- ☒ **Operating systems 2**
- ☒ **Project 5: Make a square**

Description:

project is a program that is supposed to take a set of pieces and use them to form a 4x4 square without any gaps, it should automatically rotate or flip the pieces or keep them on their original state to form that perfect 4x4 square. The inputs are given as:

1- **number of rows and columns that specify the piece's shape as the first input.**
2- **Represent the piece by putting 1 as the solid part of the piece and 0 as the place holder as the second input. Example on the input:**

```
2 3
101
111
```

The program should find a way to combine those pieces together to form the square and represent each given piece by its number (piece number 1 is represented by '1' and piece number 2 is represented by '2'). Example on the output:

```
1112
1412
3422
3442
```

Steps:

- **We used a 2D arrays to specify the pieces and the grid (the square), we used object oriented programming and Java threads to implement this project.**

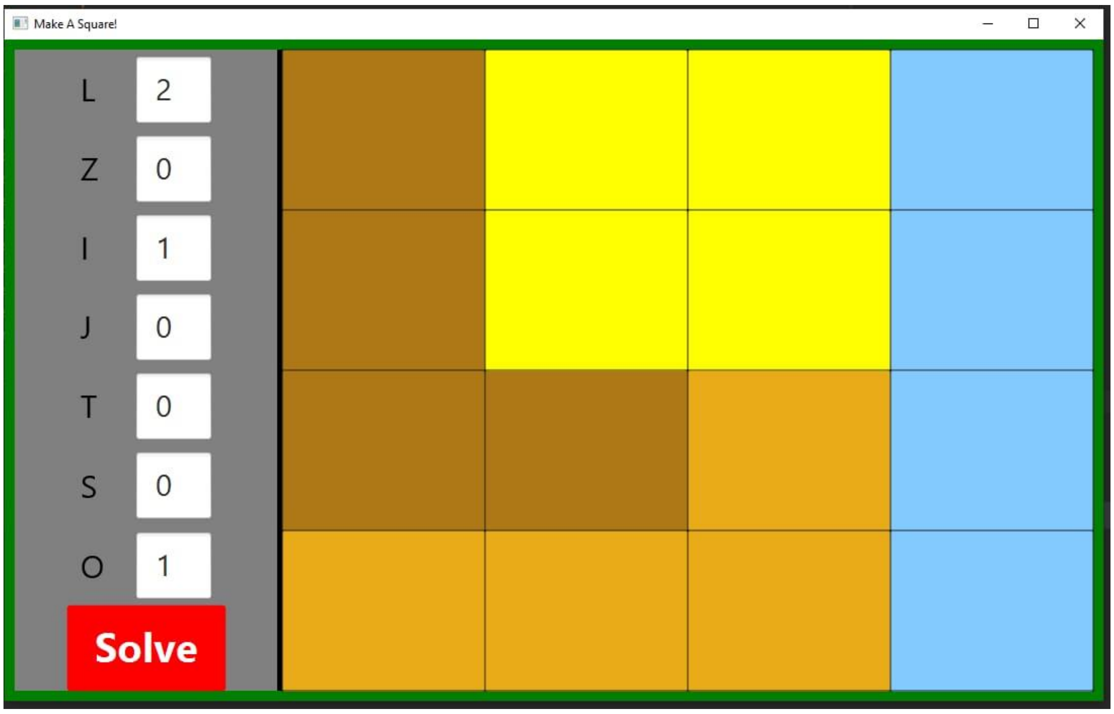
- **The idea is that the number of solutions is divided on the 5 threads which are running in parallel so that when one of the threads find the solution it interrupts the rest of the threads.**

