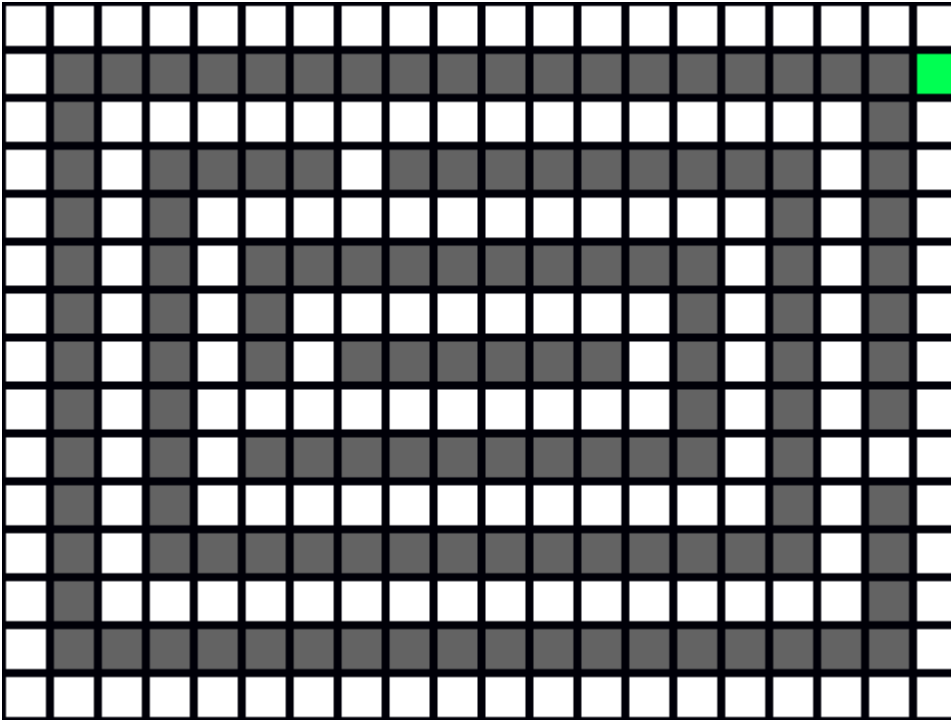


# **Week 14 Lecture 1**

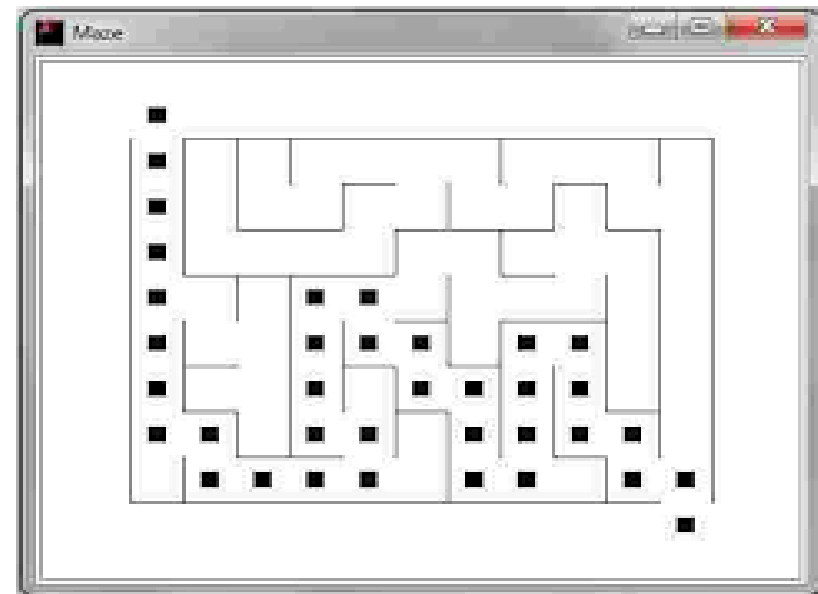
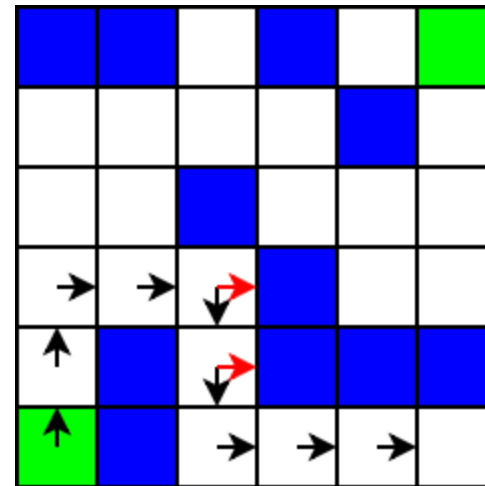
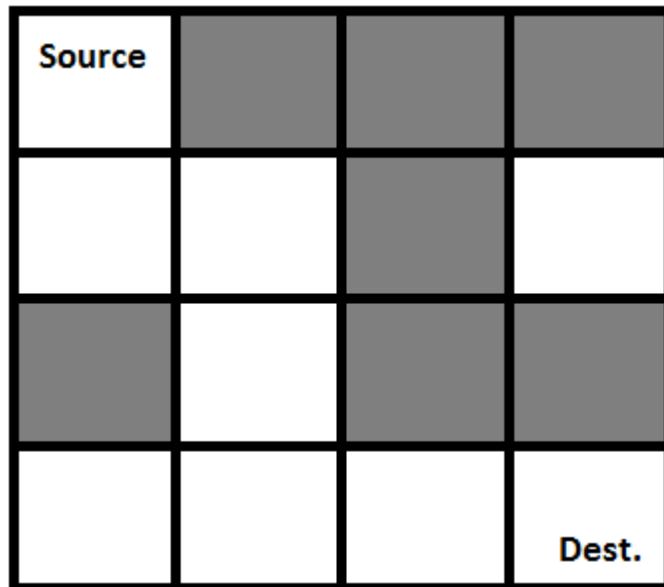
# Solving a Maze By Recursion



