Create a new Java project  
called ***Maze***

**Purpose:**

The purpose of this project is for you to demonstrate your understanding and use of a variety of data structures in a program with a complex algorithm. This is a capstone application in which you are expected to apply most of the various techniques learned in this course.

**Specifications:**

The objective of this project is for you to develop and code an algorithm that will simulate a ***Mouse*** traversing a ***Maze***. The program must display the final route through the ***Maze*** (see the example at the end of the specifications). No dead-ends or re-loops should be shown. The ***Maze*** is read in from a file chosen by the user. After loading the ***Maze***, display it for the user to select the entry point. There may be multiple entry points, not all of which result in reaching an exit.

Begin the program by prompting the user to enter the ***Row*** and ***Column*** of the point where the ***Mouse*** will enter the ***Maze***. The entry point must be an open ***Cell***; you must not allow the user to crash through a wall. Should the user select a ***Cell*** that is a wall, display an error message and return to the selection point. If an open ***Cell*** is selected, the ***Mouse*** begins at this starting location and attempts to find an exit from the ***Maze***. Not all entry points will lead to a valid exit.

It is recommended that you use either a two (2) dimensional array or a one (1) dimensional array that simulates a two dimensional array to hold the ***Maze*** itself. Each element (***Cell***) of the array can be a 1 (closed) or 0 (open).  A closed ***Cell*** is a square that the ***Mouse*** cannot enter (a wall).  An open square is a ***Cell*** that the ***Mouse*** can enter (e.g. a passage). The ***Mouse*** traverses the ***Maze*** one ***Cell*** at a time. You must use a ***Stack*** for recording the traversal sequence and decision points and an ***ArrayList***, ***LinkedList***, ***Vector***, or other container for recording the final ***Path***.

When the ***Mouse*** enters a ***Cell***, it may determine the type of the ***Cell***. The type is one of the following:

***Entrance***- if the ***Cell*** is on the outer edge of the ***Maze***, the ***Mouse*** enters the solution and traversal begins – this can occur only once.

***Passage*** - if one and only one of its adjoining ***Cells*** (excluding the previously occupied ***Cell***) is an open ***Cell***.

***Intersection*** - if two or more of the adjoining ***Cells*** (excluding the previously occupied ***Cell***) are open ***Cells***.  This requires a decision as to which ***Path*** to take. These need to be recorded in the ***Stack***.

***Dead End*** - if none of the adjoining ***Cells*** (excluding the previously occupied ***Cell***) is an open ***Cell***.  The ***Mouse*** must *backtrack*. The ***Stack*** contains the traversal sequence and decision point(s) to which the ***Mouse*** can return and resume traversal in a different direction.

**Note:** If backing out results in the return to the ***Entrance***, that entry point to the ***Maze*** has no solution and an error message should be displayed and a restart enabled.

***Exit***- if the ***Cell*** is on the outer edge of the ***Maze***, the ***Mouse*** exits and the solution is displayed. There may be more than one exit from the ***Maze***; only one needs to be found.

The ***Maze*** definition must be read from a file with the following format:

* Line 1 – contains the number of rows and columns of the maze.
* Lines 2 to n – contain one row for each row of the maze. Each row contains 1's and 0's separated by spaces that define the maze. The spaces between the 1’s and 0’s allow the values to be read into the maze as integers. It may also be read as characters but allowance has to be made for the intervening spaces.
  + 0 represents an open cell
  + 1 represents a closed cell

Programming Requirements:

1. Design an overall algorithm for traversal.
2. Have a strategy to handle each move outcome.
3. Use only the five standard structured techniques.
4. Project must make full use of classes and methods.
5. Use the necessary abstract data types.
6. ***main*** logic must make full use of functions.
7. Use good programming style and object oriented techniques.
8. Properly use the collection structures.
9. Clear, concise and accurate user messaging.
10. Good file handling techniques.
11. Good class and program documentation.

Example displays:

1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1

0 0 0 0 1 1 0 1 1 1 0 0 0 0 0 1 1 0 1 1

1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1

1 1 0 1 0 1 1 0 1 1 1 1 0 1 0 1 1 0 1 1

1 1 1 1 OR x x x 1 1 x x x 1 1

1 1 1 1 1 1 1 x 1 1 x 1 x x x

1 1 1 1 1 1 1 1 x x x x 1 1 1 1

1 1 0 1 1 1 0 0 0 1 1 1 0 1 1 1 0 0 0 1

1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1

1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1

There is a maze definition file that can be used for testing your program (mazetest.txt). It has several paths that can be used to test your traversal process.

There are 5 maze definition files to be used for grading the sum of which is worth 62.5% of the grade. The fifth and final maze (maze5.txt) is required to achieve the full 200 points; it is depicted below:



Submit the project as a .zip file