
Os Project 5
is Make A Square.
With multithreading & Gui with real time and synchronization

Make a square

Project description:

Make a square with size 4X4 by using 4. The pieces can be rotated only and all pieces should be used to form a square.

Input:

For example, piece A above would be specified as follows:

3 2

111

101

Output:

Our program should report all solution, in the format shown by the examples below. A 4-row by 4-column

square should be created, with each piece occupying its location in the solution. The solid portions of

Sample output that represents the figure above could be:

1112

1412

3422

3442

What we have actually did:

It was one of the hardest project we actually did , and it took us awhile to even know the idea of the project , but finally we did it , with the help of each other as a team members .

So now let me explain to you what we did , We used a 2D arrays to specify the pieces and Each piece is represented as a combination of 1s and 0s, with 1s indicating the solid part of the piece and 0s acting as placeholders , also we used object oriented programming and Java threads to implement this project.

Team members roles:

InputPieces: Alaa Fouad

MasterThread: Fatma Mohammed , Amr Gamal , Ahmed Mahmoud

PrintSolutionsThreads: Alaa Hamdy

multithreading: Issra Hossam

Documentation: Fatma Mohammed, Alaa Fouad

Gui: Alaa Fouad

Code documentation:

We started off by implementing **InputPieces** class which Include jButton1ActionPerformed method, This method is triggered when the "SOLVE" button is pressed .

It collects the input data from text fields and store them at inputPieces array.

```
public class InputPieces extends javax.swing.JFrame {
    public InputPieces() {
        initComponents();
    }
    @SuppressWarnings("unchecked")
    Generated Code
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

        int[] inputPieces = new int[7];
        JTextField [] jTextFieldsArray = {INput0, INput1, INput2, INput3, INput4, INput5, INput6};
        String numString;
        int numInt = 0;
        for(int i = 0 ; i< 7 ; i++ ){
            numString = jTextFieldsArray[i].getText();
            if(!numString.isEmpty()){
            inputPieces[i]=Integer.parseInt(numString);
            numInt += Integer.parseInt(numString);
            }
            else inputPieces[i] = 0;
        }
        System.out.println(numInt);
        if(numInt < 4){
                    MasterThread s1 = new MasterThread(inputPieces);
                    Thread t100 = new Thread(s1);
                    t100.start();
            try {
                t100.join();

            } catch (InterruptedException ex) {
                JOptionPane.showMessageDialog(null, "Error","Error",JOptionPane.ERROR_MESSAGE);
```

Then we created **MasterThread** class that Implements Runnable and manages the main threading logic.

Then we created **run** method and first of all we initialized four HashMap for pieces to help us with multithreading , then we initialized usablePieces that will be send to **board1setup** as arguments.

```
public class MasterThread   implements Runnable {
    static int[] inputPieces;
    static int keyThreadLJT=0;
    static int keyThreadSZI=0;
    public Thread t1,t2,t3,t4;
    public static int[] keyOfSolve;

    public  MasterThread(int[] inputPieces){
        this.inputPieces =inputPieces;
    }

    @Override
    public void run() {
        try {
            HashMap<Integer,ArrayList<int[][]>> pieces1 = new HashMap<Integer,ArrayList<int[][]>>();
            HashMap<Integer,ArrayList<int[][]>> pieces2 = new HashMap<Integer,ArrayList<int[][]>>();
            HashMap<Integer,ArrayList<int[][]>> pieces3 = new HashMap<Integer,ArrayList<int[][]>>();
            HashMap<Integer,ArrayList<int[][]>> pieces4 = new HashMap<Integer,ArrayList<int[][]>>();

            int[] usablePieces1 = null;
            usablePieces1 = board1setup(pieces1,inputPieces);
            int[] usablePieces2 = null;
            usablePieces2 = board1setup(pieces2,inputPieces);
            int[] usablePieces3 = null;
            usablePieces3 = board1setup(pieces3,inputPieces);
            int[] usablePieces4 = null;
            usablePieces4 = board1setup(pieces4,inputPieces);

            keyOfSolve = usablePieces1;
```

As for **rotation** for pieces, we think of every possible shape for the piece and write them and initialized an arraylist for each piece to store the different shape of the same piece.

```
int[][] pieceJa = {{2,0,0,0,0},
                   {2,2,2,0,0},
                   {0,0,0,0,0},
                   {0,0,0,0,0}};

int[][] pieceJb = {{0,2,0,0,0},
                   {0,2,0,0,0},
                   {2,2,0,0,0},
                   {0,0,0,0,0}};

int[][] pieceJc = {{2,2,0,0,0},
                   {2,0,0,0,0},
                   {2,0,0,0,0},
                   {0,0,0,0,0}};

int[][] pieceJd = {{2,2,2,0,0},
                   {0,0,2,0,0},
                   {0,0,0,0,0},
                   {0,0,0,0,0}};


ArrayList<int[][]> pieceJ = new ArrayList<int[][]>();

pieceJ.add(pieceJa);
pieceJ.add(pieceJb);
pieceJ.add(pieceJc);
pieceJ.add(pieceJd);
```

