Name: Jung Woo (Noel) Park
Student ID: 1162424

<u>**Etude 13 - Coroutine Maze:**</u>

**Example Solutions for Coroutine Puzzles:**

Notations:

- P0: Penny 0 starting at ‘0’ marker.
- P1: Penny 1 starting at ‘1’ marker.
- Directions (the directions are based on a 3x3 matrix):
    - N: up
    - S: down
    - E: right
    - W: left
    - NE: top right
    - NW: top left
    - SE: bottom right
    - SW: bottom left

| **Moves** | **Coroutine Puzzle 1** | | **Moves** | **Coroutine Puzzle 4** | |
|---|---|---|---|---|---|
| | **Penny** | **Direction** | | **Penny** | **Direction** |
| 1 | P0 | S | 1 | P0 | S |
| - | P1 | PASS | 2 | P1 | W |
| 2 | P0 | S | 3 | P0 | N |
| 3 | P1 | W | 4 | P1 | W |
| 4 | P0 | N | 5 | P0 | E |
| 5 | P1 | NE | - | P1 | PASS |
| 6 | P0 | N | 6 | P0 | E |
| 7 | P1 | N | 7 | P1 | N |
| 8 | P0 | S | 8 | P0 | W |
| - | P1 | PASS | 9 | P1 | SE |
| 9 | P0 | S | 10 | P0 | SE |
| 10 | P1 | W | 11 | P1 | E |
| - | P0 | PASS | 12 | P0 | N |

Name: Jung Woo (Noel) Park
Student ID: 1162424

| | | | | | |
|---|---|---|---|---|---|
| 11 | P1 | W | 13 | P1 | N |
| 12 | P0 | N | - | P0 | PASS |
| 13 | P1 | E | 14 | P1 | W |
| 14 | P0 | SE | | | |
| 15 | P1 | SE | | | |
| 16 | P0 | N | | | |

Table 1: Some Answers.

**How to find these solutions:**

If we look for the solution manually, there are four things we must keep track of:

- Move count for checking if our solution mets the puzzle move criteria.
- Penny associated with the move count.
- Direction a penny moved.
- A list of all alternative moves.

Here is an example of the process:

1) We place the pennies at 0 and 1, and start our move with penny 0.
2) We look at all valid moves using the routes underneath penny 1.
3) We make a move for each valid move.
4) If there are more than one valid move, then copy the current sequence and start a alternative sequence to track the alternative move we took.
5) Otherwise if there are no valid moves, we pass and record the pass in our list of moves.
6) We repeat the same process for penny 1’s move.
7) We keep repeating these steps until a penny in one of the list of moves we are tracking reaches the center of the puzzle. That becomes the solution like in  table 1.

**Data Structure:**

The coroutine puzzle solver has 6 main data structures for representing and solving the puzzle:

- Penny Class: Represents the pennies we place down on the puzzle board.
    - The penny class has a name field which stores the name of the penny (ie. penny 0).

- ○ The penny class has a position p with x and y coordinates representing the location in the matrix index space (so rows (y) ranges from 0-2 and columns ranges from 0-2 ).
- Move Class: Represents a particular move a penny made for a given turn.
  - The move class contains a enumerated type of all moves (N, S, E, W, NE, NW, SE, SW). All enumerated instances contains an offset in relation with the 3x3 matrix coordinate system that represents the board.
  - The move class has a string ‘name’ field to store the direction the penny moved. This will be later useful for printing all moves each penny made for the solution.
  - The move class has two fields to store the current position of the pennies after each move instance.
- Spots Class: Represents a spot on the puzzle board.
  - The spot class has a string array of moves that holds the different directions a penny can make. The values are labelled like the moves enumerated type in the move class.
  - The spot class also has a position which representation a location on the board. This is not used in our search but very useful for debugging and knowing if spots have been correctly assigned the puzzle.
  - The spot class has a computeValidMoves method that computes all the positions a penny can move. The validity of a move is determined by checking if the new position is inside the matrix bounding box (positive integers 0-2), and if the new position won’t overlap with the position of the other penny.
- ListedLinkDeque:
  - The Java ListedLinkDeque is a doubly listed link queue. This is used to keep track of all the sequence of alternative moves. With the listed link queue, we are able to always query the last move made for computing the next valid move. We always need the last move because we need to retrieve the position of the other penny for computing valid moves. The queue also guarantees a FIFO principle so it will maintain the order of moves made.
- PriorityQueue:
  - The Java Priority queue is used for deciding a sequence of moves we want to continue searching. For example if we have a list of alternative move sequences, we need some way to search all the recent list of moves, so the priority queue will priorities the list of moves that has the smallest length (lower moveCount). This way we know that all list of moves will be updated in parallel.

- Maze Class: Represents the puzzle board.
    - The maze class has a 3x3 matrix of spots. This stores the puzzles and their possible moves.
    - The maze class also has a reference to both pennies on the board. This initial instance will be used when we start our search.
    - The maze class also has breadth first search based search function for solving the coroutine puzzle.

**Algorithm:**

The search algorithm is based on the breadth first search. It uses breadth first search in order to keep track of all alternative list of moves in parallel. The algorithm works like described below:

Initialize a priority queue for picking the list of moves with lowest move count.
Initialize the first move:

- Initialize the linked list queue.
- Add the starting move to the new instance of the listed linked queue.
- Push new instance list of moves to priority queue.

While the priority queue is not empty:

- Pop the list of moves with the priority of lowest move count.
- Determine the new moveCount -> length of list of moves - 1.
- Extract pennies of most recent move from the current list of moves.
- If penny zero or one is finished (at position 1,1)
    - Print out list of moves.
    - Break from while loop.
- Declare list of valid moves variable.
- If moveCount is even (even moveCounts means penny 0’s turn):
    - Obtain list of all valid moves by:
        - Querying the position of penny 1 and obtain the corresponding spot that contains corresponding possible directions.
    - If the list of valid moves are empty:
        - Make deep copy of the current list of moves.
        - Make deep copy of penny zero and one.
        - Add a new move to the deep copy of list of moves:
            - Passing the string “passed”, the two new instances of penny zero, one and an isZeroMove flag (true) as the parameter of the new move.
        - Add the new list of moves to the priority queue.
- Else if is odd (odd moveCounts means penny 1’s turn):
    - Obtain list of all valid moves by:

```
            - Querying the position of penny 0 and obtain the corresponding
              spot that contains corresponding possible directions.
        - If the list of valid moves are empty:
            - Make deep copy of the list of moves
            - Make deep copy of penny zero and one.
            - Add a new move to the deep copy of list of moves:
                - Passing the string “passed”, the two new instances of
                  penny zero, one and an isZeroMove flag (false) as the
                  parameter of the new move.
            - Add the new list of moves to the priority queue.
    - Else if the list of valid moves is not empty:
        - For all valid moves:
            - Make a deep copy of the previous list of moves
            - Add the new valid move to the new deep copy of list of moves.
            - Add the new list of moves to the priority queue.
Repeat while loop ...
```

**Algorithm for generating new puzzles:**

A simple algorithm for generating new puzzle could go like this:

1) Assign random list of directions to each spot on the puzzle.
2) Use the bread first search to see if the new puzzle is valid.
3) If the search cannot find a solution then we quit and try generate a new puzzle.
4) Else if the search finds a solution, print the numbers of moves, list of moves and the puzzle it generated.