COMP1406 - Assignment #10

(Due: **Monday, April 3rd @ 10:00 am**)

In this assignment you will gain practice with **recursion** by implementing some recursive methods that will be called from a user interface.

(1) Getting Started

To begin, download the code for this lab which is provided on the course cuLearn page. The code contains various model and GUI classes that you will use to visualize your solutions.

(2) Maze Generation

In the downloaded code, you will see classes called **Maze**, **MazeCreator** and **DimensionsDialog**. These are needed for this portion of the assignment. The **Maze** class represents a maze of size **rows** x **column**. Each location in the maze is either a *wall* (black) or an *open* space (white). Take note of the available methods.

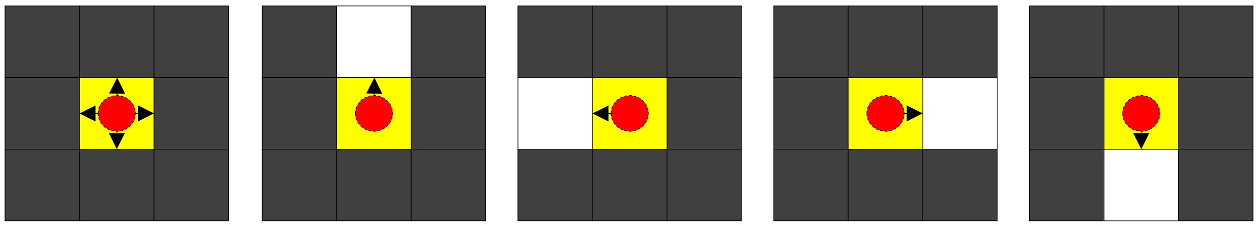
```
import javafx.geometry.Point2D;

import java.util.ArrayList;

public class Maze {
    private static final byte  OPEN = 0;
    private static final byte  WALL = 1;
    private static final byte  VISITED = 2;

    private int          rows, columns;
    private byte[][]     grid;

    // A constructor that makes a maze of the given size
    public Maze(int r, int c) {
        rows = r;
        columns = c;
        grid = new byte[r][c];
        for(r=0; r<rows; r++) {
            for (c = 0; c<columns; c++) {
                grid[r][c] = WALL;
            }
        }
    }

    public int getRows() { return rows; }
    public int getColumns() { return columns; }

    // Return true if a wall is at the given location, otherwise false
    public boolean wallAt(int r, int c) { return grid[r][c] == WALL; }

    // Return true if this location has been visited, otherwise false
    public boolean visitedAt(int r, int c) { return grid[r][c] == VISITED; }

    // Put a visit marker at the given location
    public void placeVisitAt(int r, int c) { grid[r][c] = VISITED; }
    // Remove a visit marker from the given location
    public void removeVisitAt(int r, int c) { grid[r][c] = OPEN; }

    // Put a wall at the given location
    public void placeWallAt(int r, int c) { grid[r][c] = WALL; }

    // Remove a wall from the given location
    public void removeWallAt(int r, int c) { grid[r][c] = 0; }

    // Carve out a maze
    public void carve() {
        int startRow = (int)(Math.random()*(rows-2))+1;
        int startCol = (int)(Math.random()*(columns-2))+1;
        carve(startRow, startCol);
    }

    // Directly recursive method to carve out the maze
    public void carve(int r, int c) {
        // Fill in the missing code
    }
}
```

Notice that there are two methods called **carve()** and **carve(int r, int c)**. Initially, all created mazes are filled with walls, so that there are no open spaces. When called, the carve method calls **carve(int r, int c)** which is supposed to "carve" out the maze by removing walls in a random fashion by starting at position **(r, c)** in the maze. You must complete the **carve(int r, int c)** method so that it does this in a recursive manner.