Another example, but here the piece we have, has only two possible shapes, so we write them down and store them at arraylist initialized for this piece.

```
int[][] pieceZa = {{5,5,0,0,0},
                   {0,5,5,0,0},
                   {0,0,0,0,0},
                   {0,0,0,0,0}};

int[][] pieceZb = {{0,5,0,0,0},
                   {5,5,0,0,0},
                   {5,0,0,0,0},
                   {0,0,0,0,0}};


ArrayList<int[][]> pieceZ = new ArrayList<int[][]>();

pieceZ.add(pieceZa);
pieceZ.add(pieceZb);
```

placePiece method is used to determine whether a piece could be placed on the board or not.

```
public static boolean placePiece
(int boardY, int boardX, int currentPiece,int[][] currentPerm, int[][] currentBoard)
{
        for(int pieceY = 0; pieceY < pieceYdim; pieceY++) {
                for(int pieceX = 0; pieceX < pieceXdim; pieceX++) {
                        if(currentPerm[pieceY][pieceX] != 0) {

                                int y = boardY+pieceY;

                                if(y >= boardYdim) {
                                        return false;
                                }
                                int x = boardX+pieceX;

                                if(x >= boardXdim) {
                                        return false;
                                }

                                if(currentBoard[y][x] != 0) {
                                        return false;
                                }

                                currentBoard[y][x] = currentPiece;
                        }
                }
        }
        return true;
}
```

Solve method Uses a recursive approach to attempt various combinations of pieces on the board and Tries different permutations of pieces and their placements on the board and Checks for valid solutions and avoids duplicate solutions .

```
public static void solve
(int[][] board, int[] usablePieces, int depth,ArrayList<int[][]> solutions ,HashMap<Integer,ArrayList<int[][]>> pieces) {
        for(int index = 0; index < usablePieces.length; index++) {
                ArrayList<int[][]> permutations = pieces.get(usablePieces[index]);
                for(int perm = 0; perm < permutations.size(); perm++) {
                        for(int boardY = 0; boardY < boardYdim; boardY++) {
                                for(int boardX = 0; boardX < boardXdim; boardX++) {
                                        int[][] newBoard = new int[board.length][board[0].length];
                                        for(int y = 0; y < board.length; y++) {
                                                for(int x = 0; x < board[0].length; x++) {
                                                        newBoard[y][x] = board[y][x];
                                                }
                                        }
                                        boolean returnValue = placePiece(boardY, boardX, usablePieces[index],
                                                                                permutations.get(perm), newBoard);

                                        if(returnValue) {
                                                int[] newPieces = new int[usablePieces.length-1];
                                                int indexCounter = 0;
                                                for(int i = 0; i < usablePieces.length; i++) {
                                                        if(usablePieces[i] != usablePieces[index]) {
                                                                newPieces[indexCounter] = usablePieces[i];
                                                                indexCounter++;
                                                        }
                                                }
                                                        else if(doesSolutionExist(newBoard,solutions) == false) {
                                                                solutions.add(newBoard);
                                                        }
                                                }

                                                else {
                                                        solve(newBoard, newPieces, depth+1, solutions,pieces);
                                                }
                                        }
                                }
                        }
                }
        }
}
```

doesSolutionExist method is a Boolean method Checks if a solution already exists in the solutions list to avoid duplicates.

```
public static boolean doesSolutionExist(int[][] sol,ArrayList<int[][]> solutions) {
        boolean match;

        for(int index = 0; index < solutions.size(); index++) {
                match = true;
                for(int y = 0; y < 2; y++) {
                        for(int x = 0; x < 2; x++) {
                                if((solutions.get(index))[y][x] != sol[y][x]) {
                                        match = false;
                                }
                        }
                }
                if(match == true) {
                        return true;
                }
        }
        return false;
}
```

PrintSolutionsThreads class to display solutions and handling the GUI display for showing the generated solutions and Utilizing Swing components to represent the grid layout.

```
public class PrintSolutionsThreads extends javax.swing.JFrame implements Runnable {



    int[][] sol;
    static int [] num0fkey;
    public int no=0;

    public PrintSolutionsThreads(int[][] sol ,int num0fthread) {
        initComponents();
        no =1;
        this.sol = sol;
        num0fkey = MasterThread.key0fSolve;
        jLabel2.setText(""+ num0fthread);

    }

    public PrintSolutionsThreads() {
                initComponents();
                no=0;
    }
```