# Maze Solving

- For the next programming project, you and some teammates will write a program to solve a maze
- In solving a maze
  - **The starting point (or source) is specified**
  - **The exit point (or destination) is specified**
  - **Your program finds the path from the start to the end**

# Mazes of varying sizes and difficulties



# Specifying a Maze for Your Program

```
. . . . . . . . G
. #####
. #####
. # . . . #####
. # . #####
. . . #####
. #####
. #####
. . S #####
. #####
```

You will specify a maze as shown at left.

'.' means that the cell is open and may be part of the solution path

'#' means the cell is blocked

'S' indicates the starting point in the maze

'G' indicates the goal or destination point in the maze

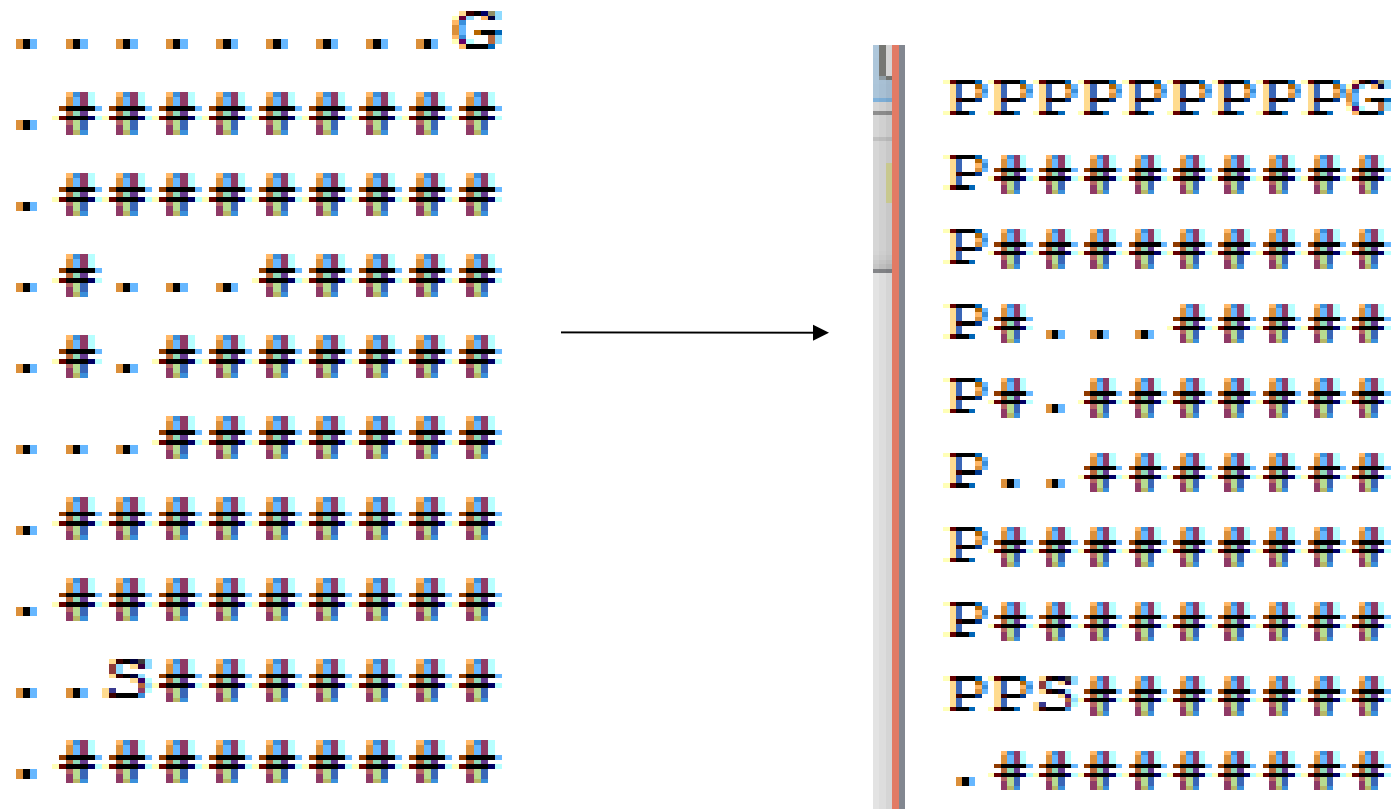
You will add 'P's" to the grid to indicate that the grid is part of the solution path.

