CSE 8B Homework 4:
Maze Solver

Learning Goals:

- Write code to navigate through 2D arrays
- Write code to modify 2D arrays within a method
- Read data from a file
- Write methods to test other methods

Get started

1. Make sure there is no problem with your Java coding environment. If there is any, then please review Assignment 1 or come to office hours before you start this assignment.
2. Download the starter code here. You should implement your code in `PA4.java`.
3. Read and understand the MazePoint class. **You will not modify this class.**
4. You should compile `PA4.java` via `javac PA4.java`. You will run PA4 via `java PA4` for this assignment.

NOTE: You should not change/delete any given starter code (except some dummy return values and comments).

Reminder: Coding Style

When coding in Java, there are several style rules that people usually follow to make the code clean and readable. **In this course, you are asked to follow rules specified in link below:**

https://cseweb.ucsd.edu/classes/fa20/cse8B-a/styleguide.html

This link is also found on Canvas. Please read the coding style guide carefully and refine your code for this and all future assignments. Sometimes, coding style can help debug your code, so doing so is a win-win!

Part 1: ReadMaze (implementation and testing) (Pair programming allowed)

In class PA4 of the starter code, eight methods are already declared: `readMaze`, `escapeFromMaze`, `printMaze`, `mazeMatch`, `testRead`, `testEscape`, `unitTests` and `main`. Your task in *this part* (Part 1) is to implement the following

1. `public MazePoint[][] readMaze(String fileToRead)`

   This method will read the maze file and parse the maze to a 2D array of `MazePoint` objects. The input `fileToRead` is the filename of the maze you want to read. Apply what you have learned in your Zybooks reading and in-class to read the input file (HINT: use `Scanner` to read in the input file).

   The input file contains the information of an input maze.

   1. The first line of the file will always contain two numbers that are space-separated.
      i. The first will represent the number of rows in the maze, while the second represents the number of columns. These numbers need not necessarily testreadbe equal (*i.e.* the maze could be a rectangle). You can assume these numbers will always be positive.
   2. The rest of the lines of the file is the input maze:
      i. Open spaces in the maze are represented by a `-` (minus sign). Walls are represented by a `X` (capital X). No other characters will be in the maze.
      ii. Between every column, there will be a single space. Every row is separated with `'\n'` (a new-line character).
      iii. There are no leftover blank lines in the input file (*i.e.* no trailing whitespace). There are no extra spaces between rows or between columns. There are no extra spaces before nor after each line.

   Read the input file, and convert the input file into a 2D array. Do NOT forget to close the Scanner for the input file after you finish the maze conversion.

   NOTE: You can assume that the file always exists, and that the file is properly formatted (*i.e.* no corrupt/bad inputs).

We have provided two maze files as examples. They are titled `input1` and `input2`. Open them using any text editor.

2. `public boolean testRead(String fileToRead, MazePoint[][] expected)`

   This method takes in two parameters - a file name called `fileToRead`, and a 2D array of `MazePoint` objects called `expected` which represents the expected array that the `readMaze` method should yield after reading the file. The function will print out the expected array (`expected`) and the array returned by `readMaze`. It returns a `boolean` value - `true` if these two arrays are matched, and `false` otherwise. If `expected` does NOT match the array returned by `readMaze`, then beside the 2 arrays, this method should also print an error message. Refer to Part 3 for an example output.

   The `printMaze` and `mazeMatch` helper methods:

   We have provided empty declarations of two functions: `printMaze` (which prints the characters in the maze) and `mazeMatch` (which checks to see if two mazes contain the same symbols in the same locations). You must write these methods, and then call them in the appropriate places in `testRead`. You should also call them in `testEscape` (Part 2).

3. Add tests to `public boolean unitTests()`

   Add code to call `testRead` on *at least three* different maze files. One test is already given to you. In other words, you need to create and add *at least two* more tests.

   NOTE 1: you will need to make sure that `unitTests` returns false if **any** of the calls to `testRead` fail.

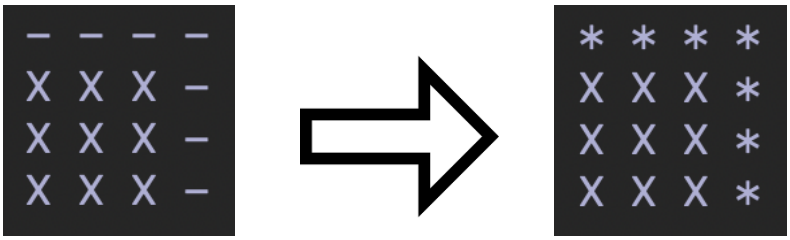
   NOTE 2: `unitTests` is already called from `main`, so when you add your tests to `unitTests`, they will automatically be called when you run PA4 via `java PA4`.

