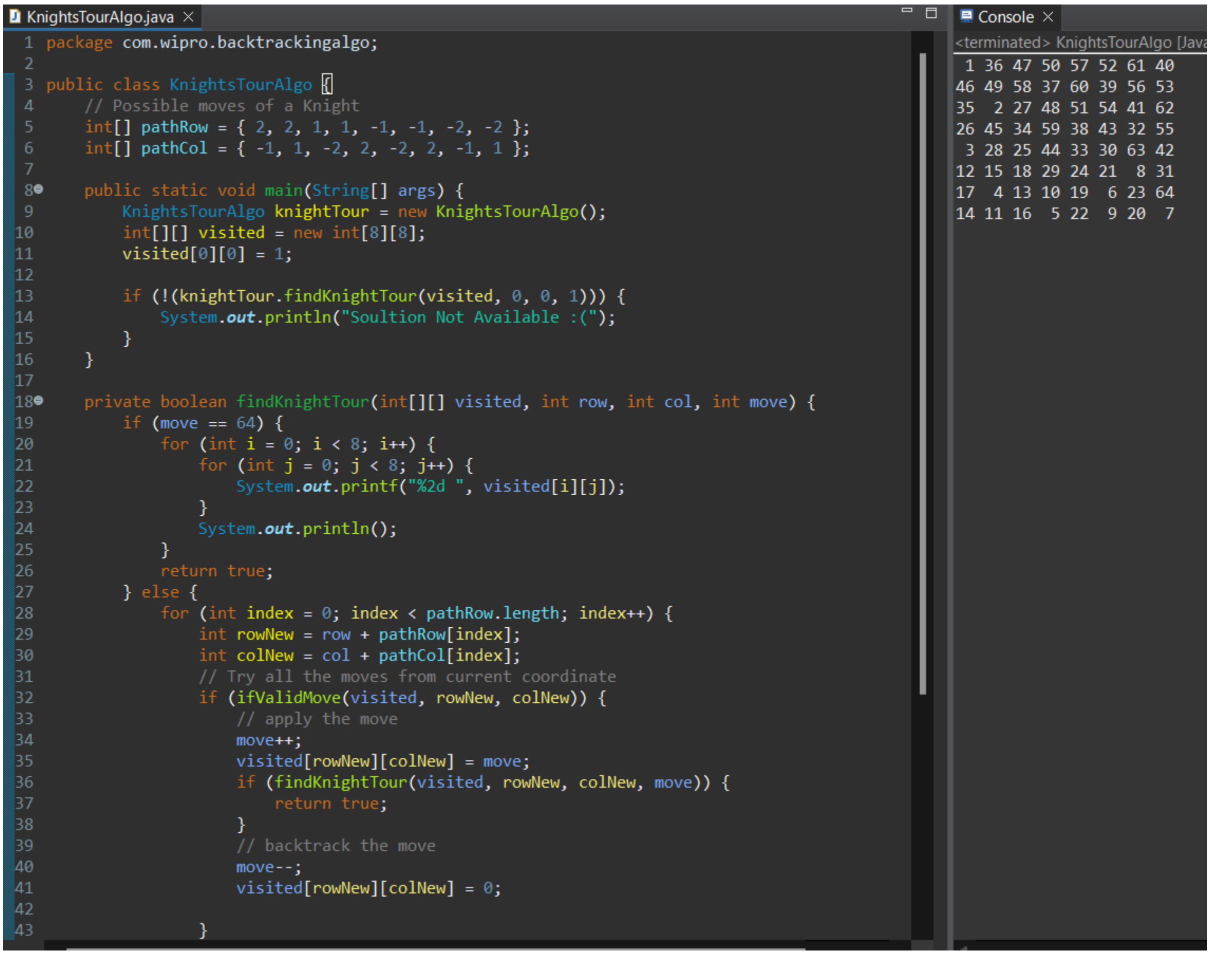
DAY-16 & 17 JAVA ASSIGNMENT

Day 16 and 17:

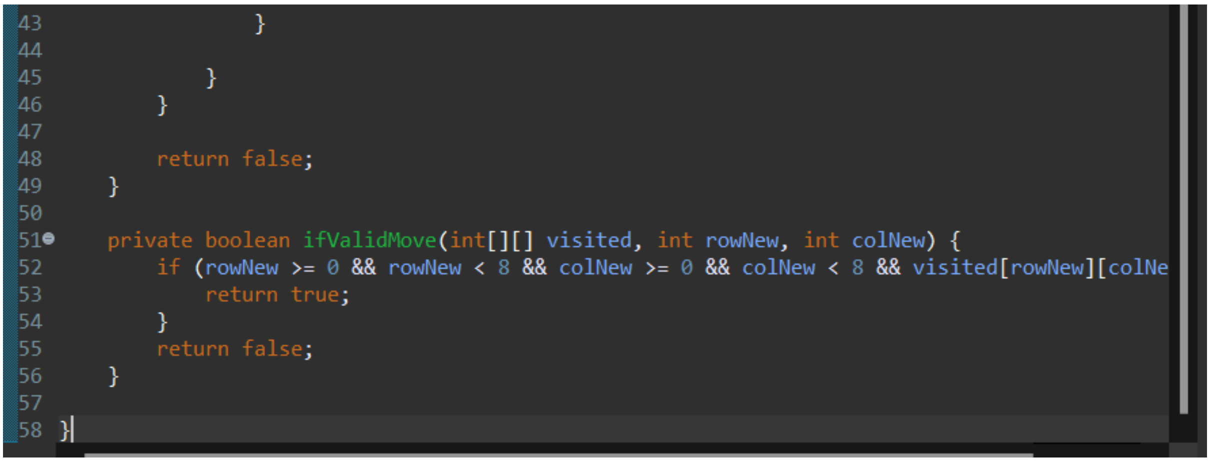
Task 1: The Knight's Tour Problem

Create a function bool SolveKnightsTour(int[,] board, int moveX, int moveY, int moveCount, int[] xMove, int[] yMove) that attempts to solve the Knight's Tour problem using backtracking. The function should return true if a solution exists and false otherwise. The board represents the chessboard, moveX and moveY are the current coordinates of the knight, moveCount is the current move count, and xMove[], yMove[] are the possible next moves for the knight. Fill the chessboard such that the knight visits every square exactly once. Keep the chessboard size to 8x8.



```java
package com.wipro.backtrackingalgo;

public class KnightsTourAlgo {
    // Possible moves of a Knight
    int[] pathRow = { 2, 2, 1, 1, -1, -1, -2, -2 };
    int[] pathCol = { -1, 1, -2, 2, -2, 2, -1, 1 };

    public static void main(String[] args) {
        KnightsTourAlgo knightTour = new KnightsTourAlgo();
        int[][] visited = new int[8][8];
