FA21 CSE 8B Homework 7:
Maze Solver

Due Date: Wednesday, November 17, 11:59 PM

Learning goals:

- Objects (and object-oriented thinking)
- Inheritance
- Exception Handling
- Assertions
- 2D Arrays and Loops
- Text File I/O

NOTE: This assignment should be completed INDIVIDUALLY. Pair programming is NOT allowed for this assignment.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 8B Coding Style Guidelines. These guidelines can also be found on Canvas. Please ensure to have *COMPLETE* file headers, class headers, and method headers, to use descriptive variable names and proper indentation, and to avoid using magic numbers.

Part 0: Getting started with the starter code (0 points)

1. Make sure there is no problem with your Java coding environment. If there are any problems, then review Assignment 1, or come to the office/lab hours before you start Assignment 4.
2. Download the starter code.

a. If you are working on your local machine, then you can download the starter code from Piazza → Resources → Homework → `Assignment7.zip`. Download the starter code to a directory of your choice, then extract the `zip` file. It should have the following structure:

```
+-- starter/
|     +-- MazeElement.java                  Edit this file
|     +-- MazeSolver.java                   Edit this file
|     +-- InvalidSymbolException.java       Edit this file
|     +-- UnsolvableMazeException.java      Edit this file
|     +-- Assignment7.java                  Edit this file
|     +-- input1                            Do NOT Edit
|     +-- input2                            Do NOT Edit
```

b. If you are working via UCSD Linux Cloud through your CSE 8B account, then use the commands below to copy the starter code to a new directory called `PA7`, to change your current directory to `PA7/starter`, and to print the files in the `starter` directory:

```
$ cp -r ~/../public/assignments/PA7 ~
$ cd ~/PA7/starter
$ ls
```

Overview

In this assignment you will be implementing tools used to read a maze from a file, solve a maze, and write a maze to file. Take a look at all files to understand them before you start coding. Make sure you follow the instructions below carefully.

In the starter code, there are two exceptions you will implement yourself. Both exceptions are used when there is an anomaly when handling a maze. Additionally, the class `MazeElement` is used to represent a single element in the maze, and the class `MazeSolver` encapsulates a 2D array of `MazeElement` and includes methods to read, solve, and write a maze.

Finally, you have a class called `Assignment7` that contains the unit tests and main method to execute the programming assignment.

IMPLEMENTATION NOTE: you CANNOT change any data field or method signature in the starter code. Additionally, you CANNOT change any methods that are fully implemented for you (those methods without TODO).

Part 1: UnsolvableMazeException.java (4 points)

First, you need to implement the class `UnsolvableMazeException`, which extends the Java class `RuntimeException`. An instance of `UnsolvableMazeException` is thrown when the maze is unsolvable (we will discuss this later). You only need to implement the no-arg constructor by initializing the `message` member variable in its superclass with the constant string. To learn more about the `RuntimeException` class, please take a look at the official doc here.

Question for you to think about: Is `UnsolvableMazeException` a checked exception or unchecked exception?

Part 2: InvalidSymbolException.java (4 points)

You need to implement the class called `InvalidSymbolException`. Unlike the previous class, this class extends `Exception` and is thrown when you encounter an invalid symbol when loading the maze from a file (again, we will discuss this later). You need to implement the single-arg constructor by initializing the message member variables in its superclass. The message format is given as a constant string and you need to format the parameter `symbol` (Hint: use `String.format` method). To learn more about the `Exception` class, please take a look at the official doc here.

Note: you may assume the parameter `symbol` is not `null.`

Question for you to think about: Is `InvalidSymbolException` a checked exception or unchecked exception?

Part 3: MazeElement.java (8 points)

This class encapsulates a String `symbol` which represents a maze entry in the 2D maze array. There are many methods that have already been implemented for you. Please look at the provided source code and understand what each method does, since you might leverage them. A maze element can have one of the three following symbols:

1. "X": A wall that cannot be passed through.
2. "-": An empty position that can be "stepped" on.
3. "*": A symbol representing part of the solution path to get out of the maze.

The `MazeElement` class has the following constructor for you to implement:

```
public MazeElement(String symbol) throws InvalidSymbolException
```

Before initialization, it checks whether the parameter `symbol` is one of the valid symbols discussed above. If the parameter `symbol` is invalid (i.e., not of the three symbols "X", "-", or "*"), then it throws an `InvalidSymbolException` object; otherwise, this constructor initializes the member variable `symbol` with the parameter `symbol`.

Note: you may assume the parameter `symbol` is not `null`.

Question for you to think about: What happens if the method signature does not have "`throws InvalidSymbolException`"?

Part 4: MazeSolver.java (64 points)

This class encapsulates a 2D `MazeElement` array called `maze` representing our maze, which is either yet to be solved or already solved. Do NOT change the provided setter and getter methods, and no-arg constructor. You need to implement the following four methods:

1. `public MazeSolver(String fileToRead)`

This constructor will read the maze file and parse the maze to a 2D array of `MazeElement` objects. The input `fileToRead` is the filename of the maze you want to read. Apply what you've learned in lecture to read the input file (hint: use Scanner). The file contains the information of an input maze (see the example files `input1` and `input2`).

The format of the file follows as:

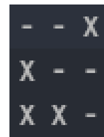
a. The first line of the file will **always** contain two numbers. The first will represent the number of rows in the maze, while the second represents the number of columns. These numbers need not necessarily be equal. **You may assume both will be positive integers.**
b. The rest of the file is the input maze:
    i. Each position is represented as the String symbol described in Part 3.
    ii. Between every column, there will be a single space.
    iii. Between every row, there will be a newline character (that's why they are on different lines).
    iv. No blank lines in the input file. No extra spaces between rows and columns. No extra spaces before and after each line. **A newline character will be added to the end of the last row**.

