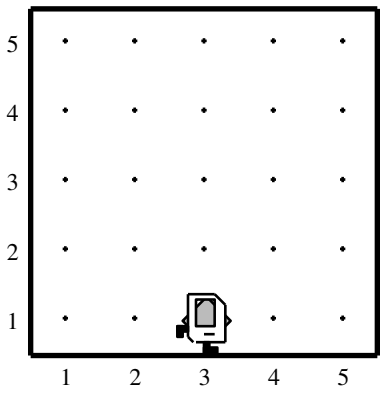
- Karel starts at 1st Avenue and 1st Street, facing east, with an infinite number of beepers in Karel's beeper bag.
- The columns Karel needs to rebuild are exactly four units apart, on 1st, 5th, 9th Avenue, and so forth. The world will be sized so that the final column Karel needs to build will be flush up against the right wall of the world.
- Although in the sample worlds above there are four columns for Karel to rebuild, your program should work in worlds where there are any number of columns that need to be rebuilt. For example, if the world were to be 17 avenues wide, Karel would need to build five beeper columns (at 1st, 5th, 9th, 13th, and 17th Avenues). If the world were to be 1 avenue wide, Karel should just build one beeper column at 1st Avenue.
- 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 Three: `MidpointFindingKarel`

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



Karel should end 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, but must pick them all up again before finishing.

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

- Karel starts at 1st Ave. and 1st St., 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.

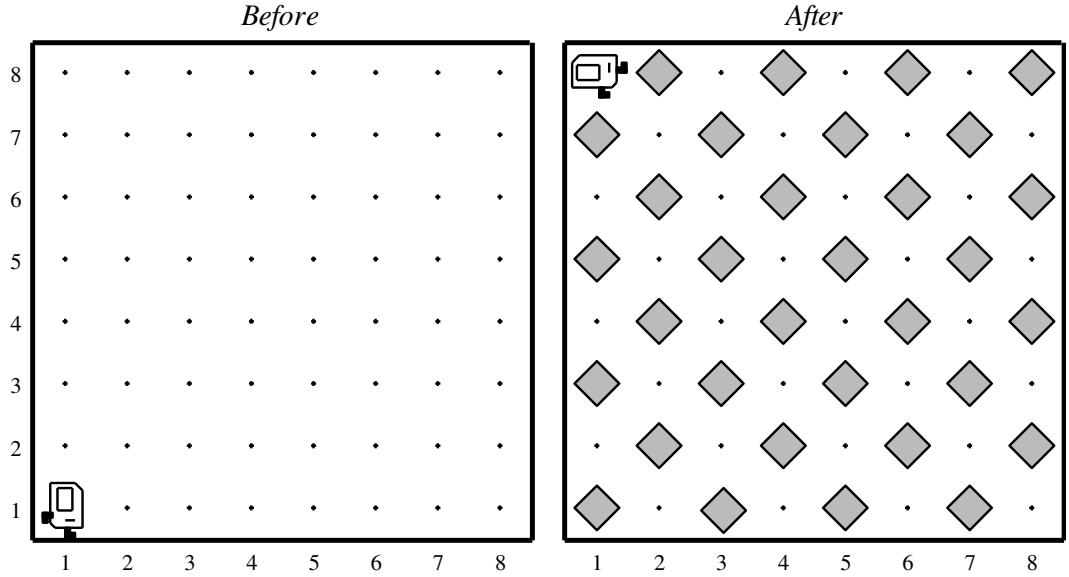
Your program, moreover, can assume the following simplifications:

- 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.

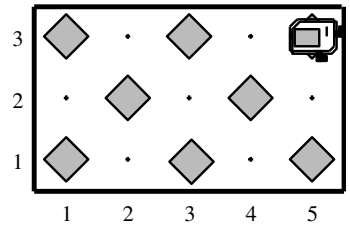
The interesting part of this assignment is to come up with a strategy that works. There are *many* different algorithms you can use to solve this problem (we know of at least ten different approaches), so be creative and have fun coming up with one!

Problem Four: `CheckerboardKarel`

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.)



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 8×8 checkerboard shown in the example. Odd-sized checkerboards are tricky, and you should make sure that your program generates the following pattern in a 5×3 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.

There are many, many different approaches to solving this problem, some of which are significantly easier than others. One specific note we'd like to make: it's perfectly fine for Karel to retrace its steps; Karel's route doesn't have to be "optimized."

Testing Your Programs

All of the Karel problems you will solve (except for `CollectNewspaperKarel`) should be able to work in a variety of different worlds. 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*.

When testing a program, it's important to choose test cases that will each exercise some different aspect of the program's behavior. For example, for the `MidpointFindingKarel` program, you might want to choose a test world with an even number of columns and an odd number of columns. It's also useful to choose test cases that exercise "edge cases," special inputs that are technically legal but might require unusual behavior. For example, in the `CheckerboardKarel` program, a world with just one row or just one column is one type of edge case – it's a legal world for `CheckerboardKarel`, but so small that it might break some assumptions you might have made when writing your programs. What might an edge case look like for `MidpointFindingKarel`?

General Advice

When writing your Karel programs, to the maximum extent possible, try to use the top-down design techniques we developed in class. Break the task down into smaller pieces until each subtask is something that you know how to do using the basic Karel commands and control statements. These Karel problems are somewhat tricky, but appropriate use of top-down design can greatly simplify them.

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 these:

- **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`
	` * at each corner.`
	` *`
	` * Precondition: Karel is facing`
	` *  East at the start of an`
	` *  empty row.`
	` * Postcondition: Karel is facing`
	` *  East at the end of that row,`
	` *  and every corner in the row`
	` *  will have a beeper on it.`
	` */`
	`public void fillRowWithBeepers() {`
	`  while (frontIsClear()) {`
	`  putBeeper();`
	`  move();`
	`  }`
	`  putBeeper();`
	`}`

- **Did you decompose the problems into smaller pieces?** There are many ways to break these Karel problems down into smaller, more manageable pieces. Breaking the problem apart elegantly will result in a small number of easy-to-read methods, each of which performs just one small task. Breaking the problem apart 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)?
- **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();`
	`}`
	`}`
	`}`

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 on by the Tresidder LaIR with questions, come visit us during office hours, or email your section leader with questions!

Good luck!