        visited[0][0] = 1;

        if (!(knightTour.findKnightTour(visited, 0, 0, 1))) {
            System.out.println("Soultion Not Available :(");
        }
    }

    private boolean findKnightTour(int[][] visited, int row, int col, int move) {
        if (move == 64) {
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                    System.out.printf("%2d ", visited[i][j]);
                }
                System.out.println();
            }
            return true;
        } else {
            for (int index = 0; index < pathRow.length; index++) {
                int rowNew = row + pathRow[index];
                int colNew = col + pathCol[index];
                // Try all the moves from current coordinate
                if (ifValidMove(visited, rowNew, colNew)) {
                    // apply the move
                    move++;
                    visited[rowNew][colNew] = move;
                    if (findKnightTour(visited, rowNew, colNew, move)) {
                        return true;
                    }
                    // backtrack the move
                    move--;
                    visited[rowNew][colNew] = 0;

                }
```

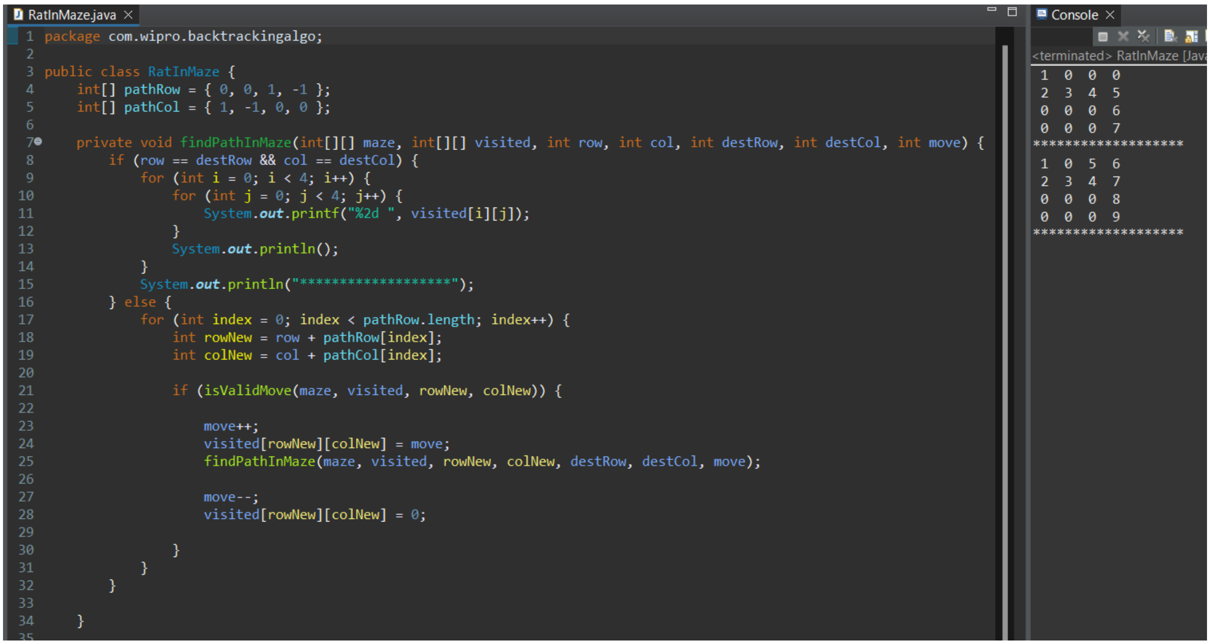
```
<terminated> KnightsTourAlgo [Java
 1 36 47 50 57 52 61 40
46 49 58 37 60 39 56 53
35  2 27 48 51 54 41 62
26 45 34 59 38 43 32 55
 3 28 25 44 33 30 63 42
12 15 18 29 24 21  8 31
17  4 13 10 19  6 23 64
14 11 16  5 22  9 20  7
```



```
                }

            }
        }

        return false;
    }

    private boolean ifValidMove(int[][] visited, int rowNew, int colNew) {
        if (rowNew >= 0 && rowNew < 8 && colNew >= 0 && colNew < 8 && visited[rowNew][colNe
            return true;
        }
        return false;
    }

}
```

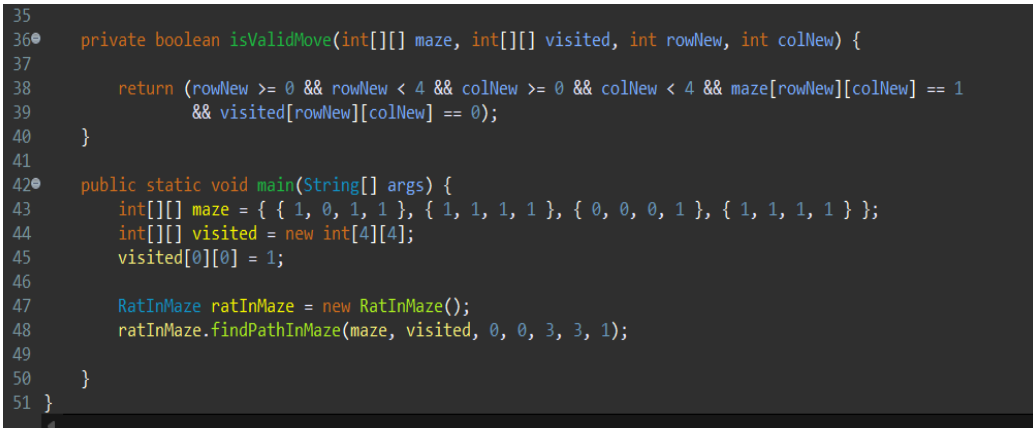
Task 2: Rat in a Maze

mplement a function bool SolveMaze(int[,] maze) that uses backtracking to find a path from the top left corner to the bottom right corner of a maze. The maze is represented by a 2D array where 1s are paths and 0s are walls. Find a rat's path through the maze. The maze size is 6x6.



