Assignment 3: Rat Race

In this assignment, we'll be playing with a pre-built graphical maze and the rats that navigate it. **See Moodle for the assignment package.**

RatRaceGUI.java

In the attached code, the GUI is defined and built within the **RatRaceGUI.java** file. You will also find the main method there so you can run the code.

Once you start the program, a GUI should open to a blank screen. Select a maze (either 1, 2, or 3) from the source folder, then in the dropdown select **`class IdiotRat`**. Press start to test it.

The **IdiotRat** is a concrete class that implements the **Animal** interface. That interface, provides all the methods that need to be implemented by the concrete class for it to be compatible with the maze.

Read through the **IdiotRat** file, and the class **Maze** file to get an understand of how you need to implement your methods and how to use the **Maze** object in the **move(Maze maz)** method

Note how in the move method, you iterate until you **find and perform only one move**!

You can use the following method to check if you can move to the new position before performing the move: **maz.canMove(x, y)**

After that check, modify your x and y (current row and current column) to move.

Smarter Rats

Let's make some smarter rats to run through the maze! The goal is to implement rats which can solve the maze faster (or in less moves).

Ensure that you are making these rats in new public class files within the same folder as RatRaceGUI, otherwise it won't work.

Make new rats that implement the Animal interface that are called the following, and move in the way specified:

- **WallHuggerRat**

  This rat hates going left so instead of doing things randomly, it **always goes right relative to its orientation**. The orientation can be represented by an int variable with 4 values, for example: (0, 1, 2, 3) for (Up, Down, Left, Right) facing.

  For instance, if the rat is facing upwards, then it will try to move Right. If it can't move right, then it will move Up. If it can't move Up, then it will try to go left. Finally, if it can't go Left, it will try to go Down. Remember to change the orientation as you move.

  However, if the Rat is facing Left, then it will try to move to it's relative right which is the up direction on the map. The same applies for the other orientations.

  Note that this rat will get infinitely confused and lost in some situations.

- **RandomRat**

  This rat is relatively smarter than the other two. Rather than pure randomness, it has a small amount of memory that allows it to remember where it has been. It's recommended that this memory be implemented as a single variable as you only need to remember the previous position.

  Not only that, but it's smart enough to know that it's at the end of a path (walls on 3 sides of the rat) and go back the other way or the only way available. Remember that the rat remembers that it was at the end of the path because it remembers it's previous position, so this would allow it to escape a dead-end.

  In summary, this rat will move in a random direction to a cell that is not the previous cell that the rat was at. Also, if we reach a dead end, we move to the previous cell and continue from there.

  Hint: Give your rat an orientation that can be used for checking if you're at a dead end.

- **CustomRat**

  For this rat, you have a choice of implementing the movement in one of the following ways:

  - **Breadth-First Search**

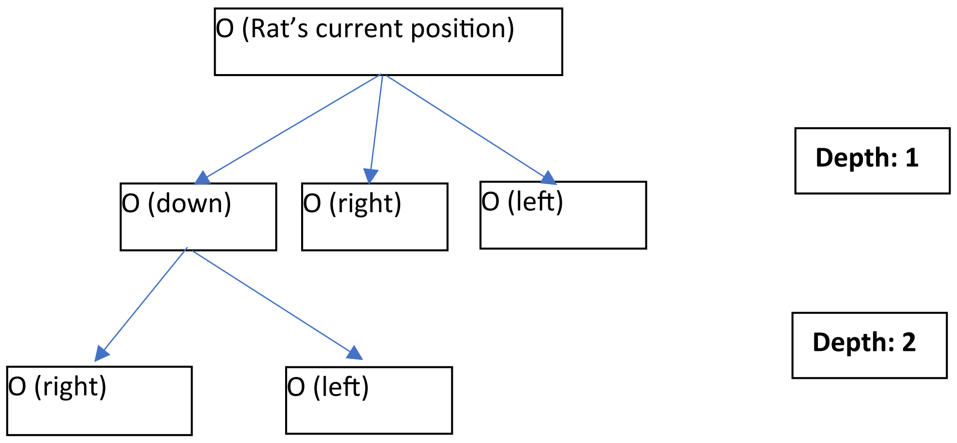
    This rat implements the Breadth-First Search technique to find it's way through the maze.

    In this method, the maze is considered to be a tree with each choice in direction being an edge to another node in the tree.

    For instance, if you have a maze like this (O is where you can move, X is a wall):

```
X X X   X X
X O R O X
X O O O X
X X   X X X
```

    If we go counter-clockwise, starting with the down direction. This would give a tree (a BFS tree) that is similar to this:



    In this case, we would visit all the nodes in the first level of Depth, add the child nodes found at each visited node to a queue, then proceed to the next element in the queue until we reach the finish cell in the maze. Before moving to a position/node found in the Queue, you need to ensure that it is next to the rat. **Note how the nodes that were already visited do not appear in the tree**. **Your rat cannot jump through walls or teleport.**

    See below Depth-First Search for a hint.

- **Depth-First Search**

  This path-finding algorithm is similar to the method above except that it goes as deep as possible in the tree before going to a neighbouring node (in contrast, breadth checks all neighbouring nodes before digging deeper). In the tree example above, we should check Right, Left, then check Down and go to Down-Right immediately after that as it's the deepest node we know of. Remember that if you are very deep in the tree, you will need to backtrack to the next neighbour to check. **Your rat cannot jump through walls or teleport.**

  Hint: See how stacks are used in DFS algorithms. BFS/DFS Trees can help with remembering how to get back to other positions.

- **BONUS-PRIZE: A* (A-star)**

  <u>This is a very difficult rat to implement, much harder than the two above. You don't need to do one of the two choices above if you implement this one.</u>

  Read about the A* algorithm online, and create a rat that implements this efficient path-finding algorithm.

Grading Criteria:

Style/submission guidelines: https://gmierzwinski.github.io/bishops/cs321/style_guidelines.html

Comments, Formatting, & Readability	**5 Marks**
Submission Guidelines	**5 Marks**
RandomRat	**10 Marks**
WallHuggerRat	**10 Marks**
CustomRat	**20 Marks**
Total	**50 Marks**