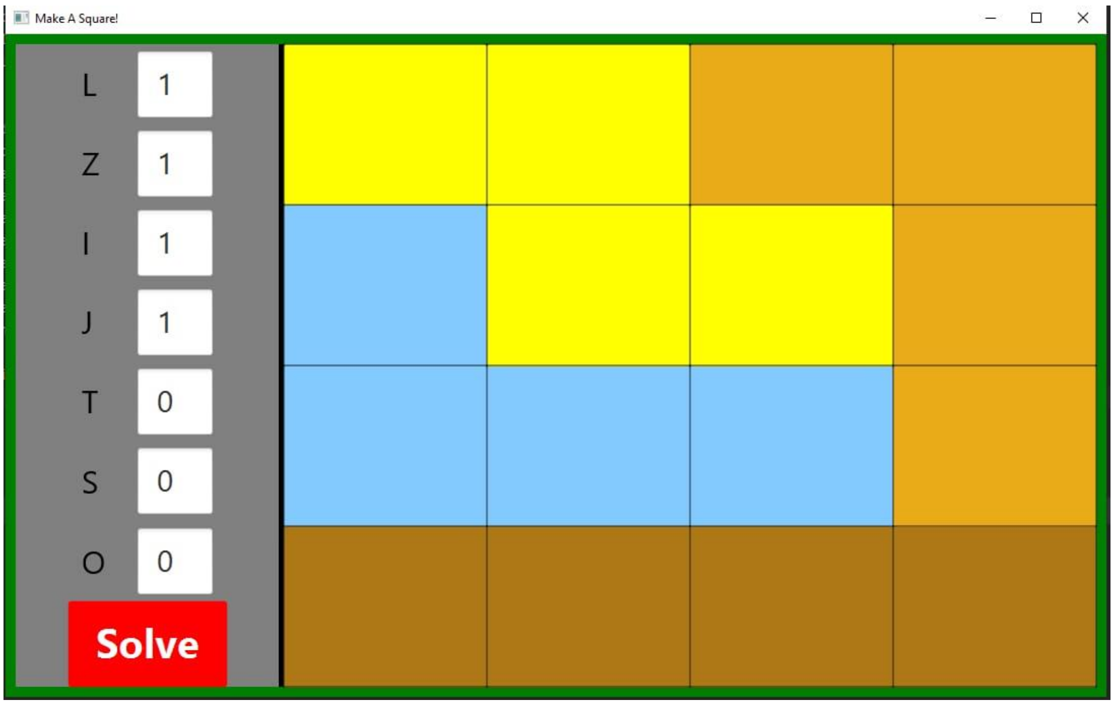
- **We take all the possible combinations of pieces starting from piece A1A1A1A1 which is the first possible combination to E4E4E4E4E4 which is the last possible combination in our project,** between those two we have different combinations that we can actually use like A1B3C4D0E2, B4A0D3E5C2.

- Each piece is then represented by a binary number consists of 5 bits, 3 bits for the piece and 2 bits for the move id which works with this piece on the board .

Example:

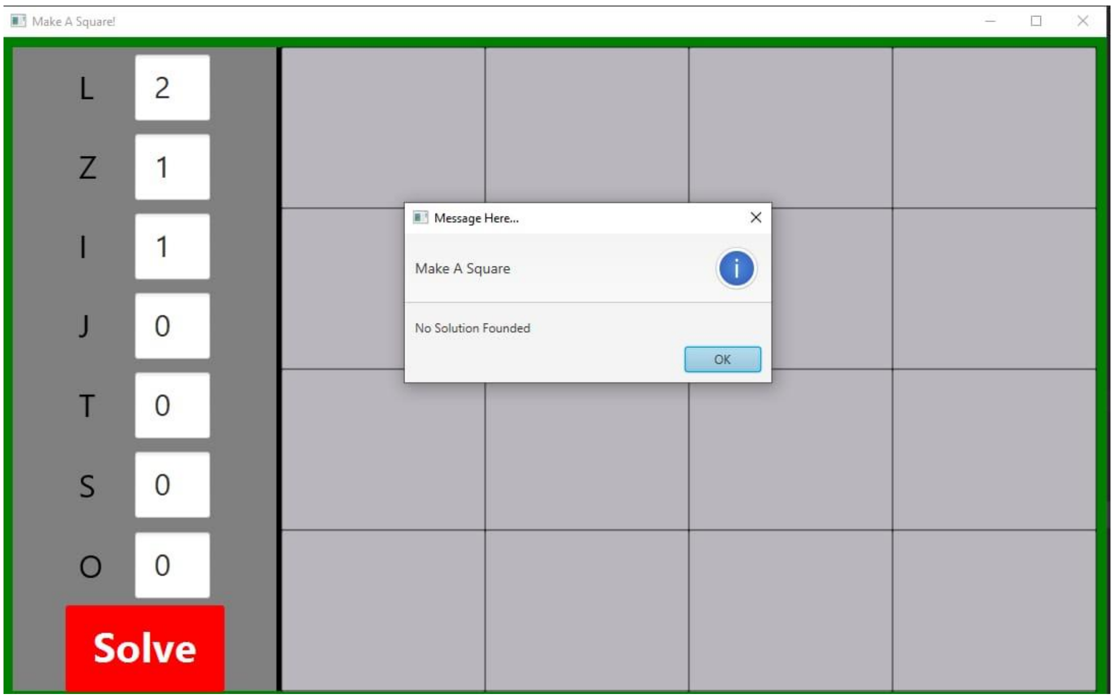
Each piece is given to the program by writing how many pieces of the same shape we need in the text box beside the shape of the piece in the GUI.