```java
package com.wipro.backtrackingalgo;

public class RatInMaze {
    int[] pathRow = { 0, 0, 1, -1 };
    int[] pathCol = { 1, -1, 0, 0 };

    private void findPathInMaze(int[][] maze, int[][] visited, int row, int col, int destRow, int destCol, int move) {
        if (row == destRow && col == destCol) {
            for (int i = 0; i < 4; i++) {
                for (int j = 0; j < 4; j++) {
                    System.out.printf("%2d ", visited[i][j]);
                }
                System.out.println();
            }
            System.out.println("*******************");
        } else {
            for (int index = 0; index < pathRow.length; index++) {
                int rowNew = row + pathRow[index];
                int colNew = col + pathCol[index];

                if (isValidMove(maze, visited, rowNew, colNew)) {

                    move++;
                    visited[rowNew][colNew] = move;
                    findPathInMaze(maze, visited, rowNew, colNew, destRow, destCol, move);

                    move--;
                    visited[rowNew][colNew] = 0;

                }
            }
        }

    }
```

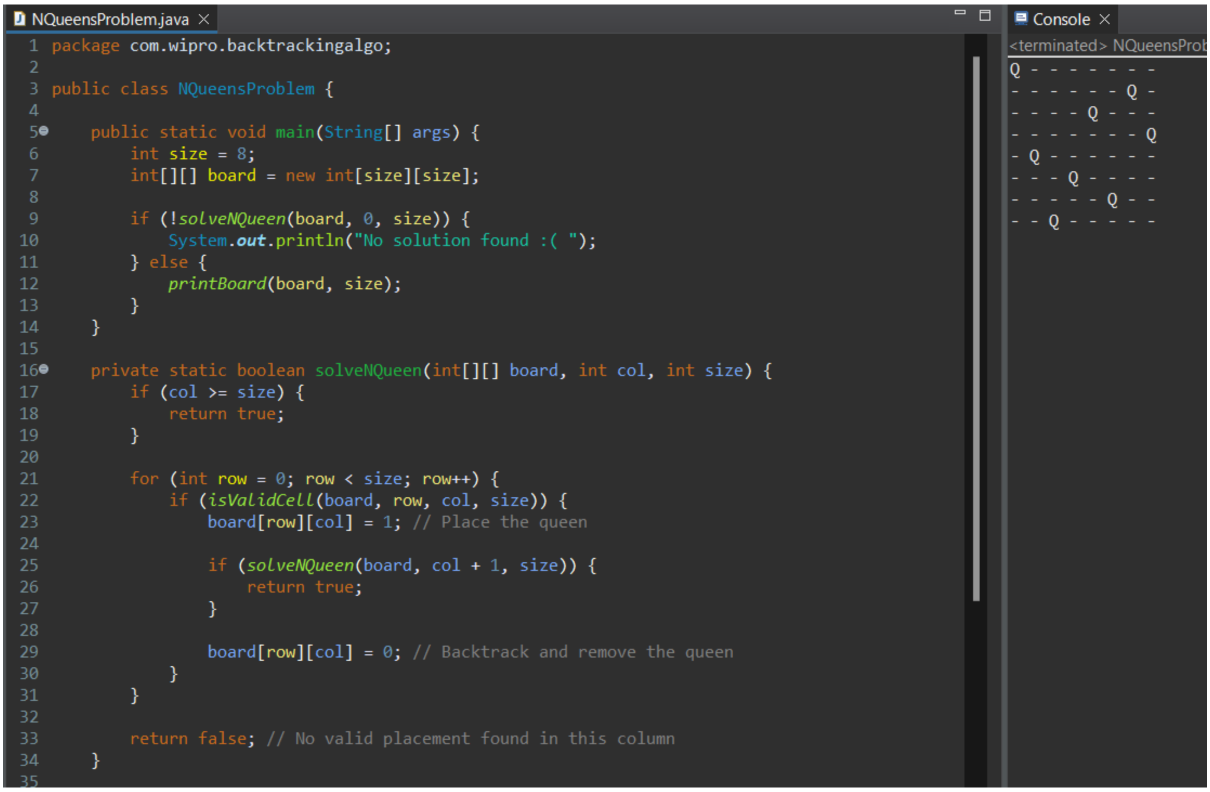
```
 1  0  0  0 
 2  3  4  5 
 0  0  0  6 
 0  0  0  7 
*******************
 1  0  5  6 
 2  3  4  7 
 0  0  0  8 
 0  0  0  9 
*******************
```



```java

