## Lab 5. Stacks

---

**Theme.**   In this lab, you will:

  -use a stack to implement a maze search

**Key concepts:**   stacks, depth-first search

**Required file(s):**   lab5.zip

---

## 1   Getting started…

1. In a web browser, download the archive `lab5.zip` from Blackboard and extract its contents into your `eclipse` workspace for this module.

2. Start up `eclipse`.

3. Making use of the contents of your extracted archive, create a new Java project named **maze_search** in your `eclipse` workspace using the contents of your extracted archive.

## 2   Search for a Path within a Maze

The project `maze_search` is an extended version of the maze search problem discussed in Unit 6. A maze is modelled as a 2D grid of cells, with each cell representing one location of the maze. A location can either be a wall or a path. The maze is modelled as a two-dimensional array of 0's and 1's, with 1 representing a path and 0 representing a wall. Finding a path within the maze means to locate a continuous sequence of cells containing the value 1 from the start to the end of the maze: To discover a path in the maze, an explorer can simply use a trial-and-error approach:



```
{ {1,1,1,0,1,1,0,0,0,1,1,1,1},
  {1,0,0,1,1,0,1,1,1,1,0,0,1},
  {1,1,1,1,1,0,1,0,1,0,1,0,0},
  {0,0,0,0,1,1,1,0,1,0,1,1,1},
  {1,1,1,0,1,1,1,0,1,0,1,1,1},
  {1,0,1,0,0,0,0,1,1,1,0,0,1},
  {1,0,1,1,1,1,1,1,0,1,1,1,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0},
  {1,1,1,1,1,1,1,1,1,1,1,1,1} }
```
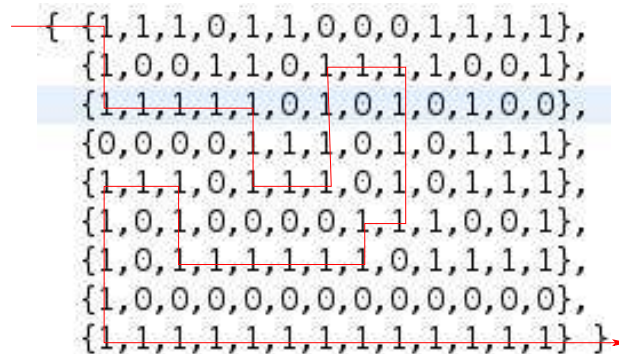
Figure 1: A path in the maze.

keep moving to an adjacent cell containing 1, at each move also take note of the alternative path(s) and visited cells (by marking each visited cell as 2); when there is no further path amongst the adjacent cells, backtrack to a recent alternative path and continue. The stack ADT can be used to keep track of:

- the alternative paths collected along the way, and

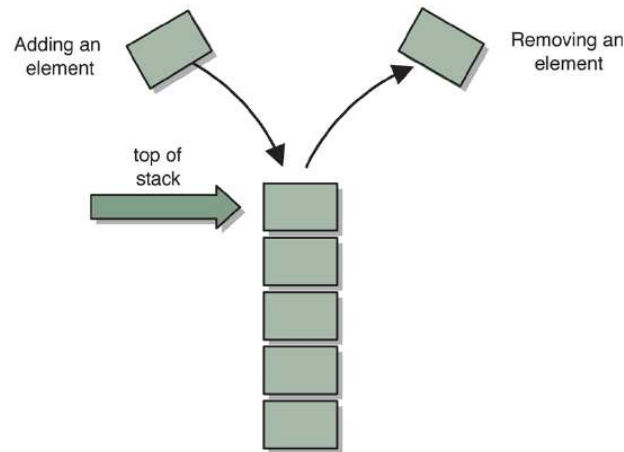- the sequence of cells which make up the path discovered so far.

---

Figure 2: Elements are added to, and removed from, the top of the stack only.

Project maze_search contains the followings:

1. Package maze contains:

   - Maze - models a maze
   - Position - models a position (cell) in a maze
   - PathFinder - models an explorer who tries to find a path within a maze which connects the entrance and the exit
   - MazeSearchDemo - a top level class for modelling a maze search demonstration

