Stanford University, CS 106A, Homework Assignment 1
Karel the Robot (INDIVIDUAL ASSIGNMENT; NO PAIRS)

Based on past editions of this assignment as created by Mehran Sahami, Keith Schwarz, Eric Roberts, and others.

The purpose of this assignment is to introduce you to basic programming and Java using Karel the Robot. You will solve four Karel programming problems. We recommend you read the **"Using Eclipse and Karel"** handout posted on the class web site, which describes how to set up your project in Eclipse, before starting HW1.

This is an **individual assignment**. You should write your own solution and not work in a pair on this program.

There is a **starter project ZIP archive** including all of these problems on the CS 106A web site in the area for Homework 1. To work on these programs, download that starter folder as described in the Homework web page. 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.

You will **turn in** your entire project via the Stanford Eclipse submitter. We will grade the following files:

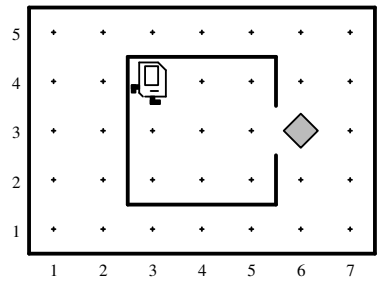
- **CollectNewspaper.java**, **StoneMason.java**, **Checkerboard.java**, **MidpointFinder.java**, and **AnythingGoes.java**, your Java code solutions to Problems 1-5
- **AnythingGoes.w**, a world file that accompanies your solution to Problem 5

The ZIP archive contains other files and libraries; you should not modify these. When grading/testing your code, we will run your **.java** code with our own original versions of the support files, so your code must work with them.

When finished, you will **submit your assignment** using the Submit Project entry under the Stanford Menu. 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.

On this assignment you must limit yourself to using only Java commands and syntax discussed in lecture or in the ***Karel the Robot Learns Java*** **in the `Karel` and `SuperKarel` classes.** Do not use other features of Java, even though Eclipse accepts them. For example, **do not use variables or data fields** in this program. Doing so will heavily reduce your score.

Problem 1: `CollectNewspaper`



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

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

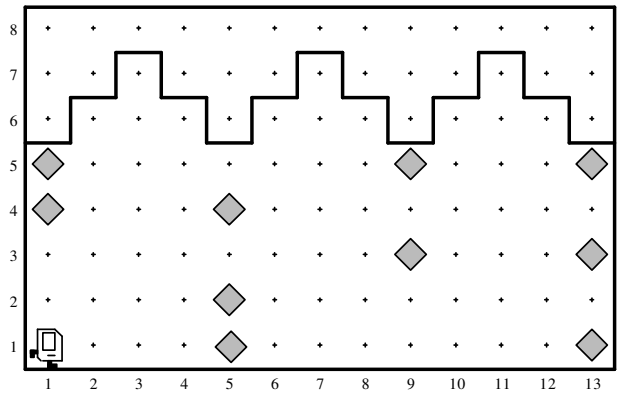
Even though the program is short, it is still worth getting practice with decomposition. In your solution, include a **method** for the first and third step shown in the outline.

NOTE: Every year, a handful of students run into some kind of problem with the electronic submission. If you wait until right before the deadline before you submit your work, you may discover that there is an issue after it's too late to get help. So we suggest that you test the submitter early. As soon as you finish `CollectNewspaper`*, click* ***Submit Project****. Once you've done so, you'll know that the submission process works. We only look at your last submission for grading, so it doesn't hurt to re-submit after each problem.*

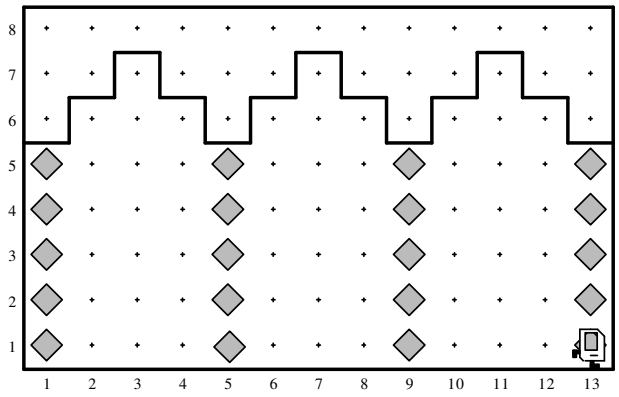
Problem 2: StoneMason

Karel has been hired to repair the damage done to the Quad in the 1989 earthquake. Karel is to repair a set of arches where some stones (represented by beepers) are missing from the columns supporting the arches, as in the "Before" diagram below at left. When Karel is done, **the missing stones in the columns should be replaced by beepers**, so that the final picture will match the "After" diagram shown below at right.

Before:



After:



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 below. There are several example worlds in the starter folder, and your program should work correctly with all of them.

