Assignment 4

Objectives

- Experience implementing the Stack ADT using an array
- Experience using Exceptions in Java
- Experience writing recursive methods

Objectives

This assignment requires you to implement a Stack interface using an array.

Although there is a tester provided for this assignment, it does not include a comprehensive set of sets for each method. You should add your own tests to test cases not considered.

Note: The automated grading of your assignment will include different and additional tests to those found in the A4Tester.java file.

After implementing the Array in A4Stack.java, you will use the stack to solve a text-based maze. The strategy will be to push all of the maze positions onto the stack. If a dead-end is reached, recently visited positions can be popped off the stack (equivalent to backtracking in the maze) until the last fork in the road, and a different route can be attempted. When the end location of the maze is reached, the stack is holding all of the positions traveled on the path from the start to finish.

Quick Start

1. Download all of the .java files found in the Resources > Assignments > a4 folder.
2. Read through the documentation provided in the `Stack.java` interface, there is a lot of very useful information that will help you with your implementation of each method in `A4Stack.java`.
3. Compile and run `A4Tester.java`. Work through implementing each method one at a time. Debug each method until all tests pass for that method before proceeding to the next method.
4. Once you have completed `A4Stack.java`, begin working on `MazeRunner.java`.
5. More detailed Maze instructions are found on the next page.
6. Here is a video explaining the process of solving a maze.

CRITICAL: You **must** name the methods in A4Stack.java and MazeRunner.java as specified in the documentation and used in `A4Tester.java` or you will receive a **zero grade.** Remember that all methods specified in an interface must be implemented, or your code will not compile correctly. Any compile or runtime errors will result in a **zero grade** (as if the tester crashes it will not be able to award you any points for any previous tests that may have passed).

Submission and Grading

Submit **A4Stack.java** and **MazeRunner.java** with your name and student ID at the top using conneX**.**

If you chose not to complete some of the methods required, you **must provide a stub for the incomplete method(s)** in order for our tester to compile. If you submit files that do not compile with our tester, you will receive a **zero grade** for the assignment. It is your responsibility to ensure you follow the specification and submit the correct files. Additionally, your code must **not** be written to specifically pass the test cases in the tester, instead, it must work on all valid inputs. We may change the input values when we run the tests and we will inspect your code for hard-coded solutions.

Be sure you submit your assignment, not just save a draft. **ALL late** and **incorrect** submissions will be given a **ZERO** grade. A reminder that it is OK to talk about your assignment with your classmates, but not to share code electronically or visually (on a display screen or paper). We will be using plagiarism detection software.

Solving the Maze

There are a number of different .java files you will use to solve a text-based maze. Before looking at the Java files, first we will look at the format of the maze input files:

Each input file should begin with 2 integers on the first line.

The first integer is the number of rows the maze contains, the second is the number of columns. The size of the maze in the input file to the right is 9 rows by 11 columns.

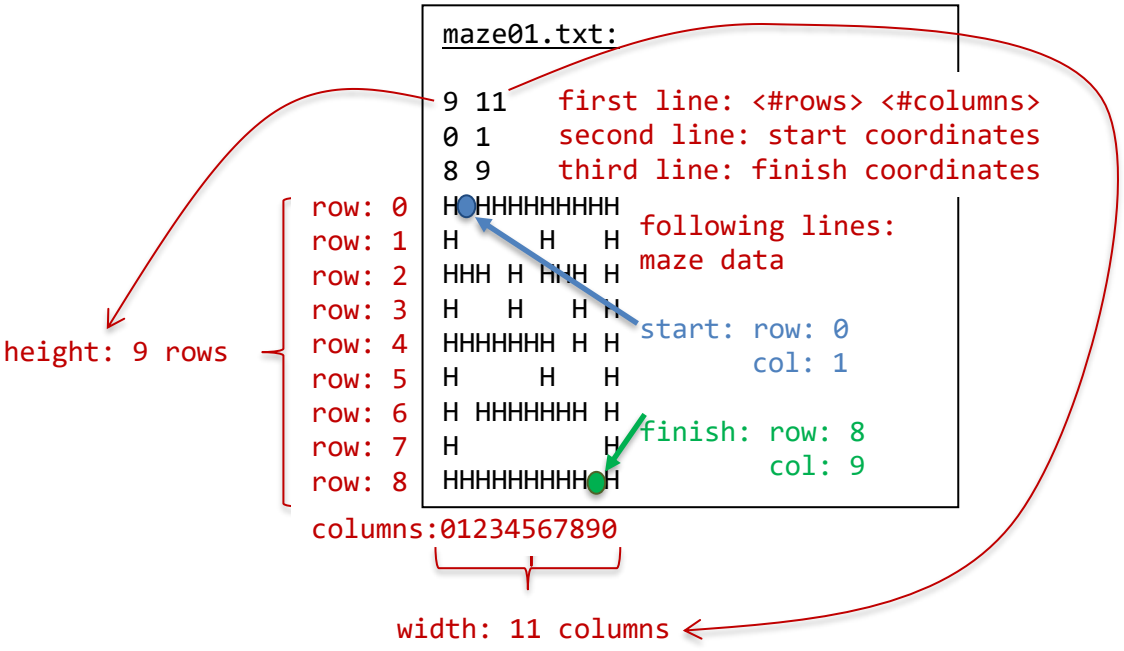
The second line has two integers representing the starting coordinates, which is where the maze begins. In the input file shown on the right, the starting coordinates are (0,1), representing row 0 and column 1.