2. Package stack contains classes LinearNode, LinkedStack and StackADT as discussed in the lectures.

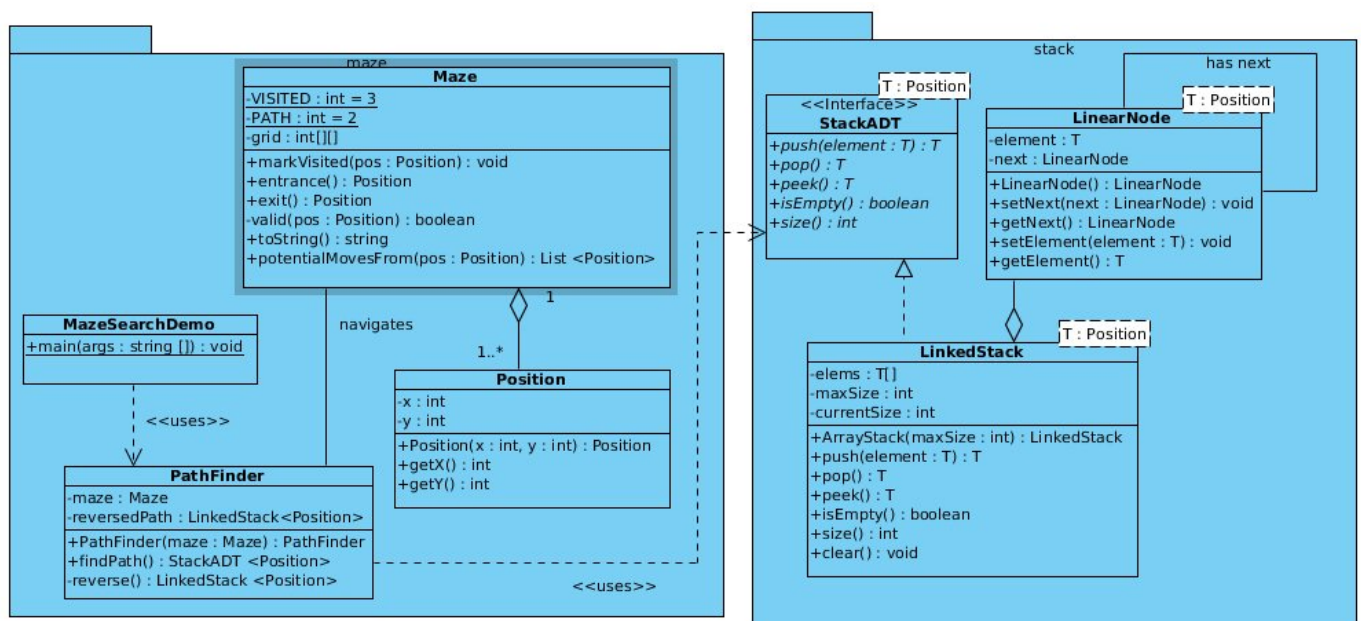The relationship between these classes is shown on the UML diagram in Figure 3.



Figure 3: A UML class diagram for the maze traversal application

Detailed implementation hints have been given throughout the source code.

> **Hint**: The rough locations where you are expected to add your Java code and relevant hints for accomplishing the tasks have been marked throughout the given Java programs. Look out for **block comments** that include a sequence of *four* exclamation marks, i.e.:
>
> ```
> /* !!!! ...      ...  */
> ```
>
> The locations where you are expected to pay particular attention on the given Java code have also been annotated. Look out for **block comments** that include a sequence of *four* plus signs, i.e.:
>
> ```
> /* ++++ ...      ...  */
> ```

**Your Task**

Complete the implementation of class `PathFinder`:

1. Constructor `PathFinder(Maze)` - initialise the fields of a new `PathFinder` object.

2. Method `reverse()` - reverses the positions in the field `reversedPath` and returns a new stack which contains the positions in a reversed order.
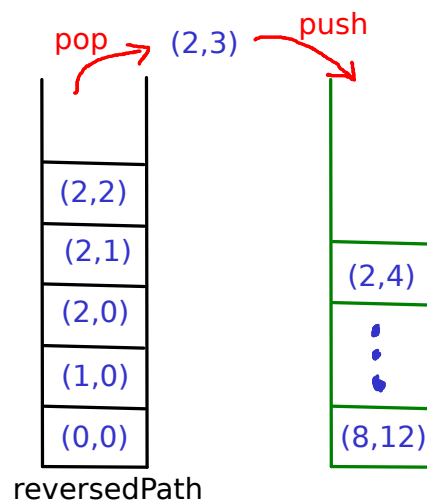


Figure 4: Reversing a path in the maze.

3. Method `StackADT<Position> findPath()` - attempts to find a path between the entrance and the exit by traversing the maze.

   During the maze traversal, this method records the positions that make up a path in a stack and marks each visited position (cell) as 'visited'.

   Unlike the simpler maze traversal algorithm considered in Unit 6, this method returns a path from entrance to exit as a stack of positions, with the top of the stack holding the location of the entrance and the bottom of the stack holding the location of the exit. If no path can be found, the method returns an empty stack.

   Like the example in the lecture, the maze traversal employs a trial-and-error approach using depth-first search.

# 3 Testing

1. You may use the `main` method in class `PathFinder` to test the correctness of your implementation. A sample output can be found in the file `PathFinder-output.txt` in this project.

2. To test the `MazeSearchDemo` application, use the given `main(String[])` method in `MazeSearchDemo`. If your implementation is correct and provided that you have not changed the given maze structure, the output should be:

```
The maze before traversal:

1110110001111
1001101111001
1111101010100
0000111010111
1110111010111
1010000111001
1011111101111
1000000000010
1111111111111


The maze after traversal:

2330110003333
2001102223003
2222202020300
0000222020333
1110333020333
1010000122003
1011111102223
1000000000020
1111111111122
```

The first grid shows the original maze layout. The second grid shows a path from the entrance to the exit (marked with 2's) and other visited positions leading to dead ends (marked with 3's).

3. Modify the maze layout slightly to make it impossible to find a path from the entrance to the exit. Run the `MazeSearchDemo` application again and observe the result.

# 4 Further Challenges

- Modify the methods `entrance` & `exit` of `Maze` so that the start and finish points need not necessarily be at the top left-hand and bottom right-hand corners of the maze.

- When no path can be found, the Maze Search application simply shows a message.

  Modify the output of the Maze Search application so as to also show the final state of the maze if no path can be found.

- Modify the method `findPath` in `PathFinder` (around line 132) so that a random move is chosen from the list of possible moves rather than always choosing the first move from the list.