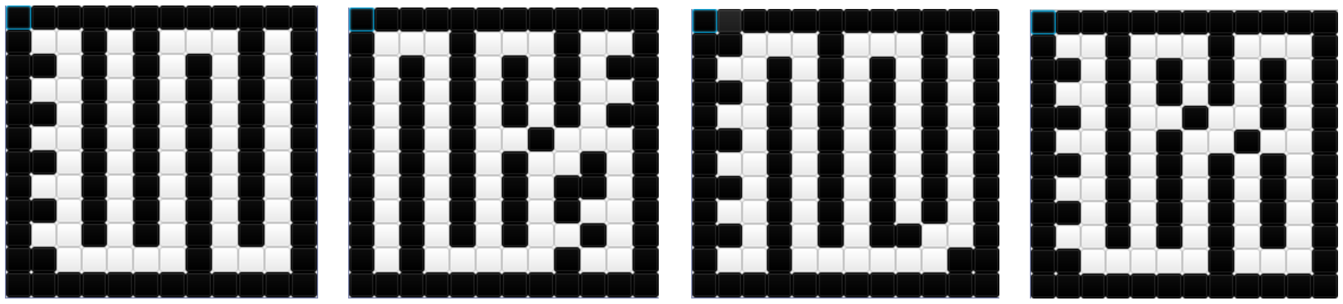
The idea behind the algorithm is as follows:

```
CARVE(r, c) {
    if location (r, c) is along the border of the maze, then
        don't remove the wall here, size we want a solid border around the maze at all times.
    otherwise, if location (r, c) is an open space,
        we already removed the wall here, so there is nothing more to do.
    otherwise
        we should remove the wall at this location if there are at least 3 walls around it ... then
        we should recursively carve out the maze in all 4 locations around (r,c).
}
```

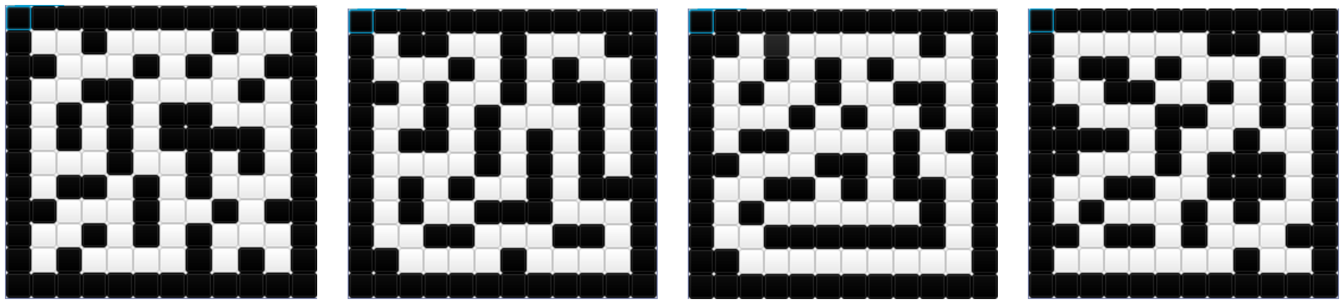
The image here shows the only 5 situations in which the yellow WALL location (r, c) should be changed into an OPEN location since there are 3 or more walls around it.



The part of the algorithm that says to carve out the locations all around (r, c) will require you to call the recursive function again, but with a different (r, c) each time. In order to get a random maze, it is important to go recursively in the 4 directions in random order. That is, for example, you cannot always go up, then down, then left, then right. Otherwise, this will produce poor mazes that are always very similar with long paths like this:

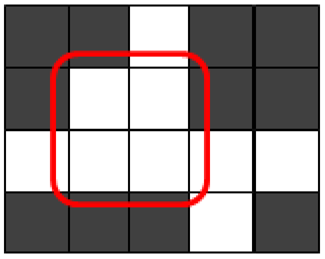


Instead, we would like to produce mazes which are more random like this:



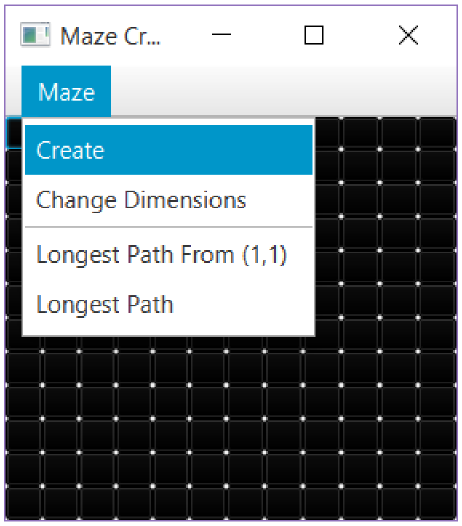
In order to do this, in your recursive calls, you must vary the order in which you call the method recursively. One way to do this is to make two **ArrayLists** of offsets as follows:

```
  ArrayList<Integer> rowOffsets = new ArrayList<Integer>(Arrays.asList(-1, 1, 0, 0));
  ArrayList<Integer> colOffsets = new ArrayList<Integer>(Arrays.asList(0, 0, -1, 1));
```



