Program #4

Due: **11/14/2017 at 10:00 PM**
Late Collections: **11/15/2017 at 10:00 PM**
11/16/2017 at 10:00 PM
Program Files (1): **MazeSolver.java**

In this program we will implement a recursive algorithm to solve a maze. Let's take a step back and think about a way to solve a maze that will always reach a solution.

1. Begin at the start and follow the available path
2. When you reach an intersection select one of the paths to follow
3. Follow that path.
4. If the path dead ends return to the last intersection and select a different path
5. If you reach another intersection before the path dead ends, select a direction and follow it using the same logic as in step 4.

As you can see, this algorithm will explore every path that is available in the maze. This kind of algorithm is known as Depth First Search (DFS) and is commonly used in programming. DFS is one example of an algorithm that visits each node in a connected graph, or in our case each cell of the maze. In particular, DFS explores every possible path until it finds a solution. Thus, if there is a path from the start to the end, a DFS algorithm will eventually find it (assuming there are no infinite paths).

For this assignment, we provide you with two classes that can be downloaded from CourseSite. The Cell class represents a location inside the Maze. The Maze class is a collection of cells that are connected to each other. Descriptions of the relevant methods are provided after the project requirements.

Project Requirements:
You need to implement a **MazeSolver** class that has the following methods. Note, that these methods are all **static**. Although you only need to write four methods, this assignment is conceptually difficult, and hard to get functioning properly. Please start early!

+findPath(currentMaze:Maze):ArrayList<Cell> – This method solves the maze passed in as a parameter. It uses the recursive helper method findPath(currentMaze:Maze, current:Cell, path:ArrayList<Cell>) to recursively find the solution to the maze. It returns an ArrayList of Cells, one for each step from start to finish through the maze.

-findPath(currentMaze:Maze, current:Cell, path:ArrayList<Cell>):ArrayList<Cell> – This method recursively finds a path from the start of the **currentMaze** to its end that goes through the **current** Cell. The **path** is an ArrayList of the sequence of cells that was followed to get from the start of the maze to the **current** cell (i.e., the path explored so far). In order to avoid paths that are longer than needed, the algorithm should avoid revisiting cells already in this **path**. The algorithm should return null if there is no path from **current** to the end that only goes through each Cell at most once. Otherwise, it should return the complete path from the start of the maze to the end as a sequence of Cells in an ArrayList.

You must implement this as a recursive algorithm. You will have to decide what the base case(s) and recursive case(s) are. When making your recursive call(s), you will have to decide which

parameters are unchanged and which stay the same, keeping in mind their descriptions above (also see the Hint below). In order to explore all paths through neighbors that have not yet been visited, you will want to make use of Maze's **getNeighbors** method (see below).

+printSolvedMaze(m:Maze, solution:ArrayList<Cell>):void – This method prints out the solution to the maze. It will get a character array for the maze display from the Maze class (see below for details of Maze's methods). It will then add a '.' character in each cell that is part of the **solution**. It will also add a '.' in the connector spaces between cells in this solution. After the solution has been added to the display it will print the entire display to the screen. There are methods in Maze that can help you do this. Note, in order to display walls between cells, the output will include a 2x2 set of characters for each cell. See Maze's **getMazeDisplay** method below for details.

+main(args:String[]):void - Your main method should allow the user to provide an integer maze id as a command-line argument. If no command-line arguments are given, use **1** (the number one) as the maze id. If two or more command-line arguments are given, print a helpful error message and exit. Note, the program must terminate gracefully if the user gives a command-line argument that cannot be converted to an integer. Assuming valid input, the program should generate a 20x6 maze with the specified maze id (note, 20x6 means width=20 and height=6), print the maze, solve it, and then print the maze again with the solution drawn in using '.' characters. See the description of the Maze constructor below for information on how to generate a maze.

Hint:
Remember that although Java is pass-by-value, object variables are references, and thus any changes to the objects affect the actual parameter in the caller. The best way to copy an ArrayList is to use the version of the ArrayList constructor that takes a Collection as a parameter and copies the elements into a new ArrayList. This works because ArrayList implements the Collection interface. This creates a new ArrayList consisting of the same objects as the original (called a shallow copy in Java), but as long as you are not changing the data in your Cells, this will work.