Part 2: EscapeFromMaze (Implementation and Testing) (Pair programming allowed)

1. `public void escapeFromMaze(MazePoint[][] maze)`

As a parameter, you are given a **`maze`** in the form of a 2D-array of MazePoint objects. You *always* start at the **top-left corner** of this maze, and your job is to find a path to go from the top-left corner to the **bottom-right corner** of the maze (which is the exit). You can **only** go **right** OR go **down** - the maze should never give you the option to move in both directions. All the `X`'s in this 2d-array mean blockers, and all `-`'s means empty spaces you can move to. There will always be a path to the exit, and there will never be any forks or dead ends.

Your job is to trace out the path taken to solve the maze. Modify the **`maze` in-place**. In other words, you should not have to create a new 2D-array of MazePoint objects. You are required to "record" the position of each step you take on your way by using an asterisk (`*`) to replace `-`'s. For example, if you have the following input 2D-array, then that 2D-array will become the output array after this method is finished running:



input output

This function must handle edge conditions gracefully. For example, what would happen if the input maze is `null`, or if the number of rows and/or columns is less than one? For each of these cases, your method should not crash, but instead display an appropriate error message and `return`.

IMPORTANT: In essence, you need to write a program to find a way out of a maze. Keep in mind that the maze given in the input files are only two examples. Your program should also work if the maze is slightly changed. **You will receive 0 credit if you hard code your answer by just giving the fixed path.**

2. `public boolean testEscape(MazePoint[][] maze, MazePoint[][] expectedSolution)`

This function takes in two parameters: a 2D-array of `MazePoint` objects representing the test input (`maze`), and another 2D-array of `MazePoint` objects that represents the expected maze after solving the escape puzzle (`expectedSolution`). The function should run `escapeFromMaze` on `maze`, and verify that, after running `escapeFromMaze`, `maze` matches `expectedSolution`. The function returns a boolean value - `true` if the expected array (`expectedSolution`) matches the array returned by `escapeFromMaze`, and `false` otherwise. If `maze` does NOT match with `expectedSolution`, then this method should also print the expected solution maze, then print the actual maze produced by `testEscape`, then print an error message.

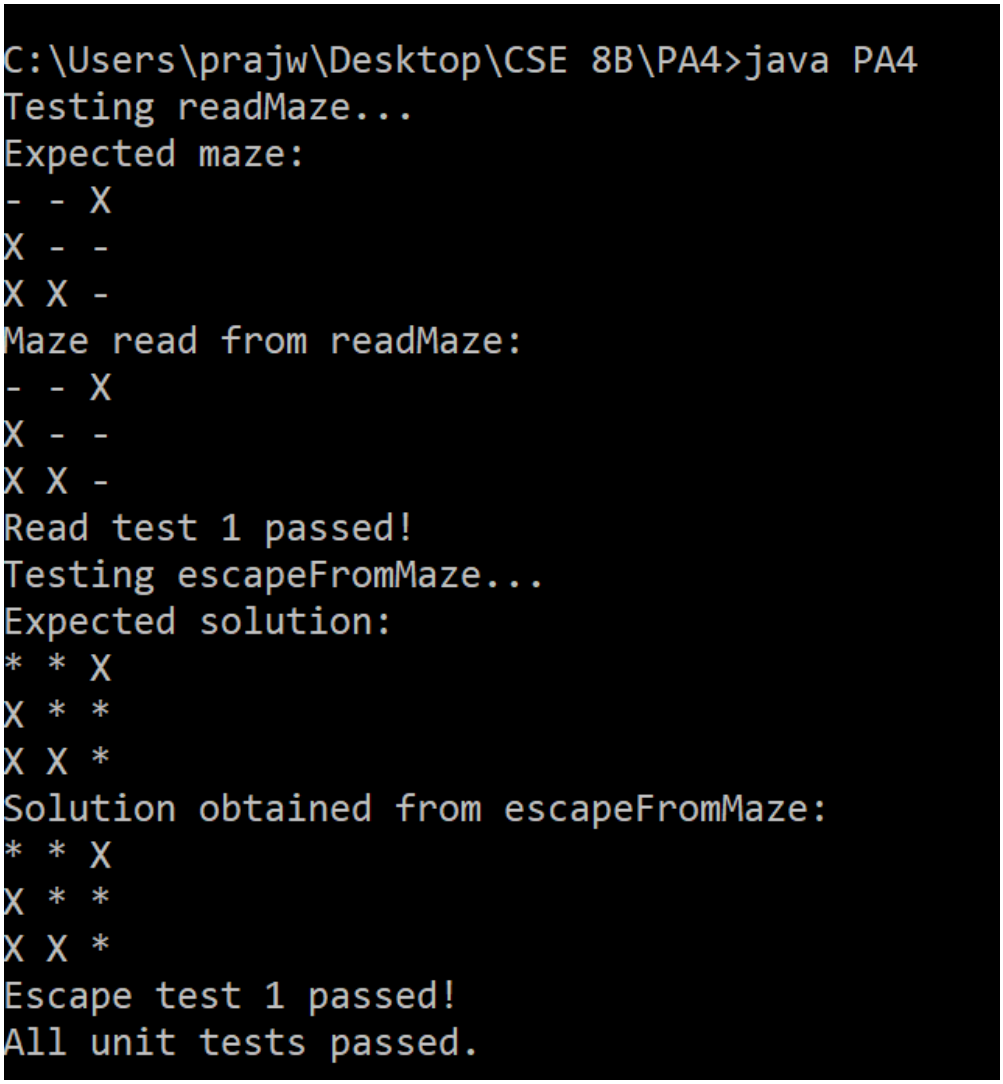
3. Add tests to `public boolean unitTests()`
   Add code to call `testEscape` on *at least **four*** different mazes. We have already provided one test for you. In other words, you should add *at least three* more tests. *At least* one of your tests should be on an invalid maze (`escapeFromMaze` should leave the invalid maze unchanged and not crash). Again, you will need to make sure that `unitTests` returns false if any of the calls to `testEscape` fail.