Then you can repeatedly call your **recursive** method 4 times by randomly selecting an index in these array lists and using those corresponding row/column offsets. For example, if you choose random number 2, then the offsets in the arrays are 0 and -1, so you can next go to location (r+0, c-1). You will know that your code works when it produces images similar to what is shown above. You should never see four open spots arranged in a 2 x 2 block (as shown here on the right), nor should you see any larger blocks of open spots. **<u>Note that if your method is not written in a properly recursive manner, you will not get marks for this part of the assignment</u>**.

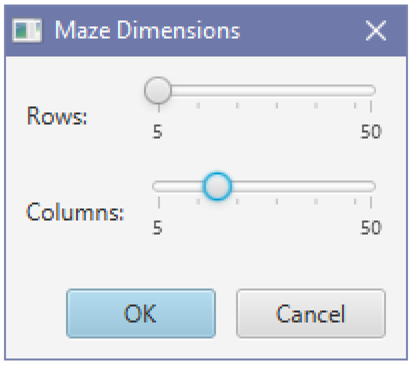
You will test your code by using the provided **MazeCreator** application which makes use of the **DimensionsDialog** class as well. Just run the code and you should see the interface look as shown on the right with a 12x12 maze completely filled with walls, and a Menu as shown.



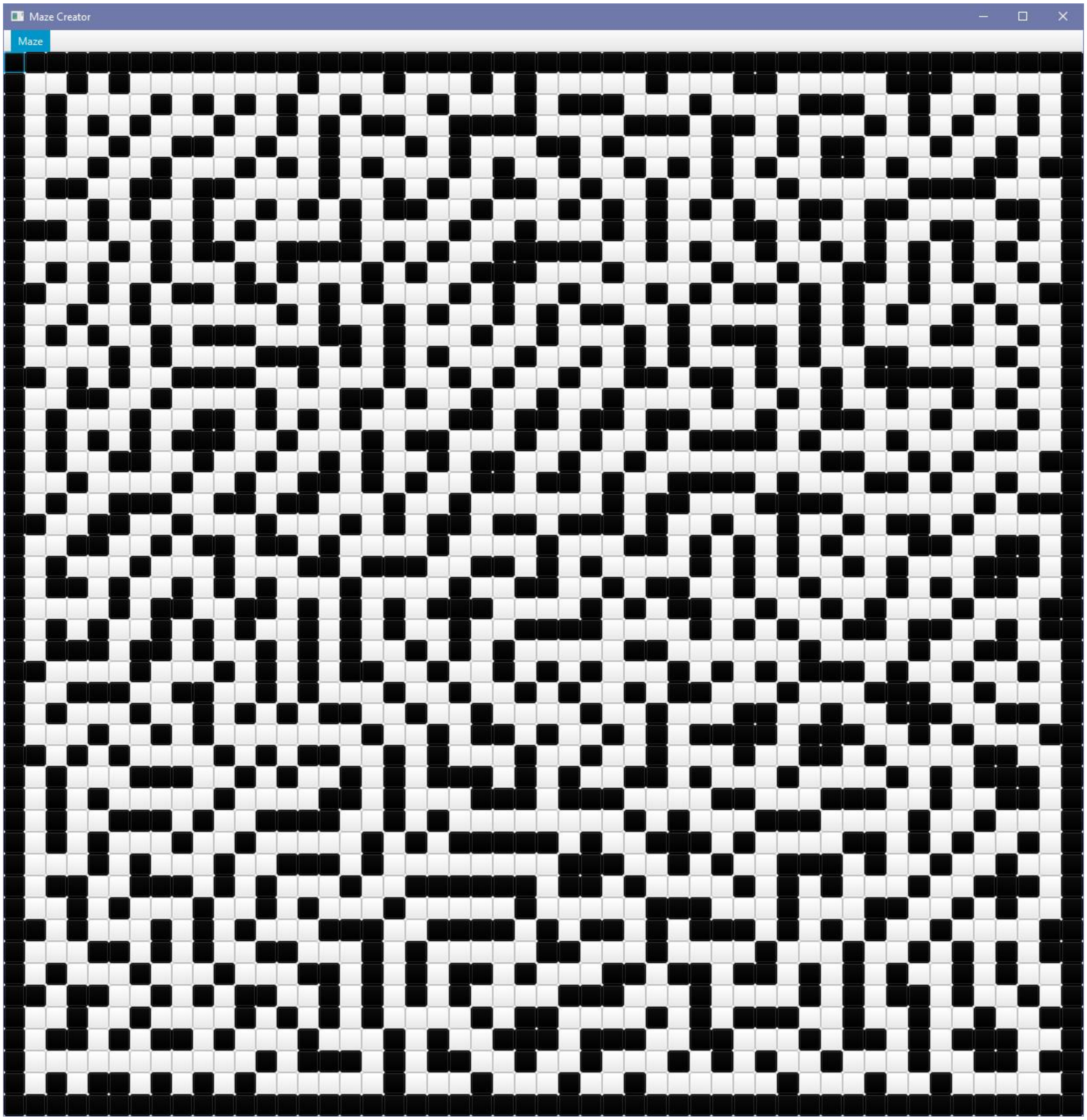
The **Create** option will create a new maze and refresh the window so that you can see what it looks like.