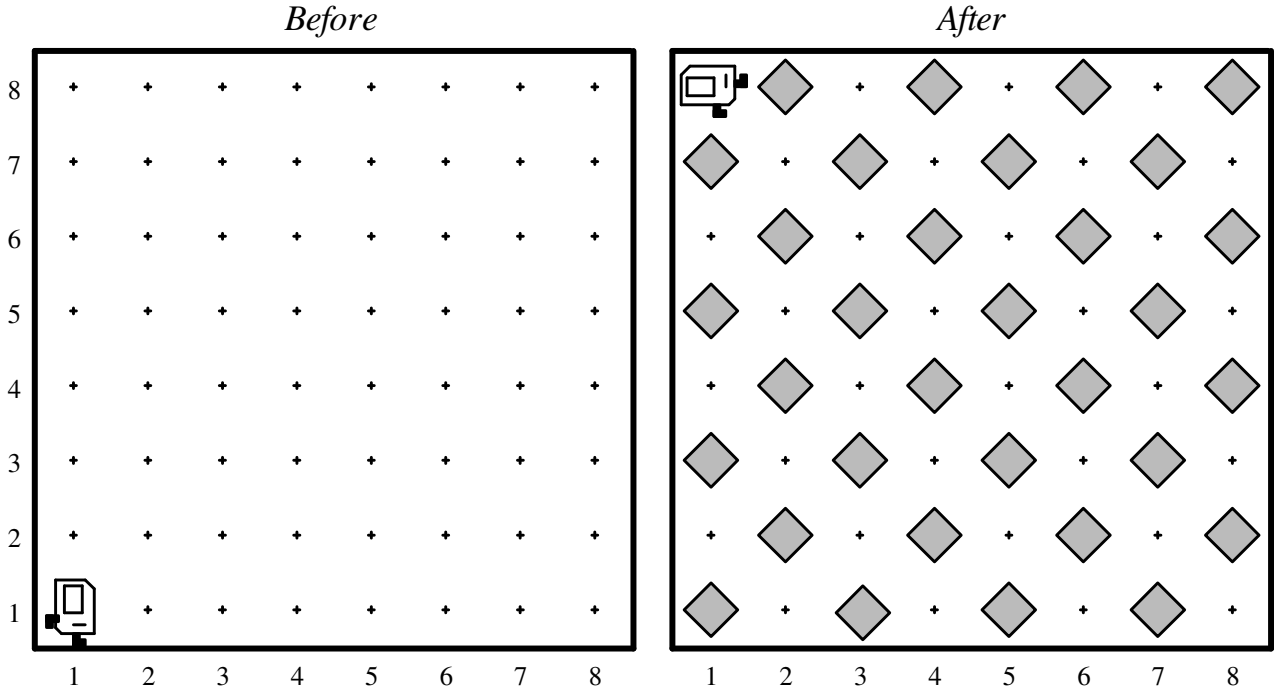
Karel's final location and the final direction he is facing at end of the run do not matter. Karel may count on the following facts about the world:

- Karel starts at 1st Avenue and 1st Street, facing east, with an infinite number of beepers in his 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 overall world will start and end with a column, so it will have a total number of avenues such as 1, 5, 9, 13, etc.
- The top of the column is marked by a wall, but Karel cannot assume that columns are always five units high, or 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.

Hint: Some students try to have Karel walk along the top of the columns and build them downward. We think that is unlikely to lead to a successful solution.

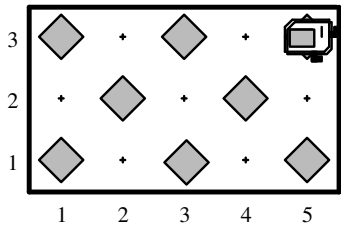
Problem 3: Checkerboard

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 he is facing at end of the run do not matter, but the program should finish running rather than repeating infinitely.)



Your program should put the beepers in *exactly* the squares shown. For example, in the picture above, Karel has placed beepers at his starting square of (1, 1), as well as (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 into the 1 square.

As you think about how you will solve the problem, make sure that your solution works with checkerboards that are different in size from the standard 8x8 checkerboard shown in the 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.

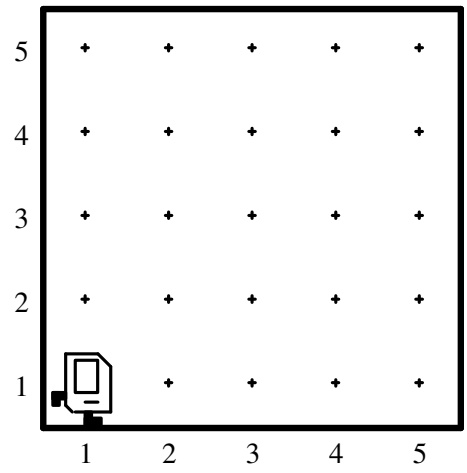
This problem can be **tricky**, so we suggest developing it in stages using the technique of "stepwise refinement" as described in the text. For example, write an initial version that just checkers a single row, or a single column, or some other chunk of the overall work. Test it until you're satisfied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

Hint: The `Roomba` program written in lecture may be a useful example to look at.

