COMP-1027 **Assignment 3** Summer 2025

Learning Outcomes

- Recursion
- Algorithms

Introduction

Everyone loves a good maze. They're fun, they make you think, and they provide a feeling of satisfaction once solved. They are also, however, annoying. Because of this, we are going to be writing a maze solver. Given a maze provided within a textfile, you are going to write a static method, **solveMaze**, to solve that maze (regardless of the size). In addition, there will be a graphical component which makes the whole thing a little more enticing (this part is written for you).

The method signature for the method you'll be writing is:

```java
public static boolean solveMaze(char[][] maze, int row, int col,
ArrayList<String> path, MazeVisualizer visualizer)
```

It takes a 2D char array representing the maze, int row (representing the row index you're currently at), int column (representing the column index you're currently at), ArrayList<String> path (representing the directions to take to solve the maze that you will build upon), and MazeVisualizer visualizer (I give you the visualization code, don't worry about that part).

There are five textfiles provided to you, each containing a unique maze of some size. Let's look at an example maze:

#########

#S #   #

#

#

##

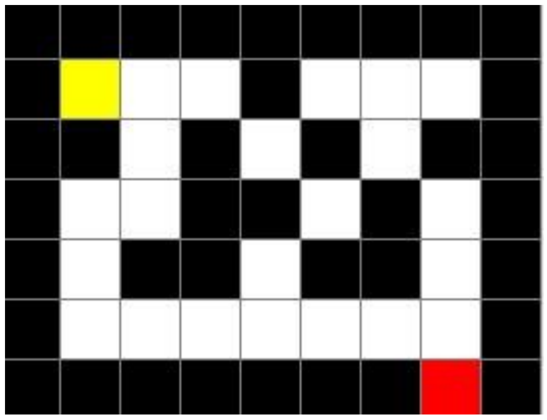
#   #

#######E#

This maze has seven rows and nine columns.

- The '#' character represents a black tile (a wall).
- The spaces represent a hallway (you can navigate through these areas).
- The 'S' character represents where the player begins.
- The 'E' character represents where the player must navigate to in order to solve the maze.

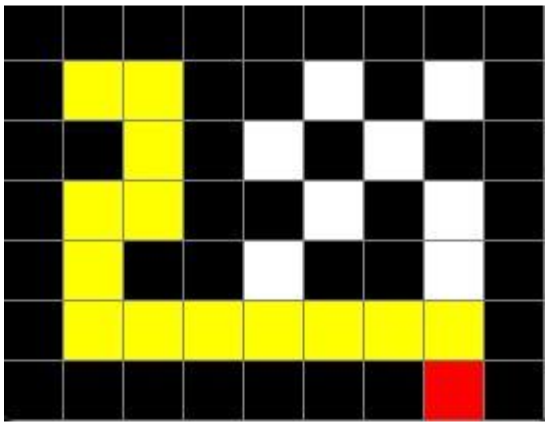
So, visually, the above maze would look like this:



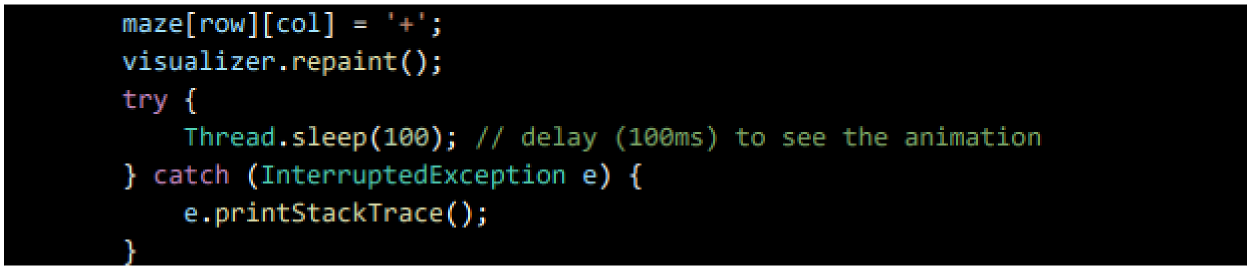
Let's take the second row of the text. It is #S #   # (#, S, space, space, #, space, space, space #) which translates to wall, starting position, open block, open block, open block, wall, open block, open block, open block, wall. We can see this in the second row of the above graphic.

Your objective is to figure out a way to solve mazes in this format. You will write your algorithm in a static method '**solveMaze'** within class **MazeSolver**. Everything else (including the visualization) is taken care of for you.