Example:

IF the program doesn’t find a way to combine the pieces, it prints no solution found



Make A Square

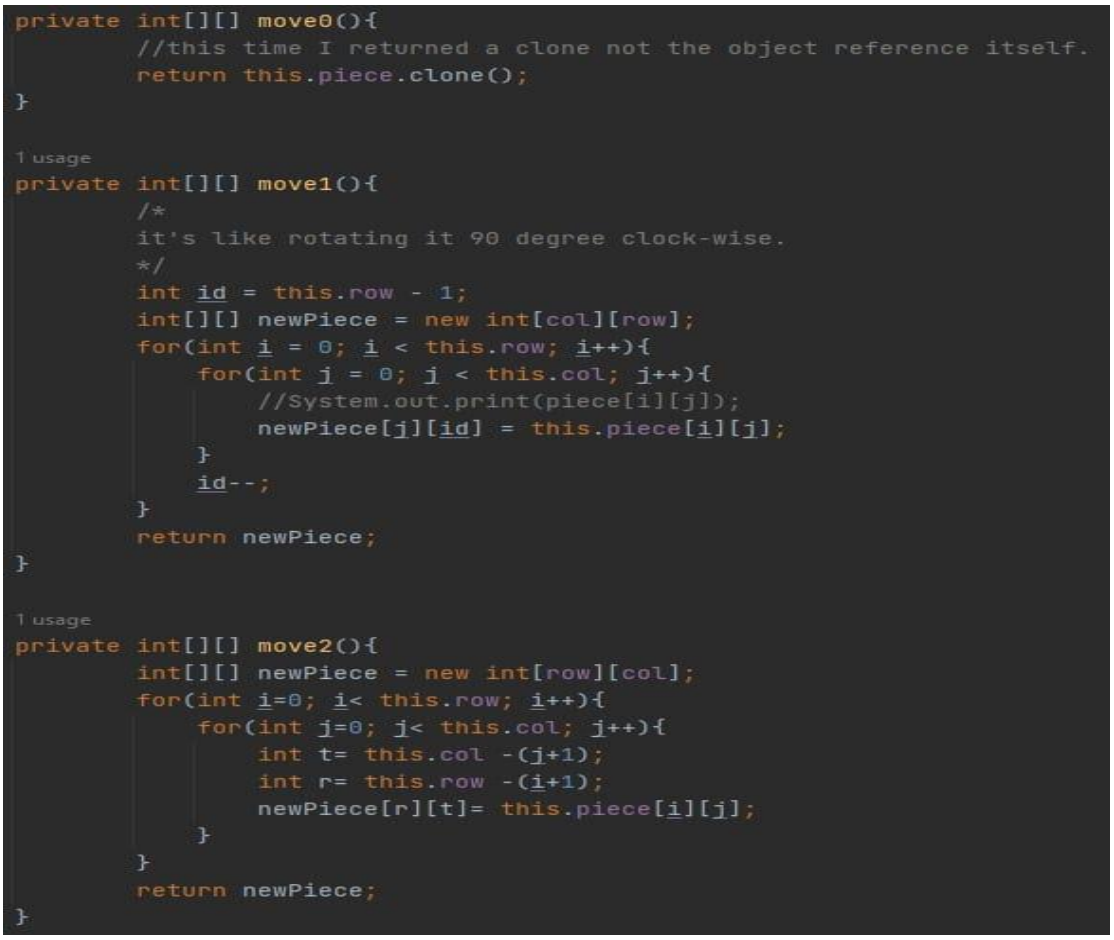
No Solution Founded

Code documentation:

Move methods:

We have created a punch of private methods to deal with rotations inside the class either rotating the piece 90, 180, 270 degrees or to keep it on its original

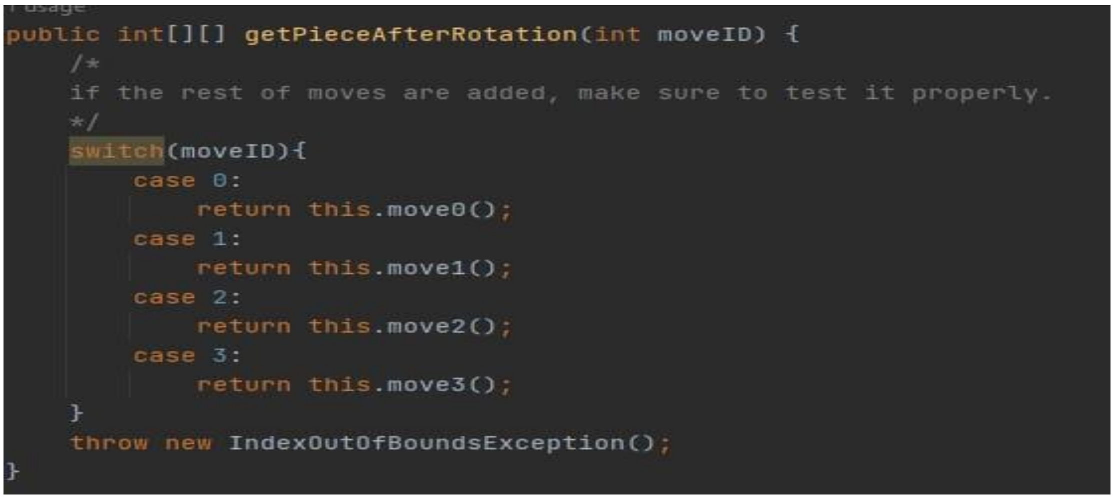
State (cloning the piece) and pass the new pieces after rotations to the public method getPieceAfterRotation(). Example on rotations code:



```
private int[][] move0(){
        //this time I returned a clone not the object reference itself.
        return this.piece.clone();
}

1 usage
private int[][] move1(){
        /*
        it's like rotating it 90 degree clock-wise.
        */
        int id = this.row - 1;
        int[][] newPiece = new int[col][row];
        for(int i = 0; i < this.row; i++){
            for(int j = 0; j < this.col; j++){
                //System.out.print(piece[i][j]);
                newPiece[j][id] = this.piece[i][j];
            }
            id--;
        }
        return newPiece;
}

1 usage
private int[][] move2(){
        int[][] newPiece = new int[row][col];
        for(int i=0; i< this.row; i++){
            for(int j=0; j< this.col; j++){
                int t= this.col -(j+1);
                int r= this.row -(i+1);
                newPiece[r][t]= this.piece[i][j];
            }
        }
        return newPiece;
}
```