The third line has two integers representing the finish coordinates. For the input file on the right, the finish coordinates are row 8 and col 9.

```
9 11
0 1
8 9
H HHHHHHHHH
H      H   H
HHH H HHH H
H   H   H H
HHHHHHH H H
H      H   H
H HHHHHHH H
H          H
HHHHHHHHH H
```

The remaining lines are maze data. The file shown on the right has 9 rows of maze data, each with 11 columns. H's are considered walls, spaces are considered open pathways.

The following diagram depicts the information explained above more visually:



You have been provided with a number of Java classes to help you write a maze solver.

The first class is `MazeLocation.java`. This class represents a position in the maze, which consists of a row and column. For example, we could make two `MazeLocation` variables to represent the start and finish of the maze shown above:

```
MazeLocation start = new MazeLocation(0, 1);
MazeLocation start = new MazeLocation(8, 9);
```

The next class is `Maze.java`. This class has three fields: the start and finish maze locations (described already), and a two-dimenstional character array called `mazeData`. Maze data is an array of characters representing the current state of the maze (which starts with H's as walls and spaces as open paths).

For example, `mazeData` for the sample maze would be the following:

	0	1	2	3	4	5	6	7	8	9	10
0	H		H	H	H	H	H	H	H	H	H
1	H						H				H
2	H	H	H		H		H	H	H		H
3	H				H				H		H
4	H	H	H	H	H	H	H		H		H
5	H						H				H
6	H		H	H	H	H	H	H	H		H
7	H										H
8	H	H	H	H	H	H	H	H	H		H

Remember, the MazeLocation class allows us to locate a position in the maze. For example a MazeLocation with fields **row:2** and **col:3** would be at the following position:

	0	1	2	3	4	5	6	7	8	9	10
0	H		H	H	H	H	H	H	H	H	H
1	H						H				H
2	H	H	H		H		H	H	H		H
3	H				H				H		H
4	H	H	H	H	H	H	H		H		H
5	H						H				H
6	H		H	H	H	H	H	H	H		H
7	H										H
8	H	H	H	H	H	H	H	H	H		H

The mazeData array is updated as the maze is explored. For example, when starting at MazeLocation (0, 1), an 'o' is written to the maze with the statement `mazeToSolve.setChar(0,1,'o');` (representing the fact that the location (0,1) has been visited):

	0	1	2	3	4	5	6	7	8	9	10
0	H		H	H	H	H	H	H	H	H	H
1	H						H				H
2	H	H	H		H		H	H	H		H
3	H				H				H		H
4	H	H	H	H	H	H	H		H		H
5	H						H				H
6	H		H	H	H	H	H	H	H		H
7	H										H
8	H	H	H	H	H	H	H	H	H		H

	0	1	2	3	4	5	6	7	8	9	10
0	H	**o**	H	H	H	H	H	H	H	H	H
1	H						H				H
2	H	H	H		H		H	H	H		H
3	H				H				H		H
4	H	H	H	H	H	H	H		H		H
5	H						H				H
6	H		H	H	H	H	H	H	H		H
7	H										H
8	H	H	H	H	H	H	H	H	H		H

The video outlines how you will continually update the mazeData array as you traverse through the maze, until (hopefully) finding a path to the finish location. The next page overviews a strategy to visit different locations until a solution is computed.

The `MazeRunner.java` file is what you will be modifying. You will be implementing the **`solve`** method, which is a recursive method that uses a stack to solve the maze.

NOTE: the **only** method you should modify is the **`solve`** method.

The general strategy to implement the **`solve`** method is the following:

1. Update mazeData array by writing an 'o' to the current location
2. Check if you are the finish location, if so, you have solved the maze!
3. If not, check if there is an open path in a neighbouring location (meaning you can progress that way through the maze), and if so, push that location onto the stack and visit it (there are 4 neighbouring locations we can try – the order you want to progress through the maze is up to you).
4. If none of the neighbouring locations lead to the finish (i.e., we're at a dead end) then pop off the stack so we can backtrack and attempt a different route (maybe we tried going down when we hit a fork in the road the first time, and found a dead end. This allows us to backtrack to that fork and go in a different direction). Write an 'x' every time we backtrack to keep track of the routes that resulted in a dead end.
5. If we have tried all of the options, the maze just isn't solveable.

The last file, `MazeTester.java`, allows you to test the different maze files by passing in a maze input file using the command line. After compiling, it is executed the following way:

```
java MazeTester <inputfile>
```

For example: `java MazeTester maze01.txt`

Good luck!