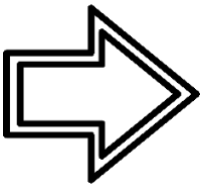
Assignment

Overview

For this assignment you are going to create and solve the path finding for a maze. You will create the maze yourself in a 2D array and Recursion to step through the maze finding the end. The exit and entrance will be marked by -1, walls are marked with 1 and the open spots marked with a 0. I have provided an algorithm to search through the maze. You can research others to use as well. The program should work with any size maze. I have given you some to start with but you can definitely make your own.



Directions

To Get Started:

1. Create a new project in IntelliJ(or your preferred IDE) called **YOUR_MCC_Name-FinalProject**
2. Then follow the directions below to create a **Maze.java**, **MazeSolver.java** and **MazeDriver.java** files.

Part 1: Maze Object

Reference the following UML diagram to build an object that will represent the maze on the computer.

Maze.java
-int [][] map -int entranceRow
<<constructor>> Maze(int[][] m) -findEntranceRow() : int +getEntranceRow() : int +getExitColumn() : int +getCell(int r, int c) : int +setCell(int r, int c, int val) : void +isOpenSpace(int r, int c) : boolean +printMaze()

Discussion on Maze

A maze is defined by a 2 dimensional int array. Walls in the maze are denoted by the number 1. Places that can be traversed are denoted by the number 0. The entrance and exit are denoted by number -1. The entrance is on the left(column 0), the exit is on the right(rowlength -1).

Example of a maze:

```
 111111111111
 100010000001
-101010111101
 111010000101
 10000111010-1
 111101010101
 100101010101
 110101010101
 100000000101
 111111011101
 100000010001
 111111111111
```

Notice the Maze itself is surrounded by walls except for the entrance and exit.

Constructors

The maze will have one constructor that accepts an int[][] that assigns to the map. We also need to find the starting row for the entrance. To do this we will call upon the findEntranceRow() and assign it to the entranceRow field.

Getters/Setters

getCell/setCell will get or set the information within the int[][] array. The set will be used a lot as we set the cell to a different value as we search.

getEntranceRow() reports back its value.

getExitColumn() reports back the last column on the right. Use row [0] and report its length. We are going to assume our 2D array is rectangular and we can't hardcode the length/size of the array, but rather use the lengths for any size maze.

Other Methods:

isOpenSpace(int r, int c) will be our helper method to make sure we are not out of range of the array AND that we aren't at a wall. This is a boolean method. If false is returned we know we can't go to the space. This will be used in our recursive method to help solve the maze.

- So that means you need to check if the values r & c sent are within the range of the 2D array and if it contains the value '0'. Return true
- otherwise return false.

findEntranceRow() will search through the first column of the maze and report back the row that the entrance is at. *Note: this is a private method and only used inside the class, the getter will be used to report it back*

printMaze()

When printing the maze you should use '*' (asterisks) as the walls, the cells visited/final result should be marked with '@' the exit should be a [ ]. Note that you should use if or switch to match the -1, 0 and 1 values saved in the array. Think about adding extra spaces around characters as you print. This will also be in a nested loop structure.

```
************************
**   **      **
[] ** ** ******** **
****** **    ** **
```

```
**    ****** ** []
******** ** ** ** **
**  ** ** ** ** **
**** ** ** ** ** **
**       ** **
*********** ****** **
**      **   **
**********************
```

Part 2: Solving the Maze

Reference the following UML diagram to build the MazeSolver.java file.

MazeSolver.java
-Maze maze
<<constructor>> MazeSolver(Maze m) +solveMaze(int r, int c) : boolean

Discussion of MazeSolver

The MazeSolver will hold a single Maze object. This will have our recursive method to solve the maze

Constructor

Takes on a simple Maze object and assigns it to the field

Maze Solver Recursion

solveMaze(int r, int c) is the most important method here that uses **Recursion** to find the end of the maze. The method accepts two ints representing the current location in the maze. For the first call upon the method the entrance of the maze will be passed as the current location. This method attempts to locate the exit marking an '@' in each

square that has been reached. So as you traverse through blank spaces you need to mark/change the cell. Note: the '@' comes from the print maze, since our maze is in int[][] format pick another number to fill in the cell that you can later represent as an '@'

Here is the basic algorithm you can follow:

```
solveMaze(int r, int c)
    First use the isOpenSpace method on the maze, then nested inside
        if the current cell is the exit, then return true <-- this is our base case
              What is the exit?  well an exit is -1 BUT so is the entrance...
                  which means you need to make sure you are on the exit column

        otherwise:
        mark the space with another number that will represent the path taken
              (for example number 2)
        if not the end of the path then recursive call all below:
        Check the location above us if is a valid spot and not visited yet
            solveMaze(cur X, curY - 1)
            return true
        Check the location left of us if is a valid spot and not visited yet
             solveMaze(curX-1, curY
             return true
        Check the location right of us if is a valid spot and not visited yet
             solveMaze(curX+1, curY)
             return true
        Check the location down from us if is a valid spot and not visited yet
             solveMaze(curX, curY+1)
             return true

        If none of the above recursive calls work we need to set the space back to a
           '0' value as it is not the path taken to find the exit.

    return false if the isOpenSpace check doesn't work
```

