Algorithm:

- Create a global boolean vis[][] 2D array of size 20 by 20 indicating if a point is visited or not
- Create a global boolean check[][] 2D array of size 20 by 20 indicating if a point can lead to the exit or not
- Create a global char maze[][] 2D array of size 20 by 20 that stores the maze generated or loaded.
- Create a global variable int maxR that stores the user entered or loaded row number
- Create a global variable int maxC that stores the user entered or loaded column number
- Create a global variable char barrierChar that stores the character that represents a barrier
- Create a global variable char exitChar that stores the character that represents the exit
- Create a global variable char startChar that stores the character that represents the mouse's starting position
- Create a global variable char openChar that stores the character that represents an open block.

Generating the maze:

- Method name: MazeGen
- Description: randomly generate a maze with user imputed dimensions(rows&columns).
- No parameters
- Returns void, because the maze is already stored in the global maze array
- Method:
    - Do while loop to prompt the user for row and column until the given dimension is within the size range (2x3/3x2 to 20x20)
    - All char of the maze are stored in the global char 2D array maze[][]
    - Set the first row, last row, first column, and last column to 'B', indicating a border.
    - Randomly generate a number 1-4 inclusive determine the placement of exit
        - 1 indicating the exit is on the first row

        - 2 indicating the exit is on the last row
        - 3 indicating the exit is on the first column
        - 4 indicating the exit is on the last column
    - special cases: if the maze is a 2x3 or 3x2, the exit, must be on the side with length 3.
    - According to the number generated above, randomly generate a number from 0 to (row-1) or (column-1) exclusive (so the corners are not included).
    - Set the position generated on the border above to ‘X’ indicating exit
    - Do while loop to randomly generate a mouse starting position:
        - if the position generated is the 4 corners or ‘X’, the do-while loop keeps running.
    - Set the mouse position generated above to ‘S’
    - Use a nested for loop to loop through the middle section (excluding the borders):
        - If the position the nested for loop is on in the 2D char array maze[][] is ‘S’
            - Continue; //because this block is already assigned
        - Randomly generate a number from 1-100 inclusive
        - If the number is less than 76, then set that position to ‘O’ indicating an open position (75% chance of creating an open path)
        - Otherwise, set it to ‘B’ indicating a barrier.

Finding and marking all paths:

- Method name: findPath
- Description: Recursive method to find all paths that lead the mouse to the exit. When calling this method, posR and posC parameters should be the position of the mouse
- @param posR - an integer indicating the row# the mouse is currently on
- @param posC - an integer indicating the column# the mouse is currently on
- @param exitChar - char indicating the character of the exit
- @param barrierChar - char indicating the character of barrier blocks
- @return boolean - indicating if the block with posX and posY can lead to the exit.

- Method:
    - Create a boolean called checkExit = false, indicating whether the current block can lead to exit or not.
    - If vis[posR][posC]:
        - If posR+1 is within the user entered dimension:
            - checkExit = checkExit || check[posR+1][posC];
        - If posR-1 is within the user entered dimension:
            - checkExit = checkExit || check[posR-1][posC];
        - If posC+1 is within the user entered dimension:
            - checkExit = checkExit || check[posR][posC+1];
        - If posC-1 is within the user entered dimension:
            - checkExit = checkExit || check[posR][posC-1];
        - check[posR][posC] = checkExit;
        - Return checkExit;
        - All the if statements in here are to make sure that in a situation like an example below. From the simulation of my algorithm, I realized position 2,2 will be marked visited but it will be marked false in the check 2d array. This means 1 of the paths is not found. Therefore, to solve this problem, I added the above steps so that even if a position is already visited, it will still check the surrounding positions for the ability to lead to an exit so all paths will be included

	0	1	2	3	4
0	B	E	B	B	B
1	B	O	O	O	B
2	B	B	O	O	B
3	B	B	B	S	B
4	B	B	B	B	B

    - If the current position is not visited, set vis[posR][posC] to true.

- If maze[posR][posC] equals to exitChar:
    - Don't set check[posR][posC] to true because we are not marking the exit blue, only the paths
    - return true;
- If maze[posR][posC] equals to barrierChar:
    - return false;
- Now, I'm going to do all the moves possible
- If posR+1 is within the user entered dimension:
    - checkExit = checkExit || findPath(posR+1, posC, exitChar, barrierChar);
- If posR-1 is within the user entered dimension:
    - checkExit = checkExit || findPath(posR-1, posC, exitChar, barrierChar);
- If posC+1 is within the user entered dimension:
    - checkExit = checkExit || findPath(posR, posC+1, exitChar, barrierChar);
- If posC-1 is within the user entered dimension:
    - checkExit = checkExit || findPath(posR+1, posC-1, exitChar, barrierChar);
- The || logical operator is used because as long as one of the 4 adjacent positions can lead to the exit, the current position will also lead to the exit.
- check[posR][posC] = checkExit; //set the block mouse is currently on to checkExit so when doing later execution, we know if the block should be marked blue or not.
- Return checkExit;

How to load maze from file:

- Method name: MazeLoad
- Description: load maze from a file user entered, procedure method
- No parameters
- Returns void - because the chars are already stored in the global 2D char array maze[][]
- Method:
    - Prompt user for the file to name to load the maze from

- Scan the first number and store it in the global variable maxR
- Scan the second number and store it in the global variable maxC
- Scan the third char and store it in the global variable barrierChar
- Scan the fourth char and store it in the global variable openChar
- Scan the fifth char and store it in the global variable startChar
- Scan the sixth char and store it in the global variable exitChar
- For loop from i=0 to i<maxR:
    - Scan the String
    - For loop through the String from j=0 to j<String.length:
        - Store each char in global variable maze[i][j]