جامعة حلوان HELWAN UNIVERSITY	Operating Systems 2 (Fall 2021) Project Discussion	كلية الحاسبات والذكاء الإصطناعي Faculty of Computers & Artificial Intelligence

Operating systems 2 project documentation

Project description:

Our project is more like a game called "Tetris", it's about putting pieces falling out of the top together at the bottom to form a straight line and then the straight line vanishes.

our 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

010

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' … etc).

Example on the output:

```
1112
1412
3422
3442
```

What we have actually did:

It took us a while to figure out what to do in this project, it wasn’t easy but we did it, we found a way of cooperating with one another as a team and we have done the project exactly as it was described.

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.

Team members roles:

Board: Mohamed Hany, Mohamed Wael and Nada Sabry.

Pieces: Nada Sabry, Eman Ashraf and Marwan Nabil.

Testing the code: Marwan Nabil and Mohamed Wael.

Documentation: Eman Ashraf.

Gui: Mohamed Hany.

Code documentation:

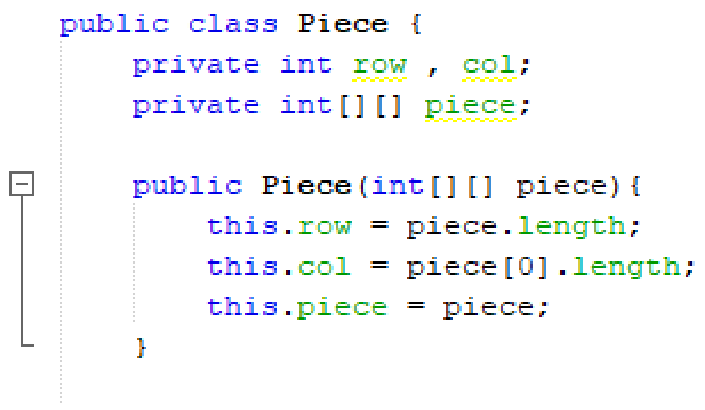
1-Pieces:

Those are the four essential rotations (counter clock-wise) for a piece used in our project, we created a method for each rotation beside the constructor.

The rotation methods are private but the getRotationById() method is public.

Piece class and constructor:

We started off by creating the main piece class which has the constructor “piece”, each piece is defined by the number of rows and columns of a 2D array which are passed to the constructor as follows:

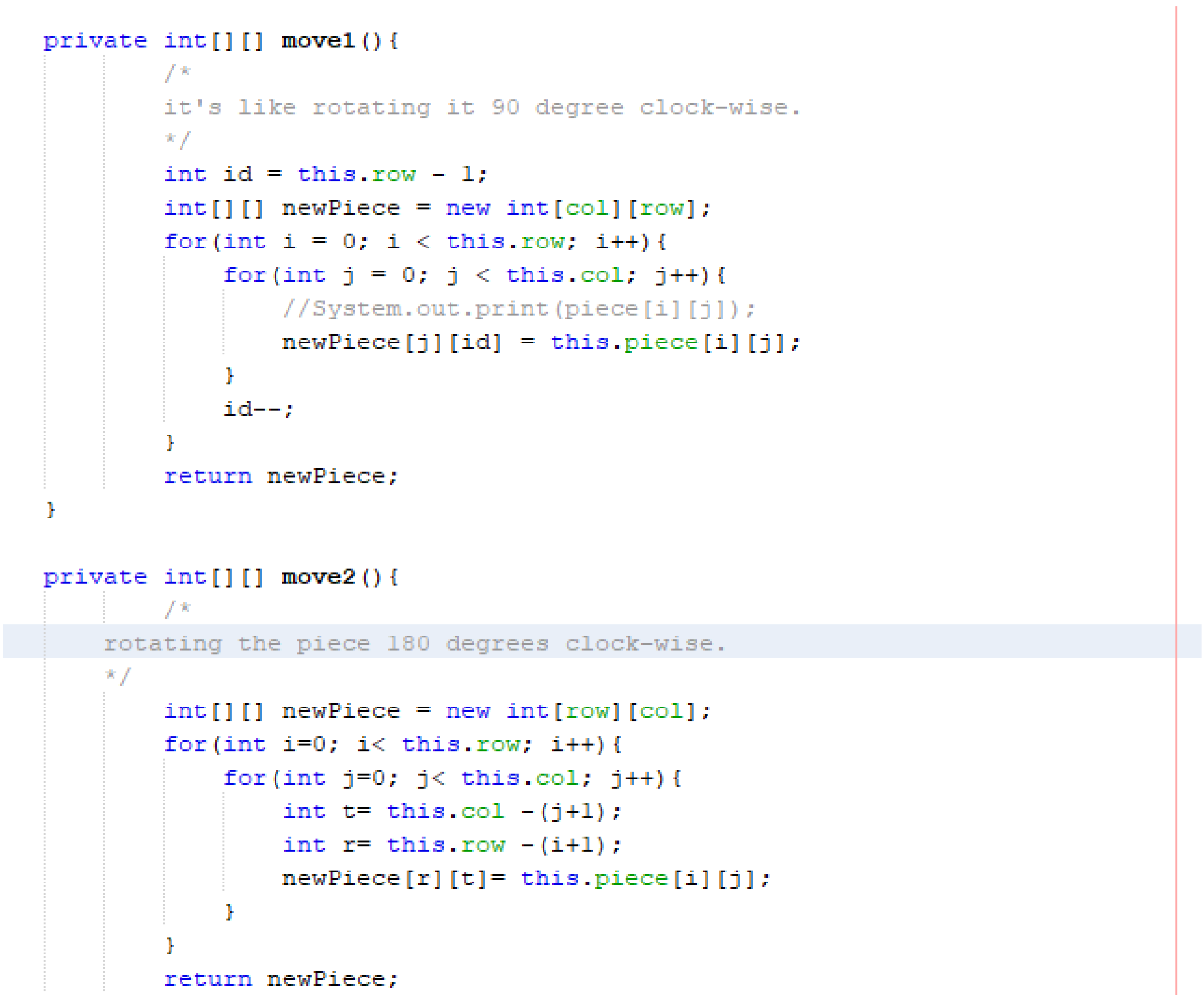


```
public class Piece {
    private int row , col;
    private int[][] piece;

    public Piece(int[][] piece){
        this.row = piece.length;
        this.col = piece[0].length;
        this.piece = piece;
    }
```