The **Change Dimensions** option brings up a new **DimensionsDialog** that allows you to change the size of the maze:

Try various maze sizes and have fun with it, once you get your **carve()** method working.



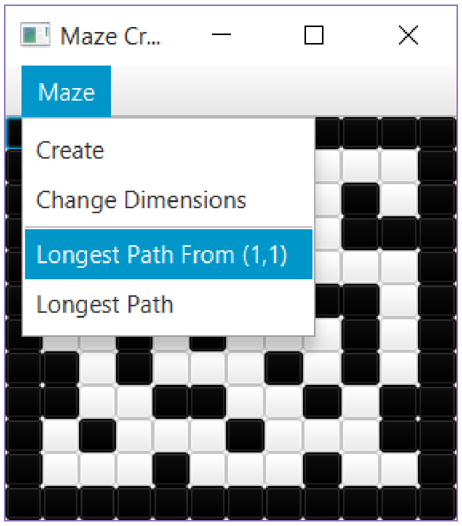
Below is a carved out 50x50 maze:



(3) Finding the **Longest Path**

In the **Maze** class, add the following methods to the end of the code:

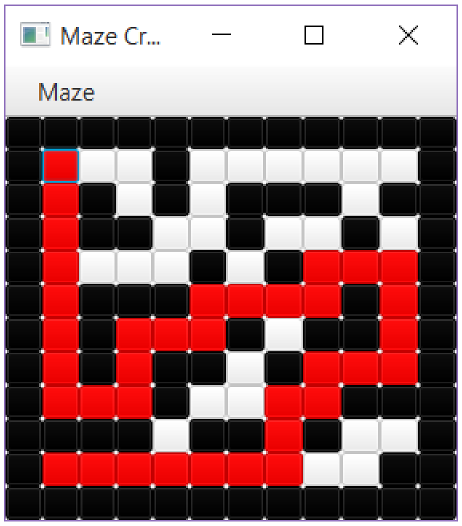
```
    // Determine the longest path in the maze from the given start location
    public ArrayList<Point2D> longestPath() {
        return longestPathFrom(1, 1);   // Replace this with your code
    }

    // Determine the longest path in the maze from the given start location
    public ArrayList<Point2D> longestPathFrom(int r, int c) {
        ArrayList<Point2D> path = new ArrayList<Point2D>();

        // Fill in the missing code

      return path;
    }
```



These methods are to be called AFTER you carve out a **Maze** by selecting the options from the Menu:

The Longest Path From (1,1) must be written recursively. That means that the method must have no WHILE loops nor FOR loops at all, otherwise you will get no marks. The code should attempt to fid the longest path in the maze that starts at location (1,1), which is the first location at the top left corner within the walls as shown in red in the image below.

