CSC205AB
Program 2 – solving a Maze

In this program you are given 2 classes to work with:

1) Maze – which is a Maze
2) MazeDisplay – which graphically displays the maze it is given.

You will also use:

1) the Stack class that you wrote and submitted on HyperGrade this semester. If you do not have it, you can use Java's built-in Stack class for a deduction on your grade.

Finally, you will write:

1) A class called MazeSolver, which will be able to create a Maze, set up a MazeDisplay, and then solve the Maze. It should have a method called **.solve()**, which contains a loop which will run until the Maze is solved or until the user says to quit. Inside the loop, the user can enter Q (or q) to quit, S (or s) to save the current state of the MazeSolver, or just ENTER to do another move. See below for details.
2) A class called StartSolvingMaze.java, which will contain a main method that will start the whole program. It should prompt the user to enter the number of rows and the number of columns, then create an instance of a MazeSolver, passing in the number of rows and number of columns. Once the new MazeSolver is created, then the StartSolvingMaze class should tell your MazeSolver instance to **.solve()**
3) A class called ResumeSolvingMaze.java, which will contain a main method that will read in a serialized MazeSolver (instead of creating a new one). So it should ask the user to enter the name of the file that contains the serialized class, then read it in and convert it to the existing MazeSolver. Then tell it to **.solve().**

The enumerated data type called Maze.Direction: Look at the Maze.java file; it contains the code for the Maze. You do not have to know or understand all the code for the Maze class, but it uses a feature called enumerated data types that you will use extensively.

- At the very end of the Maze class is defined an enumerated data type (enum in Java) called Direction. When working with a Maze, you can use an enum called Direction, which is either UP, DOWN, LEFT, or RIGHT. Since Direction is defined inside Maze, you have to refer to it as Maze.Direction and refer to the various directions as Maze.Direction.UP etc. You can use enumerated data types as a type in the same way that you would use other types or classes. For example, you could have:

    - Maze.Direction whichWay; //whichWay is a Maze.Direction…
    - whichWay = Maze.Direction.UP; //…so it can this value, for example
    - Stack<Maze.Direction> myStack; //Generics

  The Direction type will be extremely important, as the Maze methods that you have available are expecting Maze.Directions…

The public Maze methods: Once a Maze is created, these are its public methods that you will use in your MazeSolver class to decide where to move and then tell the maze to move there:

- **.getCurrentRow()** //returns the current row
- **.getCurrentCol()** //returns the current column
- **.goalReached()** //returns true or false, depending on whether the goal is reached
- **.move(Maze.Direction.UP)** //for example…this would tell the current location to move UP
- **.isOpen(Maze.Direction.RIGHT)** //for example…this would return a boolean telling whether or not //the Maze is open from the current location going to the RIGHT //(whether //or not there is a wall there)

You are to write a program called <u>MazeSolver.java</u>. It should have:

- Class data that stores a Maze and also stores the data structures you will use to solve it. To solve it, you will use a Stack and also a data structure to keep track of which cells in the Maze have been "visited" (see the algorithm below). This class data will be eligible to be serialized when the user chooses that option. **NOTE:** you will also create an instance of a MazeDisplay, but do not declare it in the class data. It uses the Thread class, which is not Serializable. So declare it locally in the solve() method.

- A parameterized constructor that will receive a number of rows and a number of columns, both as ints. The constructor will:
  - <u>Create</u> the new instance of a Maze that you declared in the data, passing in the number of rows and the number of columns.
  - Tell the Maze instance to do a **.buildMaze(0);** where 0 is the delay in milliseconds. The larger the delay, the slower the build will be animated.
  - Tell the Maze instance to do a **.setSolveAnimationDelay(n);** where n is the delay in milliseconds, which is how long it will wait between moves. This is so the animation does not happen so fast that you cannot watch it. You can choose any delay you want.
  - Create the other data structures that are declared in the class data (the Stack and the data structure to keep track of whether or not a cell has been "visited")

- A method called solve(), which will contain the logic to actually solve the Maze. It should do the following:
  - Create a new instance of a MazeDisplay, passing in the Maze that has already been created.
  - Once the Maze is built and displayed, you are to write code to solve it. The algorithm is described below. When you implement it, **you cannot change or add code to the Maze's methods, but only <u>use</u> Maze's existing <u>public</u> methods** that are also described above.

The basic algorithm is to try going UP, DOWN, LEFT, or RIGHT (in any order) to legally move through the Maze, taking care to go to a cell (location) that you have NOT already been to. If there is no new location to go to, you have to use the Stack to backtrack. This can be done as follows:

Mark the current (starting) cell you are at as "visited."

Tell the user they are to enter Q to quit, S to save to a file, or just ENTER to move.
Repeat the following actions while the goal location has not been reached and the user does not say to Quit
{
Get the user's next input
If the input is S or s, then ask for the file to serialize to and serialize this MazeSolver to that file.
But if the user does not type Q or q, then make a move, which the logic below shows:

If it is open in the Maze.Direction.UP direction and you have not visited the cell <u>above</u> you, then push Maze,Direction.UP onto your Stack and mark the cell you arrived at as "visited." Tell the maze instance to move in the direction of Maze.Direction.UP.

Else … if it is open in the Maze.Direction.DOWN direction and you have not visited the cell <u>below</u> you, then push Maze,Direction.DOWN onto your Stack and mark the cell you arrived at as "visited." Tell the maze instance to move in the direction of Maze.Direction.DOWN.

Else ….(try the same thing for the Maze.Direction.LEFT direction. Same logic)

Else …(try the same thing for the Maze.Direction.RIGHT direction. Same logic)

Else – you have come to a dead end. Pop the Stack to see the direction you came from. Tell the maze instance to move in the opposite direction, which will accomplish the backtracking.
}

If the user quits, print "Program ended without solving."
If the loop ends with the Maze being solved, print "Maze is solved."

Hints and Suggestions:

- Be sure that the black DOS window is the active window (by clicking in it first) when you enter the S, Q, or ENTER.
- To keep track of which cells have been visited, you could either
    - use a 2-dimensional array of booleans that is the same dimensions as the Maze. Then when you arrive at a new Maze location, set the corresponding entry in the array to true.
    - Or…create an ArrayList that holds Points. Then when you arrive at a new Maze location, create a new Point with the (row, col) and tell the ArrayList to add it. To check if a cell has been visited before, see if the ArrayList .contains that Point
- If your logic causes the Maze animation to just go back and forth between 2 cells, it means that you are not checking to see if a cell is "visited" correctly
- Let the user type in the complete file name when serializing. Do not put the .dat or any other extension on for them.

Testing your program: Since this assignment will not be using HyperGrade, you will have to test your program by yourself. This will be easy, since there is no data – just be sure it can successfully solve Mazes, quit, serialize, and read a serialized file. **There are also no late days to use so plan on submitting it on time!**

Comments and formatting: Please have an opening comment which briefly describes the class. Each method should also have an opening comment. There should be comments in the code to describe lengthy "sections" or to help the reader understand "tricky" parts. For additional maintainability, code should be indented correctly and the variable names should be meaningful. This is important for maintainability. Deductions for lack of comments and incorrect formatting could be as high as 10% of the program.

Please submit <u>via Canvas</u>: Your Stack.java, MazeSolver.java, StartSolvingMaze.java, and ResumeSolvingMaze.java files. Please be sure they are all named correctly and compile.