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:

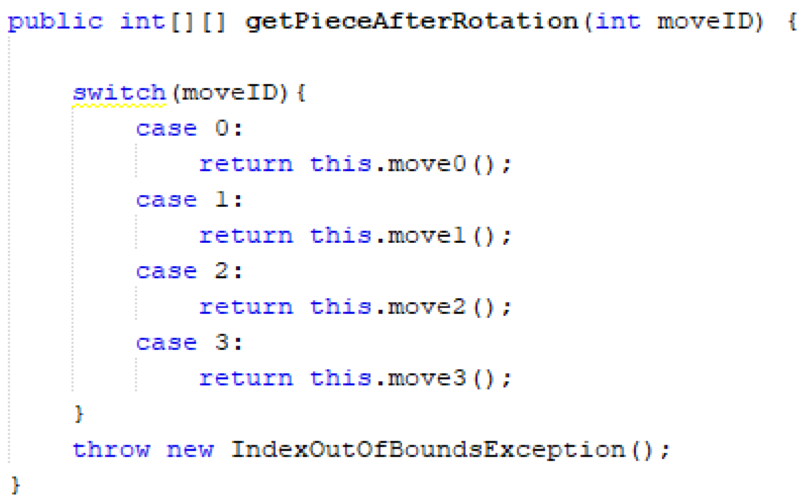


```
private int[][] move1(){
        /*
        it's like rotating it 90 degree clock-wise.
        */
        int id = this.row - 1;
        int[][] newPiece = new int[col][row];
        for(int i = 0; i < this.row; i++){
            for(int j = 0; j < this.col; j++){
                //System.out.print(piece[i][j]);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

private int[][] move2(){
        /*
    rotating the piece 180 degrees clock-wise.
    */
        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;
```

Example on rotations of ‘L’ shaped piece:

getPieceAfterRotation():

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

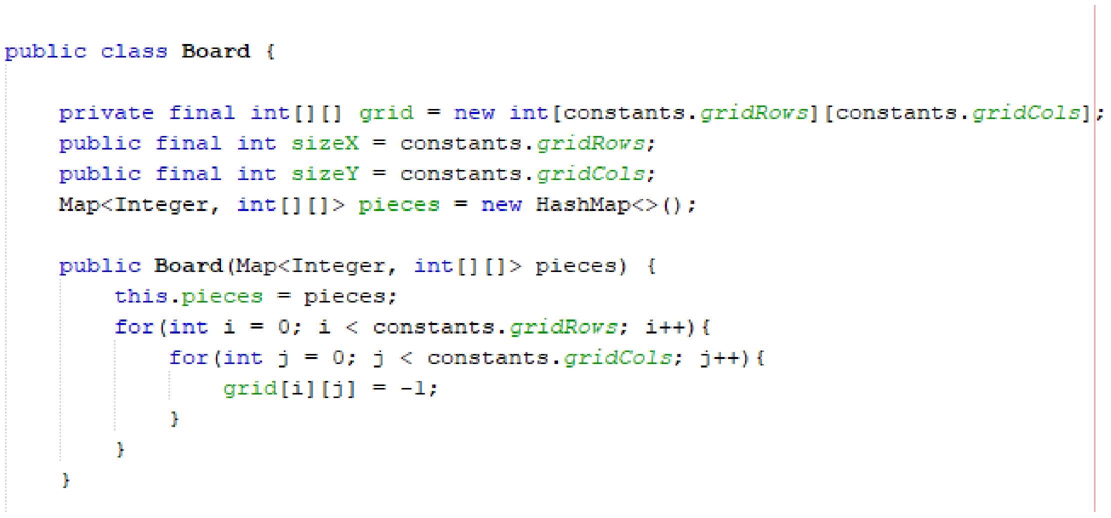


```java
public int[][] getPieceAfterRotation(int moveID) {

    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();
}
```

Board:

Board class and constructor:

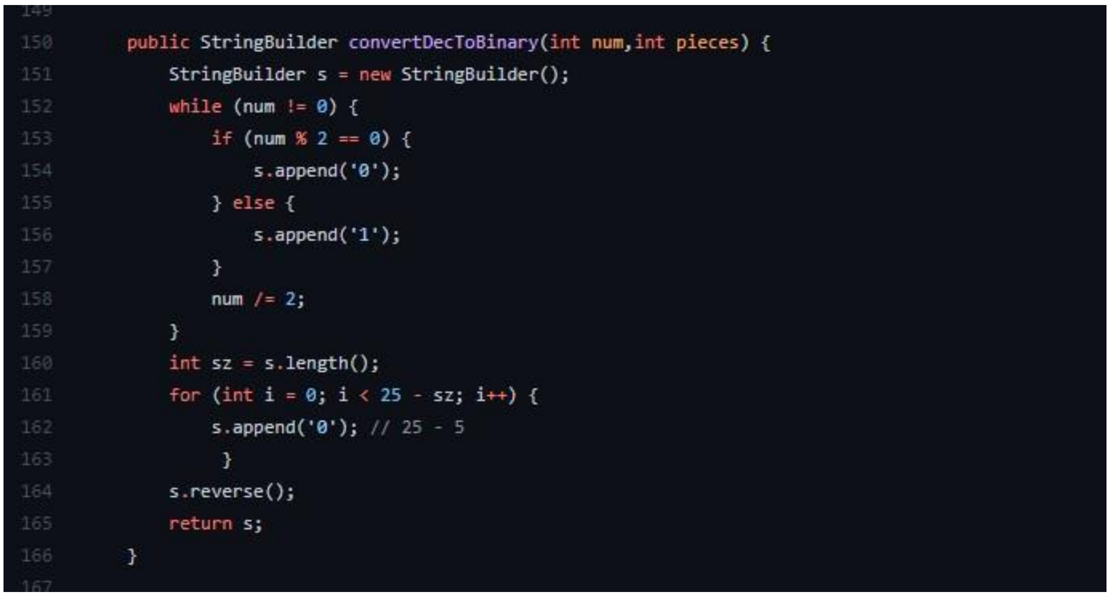
We created a board class and initialized our board (grid), specified its dimensions and we created a HashMap for pieces on the board.



```java
public class Board {

    private final int[][] grid = new int[constants.gridRows][constants.gridCols];
    public final int sizeX = constants.gridRows;
    public final int sizeY = constants.gridCols;
    Map<Integer, int[][]> pieces = new HashMap<>();

    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;
            }
        }
    }
```

ConvertDecToBinary():

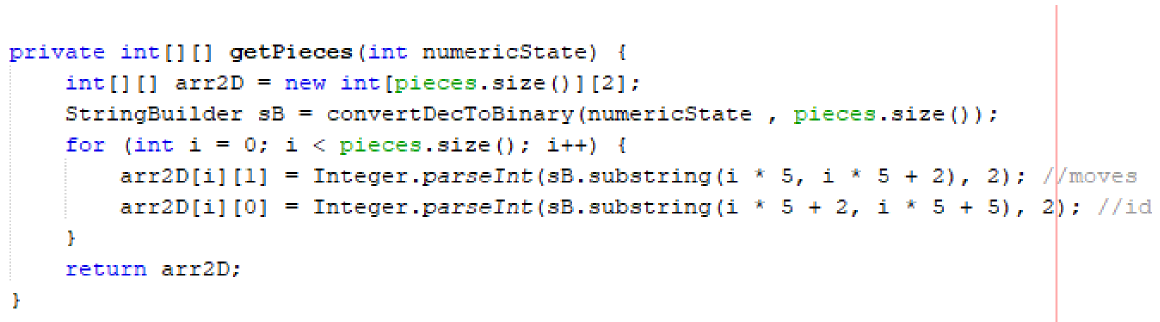
This method is used to represent the numeric state of the current combination by a binary number as a string to be used later to figure out the board state.



```java
    public StringBuilder convertDecToBinary(int num,int pieces) {
        StringBuilder s = new StringBuilder();
        while (num != 0) {
            if (num % 2 == 0) {
                s.append('0');
            } else {
                s.append('1');
            }
            num /= 2;
        }
        int sz = s.length();
        for (int i = 0; i < 25 - sz; i++) {
            s.append('0'); // 25 - 5
             }
        s.reverse();
        return s;
    }
```

getPieces():

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

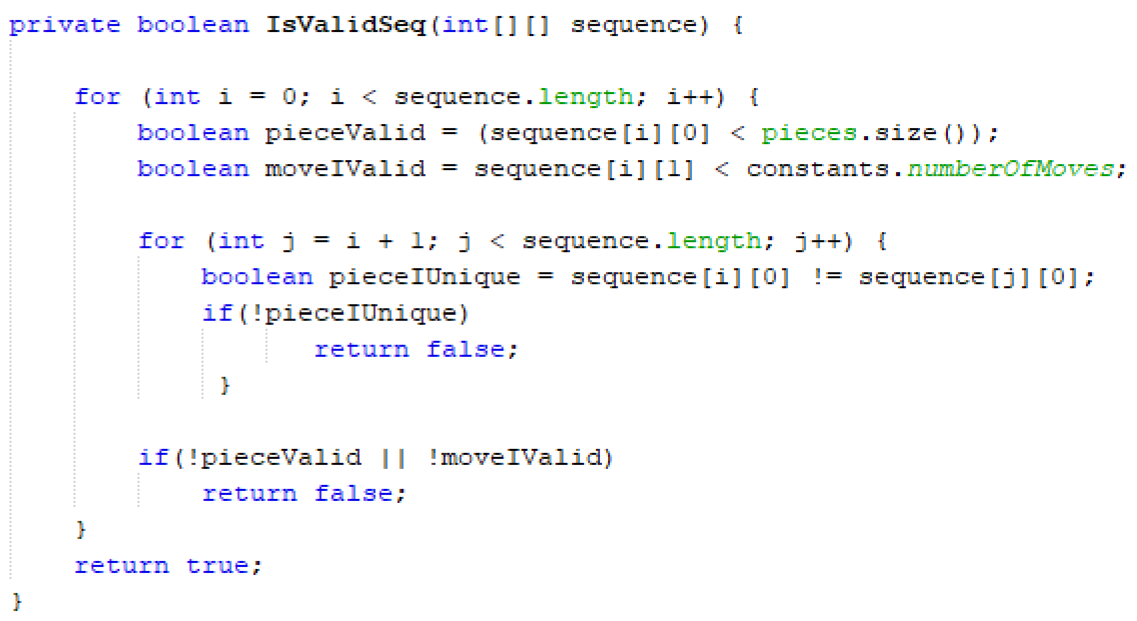


```java
private int[][] getPieces(int numericState) {
    int[][] arr2D = new int[pieces.size()][2];
    StringBuilder sB = convertDecToBinary(numericState , pieces.size());
    for (int i = 0; i < pieces.size(); i++) {
        arr2D[i][1] = Integer.parseInt(sB.substring(i * 5, i * 5 + 2), 2); //moves
        arr2D[i][0] = Integer.parseInt(sB.substring(i * 5 + 2, i * 5 + 5), 2); //id
    }
    return arr2D;
}
```

