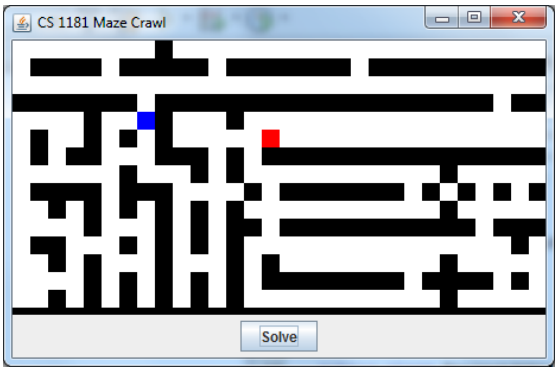
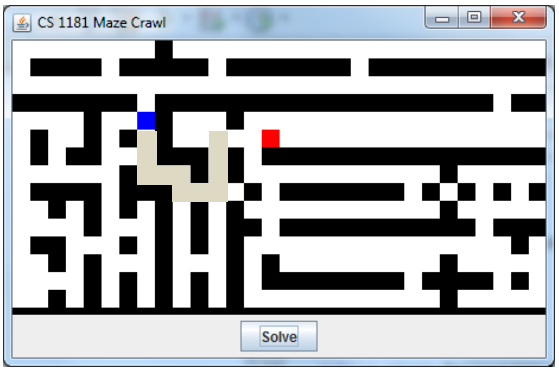
CS 1181 Project 3 – Fall 2014
Due: See the drop box for this project on course's Pilot site.
POINTS: 100

PURPOSE: In this assignment you will use recursion to solve a problem. You not only demonstrate you understanding of recursion but also your ability to integrate your solution into several software modules. This project will also reinforce the concepts of object contract software development, and other key course final objectives.

PROBLEM: MazeCrawler is an application that creates and displays a random maze with a designated beginning and end. Your program will use a graphical user interface (GUI) to show the recursive navigation path through the maze. The GUI will have one button to start your program. Your program must be able to generate a solution for the maze (if one exists).





The software architecture for your program breaks up the problem in 3 major tasks. Task #1 is that task of constructing the software necessary to represent and display randomly generated mazes. Task #2 is the task of constructing the software necessary to implement the logic to move (north, south, east, and west) and automatically solve the maze (solve), and save the maze to a file. Task #3 is the job of creating the user interface (UI) for the application.

In a real word situation, all three of these tasks might take place at the same time. Thus, the first step in software construction is usually to define application in terms of the necessary objects and their interactions (public members). Each team can then work on their portion of the problem simultaneously and eventually put them together into a working system of the whole. In this project, the software architecture (objects and their public members) are determined for you.

The overall application is to consist of 5 interactive classes in package Maze represented by corresponding java source files. Class GUI hands the UI (Task #3); GUI IS-A ActionListener, HAS-A JFrame, and HAS-A MazeCrawler. Class MazeCrawer handles the game logic (Task #2); MazeCrawler IS-A Maze. Class Maze is responsible for tracking the state of the maze and displaying it (Task #1); Maze ISA JPanel and HAS (many) MazeCells. Class MazeCell (also Task #3) keeps track of the state of each cell in the maze.

In this project, you are responsible for Task #2. Instead of just providing you with UML diagrams of the other classes, you will be provided with complete and final versions of the other four interacting objects. Please read the existing classes carefully, and consider the importance of good documentation standards as you do so! Your task is to complete the overall system by developing the MazeCrawler class. YOU MAY NOT MAKE ANY PERMENENT CHANGES TO THE OTHER FOUR CLASS FILES. Feel free to modify them temporarily if you need to do so to see how they work or test the software but your code must work with the original provided source files in order for you to receive ANY credit for the project.

SPECIFICATION DETAILS: You have a fairly limited degree of latitude in your project design and implementation. You MUST implement the public members that the other classes in the application call. You may not change the behavior of the existing public methods in the Maze and MazeCell classes – use them to handle the display and state of the maze as written! At a bare minimum, your MazeCrawler class must implement the public methods: MazeCrawler(rows: int, columns: int), isWin(): Boolean, isNoMove(): boolean, moveNorth() : void, moveSouth() : void, moveEast() : void, moveWest() : void, solveMaze() : void, and saveToFile (filename: String) : void. You will almost certainly need to develop several private variables and helper methods to accomplish these tasks.