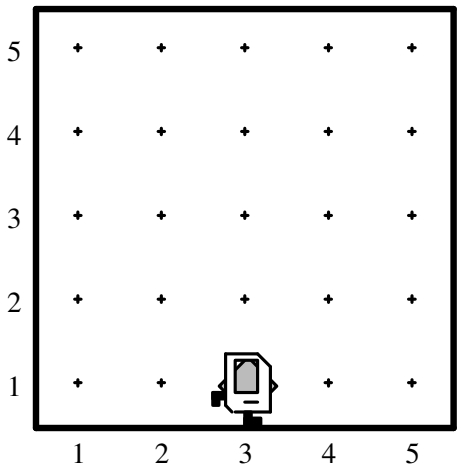
Problem 4: `MidpointFinder`

Program Karel to place a single beeper at the center of 1st Street. For example, if Karel starts in the world shown below at left, he should end **standing on** a beeper in the position shown below at right.

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 he wants, but he must pick them all up before he is finished.

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

- Karel starts at 1st Ave. and 1st St., facing east, with an infinite number of beepers in his 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.
- 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 he has placed there.

The reason that this problem is tricky 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 assignment is to come up with a strategy that works for all cases. There are *many* different algorithms you can use to solve this problem (we have seen at least ten 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 reader; you may **not** use Java variables or data fields.

Problem 5: `AnythingGoes`

For the fifth and final Karel problem, write a Karel program that does *anything you want*. The idea here is that unlike the other four problems, this one is meant to be just for fun and meant to exercise your creativity. You should also create a Karel world file named **AnythingGoes.w** that represents the world for Karel to explore. Be creative! See the "Using Eclipse and Karel" handout for information about how to create, edit, and save a Karel world file. Save it into the **worlds** subdirectory of your Eclipse project.

Your Karel's behavior must meet these constraints:

- Your file must be at least 25 lines long.
- Your code must contain at least 2 methods.
- Karel must move in some way.
- It cannot be essentially the same as Prob. 1-4.
- It should be non-offensive (rated "PG").

Your Karel world must meet the following constraints:

- It must be a valid Karel world file.
- It must be of non-trivial size (greater than 1x1).
- It must contain at least one wall and/or beeper.

Unlike the other four parts, **AnythingGoes** is **not graded on style**. So you can use any Java features you want.

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 the 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, then continue. When writing your Karel programs, as much as possible, 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 somewhat tricky, but appropriate use of top-down design can greatly simplify them.

All of the Karel problems you will solve (except for `CollectNewspaper`) 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 `CollectNewspaper`) in a variety of test worlds. Each quarter, many students submit Karel programs that work well in the default worlds but fail 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.

Grading:

Functionality: We will grade your program's behavior to give you a **Functionality score**. You can run your program with various worlds to help check your behavior for various test cases. Your code should compile without any errors or warnings, 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 has finished.

Style: 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. We will also grade your code quality to give you a **Style score**. There are many general Java coding styles that you should follow, such as naming, indentation, commenting, avoiding redundancy, etc. For a list of these, please see the **Style Guide** document on the class web site. Follow its guidelines on all assignments.

You should limit yourself to using Java syntax discussed in lecture or in assigned sections of the textbook. On this program, **your files must limit themselves to the language features described in *Karel the Robot Learns Java* in the `Karel` and `SuperKarel` classes.** Do not use other features of Java, even though Eclipse accepts them. For example, **do not use variables or data fields** in this program. Doing so will heavily reduce your score.

Procedural decomposition: Your `run` function should represent a concise summary of the overall program, calling other methods to do the work of solving the problem, but run itself should not directly do much of the work. For example, run should not directly call `move`. Instead, `run` should call other methods to achieve the overall goal.

Each function should perform a single clear, coherent task. No one function should do too large a share of the overall work. As a rough estimate, a method whose body (excluding the header and closing brace) has more than 20 lines is too large. Your functions should also help you avoid **redundant code**. If you are performing identical or similar commands repeatedly, factor out the common code into a method, or otherwise remove the redundancy.

Commenting: Your programs should have adequate **commenting**. The top of each .java file should have a descriptive comment header with your name, a description of the problem you are solving, and a **citation of all sources** you used to help write your program. Each method you write should have a comment header describing its behavior. For larger methods, you should also place a brief inline comment on any complex sections of code to explain what the code is doing. See the programs from lecture and Style Guide for examples of proper commenting.

Honor Code: Please remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also please do not give out your solution and do not place a solution to this assignment on a public web site or forum. If you need help, please seek out our available resources to help you.