IsValidSeq():

We used isValidSeq method to check if the sequence given to the board is valid by checking if the given sequence satisfy the main conditions:

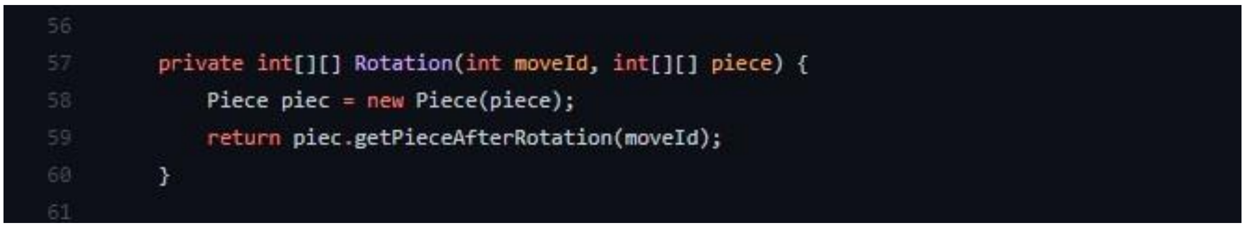
1- No two pieces have the same id
2- Id of the move is less than number of moves
3- Pieces is less than number of pieces we have



```java
private boolean IsValidSeq(int[][] sequence) {

    for (int i = 0; i < sequence.length; i++) {
        boolean pieceValid = (sequence[i][0] < pieces.size());
        boolean moveIValid = sequence[i][1] < constants.numberOfMoves;

        for (int j = i + 1; j < sequence.length; j++) {
            boolean pieceIUnique = sequence[i][0] != sequence[j][0];
            if(!pieceIUnique)
                    return false;
            }

        if(!pieceValid || !moveIValid)
            return false;
    }
    return true;
}
```

Rotation():

Returns piece after rotation.



```java
    private int[][] Rotation(int moveId, int[][] piece) {
        Piece piec = new Piece(piece);
        return piec.getPieceAfterRotation(moveId);
    }
```

FirstEmptyCeilInBoard():

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

```
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;
}
```

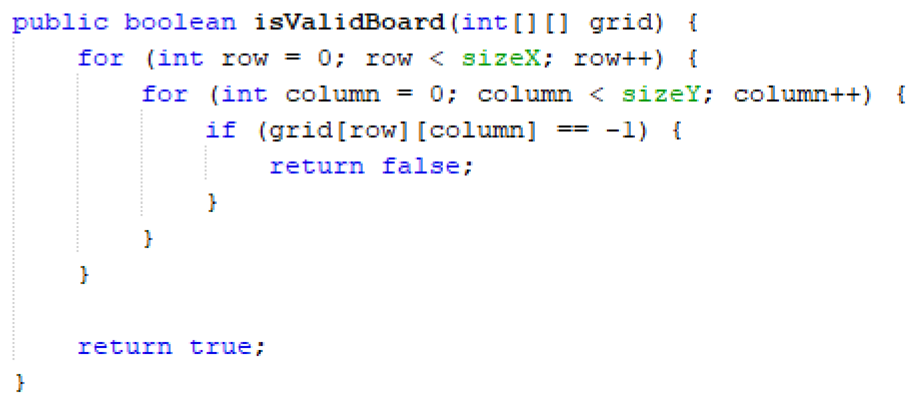
FirstFullCeilInPiece():

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

```
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;
}
```

IsValidBoard():