getPieceAfterRotation():

This method contains a switch case which takes the move id and returns the new piece resulted from this move.

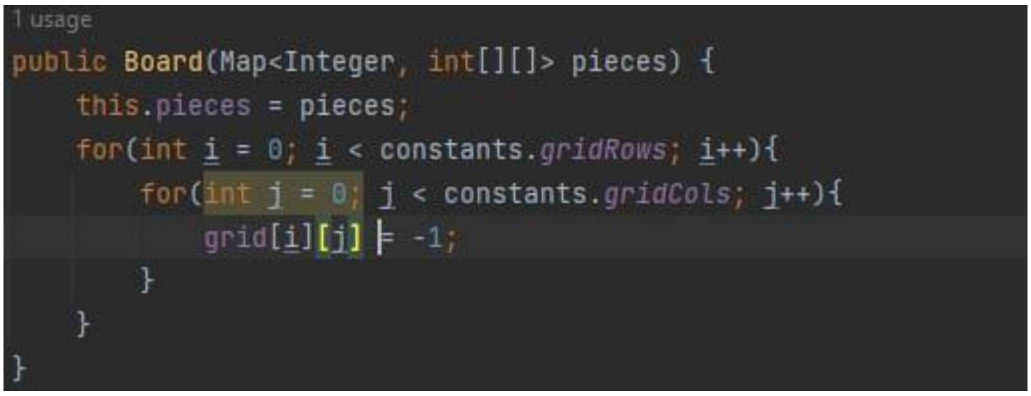


```
public int[][] getPieceAfterRotation(int moveID) {
    /*
    if the rest of moves are added, make sure to test it properly.
    */
    switch(moveID){
        case 0:
            return this.move0();
        case 1:
            return this.move1();
        case 2:
            return this.move2();
        case 3:
            return this.move3();
    }
    throw new IndexOutOfBoundsException();
}
```

<u>Board</u>:

<u>Board class and constructor</u>:

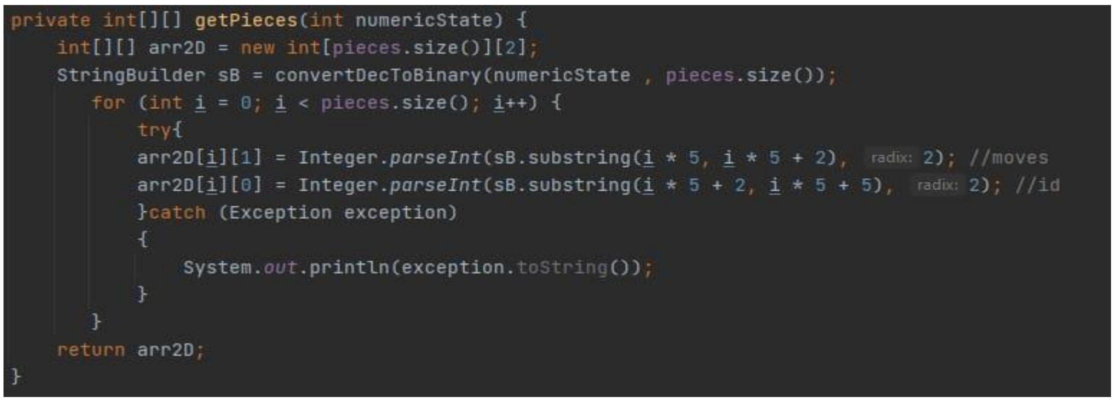
We created a board class and initialized our board (grid), specified its dimensions and we created a HashMap for pieces on the board.



```
1 usage
public Board(Map<Integer, int[][]> pieces) {
    this.pieces = pieces;
    for(int i = 0; i < constants.gridRows; i++){
        for(int j = 0; j < constants.gridCols; j++){
            grid[i][j] = -1;
        }
    }
}
```