# Two More Mazes

```
.S....#  
##.###G  
##.###.  
#...##.  
...###.  
##.###.  
##.....
```

```
S...##  
.####  
...##  
.#.##  
.#G##
```

Your goal is to find a path from the cell labeled 'S'  
to the cell labeled 'G'



You will display the maze at each step as shown below..

---

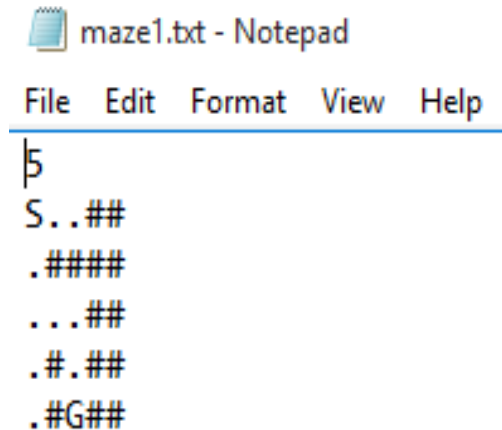
```
.....G
.#####
.#####
.#...####
.#.#####
...#####
P#####
P#####
PPS#####
.#####
Press Enter for next step
Currently checking row= 5 and column = 0 goalX = 0 goalY = 9
```

```
.....G
.#####
.#####
.#...####
.#.#####
P..#####
P#####
P#####
PPS#####
.#####
Press Enter for next step
Currently checking row= 4 and column = 0 goalX = 0 goalY = 9
```



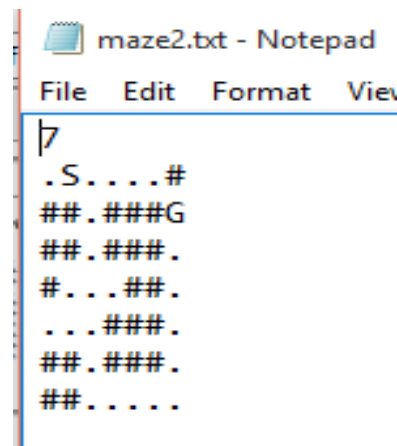
# Specifying a Maze

- Mazes will be specified in text files.
- For this assignment, we will assume that the mazes are square – that is they contain equal numbers of rows and columns
- The first data line in the file will be an integer that specifies the number of columns and rows
- The remaining data lines will be the maze itself



```
maze1.txt - Notepad
File Edit Format View Help
5
S..##
.####
...##
.#.##
.#G##
```

**A 5x5 maze**



```
maze2.txt - Notepad
File Edit Format View
7
.S....#
##.###G
##.###.
#...##.
...###.
##.###.
##.....
```

**A 7x7 maze**

# Maze Solver Class

- The `MazeSolver` class should contain the following member variables
  - An integer `n`. `n` will be read from the input file and tells you how many rows and columns the maze has.
  - An array of `char` – named `maze` – to hold the maze
  - A variable called `startX` to indicate the row in which to start
  - A variable called `startY` to indicate the column in which to start
  - A variable called `goalX` to indicate the column of the goal cell
  - A variable called `goalY` to indicate the column of the goal cell
  - A string called `filename` to hold the name of the text file containing the maze

# Member variables and constructor for Maze Solver

```
import java.util.Scanner;
import java.io.*;

public class MazeSolver
{
    static char[][] maze;
    int n, startX, startY, goalX, goalY;
    String filename;

    public MazeSolver(String filename)
    {
        this.filename = filename;
    }
}
```