Once your solveMaze method is written, your method will allow for exploration of the gridworld. Once solved, the solution will look like this:

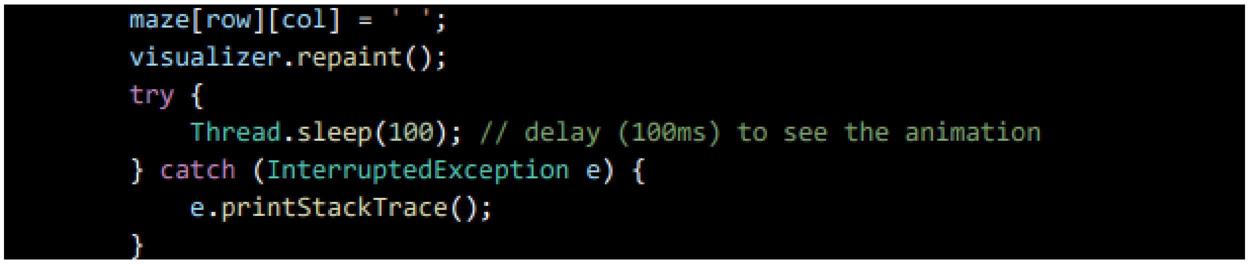


The yellow tiles represent the ‘path’. You have control over this path in the solveMaze method. To effectively control this visualization, you will want to specify ‘visited’ and ‘unvisited’ tiles. To ‘visit’ a tile (turn it yellow), write:



```
        maze[row][col] = '+';
        visualizer.repaint();
        try {
            Thread.sleep(100); // delay (100ms) to see the animation
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

Conversely, to ‘unvisit’ a tile (turn it back to white), write:



```
        maze[row][col] = ' ';
        visualizer.repaint();
        try {
            Thread.sleep(100); // delay (100ms) to see the animation
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

All spaces marked ‘+’ will be yellow, while all spaces marked ‘ ‘ are white (as we know from the text representation of the maze that we described above). Writing ‘Thread.sleep(100)’ provides a slight delay to allow for a smooth and slow representation of the algorithm running.

Additionally, your code should collect the necessary movements from start to finish (in reverse order) in the ArrayList of Strings ‘**path’** passed as an argument to the solveMaze method. So, for the 7x9 example maze we discussed previously, if we printed the contents of the ArrayList in reverse order, the contents would be:

right, down, down, left, down, down, right, right, right, right, right, right, down Which, as we notice, are the steps required to complete the maze.

Great! So now, how do we approach this problem recursively? **Consider the following**:

1. The solveMaze method starts at the tile ‘S’ (represented by the row and column of ‘S’ which is passed as an argument to the method).
2. If maze[row][column] == ‘E’ (indicating that we are at the end of the maze), this is a base case, and we can return true.
3. Otherwise, maze[row][column] is out of bounds, a wall, or previously visited (marked with a ‘+’), then return false (this is also a base-case).
4. Otherwise, we are neither at the end of the maze or at an invalid tile. Thus, we can mark the current position as ‘visited’ (with a ‘+’). We should use the code described above to allow for updating the visualization and changing maze[row][column] to ‘+’.
5. Now, we can try to move in all four possible directions. This is where the recursion comes into play. We should try moving **up**, **down**, **left**, and **right** by recursively calling the solveMaze method with adjusted arguments (think about how we might move ‘up’ for example. How would we adjust the arguments to solveMaze?)
6. If any of the recursive calls return true, we know that this tile is in the path to the exit. Thus, we do two things if that recursive call evaluates to true:
    a. We’ll want to add a String (think about what that will be) to path
    b. Return a boolean value (think about what)
7. If all recursive calls (up, down, left, and right) returned false, that means that this tile is not on the path to the exit. Thus, we can unmark the current position by changing ‘+’ to ‘ ‘ (and repainting the visual with the code described above). Then, we can return false to backtrack and explore other potential paths.

Again, you don’t need to solve the problem this way, but it is effective.

Provided Files

The following is a list of files provided to you for this assignment.

- Maze.java (reads the maze from a text file and holds the maze) **Do not alter this file**
- MazeVisualizer.java (for visualization) **Do not alter this file**
- maze1.txt, maze2.txt, maze3.txt, maze4.txt, maze5.txt **Do not alter these files**
- sample_solve.mp4 (to see a visual of how it should look)
- MazeSolver.java (write functionality for the solveMaze method in this file)

Marking Notes

Functional Specifications

- Does the program behave according to specification?
- Does the program produce the correct output and pass all tests?
- Does the code run on another machine (i.e., is anything hardcoded? It shouldn't be) - Does the code produce compilation or runtime errors?
- Does the program follow all stated rules and match specifications?

Non-Functional Specifications

- Are there comments throughout the code?
- Are variables within the methods given appropriate and meaningful names?
- Is the code clean and readable with proper indenting and white space?
- Is the code consistent regarding formatting and naming conventions?
- Submission errors (i.e., missing files, too many files, etc.) will receive a penalty of 5%.
- Including a 'package' line at the top of a file will receive a penalty of 5%.

** Remember you must do all the work on your own. Do not copy the work of another student. All submitted code will be run through similarity detection software.

Rules

- Please only submit the files specified below.
- Do not upload the .class files! A 5% penalty will be applied for this.
- Submit your assignment on time. Late submissions will receive a penalty of 10% per day up to three days.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope.
- Assignment files are not to be emailed to the instructor or TAs. They will not be marked if sent by email.

Files to Submit

- MazeSolver.java

Grading Criteria

Total Marks [11]

Passing custom tests (your code will be tested on custom mazes with only one path from start to finish) **[9.35]**

Following document specifications [1.65]

- Does your code comply with all 'non-functional specifications' discussed above?

Good luck, and have fun!