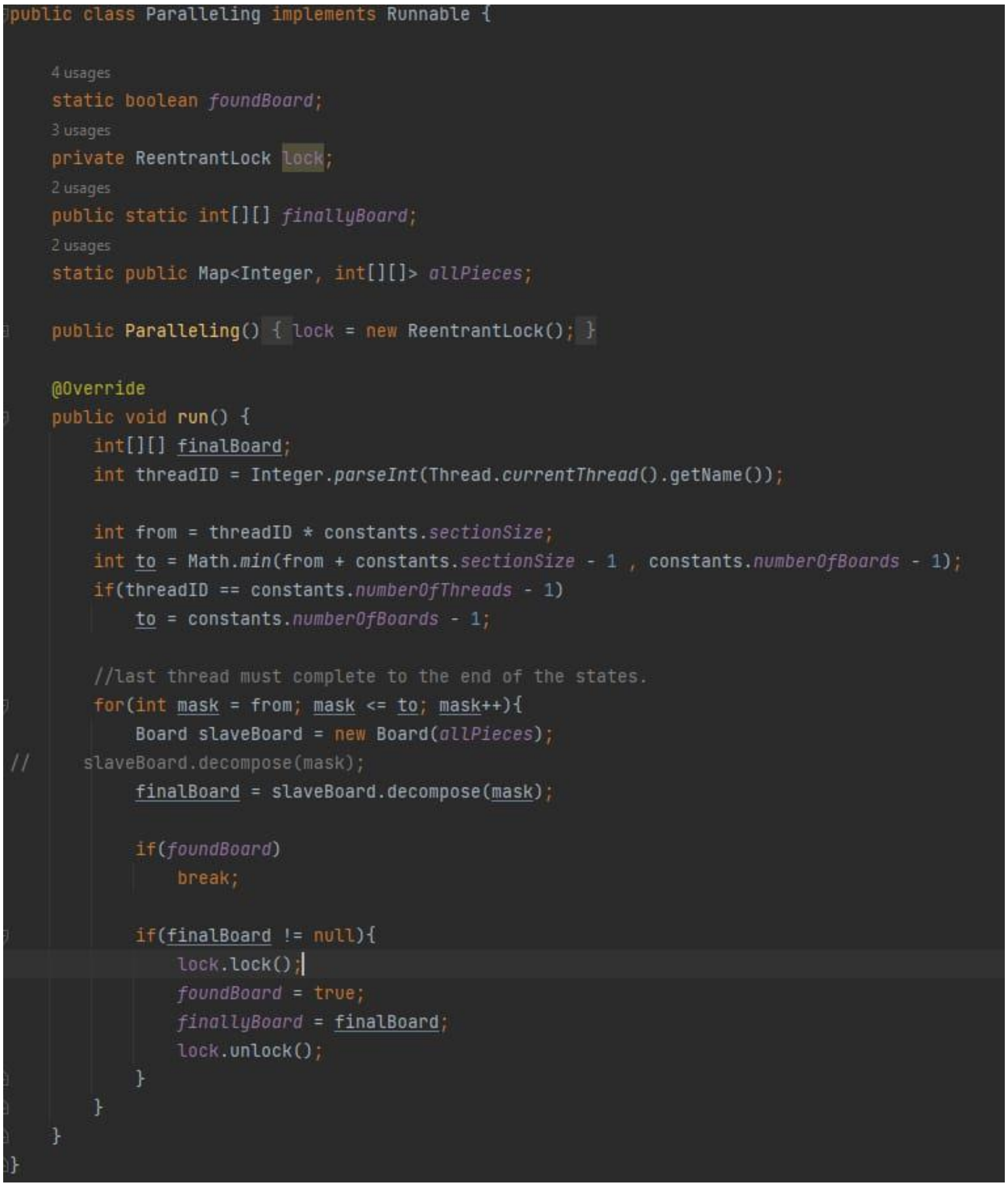
<u>getPieces():</u>

get pieces method returns 2D array that specifies the id and the move of each piece used in a specific combination.



```java
private int[][] getPieces(int numericState) {
    int[][] arr2D = new int[pieces.size()][2];
    StringBuilder sB = convertDecToBinary(numericState , pieces.size());
        for (int i = 0; i < pieces.size(); i++) {
            try{
            arr2D[i][1] = Integer.parseInt(sB.substring(i * 5, i * 5 + 2),  radix: 2); //moves
            arr2D[i][0] = Integer.parseInt(sB.substring(i * 5 + 2, i * 5 + 5),  radix: 2); //id
            }catch (Exception exception)
            {
                System.out.println(exception.toString());
            }
        }
    return arr2D;
}
```