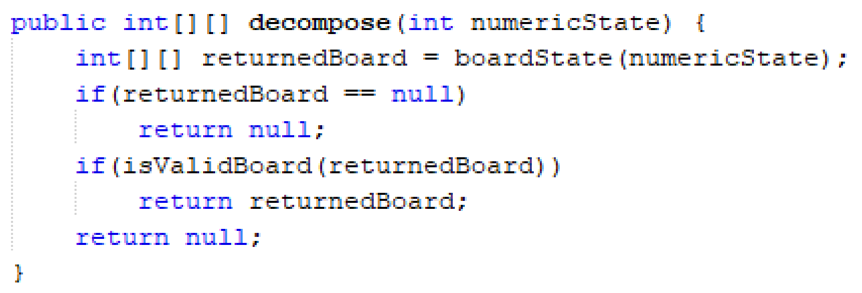
This method makes sure that the state board passed to it doesn't cross the boundaries of the grid so that no combination is having 5 rows or 5 columns, the grid must always be 4x4 square.



```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():

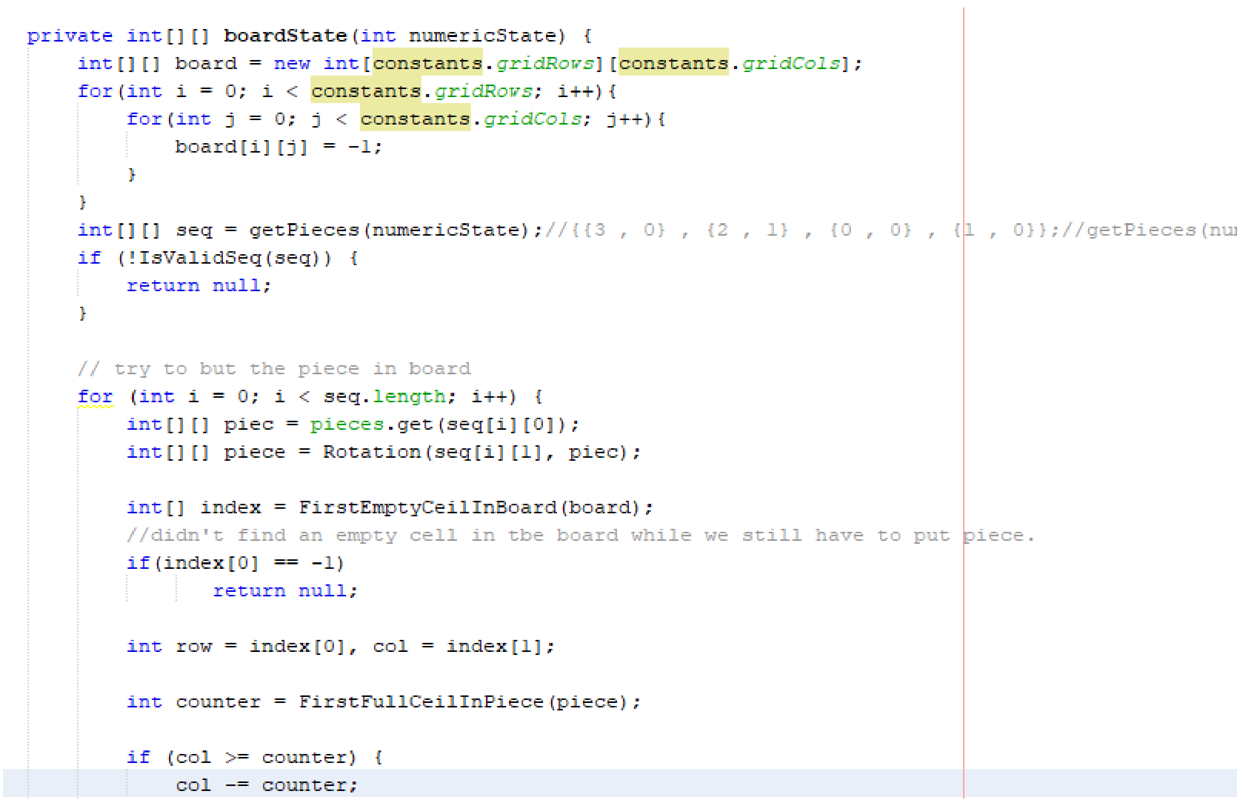
This method takes the numeric state of the boards, it returns the valid board and returns NULL if the board state is not valid.



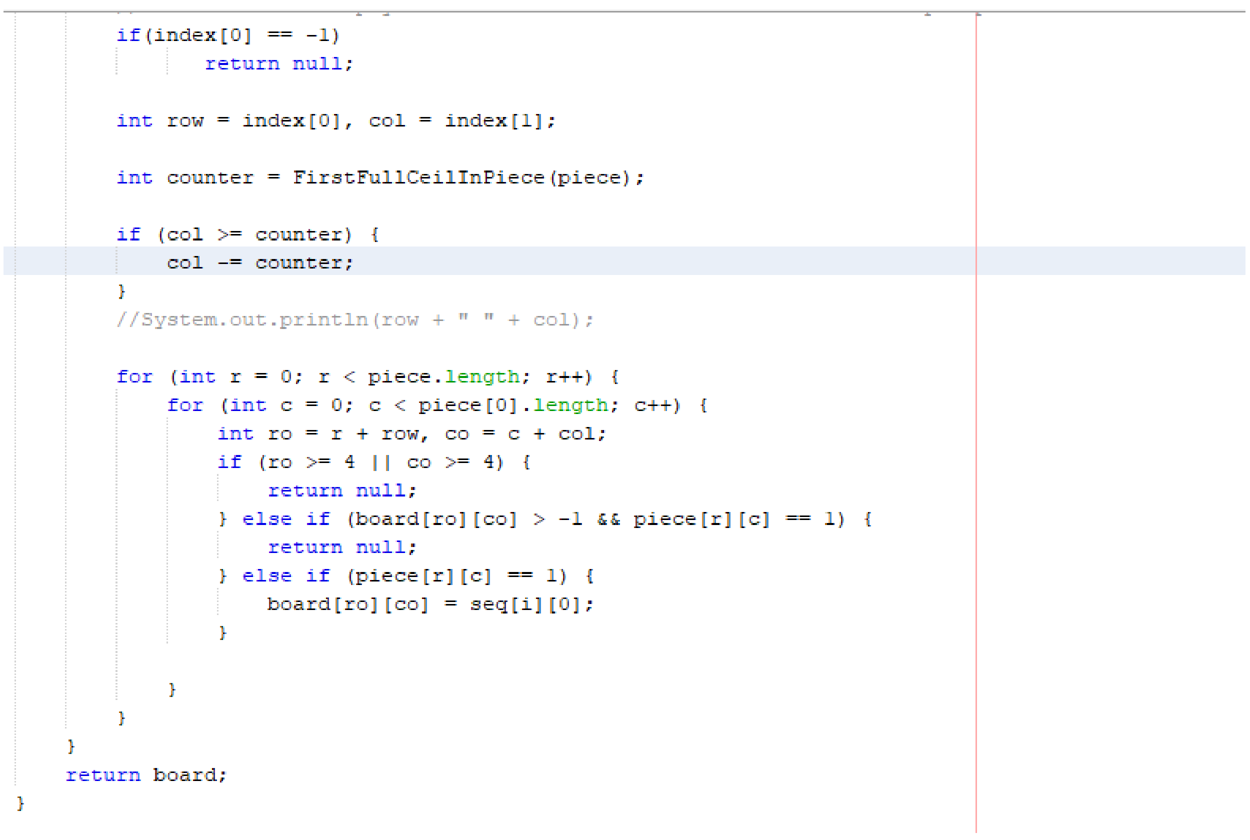
```java
public int[][] decompose(int numericState) {
    int[][] returnedBoard = boardState(numericState);
    if(returnedBoard == null)
        return null;
    if(isValidBoard(returnedBoard))
        return returnedBoard;
    return null;
}
```

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.



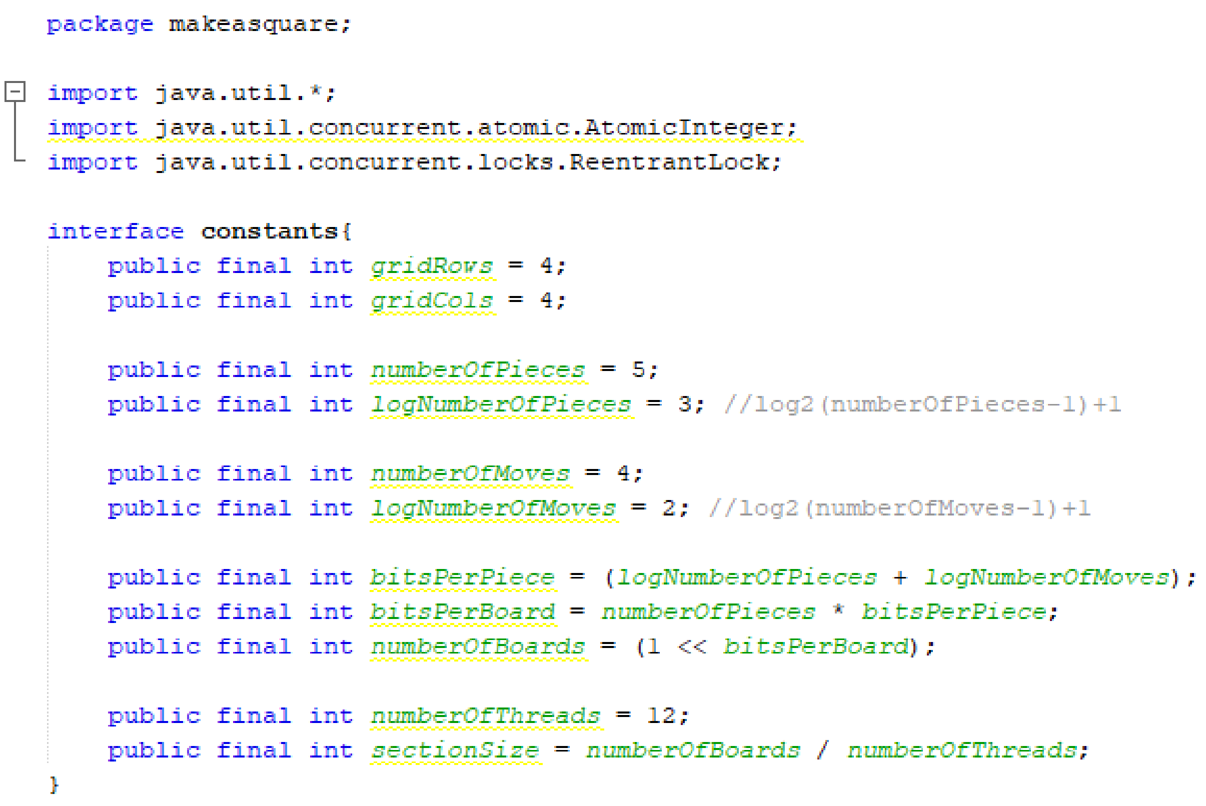
```
private int[][] boardState(int numericState) {
    int[][] board = new int[constants.gridRows][constants.gridCols];
    for(int i = 0; i < constants.gridRows; i++){
        for(int j = 0; j < constants.gridCols; j++){
            board[i][j] = -1;
        }
    }
    int[][] seq = getPieces(numericState);//{{3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



```
        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);

        for (int r = 0; r < piece.length; r++) {
            for (int c = 0; c < piece[0].length; c++) {
                int ro = r + row, co = c + col;
                if (ro >= 4 || co >= 4) {
                    return null;
                } else if (board[ro][co] > -1 && piece[r][c] == 1) {
                    return null;
                } else if (piece[r][c] == 1) {
                    board[ro][co] = seq[i][0];
                }

            }
        }
    }
    return board;
}
```

