Week 4: Escape the Maze!

This exercise focusses upon testing and strengthening your understanding of recursion, object references, inheritance and logic. The aim is for you to gain experience of writing more algorithmically complex code in Java while maintaining code elegance.

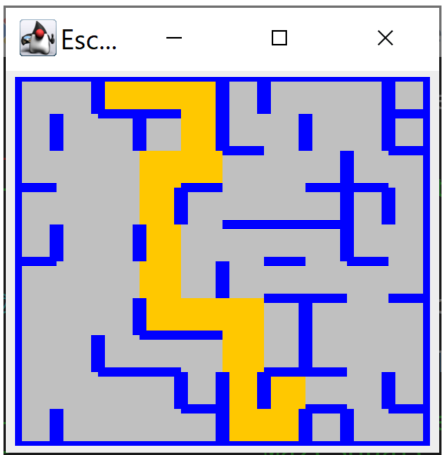
The Task

Your task this week is to implement the logic to solve and arbitrary maze using recursion.

The graphical functionality for the game has been provided for you, as has a template for an elegant object model. See **resources.zip** alongside this specification on moodle.

For this task, you may only modify the **MazeSquare** class. We have provided a basic template for this class. You are not permitted to modify the GameBoard, GameSquare or Driver classes. Restrict your work to the **MazeSquare** class alone.

Full JavaDoc documentation has also been provided for the GameBoard and GameSquare classes. Attempt to solve the task based on the documentation provided (see index.html in the doc folder). **YOU SHOULD NOT NEED TO INSPECT THE SOURCE CODE FOR THESE CLASSES**.



Requirements

You have been supplied with existing software that is capable of generating and displaying a graphical maze in Java. You have been tasked with writing code to find and display the **shortest route** from one place in the maze to another. The maze consists of 100 squares arranged in a 10x10 grid, with randomly positioned walls on the left, right, top or bottom of each square.

More specifically, your software should:

- Allow the user to define the target position in the maze by clicking the relevant square with the left mouse button.
- Allow the user to define the start position in the maze by clicking the relevant square with the right mouse button. This should also initiate the search for the **shortest** path to the target square.
- You are required to use a **recursive depth first algorithm**. This is one of the simplest ways to solve the problem (but not always the most efficient, as you learned in SCC120!). See the section below for more details if you need it.
- You **may only change the MazeSquare class**. You may not change any of the other classes. A template for this class has been provided for you. You may however, add any new code to this class as you wish.
- The shortest path from the start square to the target square should be highlighted on the maze (as illustrated above). The length of this path should also be printed out to the console using the System.out.println() method already provided in the template.

Hint: You'll need to study the methods you have inherited from the **GameSquare** class and those available in the **GameBoard** class.

Running and Testing your Code

As described earlier, much of the code required for this problem has already been written. To run the code, simply extract and compile all the .java files provided in the resources.zip file, then run the Driver class (which contains the main method):

```
javac *.java
java Driver
```

The above code will always generate the same maze. To test your code on a different maze, simply add a different level number on the command line to the java command. E.g. to run level 74:

```
javac *.java
java Driver 74
```

Recursive Depth First Search

A recursive depth first search algorithm can be used navigate from any start point in a maze, to any target point. It is also remarkably simple if you use recursion. The basic algorithm is described below. If you don't understand the idea, talk to me or one of the teaching assistants in your lab session.

1. Choose the two squares you are navigating to/from.
2. Make the start square the "current" square.
3. Choose one of the four possible directions of travel that is accessible (no wall) and has not yet been visited, and make that the current square, and so on.
4. If there are no such squares (either all walls or have been visited already), go back to the previous square and choose a different direction.
5. If you return back to the start square and there are no more possible routes to explore, then it is impossible to reach the target square.
6. If you reach the target square, success.

HINT:

Remember, for this task you are required to write a **recursive** algorithm to solve the problem… and we are **not just looking for any solution**, but the **shortest solution**… so you will need to adapt your use of the algorithm above a little. 😉

Remember to maintain your use of elegant, OO programming. I don't want some random procedural, global state algorithm from stack overflow pasted into the **MazeSquare** class. Solve it without breaking encapsulation, or you won't get many marks.

Exhaustive depth first searches can also be expensive in terms of CPU time… Once you've solved the problem, think about how you might optimize your implementation to complete faster for this specific problem.

Portfolio Contribution

As discussed in the introductory lecture, all practical work this term will contribute to your portfolio assessment.

This particular piece of work will carry marks for functionality and the creation of simple, elegant code. There will also be marks available for demonstrating the correct use of polymorphism, method overriding and recursion. This is in additional to general code style and commenting.