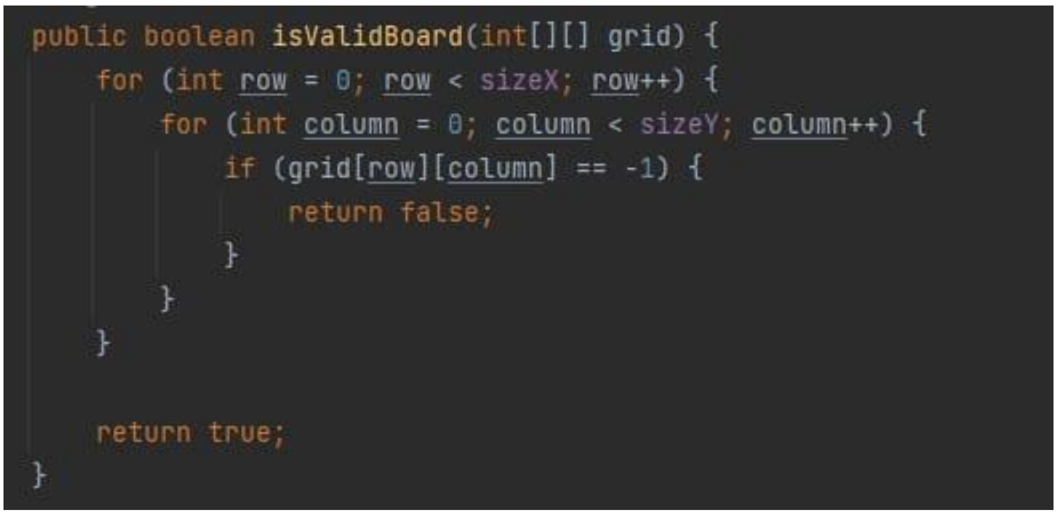
Threads:



```java
public class Paralleling implements Runnable {

    4 usages
    static boolean foundBoard;
    3 usages
    private ReentrantLock lock;
    2 usages
    public static int[][] finallyBoard;
    2 usages
    static public Map<Integer, int[][]> allPieces;

    public Paralleling() { lock = new ReentrantLock(); }

    @Override
    public void run() {
        int[][] finalBoard;
        int threadID = Integer.parseInt(Thread.currentThread().getName());

        int from = threadID * constants.sectionSize;
        int to = Math.min(from + constants.sectionSize - 1 , constants.numberOfBoards - 1);
        if(threadID == constants.numberOfThreads - 1)
            to = constants.numberOfBoards - 1;

        //last thread must complete to the end of the states.
        for(int mask = from; mask <= to; mask++){
            Board slaveBoard = new Board(allPieces);
//      slaveBoard.decompose(mask);
            finalBoard = slaveBoard.decompose(mask);

            if(foundBoard)
                break;

            if(finalBoard != null){
                lock.lock();
                foundBoard = true;
                finallyBoard = finalBoard;
                lock.unlock();
            }
        }
    }
}
```

IsValidBoard():

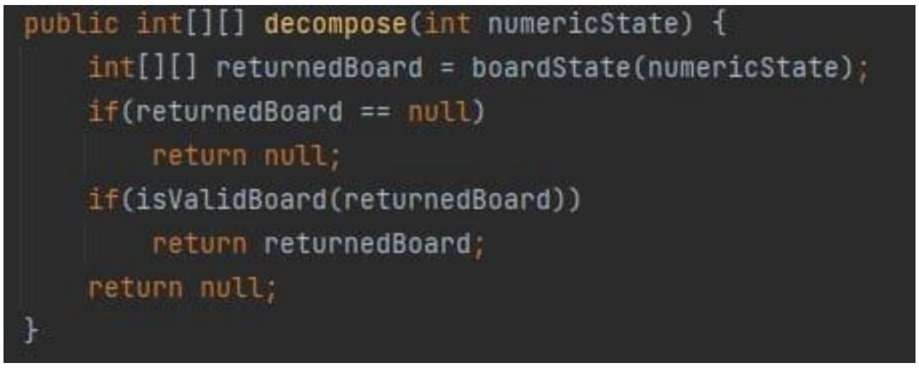
This method makes sure that the state board passed to it doesn't cross the boundaries of the grid so that no combination is having 5 rows or 5 columns, the grid must always be 4x4 square.



```java
public boolean isValidBoard(int[][] grid) {
    for (int row = 0; row < sizeX; row++) {
        for (int column = 0; column < sizeY; column++) {
            if (grid[row][column] == -1) {
                return false;
            }
        }
    }

    return true;
}
```

Decompose():

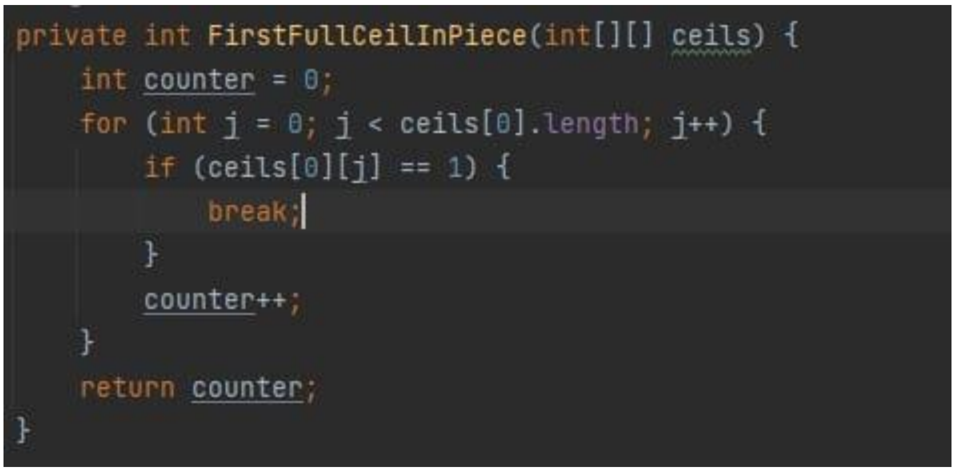
This method takes the numeric state of the boards, it returns the valid board and returns NULL if the board state is not valid.



```java
public int[][] decompose(int numericState) {
    int[][] returnedBoard = boardState(numericState);
    if(returnedBoard == null)
        return null;
    if(isValidBoard(returnedBoard))
        return returnedBoard;
    return null;
}
```