Part 3: MazeDriver

Create a class MazeDriver.java that contains the main() method. This will have a MazeSolver object to which you send it an int array to create a maze. Give the user 3 options of mazes to solve.

```
***Welcome to the Maze Project***
Select a maze to solve:
1. Maze 1
2. Maze 2
3. Maze 3
Choice:
```

Use the following as the mazes OR create your own:

```
int[][] m1 = {{1,1,1,1,1,1,1,1,1,1,1,1},
              {1,0,0,0,1,0,0,0,0,0,0,1},
              {-1,0,1,0,1,0,1,1,1,1,0,1},
              {1,1,1,0,1,0,0,0,0,1,0,1},
              {1,0,0,0,0,1,1,1,0,1,0,-1},
              {1,1,1,1,0,1,0,1,0,1,0,1},
              {1,0,0,1,0,1,0,1,0,1,0,1},
              {1,1,0,1,0,1,0,1,0,1,0,1},
              {1,0,0,0,0,0,0,0,0,1,0,1},
              {1,1,1,1,1,1,0,1,1,1,0,1},
              {1,0,0,0,0,0,0,1,0,0,0,1},
              {1,1,1,1,1,1,1,1,1,1,1,1}};


int[][] m2 = {{1,1,1,1,1,1,1,1,1,1,1,1},
       {1,0,0,0,1,0,0,0,1,1,0,-1},
       {1,0,1,0,0,0,1,0,0,0,0,1},
       {1,0,1,1,1,1,0,1,0,1,0,1},
       {1,0,0,0,1,1,0,0,0,0,0,1},
       {1,1,1,0,1,1,1,1,0,1,0,1},
       {-1,0,0,0,0,0,0,0,0,0,1,1},
       {1,1,1,1,1,1,1,1,1,1,1,1}};


int[][] m3 = {{1,1,1,1,1,1,1,1,1},
       {1,0,1,0,1,0,0,0,1},
       {1,0,0,0,1,0,1,1,1},
       {1,1,1,0,1,0,1,0,-1},
       {-1,0,0,0,0,0,1,0,1},
       {1,1,0,1,0,1,1,0,1},
       {1,0,0,1,0,1,0,0,1},
       {1,1,0,1,0,1,0,0,1},
       {1,1,1,1,1,1,1,1,1}};
```

Then you will call the MazeSolver's solveMaze() method that is boolean and use to print out if the maze is solved or unsolvable. Use a condition that if it is solvable, you print out the maze showing the path.

Example Output

```
*** Welcome to the Maze Project ***
Select a maze to solve:
1. Maze 1
2. Maze 2
3. Maze 3
Choice: 1
*** Starting Maze ***
************************
**   **      **
[] ** ** ******** **
****** **    ** **
**    ****** ** []
******** ** ** ** **
**  ** ** ** ** **
**** ** ** ** ** **
**        ** **
************ ****** **
**      **   **
************************

Maze Solved:
************************
**@@@@@@**@@@@@@@@@@@@**
@@@@**@@**@@********@@**
******@@**@@@@@@@@**@@**
**  @@@@******@@**@@[]
********@@** **@@** **
**  **@@** **@@** **
**** **@@** **@@** **
**   @@@@@@@@@@** **
************ ****** **
**      **   **
************************



*** Welcome to the Maze Project ***
Select a maze to solve:
1. Maze 1
2. Maze 2
3. Maze 3
Choice: 3
```

```
*** Starting Maze ***
******************
** ** **   **
**   ** ******
****** ** ** []
[]     ** **
**** ** **** **
**  ** **  **
**** ** **  **
******************

Maze unsolvable
```

Submission

1. Compress the IntelliJ project folder and submit it to this assignment.

   to compress: On Windows, right click -> send to -> compressed .zip file

   on Mac, right-click -> Compress

Hints/Tips (Before Submitting):

- *This project is open to interpretation. You can add extra methods or change up some of the coding as you would like. What I have provided to you is a starting point IF YOU NEED IT.*
- *Don't forget to have a header at the top of your file and include a Resource statement.*
  - ****NOTE you need this for EACH java file****
- *Note that we are getting input for the numbers 1-3 on the Driver file, you should use data validation to make sure that we get the proper input and don't have any exceptions thrown.*
- *Use comments as needed. Include comments for methods and what is going on in each method.*
- *Follow all Java Styling Guides as covered*
- *Ask me questions early and often as needed, this is a little more challenging of a program to make.*