    private boolean isValidMove(int[][] maze, int[][] visited, int rowNew, int colNew) {

        return (rowNew >= 0 && rowNew < 4 && colNew >= 0 && colNew < 4 && maze[rowNew][colNew] == 1
                && visited[rowNew][colNew] == 0);
    }

    public static void main(String[] args) {
        int[][] maze = { { 1, 0, 1, 1 }, { 1, 1, 1, 1 }, { 0, 0, 0, 1 }, { 1, 1, 1, 1 } };
        int[][] visited = new int[4][4];
        visited[0][0] = 1;

        RatInMaze ratInMaze = new RatInMaze();
        ratInMaze.findPathInMaze(maze, visited, 0, 0, 3, 3, 1);

    }
}
```

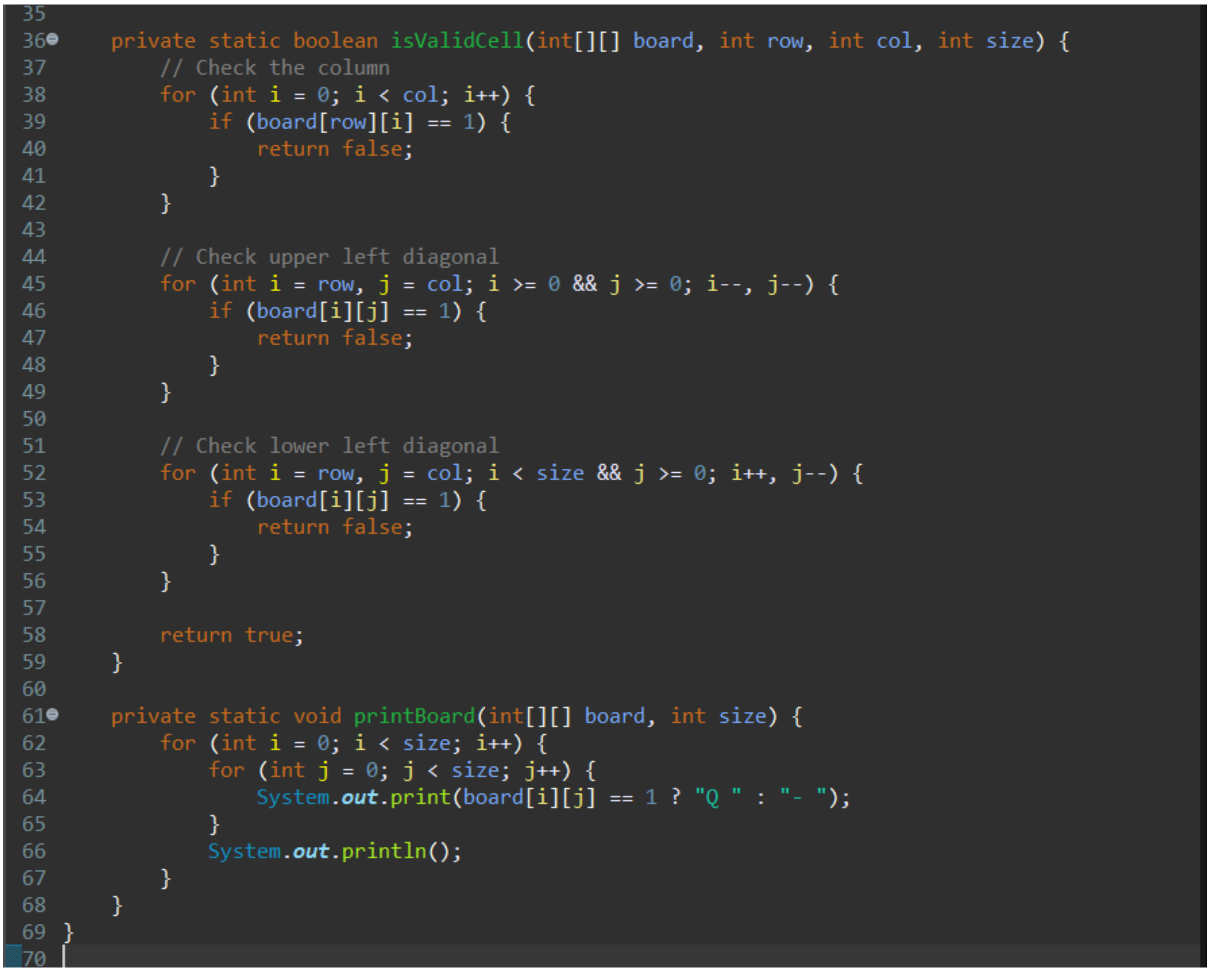
Task 3: N Queen Problem

Write a function bool SolveNQueen(int[,] board, int col) in C# that places N queens on an N x N chessboard so that no two queens attack each other using backtracking. Place N queens on the board such that no two queens can attack each other. Use a standard 8x8 chessboard.



```java
package com.wipro.backtrackingalgo;

public class NQueensProblem {

    public static void main(String[] args) {
        int size = 8;
        int[][] board = new int[size][size];

        if (!solveNQueen(board, 0, size)) {
            System.out.println("No solution found :( ");
        } else {
            printBoard(board, size);
        }
    }

    private static boolean solveNQueen(int[][] board, int col, int size) {
        if (col >= size) {
            return true;
        }

        for (int row = 0; row < size; row++) {
            if (isValidCell(board, row, col, size)) {
                board[row][col] = 1; // Place the queen

                if (solveNQueen(board, col + 1, size)) {
                    return true;
                }

                board[row][col] = 0; // Backtrack and remove the queen
            }
        }

        return false; // No valid placement found in this column
    }
```

```
Q - - - - - - -
- - - - - - Q -
- - - - Q - - -
- - - - - - - Q
- Q - - - - - -
- - - Q - - - -
- - - - - Q - -
- - Q - - - - -
```



```java

    private static boolean isValidCell(int[][] board, int row, int col, int size) {
        // Check the column
        for (int i = 0; i < col; i++) {
            if (board[row][i] == 1) {
                return false;
            }
        }

        // Check upper left diagonal
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        // Check lower left diagonal
        for (int i = row, j = col; i < size && j >= 0; i++, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        return true;
    }

    private static void printBoard(int[][] board, int size) {
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                System.out.print(board[i][j] == 1 ? "Q " : "- ");
            }
            System.out.println();
        }
    }
}

```