Read the file and initialize member variable `maze`. Close the scanner associated with the input file after the reading is finished (Hint: use try-with-resource). When the file is not found (`FileNotFoundException`) or you encounter an invalid symbol (`InvalidSymbolException`), you need to properly handle it with a try-catch clause. In either situation, you must print out the error message once you catch it (Hint: to get the error message use the inherited method getMessage() from the `Exception` class).

Note: you may assume `fileToRead` is not `null`. To make the problem easier, we're limiting the possible input mazes as follows: you may assume at any "step" in the maze, the option of going both down OR right is not a possibility. For example, the following maze will not be a possible input because at the step circled, the next step could be down or right.

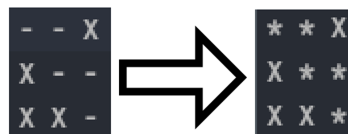
2. `public void solveMaze()`

Once you construct a `MazeSolver` object with the above constructor, a maze is loaded and stored in the member variable `maze`. The starting position (i.e., entrance) of all input mazes is at the **top left corner** of the maze. You need to find a path through the maze to the ending position (i.e., exit), which will always be at the **bottom right corner** of the maze. To make the problem even easier, you can **only** step **right** OR **down** (i.e., stepping left or up is not allowed). As shown in the following example maze, all the X's (walls that cannot be passed through) will block you, but you can move to any of the -'s.

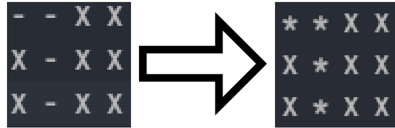


Your job is to trace out the path taken to solve the maze. Modify `maze` **in-place** (i.e., do NOT create a new maze 2D array while tracing your path). You're required to record the position of each step you take on your way using an asterisk (*****) to replace dash (-).

There are two possibilities when you try to solve a maze. The first possibility is that the maze can be solved, meaning you will be able to find a path starting from the top left corner and reaching the bottom right corner. For example, if you have the following maze on the left, it will become the maze on the right after this method completes.



The second possibility is the maze cannot be solved. In this case, you will will to convert the "-" to "*" until you hit the dead end. For example, the following will be the result of attempting to solve an unsolvable maze.



When this happens, the method will throw an `UnsolvableMazeException` object once it reaches the dead end.

Note: you may assume the member variable `maze` is not `null`. You will not be asked to solve a solved or partially solved maze.

3. `public void writeMaze(String fileToWrite)`

   This method writes the member variable `maze` to a file. The `fileToWrite` is the name of the file you want to write the maze to. Apply what you've learned in lecture to write the output file (Hint: use `PrintWriter`). The output file should follow the same format as the input file described in 1.a and 1.b above.

   You also need to handle `FileNotFoundException`. For `PrinterWriter`, a new file will be created if the input file name does not exist; otherwise, the existing file will be truncated to size zero. However, truncating a read-only file is not allowed, meaning you still need to handle the `FileNotFoundException` caused by it. Print the message while handling the exception. Close the PrintWriter you created (Hint: use try-with-resource).

   Note: you may assume `fileToWrite` is not `null` and the member variable `maze` is not `null`.

   Hint: Useful in testing this is to create a file and convert it to read-only using the following commands:

```
$ touch readOnly
$ chmod 444 readOnly
```

4. `public static boolean mazeMatch(MazeElement[][] maze1, MazeElement[][] maze2)`

This method takes in two 2D arrays of `MazeElement` references and compares them. The function returns a boolean value of `true` if two arrays are deep equal to each other, and `false` otherwise.

Note: You may assume both references are not `null`.

Part 5: Assignment7.java (10 Points)

Note: Please read below carefully, as the instructions for how to write unit tests have changed.

Inside class `Assignment7`, three testers have already been written for you to test the functionality of some methods. However, they all use assertions to check correctness rather than the boolean comparison used in previous assignments. **To get full credit, please create *at least five* additional test cases that test all your different methods using assertions, not using a boolean return value.** This also means all testers, including `unitTests()`, now have a `void` return value. Please carefully look at the three testers provided to understand how to write similar testers using assertion.

To enable assertion in Java and run your testers, do the following:

```
$ javac *.java
$ java -ea Assignment7
```

Submission

VERY IMPORTANT: Please follow the instructions below carefully and make the exact submission format.

1. Go to Gradescope via Canvas and click on `PA7`.
2. Click the DRAG & DROP section and directly select the required files (**`Assignment7.java`**, **`UnsolvableMazeException.java,`**

InvalidSymbolException.java, **MazeElement.java**, **MazeSolver.java**,). Drag & drop is fine. **Please make sure you don't submit a zip. Make sure the filenames are correct.**

3. **You can resubmit an unlimited number of times before the due date.** Your score will depend on your last submission, even if your former submissions have a higher score.
4. The autograder is for grading your uploaded files automatically. Make sure your code can compile on Gradescope. You will receive 0 out of 80 on Autograder if your code does not compile on Gradescope.

   NOTE: The Gradescope Autograder you see is a minimal autograder. For this particular assignment, it will only show the compilation results and the results of a few testers. After the assignment deadline, a thorough Autograder will be used to determine the final grade of the assignment. **Thus, to ensure that you would receive full points from the thorough Autograder, it is your job to extensively test your code for correctness via `unitTests`.**