HINT: You are expected to be able to read the existing code and figure out how/why it works. You may find that creating UML diagrams for the existing code may be of use. To begin, I recommend connect the two existing portions of the system together with your portion of the system. The easiest way to do this is to simply create stubs for all of the required public methods for handling movement, solving the maze, etc that initially, do nothing. Once you are able to display the UI with the maze, implement your methods one at a time. Remember, MazeCrawler IS-A Maze – thus you will have many methods available to you via inheritance.

HINT: As this application is not an animation with a regular refresh rate, you may find it most efficient to call objectToPaint.paint(objectToPaint.getGraphics()) directly to update a Jpanel objectToPaint. Use GUI.pause() after each call to paint if you need to guarantee a pause before you again updating the graphics.

HINT: The lion's share of the work in this project is NOT necessarily implementing the code. Instead, a great deal of your time will be spent understanding the overall project, looking at the portions that are provided, and considering HOW/WHAT needs to be done to implement "your portion" of the project. Take the time to read everything CAREFULLY!

SAMPLE OUTPUT: The output below represents, as text, the maze and partial solution that corresponds to one of the images shown earlier in this document. The maze symbol's meanings are defined in MazeCell.java.

```
. . . . . . . . # . . . . . . . . . . . . . . . . . . . . . . . . . .
. # # # # . # # # # # # . # # # # # # # # # . # # # # # # # # # # # #
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
# # # # # # # . # # # # # # # # # # # # # # # # # # # # # # # # . # #
. . . . # . . S # . . . # . . . . . . . . . . . . . . . . . . . . . .
. # . . # . # * # . . * . . X . . . . . . . . . . . . . . . . . . .
. # . # # . . * # # # * # . # # # # # # # # # # # # # # # # # # # # #
. . . . . . # * * * # * # . . . . . . . . . . . . . # . . . . . . .
. # # # # . # # # * * * . # # # # # # # # # # # # . # . # . # . # .
. . # . # . # . # . # . # . . . . . . . . . . . . . # . . . . . . .
. . . . # . . . # . # . # # . # # # # # # # # # # # # # # . # # # .
. # # . . . # . # . # . # . . . . . . . . . . . . . . . . . . # .
. . # . # . . . # . . . # . # . . . . . . . . . . . . . # . . . . .
. . . . # . # . # . # . # . # # # # # # # # # # # # . # # # # . # .
. . # . # . # . # . # . # . . . . . . . . . . . . . # . . . . . .
```

SOLVING A MAZE USING RECURSION:

This may be your first use of "Artificial Intelligence" (AI) to solve a non-trivial problem using a computer. Essentially, pathing a maze using recursion requires the exploration of every possible path through the maze (this is a "brute force" search). From any position, there are 4 possible moves (north, south, east, west). Essentially, we should try one direction and then attempt to solve the (now smaller) maze from that position using a recursive step. If we succeed in finding a path from that position, then we don't need to try the other directions. On the other hand, if we fail to find a path from that first direction, then we should try another direction. If no path exists no matter which direction we step, then we have to admit that no path exists from the location in question and return a failure. Understanding recursion is likely to difficult. Leave yourself plenty of time to thoroughly explore this new programming concept.

SUMBISSION: You will need to submit your MazeCrawler.java to the Pilot drop box. Be certain to use good stylistic practices, including good commenting style. You are required to use JAVADOC style method header comments. Include your name, lecture instructor, and lab TA in your program header comments for each class.

GRADING:

Architecture [20] Correctly display a randomly generated maze using the specified architecture. Your program must utilize the code in the provided *.java files. Those files ARE NOT TO BE MODIFIED.

Solve algorithm implementation [60] Correctly uses recursion to automatically generate a path from the current path position to the exit, if such a path exists.

Style [20] Good programming style, including use of Class header, Javadoc style method headers, in-line comments as necessary, variables names, good abstraction/decomposition, and use of hierarchy/methods to avoid replication of code.

Note: *There is no valid excuse for turning in a project which does not compile. A project will receive no credit if it does not compile. Period. Partial credit can be earned on a functioning project that meets some, but not all, of the design requirements or partially solves the problem*.