# Maze Solver Class

- You will also need a method in the Maze Solver to read in the maze from a text file.
  - You will need a `FileReader`
  - You will need a `BufferedReader`
  - You must enclose everything in a `try` block
    - You must catch `FileNotFoundException`
    - You must catch `IOException`
- You may ASSUME that the maze files are always correct
  - You may assume that the first line contains `n`
  - You may assume that `n` lines will follow that contain the maze

```

public void readMaze()
{
    try {
        FileReader fr = new FileReader(filename);
        BufferedReader br = new BufferedReader(fr);
        n=Integer.parseInt(br.readLine()); //read the size of the maze
        maze = new char[n][n]; ← Explain this!

        for(int i=0; i<n; i++) //for each row of the maze
        {
            String s = br.readLine(); //read an entire row of the maze
            for(int j=0; j<n; j++)
            {
                maze[i][j] = s.charAt(j); //put the characters in the maze
                if(maze[i][j] == 'S') //set start cell
                {
                    What goes here?
                }
                if(maze[i][j] == 'G') //set goal cell
                {
                    What goes here?
                }
            }
        } // end for j loop
    } //end for i loop
} //end try block
catch(FileNotFoundException e) {
    System.out.println("File not found");
}
catch (IOException e) {
    System.out.println("Invalid entry");
}
}

```

# Maze Solver Class – Displaying the Maze

- It will be necessary to display your maze after every move.
- This method is very short and very easy to write
- Recall the difference between Java print and Java println!

```
public void displayMaze()  
{  
    System.out.println();  
    for(int i=0; i<n; i++)    //for each row  
    {  
        for(int j=0; j<n; j++)    //for each column  
            System.out.print(______);  
        System.out.println(______);  
    }  
}
```

# **Adding a Method to Solve the Maze**

# Recursive Maze Overview

- The recursive maze solution goes as follows
  - If the grid is 1 x 1, the solution is easy
  - If the single cell is the goal, we return true and if not, we return false
  - If the grid is not 1 x 1, see if you can find a path to the destination from one of the neighbors of the cell in question
  - If you can, the path to destination includes the current cell plus the path from the neighbor cell to the destination



# Recursive Maze Solution in Detail

- To solve the maze recursively, we apply the following thinking:
- Suppose we are at the cell in row =  $x$  and column =  $y$ 
  - **If  $x = \text{goalX}$  and  $y = \text{goalY}$  then we are done! We return true.**
  - **If  $x < 0$  or  $x \geq n$  or  $y < 0$  or  $y \geq n$ , we return false. The cell  $(x,y)$  is off the grid that contains the maze. Return false!**
  - **If we have already visited the cell at  $(x,y)$  we return false**
- If we did not return for any of the above reasons, we have found a cell that we can enter.
- We mark the cell  $(x,y)$  with a 'P' to indicate it is part of the path
- We then recursively call the solve method with each of the neighbors of  $(x,y)$  to see if we can find a path to the destination.

# A Solution May Require Back-Tracking

```
.S....#
###.###G
###.###.
#....#.
...###.
###.###.
###.....
```

```
PSP...#
###P###G
###P###P
#PP.###P
PPP###P
###P###P
###PPPPP
```

Depending on which order we check neighboring cells, a solution to a maze may require backtracking.

The arrows show the direct solution. However the code – as we shall see – directs the algorithm to always check the cell to the left first.

# Writing the code...

First the test class

```
public class MazeTest
{
    public static void main(String[] args)
    {
        MazeSolver myMaze = new MazeSolver("maze2.txt");
        myMaze.readMaze();
        myMaze.solveMaze();
    }
}
```

# The SolveMaze

```
public void solveMaze()  
{  
    solveMaze(startX, startY);  
}
```

**A public method to solve the maze.**

**startX and startY got their values from  
readMaze**

# Private solveMaze method

```
private boolean solveMaze(int x,int y)
```

```
{
```

```
Scanner mys= new Scanner(System.in);
```

```
System.out.println("Press Enter for next step\nCurrently checking row= "+x +  
" and column = "+y+" goalX = "+goalX+" goalY = "+goalY);
```

```
String e = mys.nextLine(); //to pause output
```

```
if(x < 0 || x >=n || y<0 || y >=n)
```

```
return false;
```

```
if(x==goalX && y==goalY)
```

```
return true;
```

```
if(maze[x][y] == '#' || maze[x][y] == 'P' )
```

```
return false;
```

```
if(x==startX && y==startY); //leave an S at the start position
```

```
else
```

```
{
```

```
    maze[x][y]='P';
```

```
    displayMaze();
```

```
}
```

```
if(solveMaze(x-1,y) == true) //north
```

```
return true;
```

```
if(solveMaze(x,y-1) == true) //east
```

```
return true;
```

```
if(solveMaze(x+1,y) == true) //south
```

```
return true;
```

```
if(solveMaze(x,y+1) == true) //west
```

```
return true;
```

```
return false;
```

```
}
```

This section of code allows the user to hit the enter key before the next step is printed.

These are the base steps

This marks a new point on the path and displays the maze.

These are the recursive steps

# For you to do!

- You now have all of the instruction to write the MazeSolver and its test class.
- Begin work now – with your teammates.