FirstFullCeilInPiece():

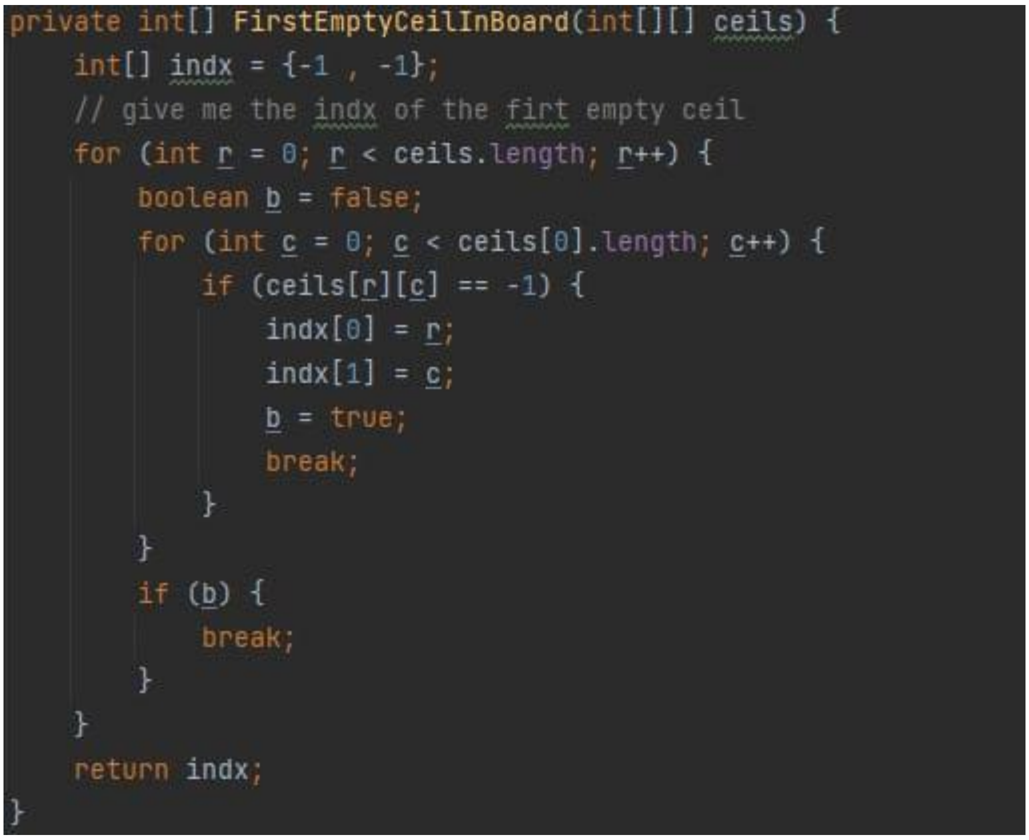
This method returns the location of the first solid part in the piece that consists of 1



```java
private int FirstFullCeilInPiece(int[][] ceils) {
    int counter = 0;
    for (int j = 0; j < ceils[0].length; j++) {
        if (ceils[0][j] == 1) {
            break;
        }
        counter++;
    }
    return counter;
}
```

FirstEmptyCeilInBoard():