Threads:

Constant variables that we used in implementing threads:

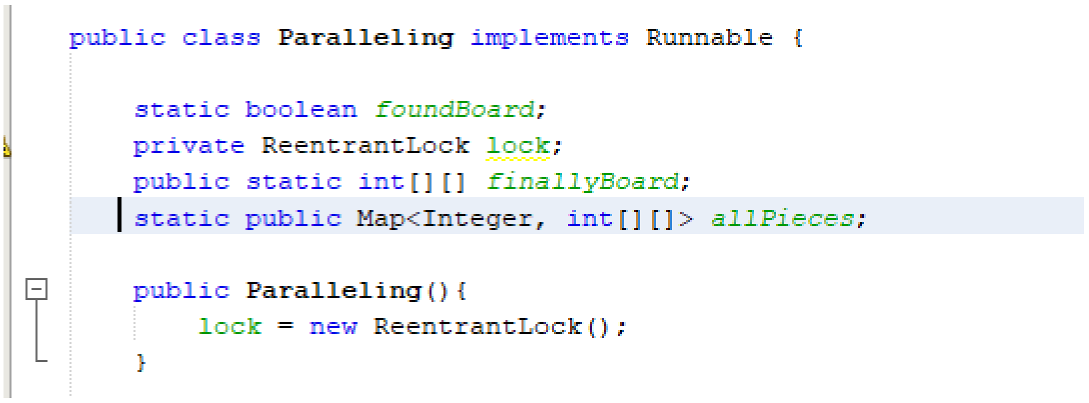


```
package makeasquare;

import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.ReentrantLock;

interface constants{
    public final int gridRows = 4;
    public final int gridCols = 4;

    public final int numberOfPieces = 5;
    public final int logNumberOfPieces = 3; //log2(numberOfPieces-1)+1

    public final int numberOfMoves = 4;
    public final int logNumberOfMoves = 2; //log2(numberOfMoves-1)+1

    public final int bitsPerPiece = (logNumberOfPieces + logNumberOfMoves);
    public final int bitsPerBoard = numberOfPieces * bitsPerPiece;
    public final int numberOfBoards = (1 << bitsPerBoard);

    public final int numberOfThreads = 12;
    public final int sectionSize = numberOfBoards / numberOfThreads;
}
```

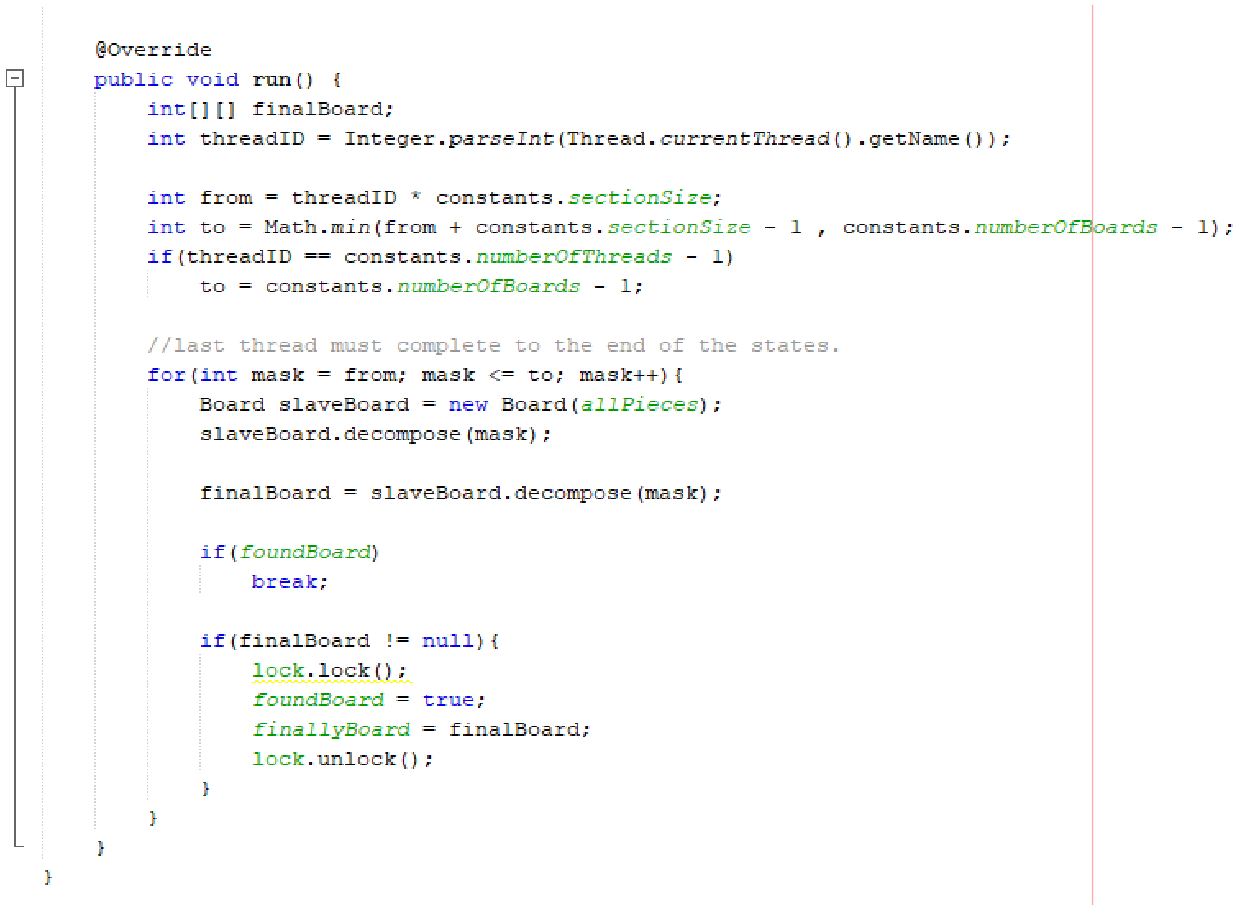
We used java runnable which is an interface to execute code on a concurrent thread, this interface is implemented by any class if we want the instances of that class to be executed by a thread.