Part 3: A Maze Solver loop (implementation and screenshot) (Pair programming allowed)

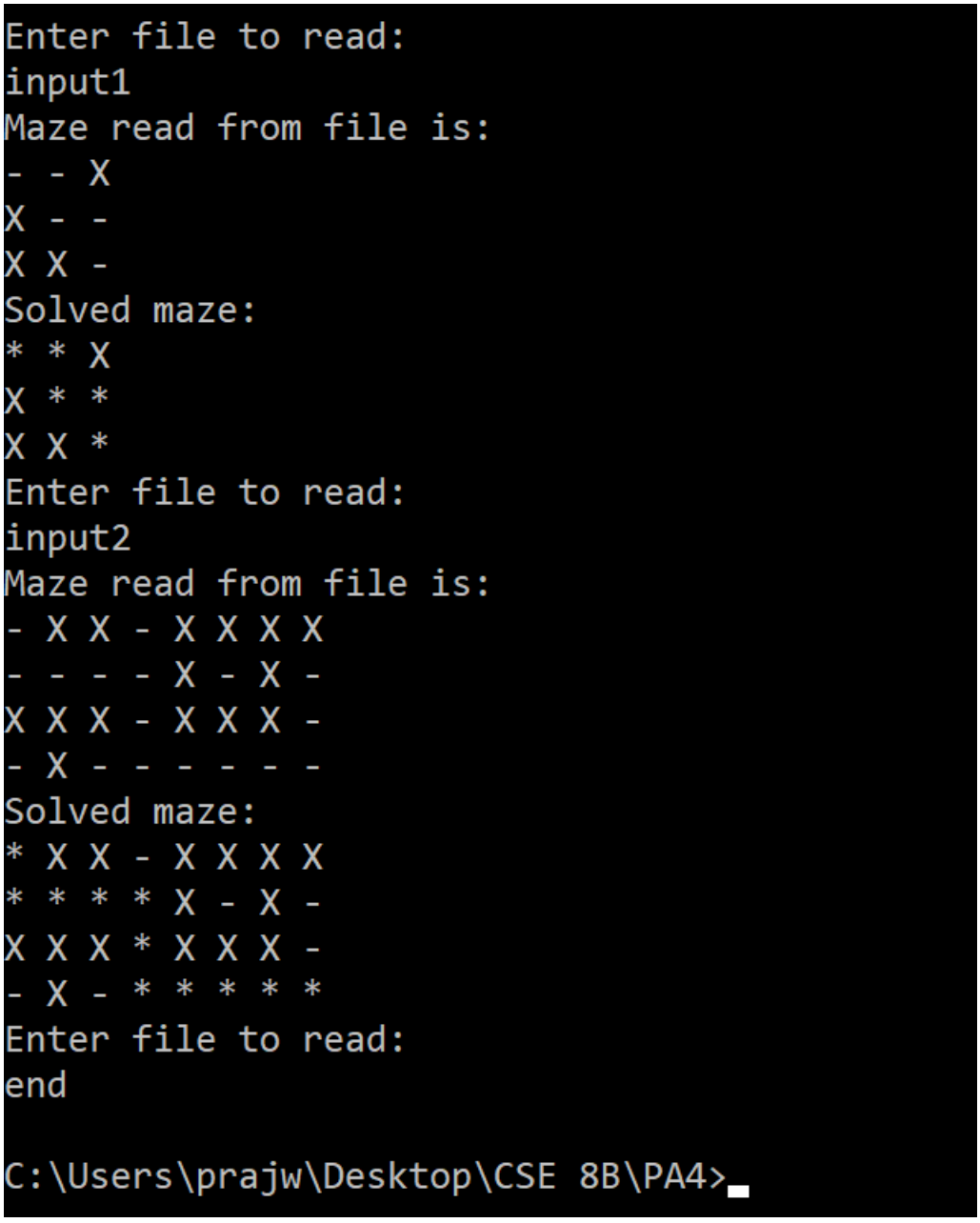
In the `main` method after the call to `unitTests`, add code to do the following:

1. Prompt the user to enter the name of a file containing a maze.
2. Read in the user's input.
3. If the user entered the string `"end"`, then the program should stop. Otherwise:
   a. Open the file
   b. Print the maze
   c. Solve the maze
   d. Print the solved maze
4. Return to step 1.

Example: you should be able to reproduce the output below with your program. These two screenshots are actually from the same program execution (`java PA4`), but they have been split into two to fit into this document.



The output above shows the results from our unit tests. After these unit tests, you should prompt for user input and follow the 4 steps above.



The output above is what you should expect after correctly implementing Part 3.

Run your program and show the result of solving at least three different mazes. Take a screenshot of the output of your program, and save your screenshot in a PDF file for submission.

Part 4: Mid-quarter feedback (individual)

This week you will receive credit for providing us with feedback about how the class is going for you. The survey is not anonymous because we need your name to give you credit, but we will review the results without names attached. You will provide this feedback via a Google Form survey, which will open on Friday of week 4 at 8am PST. (It is due Tuesday, Feb 1 at 11:59pm.)

- Survey form

Grading Information

- 1 point for `readMaze` correctness
- 1 point for `testRead` correctness
- 1 point for `testRead` tests (in `unitTests`)
- 2 points for `escapeFromMaze` correctness
- 1 point for `testEscape` correctness
- 1 point for `testEscape` tests (in `unitTests`)
- 1 point for `main`
- 1 point for screenshot of correct behavior in `main`
- 0.5 points for code style
- 0.5 points for mid-quarter feedback

Star Point Options:

1. You may implement a function `findPath` that takes in a `MazePoint[][]` and returns a `int[][]` that stores the path taken to correctly solve the maze. For example, since each maze starts at the top left corner, then your first index in your path array, i.e. `path[0]` should be `[0,0]`. If the next step along the correct path is at (1,0), then `path[1] = [1,0]`, and so on.
2. Create a function that solves mazes in which traversing the correct path includes upward or left movements.
3. Any additional methods or functionality to perform on the mazes (i.e. maze rotation). Be creative and have fun!

Submission

Very important! Please follow the instructions below carefully and make the exact submission format. This is important since we will use scripts to grade so if you do **NOT** follow the same submission format, then you probably will receive a zero.

1. Run your **PA4.java** file, screenshot the output of your program, convert that screenshot into a PDF file.
2. Go to Gradescope and click on PA4 - Tests.
3. Click the DRAG & DROP section and directly select the PDF file that contains the screenshot of your program's output.
4. Go to Gradescope and click on PA4 - Code.
5. Click the DRAG & DROP section and directly select the **PA4.java** file. Drag & drop is fine. Make sure the name of the file is correct.
6. You can resubmit unlimited times before the due date. Your score will depend on your final submission, even if your former submissions have a higher score.
7. The autograder is for the use of the instructional team. You won't see the result of the autograder. As long as you uploaded your file you're good to go.