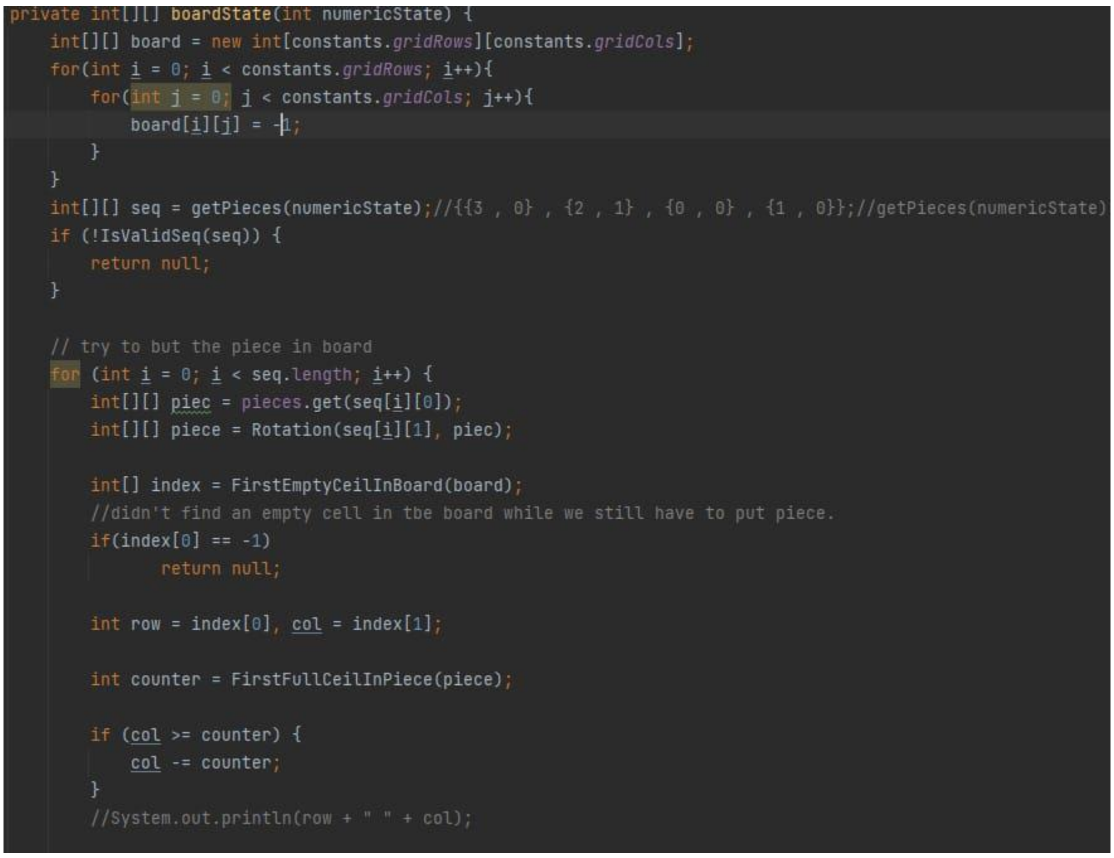
This method returns the index of first empty cell in the grid.



```java
private int[] FirstEmptyCeilInBoard(int[][] ceils) {
    int[] indx = {-1 , -1};
    // give me the indx of the firt empty ceil
    for (int r = 0; r < ceils.length; r++) {
        boolean b = false;
        for (int c = 0; c < ceils[0].length; c++) {
            if (ceils[r][c] == -1) {
                indx[0] = r;
                indx[1] = c;
                b = true;
                break;
            }
        }
        if (b) {
            break;
        }
    }
    return indx;
}
```

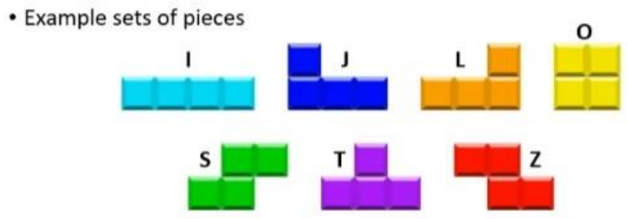
BoardState():

This is the method where all other methods glued together, it takes the sequence of pieces by getPieces(), make sure it is valid by IsValidSeq() and try to put those pieces on the board by trying the possible rotations of the piece by Rotation(), it finds the first empty cell In board in the upper right corner to place the piece in by findFirstEmptyCeilInBoard() and make sure we find empty cells while we still have pieces to place, find first solid part of the piece by findFirstFullCeilInPiece() and at last it puts the pieces on the board and returns the board



```java
private int[][] boardState(int numericState) {
    int[][] board = new int[constants.gridRows][constants.gridCols];
    for(int i = 0; i < constants.gridRows; i++){
        for(int j = 0; j < constants.gridCols; j++){
            board[i][j] = -1;
        }
    }
    int[][] seq = getPieces(numericState);//{{3 , 0} , {2 , 1} , {0 , 0} , {1 , 0}};//getPieces(numericState)
    if (!IsValidSeq(seq)) {
        return null;
    }

    // try to but the piece in board
    for (int i = 0; i < seq.length; i++) {
        int[][] piec = pieces.get(seq[i][0]);
        int[][] piece = Rotation(seq[i][1], piec);

        int[] index = FirstEmptyCeilInBoard(board);
        //didn't find an empty cell in tbe board while we still have to put piece.
        if(index[0] == -1)
              return null;

        int row = index[0], col = index[1];

        int counter = FirstFullCeilInPiece(piece);

        if (col >= counter) {
            col -= counter;
        }
        //System.out.println(row + " " + col);
```

Example sets of pieces:



TEAM MEMBERS ROLE:

مصطفى عبدالباقي ابو حمد ، دنيا احمد على

Multithreading

كريم مصطفي عبدالرحمن ابراهيم عيد ، محمد شعبان سرور احمد ، محمد بهجت فاروق

Implementation

documentation

عادل ناصر طه ، احمد محمد امام

gui