We also used a ReentrantLock class which implements the lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the final board is surrounded by calls to lock and unlock method.

Each one of the 12 threads takes a number of board states to work with in parallel with the other threads and whenever one of them finds the first valid board it locks the lock and set the valid board as the final board and break from the loop.

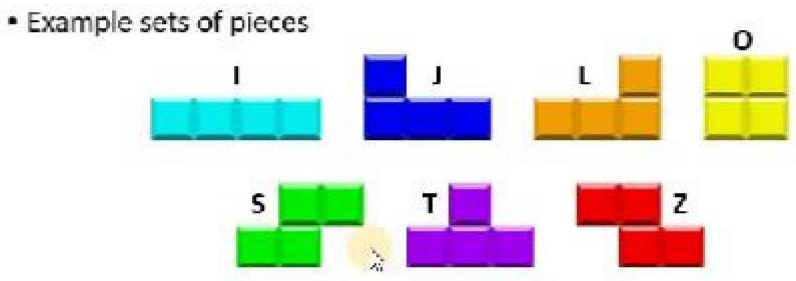


```
public class Paralleling implements Runnable {

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

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

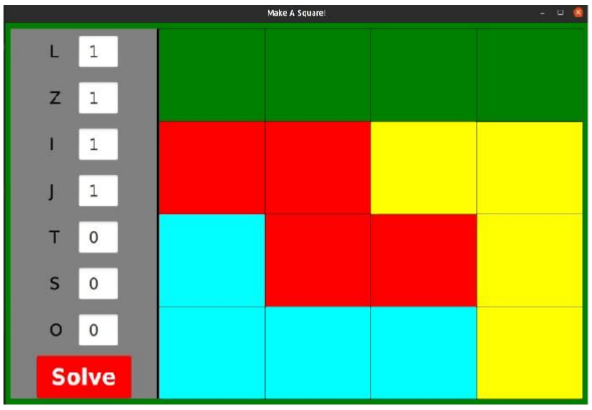


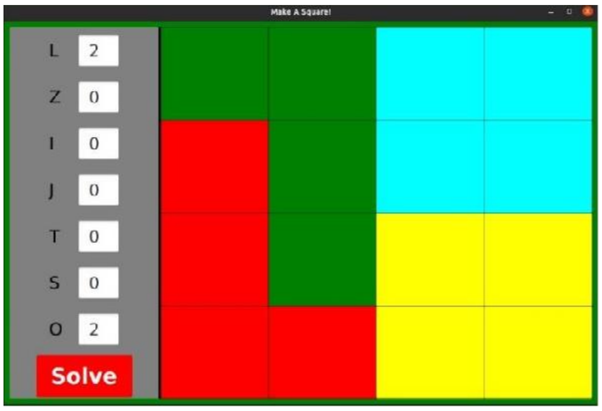
```
    @Override
    public void run() {
        int[][] finalBoard;
        int threadID = Integer.parseInt(Thread.currentThread().getName());

        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();
            }
        }
    }
}
```

Graphical user interface:



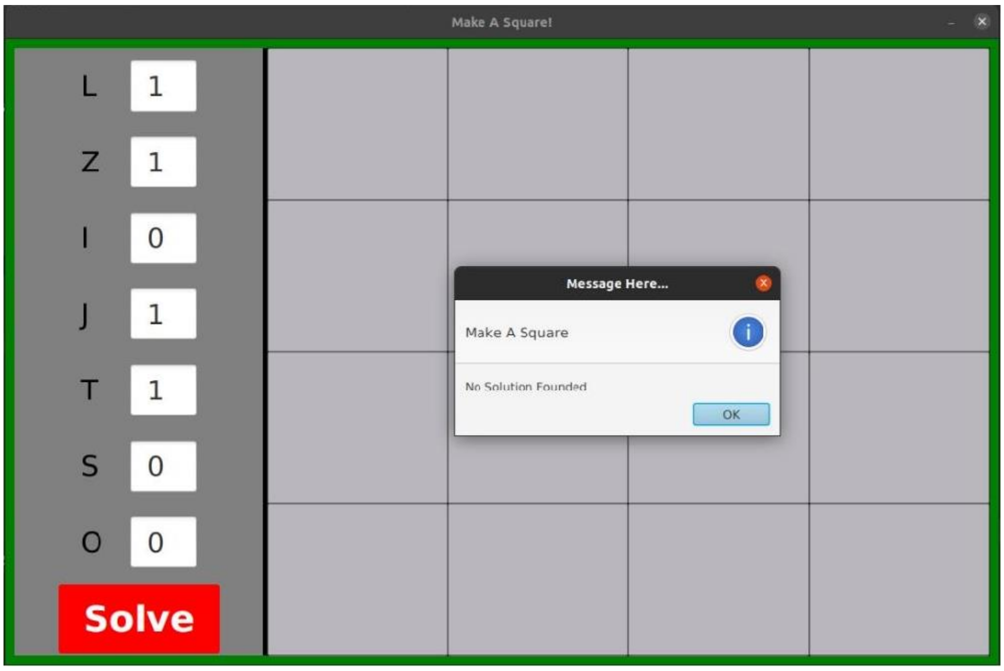
This is an example of pieces we have used in our project, each piece is given to the program by writing how many pieces of the same shape we need in the text box beside the shape of the piece in the GUI.





We tried many samples in the project, by giving it different pieces and it finds the way to combine those pieces.

if there’s a possible combination to the pieces, it will be drawn on the gui as we have shown above.



No Solution Founded

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