Provided Classes:
You will need to make use of the following classes and methods which are provided for you. Do not modify these classes! Your programs will be tested using the versions we give, and any custom modifications may cause your program to break when we test them.

Maze	A maze that consists of a set of cells
	You do not need to know the fields.
+Maze(width:int, height:int, mazeId:int)	Creates a maze of size **width** x **height**, initializes the cell objects, and creates the maze with the given **mazeId**. When given a positive mazeId, it will always draw the same maze. This allows you to test the effect of changes to your program on the same maze. The constructor will only generate mazes that have a solution. You will not have a case where you get an unsolvable maze.
+getStartCell():Cell	Returns the start Cell of the maze
+getEndCell():Cell	Returns the end Cell of the maze

+getNeighbors(currentCell:Cell): ArrayList<Cell>	Returns a list of all the cells that are connected to **currentCell**. If there is a wall between **currentCell** and its neighbor, it is not added to this collection.
+getMazeDisplay():char[][]	This method creates the 2D character array that represents the display of the maze. Because you can only draw one character at a position in the output but you need to be able to draw walls between cells, it draws a larger version of the maze to convey all information. This is why when you specify a 20x6 maze, for example, the output is actually 41x13. This array is organized so the column is specified first, and then the row, e.g., a[col][row].
+printMaze():void	Prints the maze object by calling **getMazeDisplay** and passing it to the other **printMaze** method.
+printMaze(mazeDisplay: char[][]):void	Prints the 2D character array **mazeDisplay** to the terminal.

Cell	A location in the maze. *You do not need to know the fields.*
+Cell(col:int, row:int)	Constructs the Cell and assigns it's (col, row) coordinate pair.
+getRow():int	Returns the row of this Cell.
+getCol():int	Returns the column of this Cell.
+getDisplayRow():int	Returns the row location in a character display where both cells and walls are printed as characters.
+getDisplayCol():int	Returns the column location in a character display where both cells and walls are printed as characters.
+equals(o:Object):boolean	Compares two cells and determines if they are the same cell. Two cells are equal if they have the same row and the same column.

Submission:
At a minimum provide a Javadoc comment explaining what each method does, and another one that summarizes the class. Include additional comments for lines that are especially complicated. At the top of the program include a comment with the following form:

```
/*
CSE 17
Your name
Your Lehigh e-mail
[Assisted by: tutor-name (tutor-email)]    if you used a tutor
Program #4      DEADLINE: November 14, 2017
Program Description: Maze solver
*/
```

Once the program compiles, runs, and has been tested to your satisfaction, upload **MazeSolver.java** to Course Site (you do not need to submit **Maze.java** or **Cell.java)**. To do so, click on the name of the assignment in the Course Site page, and then press the "Add submission" button at the bottom of the next page. Drag and drop the file into the area under "File submissions". If necessary, you can update your submission at any time before the deadline passes. Be very careful to ensure that you have named your files and classes correctly.

Sample Output:
If your solver is working properly, you should see something like the output below when you run the program. By specifying command line arguments other than 1, you can solve different mazes.

```
> java MazeSolver 1
#########################################
S           #       #     #   #         #
### # ##### # # ### # ### # ### # # #####
#   # #   # # #   #   #   #   # # # #   #
# ##### # ### ### ##### ##### # # ### # #
#     # #       #   #         # #     # #
# ### # ####### ### ########## ####### #
#   # # #     #   #   #            #   # #
### # # # ### ####### # ########## # # #
# # # # # # #   #   #   #     #     #   #
# # # # # # ### # # ##### ### # ####### #
#   #   #   #     #       #     #       E
#########################################

Solution:
#########################################
S...        #...    #     #   #...      #
###.# ##### #.#.### # ### # ###.#.# #####
#...# #...# #.#...#   #   #   #.#.# #...#
#.#####.#.###.###.##### ##### #.#.###.#.#
#.....#.#.....  #...#         #.#.....#.#
# ###.#.####### ###.##########.#######.#
#   #.#.#     #   #...#.........  #   #.#
### #.#.# ### #######.#.########## # #.#
# # #.#.# # #   #   #...#     #     #  .#
# # #.#.# # ### # # ##### ### # #######.#
#   #...#   #     #       #     #      .E
#########################################
>
```