Maze solver
using a queue

You will do this project **without** using the `java.util` library, which means, you'll have to do ALL the implementation work yourself except for Scanner and Random.

A maze is a 2D array of char (char [][] maze;)
Walls are marked with `'#'` and open paths are marked with space `' '`.
This maze represents 10 rows and 10 columns.
Enter the maze at row 0 column 1 and exit at any exit (a cell on an edge marked with space `' '`. The exit in the example below is at row 8 column 9.

```
10 10 0 1 8 9
# ########
# ##     #
# ## ### #
# ####   #
#     ## #
# # #### #
# # #    #
# # # ## #
# #    #  exit
##########
```

Your output should trace a path from the entry point to the exit point. Similar to the diagram below. Dots `'.'` represent a path from entry to exit.

```
#.########
#.##     #
#.## ### #
#.####   #
#.....##.#
#.#.####.#
#.#.#....#
#.#.#.##.#
#.#....#..
##########
```

Solving a Maze with a Queue
The process of solving a maze with queue is a lot like "hunting" out the end point from the start point. To do that, we must ensure that we test all possible paths. The queue will allow us to track which spaces we should visit next.

The basic routine works as follows:

- Initialize a queue with the start location (0,1)
- Loop Until: queue is empty or the exit point reached or a `' '` in the outer perimeter is reached.
    - Dequeue (remove) current location and mark it as visited
    - If current location is the exit point
        - Break! We've found the end
    - Enqueue (insert) all locations for reachable and unvisited neighbors of the current location
    - Continue the loop.

Looking over this routine, imagine the queue, full of spaces. Each time you dequeue (remove) a space you are searching around for more spaces to search out next. If you find any, you add (insert) them to the queue to eventually be analyzed later.

Because of the FIFO nature of a queue, the resulting search pattern is a lot like a water rushing out from the start space, through all possible path in the maze until the end is reached. This search pattern is also referred to as Breadth First Search (or BFS).

(15 points)
You will need to create the following classes:

1. **Location** class that represents a location on the maze to have a row number and a column number. You will use it to create a location object when you need to enqueue (insert) a location. You might also need getter and setter methods.
2. **LocationQueue** class implemented by a Linked List, that would hold Location objects. You can modify a copy of our queue example that holds numbers. So instead of int values, it should hold Location objects.
3. **Maze** class where you write your main() method.

Your main method should do the following:

(5 points)
a. Ask the user for the name of a maze file.
b. Read the first line which has six numbers like
`10 10 0 1 8 9`

This maze represents 10 rows and 10 columns.
Enter at row 0 column 1 and exit at row 8 column 9.
You can use code like the following to read these numbers:

```
int rows = inputFile.nextInt();
int cols = inputFile.nextInt();
int startRow = inputFile.nextInt();
int startCol = inputFile.nextInt();
int endRow = inputFile.nextInt();
```

    int endCol = inputFile.nextInt();

The rest of the maze file represents the maze which you will need to load onto your 2D char array.

```
# ########
# ##     #
# ## ### #
# ####   #
#     ## #
# # #### #
# # #    #
# # # ## #
# #    #
##########
```

c. Write a method to print the maze to make sure your code reads the maze correctly.

(15 points)
d. Write code to find the path and mark the path cells with `'.'` as described in the algorithm on the previous page.

e. print the maze. If your code is correct you should get this output:

```
#.########
#.##     #
#.## ### #
#.####   #
#.....##.#
#.#.####.#
#.#.#....#
#.#.#.##.#
#.#....#..
##########
```

Where you will have the character `'.'` marking a path from the start to the end point.

(5 points)
Documentation, correct alignment, use of meaningful variables, and design.

Sample maze files:
maze3.txt, maze2.txt
Create your own maze map file and test your code.

(10 points)
Each team member is required to complete (typed):

1. Write a short, individual paper summarizing what you learned from the assignment and what you contributed to the team.

2. Fill out the Peer Work Group Evaluation Form
3. Fill out the Numerical Peer Evaluation (Self Included)