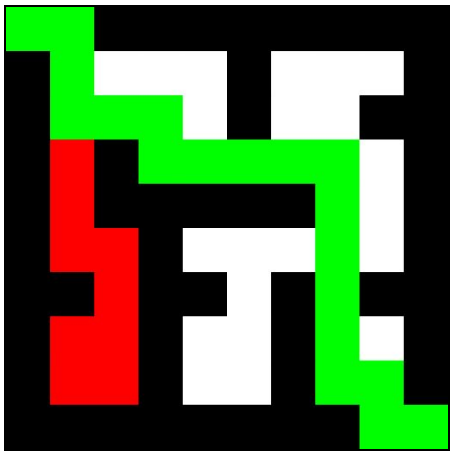
# Maze Solver

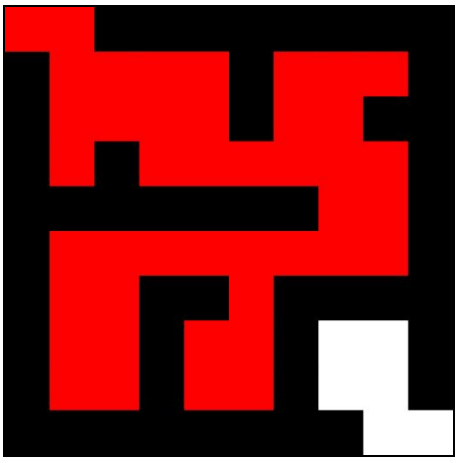
## Introduction

In this project you will implement a dummy agent to find a path out of a maze, provided that one exists. We say this agent is a dummy because there isn't any sort of intelligence in how it chooses its path. It will, however, be methodical by looking up, down, left, and right at each cell. If there is a path from the start to the finish, it will eventually reach the exit. There is no guarantee, though, that this path will be the shortest or most efficient. Stick around for the spring semester to learn about those algorithms. :)

Our maze solver will be designed to find a path from any valid location in our maze to the exit. For simplicity, we'll say that the bottom-right cell is the exit and for debugging purposes we'll begin in the top-left corner. You can modify this starting location later, if you'd like.



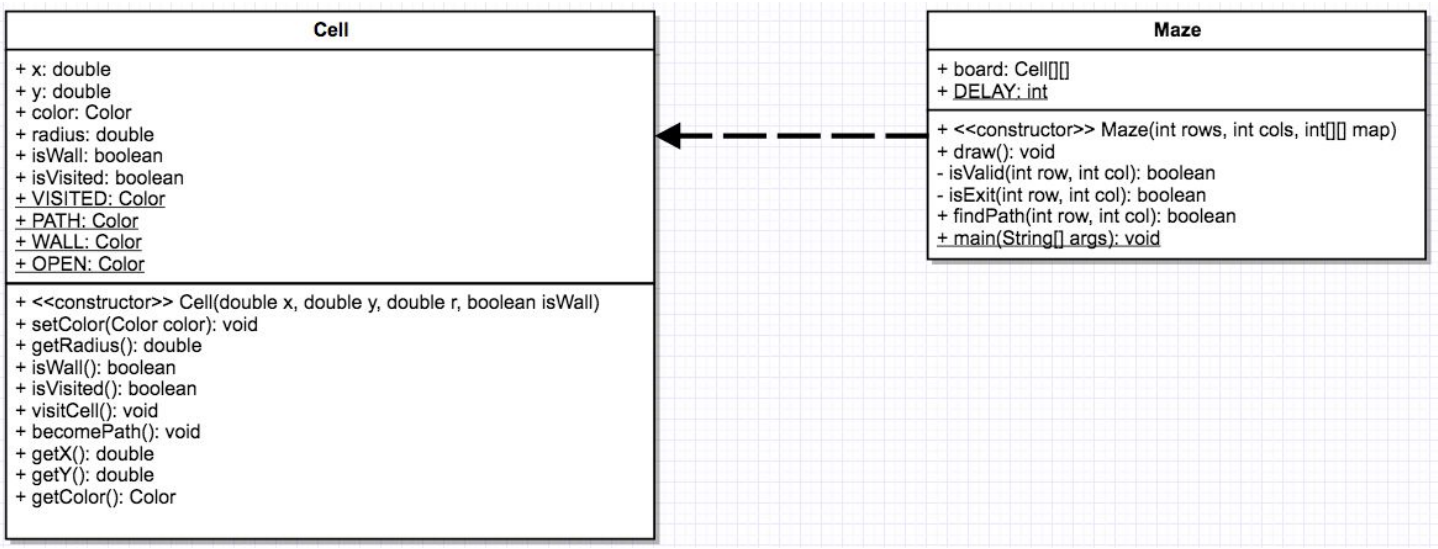
A maze with a solution path.



A maze with no solution path.

## Project Structure

You'll need to familiarize yourself with the UML diagram of the project seen below. The methods of the `Cell` class will be needed when you begin traversing the maze.



# Maze solver

## Your Responsibilities

You'll be tasked with implementing three methods in `Maze`: `isValid`, `isExit`, and `findPath`.

- `isValid(int row, int row): boolean`
  - Returns a `boolean` whether or not position `(row, col)` is an open cell in `board`.
    - Ensure that `(row, col)` is a valid index in `board`.
    - Ensure that the cell isn't a wall and hasn't already been visited.

- `isExit(int row, int row): boolean`
  - Returns a `boolean` whether or not position (`row, col)` is the exit in `board`.

- `findPath(int row, int col): boolean`
  - Returns a `boolean` indicating whether or not a path to the exit has been found beginning at the location `(row, col)` of `board`.
    - Use a `boolean` flag to record whether or not the path is complete.
    - Check to make sure that `(row, col)` is valid.
      - If it is valid, visit the cell, draw the board, and pause for `DELAY` seconds.
        - Are you at the end of the maze?
          - If yes, your flag is `true`.
          - If not, recursively find the path from the four adjacent cells.
    - Check to see if your `boolean` flag is `true`.
      - If it is, `(row, col)` should become part of the path and the board should be redrawn. Don't forget to pause for `DELAY` seconds.
    - Return the flag.
- `main(String[] args): void`
  - This is the method that runs your program. It uses a hard-coded 2D array of integers to build the maze. A value of 1 indicates an open cell, and a value of 0 indicates a closed cell. Your starter code comes with a 10x10 maze. Feel free to tinker with this.

## Extensions

Completing the basic maze solver will net you a score of 90% on this assignment. To earn a higher score, choose one of the following improvements that you find interesting:

- Hard-coding isn't a best practice. Modify your `Maze` class so that it reads a CSV file of 1s and 0s and creates a maze based on the CSV file. This shouldn't be hard-coded in `main`. Modularize your code and add a `private` helper method(s) to handle this File I/O logic.
- Creating CSV files can be tedious and time consuming, especially for large mazes. Add a `private` method that will generate a random maze when called by the `Maze` constructor. Barring the exit, the outside perimeter of the maze should be walls. Every cell inside the maze should have some probability, p, of being a wall. You'll need to tinker to find a probability that works well.
- When the our dummy agent travels down a dead-end path, we lose sight of where it is in the sea of red cells. Modify the program so that the cell in which the agent currently is visiting is a unique color. Note that it should return to its previous state when the agent leaves the cell.