CSC220 Lab12
Graph Algorithms

The goal of this week's lab is:

1. Practice working with graphs
2. Practice working with stacks and queues
3. Practice using breadth-first search (BFS) and depth-first search (DFS)
4. Practice using a nested class
5. Practice I/O

This week, you are asked to write a program to help Pacman solve mazes. We will apply BFS and DFS search strategies to find a path from one location to another in any enclosed field of obstacles. We will also compare the results to see which delivers the shortest path. The field is given as input, represented as a plaintext file. The start and end point are indicated, as well as the layout of the field (the location of the walls and obstacles). **A similar text file is produced as output, annotated with the path from the start-point (S) to the end-point (G)**. If no such path exists, the input and output text files are identical. The following figure shows an example input and output, where Pacman's search strategy here gives the shortest path:

```
5 5              5 5
XXXXX            XXXXX
XS  X            XS..X
X   X            X .X
X GX             X  GX
XXXXX            XXXXX
```

Pacman is free to move from his current location to any adjacent open space. This includes the space directly above, below, left and right of where he is. It does **not** include diagonally adjacent spaces. If any of the adjacent spaces are a wall, Pacman cannot travel in that direction. The path that your maze solver finds MUST be a **connected** path (it can't skip spaces and have Pacman "jumping" over walls or empty spaces).

You can assume that all input mazes will be rectangular. All of the border positions around the perimeter of the field will be walls (the field is fully enclosed).

To solve this problem, you must first encode the maze as a graph, and perform a search strategy from Pacman's starting location. The graph here can be represented by a 2D array of nodes. In lab, we will populate the graph and prepare the output file. For the assignment, you will implement search.

Part 0

- Create a new Java project and call it **Lab12**.
- Create a package inside your project and call it **lab12.**
- Download the **Lab12-Assignment12** ZIP folder from BB and copy in the following files:
  - **Pacman.java (inside code/)** You will be working on developing methods for this class to help Pacman solve mazes.
  - **mazes/** This directory holds the input mazes we will be working with, and the solutions. You have learned how to add a text file to your project in assignment 4. If you are in doubt consult the instruction for assignment4 on Blackboard

- Note the following directory for later:
  - **pacman/** This holds some Python scripts you can use to visualize your mazes.

Part 1 – Pacman **constructor**

The constructor for this class receives an input and output filename. The input name is used to read a maze from a file with the given input name and the output name will be used by our solve methods to output the solved maze to a file.
The constructor has already been started for you. To finish the implementation, **you must finish writing the method buildGraph()** which initializes member variables and populates the graph according to the input file. This method uses a **BufferedReader** to visit the contents of the file line by line.

The input files are in the following form:

```
5 5
XXXXX
XS  X
X   X
X  GX
XXXXX
```

The first line contains two numbers, separated by a space. The first number is the height of the field, and the second is the width. The rest of the lines contain the layout of the field. The characters have the following meaning:

- **X:** A single wall segment. Pacman cannot travel on to or through walls.
- **S:** The starting point of the path that we are trying to find (where Pacman starts). This is an open space (no wall). Be sure to set the position of startX and startY after S has been visited.
- **G:** The ending point of the path we are trying to find (where Pacman wants to be). This is an open space (no wall).
- **(space)**: An open space on the field. Pacman is free to travel in any open space, assuming he can get there.

Part 2 – writeOutput **method**

This method is responsible for writing out the contents of the maze to file as pointed to by **outputFileName**. Output the height and width at the top, just like the input. The output should also have the same layout of the walls and the same start and end points. After completing the assignment, some space characters will be replaced with dots ('.'). For now, the input and output files should be identical. **There is a single newline character after the last 'X' in the output.**

You must produce output in the exact format specified (for grading purposes). You will lose up to 10% if the format of your output file is wrong even though your solution is correct.

PrintWriter can be used to create and write to a file in Java. The method provides the incantation for you. Do not modify the path of the file.

Part 3 – Test your code

As usual you need to test the functionality of the methods you have implemented.

In order to test your code, you should make your own PacmanTester.java file with a main method for testing. (We will use our own PacmanTester.java for grading purposes)

You should now be able to instantiate a Pacman object from an input file name **(try any of the mazes from the mazes/ directory)** that populates the 2D array of nodes and sets member variables accordingly. To check the structure of your maze, try invoking the writeOutput() method to output the maze to file. You can use an online diff tool like https://text-compare.com/ to compare the input and output files. As noted before, these should both be identical at this stage.

How to debug your code?
1. Use the Eclipse debugger you learned about during the first lab.
2. If you see JavaStackOverflow, that means that you have an infinite recursive call and your recursive call is filling up the "call stack" (we talked about this concept in class). Go back and debug the method that is causing the problem.
3. Infinite loops! How would you know you have an infinite loop? As you should know from CSC120, if you have an infinite loop in your code, your code will not stop running. An easy way to inspect that in Eclipse is to look at your console window, if your code is done running the console should look like the following:
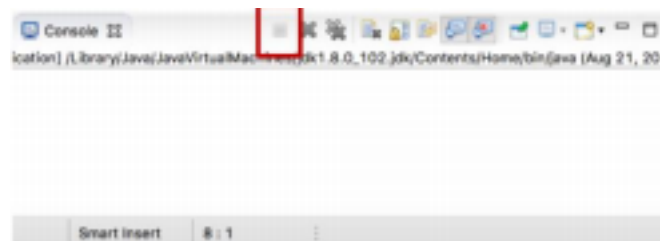
If the little square marked above is red and continues to stay red, that means your code has an infinite loop!

<u>Don't forget</u>: This lab/assignment is due Thursday night @ 11:59pm!