HW4: Programming with recursion

Due: 12PM, Noon Thursday, September 25 through turnin
Total: 20pts

You may work on this assignment with one other student. A team of two members must practice pair programming. Pair programming "is a practice in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test." Both members in a team must read the article All I really need to know about pair programming I learned in kindergarten. This article can be downloaded to any computer that is in the Marquette network.

Preparation: get familiar with GridWorld.

The Advanced Placement GridWorld Case Study is a useful starting point for this assignment. Begin by downloading GridWorld code, as well as the Student Manual for your reference. Online resources (such as this YouTube video) can help you get started with setting up the GridWorld JAR file under Eclipse, in order to beginning running the sample projects.

GridWorld already contains a variety of `Actor`s that can inhabit the grid: `Bug`s, `Flower`s, `Rock`s and `Critter`s. This assignment will only be concerned with `Rock`s, `Flower`s, and `Critter`s.

For this assignment, you will be implementing your own extension of an existing GridWorld classes to simulate the `PsychicGecko`.

The bounded grid for this assignment will always begin with precisely two `PsychicGecko`s on the board, an arbitrary number of `Rock`s, and no other `Actor`s.

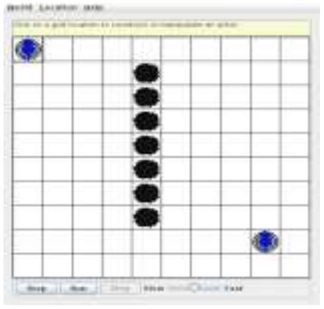
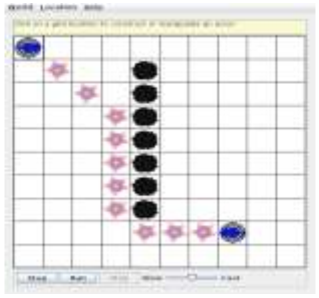
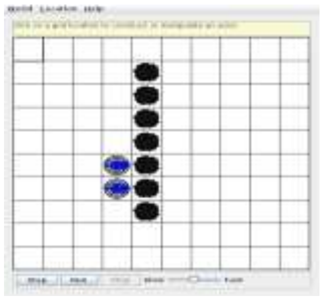
- The `Rock`s are obstacles that occupy the squares in the bounded grid. `Rock`s do not move.
- The `PsychicGecko`s are able to move. The two `PsychicGecko`s try to meet and occupy two adjacent squares (in the same row or column). However, a `PsychicGecko` does not move until it has already figured out the entire path that leads to its counterpart. The path to its counterpart should not pass through any `Rock`s.

Task: design `PsychicGecko` and `PsychicBean` classes.

Class `PsychicGecko`:
Create a new `Critter` type, `PsychicGecko`, that detects its counterpart on the grid, figure out the entire path to its counterpart, and moves toward it on each step following the path. Details include:

(1) The `PsychicGecko` can move one square at a time, similarly as the existing `ChameleonCritter` example class (`GridWorldCode/projects/critters/ChameleonCritter.java`). Unlike `ChameleonCritter`, `PsychicGecko` doesn't just randomly moves about the board, nor change its color to match adjacent `Actor`s.
(2) The `PsychicGecko` will use a new `Actor` subclass, `PsychicBean` to leave a trail of breadcrumbs in grid locations it has already explored telepathically.
(3) When the trail of `PsychicBean`s finds the other `PsychicGecko`, the seeds turn into `Flower`s, leaving a path that the `PsychicGecko`s will eat on their way to meet each other.

The following three figures show some status of the board when `PsychicGecko`s act.

 Initial board.	 An entire path is found.	 `PsychicGecko`s have eaten the flowers and reached each other.

A driver program our PsychicTestRunner.java is provided for you. You will at least override the public method act(). In addition, you will implement several private methods to help simply the logic. Two of them will help figure out the direction to its counterpart in the `PsychicGecko` class.

1. `private int getDirectionToOther(Location fromLoc, Location toLoc)`
   Returns its direction from `fromLoc` to `toLoc` in terms of the static constants in the `Location` class.
2. `private ArrayList<Location> sortByDirection(Location origin, int direction, ArrayList<Location> unsorted)`
   Returns a sorted version of the `unsorted` list, with the locations in ascending order by difference in direction from the given origin point.

   A `PsychicGecko` uses the following algorithm in `act()` to move to its counterpart:

   - Case 1: if there is an adjacent `PsychicGecko`, move nowhere.
   - Case 2: if there is an adjacent `Flower`, move toward it and eat it.
   - Case 3: if there are neither of these, execute a recursive search of all adjacent grid locations.

     For Case 3, the `PsychicGecko`'s telepathic search should use Recursive Backtracking and marking to prevent repeated searching of dead ends. It begins by examining each empty location immediately adjacent to the `PsychicGecko`:

     - If a `PsychicGecko` distinct from the original is found, the search terminates, and each `PsychicBean` along the found path should be replaced with a `Flower`. (base case for the recursive algorithm).
     - Otherwise, a new `PsychicBean` is placed at the location, marking the space so that the search does not circle back around needlessly.
     - Each empty location adjacent to this new `PsychicBean` is then recursively searched in the same way. If there are several such empty squares to consider, they are sorted in order according to their difference from the direction of the other `PsychicGecko`. (Recursive cases)

`Class PsychicBean:`
The `PsychicBean` class is merely a placeholder, like `Rock`. My implementation overrides the `act` method to remove itself from the grid after each step.

Test your design and implementation.

Points for this assignment will be awarded based upon successful completion of a battery of testcases. As you might guess, test scenarios will consist of placing two `PsychicGecko`s at various locations on the board, with diverse patterns of `Rock`s between them.

You should create many of your own testcases, but we will only evaluate your `PsychicGecko.java` source file.

Here are four typical testcases:
02-OneSpace.test
01-Barrier.test
06-BlindAlley.test

Here are PsychicGecko.gif and PsychicBean.gif image files. Placing these in the same directory with your .class files will allow the GridWorld system to use the images in your display.

Submission.

Submit `PsychicGecko.java` source file only to turnin system. You can submit multiple times but only the last submission will be saved by turnin. If you work in a team of two, always let the same user submit the file. The lower grade from two submissions will be used if each team member submits.

Pay attention to the following details for successful submissions and tests.

(1) Name your class precisely -- spelling and capitalization must match this assignment specification for you to pass any of our testcases.
(2) Do not modify any of the other `GridWorld` classes -- or, at least, do not make your solution depend upon any modifications you make to other `GridWorld` classes. Your `PsychicGecko.java` source file will be graded in a fresh copy of `GridWorld`, with our own test framework, by the TA-bot.
(3) Make certain that your code compiles correctly -- no points can be awarded if your submission will not run to pass any of the testcases.
(4) If you put any debugging `System.out.println()` statements in your code during development, please comment these out before submission. Stray output from your program will confuse TA-bot.