Note that for some mazes, there will be a wall at location (1,1). In that situation, there is no path from (1,1), so no red path will be shown.

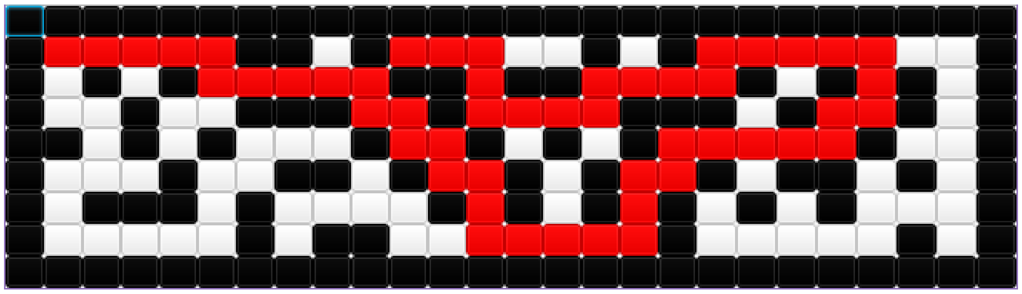


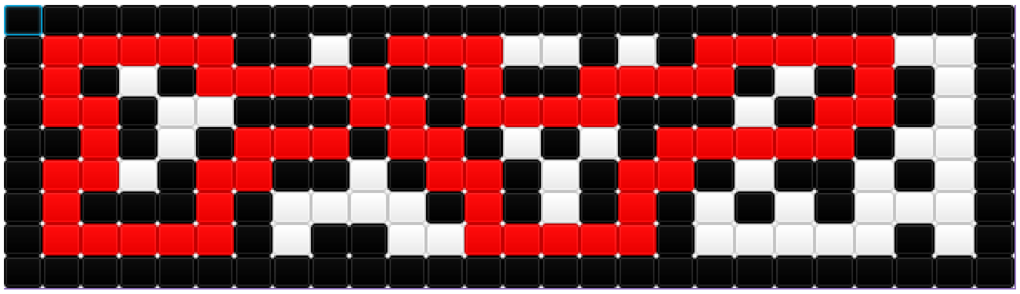
Notice that the method returns an **ArrayList<Point2D>** object. That is, it returns a collection of Point2D objects that represent the path. A **Point2D** object represents an **(x, y)** location is 2D. You can make a new one by simply supplying the **row** and **column** of a point that you want on the path as follows: `new Point2D(r, c)`

Your recursive code must build up such a path of points and return it as the answer. You will need to properly make use of the **VISITED** state of the grid cells so that you don't re-visit the same locations multiple times, otherwise your code may go into an infinite loop leading to a **StackOverflow Exception**. The GUI interface code will automatically assume that once your code completes, then only the path's points will be marked as **VISITED**. Therefore, after you are done the recursive steps, before you return from the method, you will need to remove the **VISITED** marker that you placed down that round of the recursion. You MAY NOT loop through any locations to unmark them. Once your code works, you should see the path each time as shown above. If you see ALL RED spaces, then you did not properly unmark your visited locations within your method.

Finally, the **Longest Path option in the menu will attempt to find the longest path in the MAZE altogether. This path will** likely **NOT** start at location **(1,1)**, but may start anywhere. So you will need to complete the **longestPath()** method (i.e., the one without the parameters) so that it finds that longest path. The method is not to be directly recursive, so you may use **FOR** loops. You MUST, however, call the **longestPathFrom (r,c)** method that you just wrote and make use of its results.

Below is an example that shows the difference between the **Longest Path From (1,1)** and the **Longest Path** from anywhere for a particular maze:





Make sure to hand ALL of your code in … including the GUI code that was given to you.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit your ZIPPED **IntelliJ project file** as you did during the first tutorial for assignment 0.

- YOU WILL LOSE MARKS IF YOU ATTEMPT TO USE ANY OTHER COMPRESSION FORMATS SUCH AS **.RAR**, **.ARC, .TGZ**, **.JAR**, **.PKG**, **.PZIP**.
- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !
- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not written neatly with proper indentation. See examples in the notes for proper style.