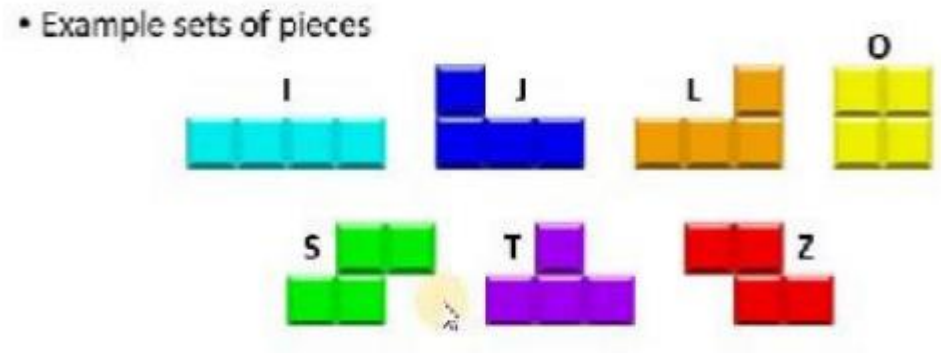
printSolution method used to Populate and color the GUI buttons based on the provided solution grid and Use different colors to represent different pieces.

```
    public void printSolution(int [][] grid, JButton []JButtonsArray) {
      if (grid[0][0] != 0){
         for (int i = 0; i < 4; i++) {
             for (int j = 0; j < 4; j++) {
                 if (grid[i][j]==0){
                    JButtonsArray[(i*4)+j].setVisible(false);
                     System.out.println("NoSolution");
                 }
                 else if (grid[i][j]==num0fkey[0])
                 {
                     JButtonsArray[(i*4)+j].setBackground(Color.red);

                 }
                 else if (grid[i][j]==num0fkey[1])
                 {
                     JButtonsArray[(i*4)+j].setBackground(Color.BLACK);
                 }
                 else if (grid[i][j]==num0fkey[2])
                 {
                     JButtonsArray[(i*4)+j].setBackground(Color.BLUE);
                 }
                 else if (grid[i][j]==num0fkey[3])
                 {
                     JButtonsArray[(i*4)+j].setBackground(Color.MAGENTA);
                 }
             }
         }
      }
}
```

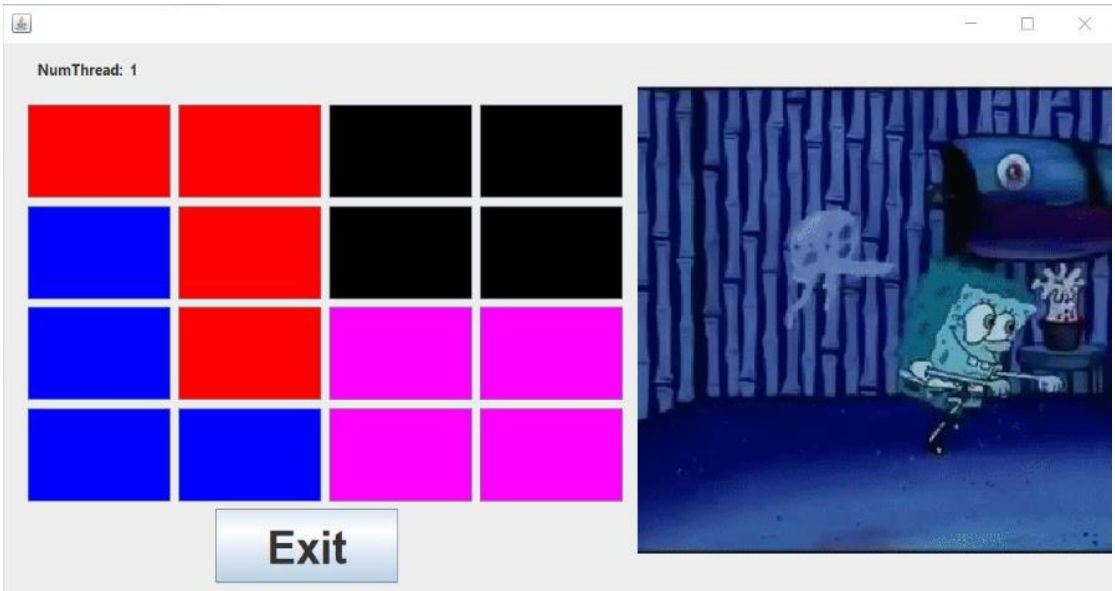
GUI

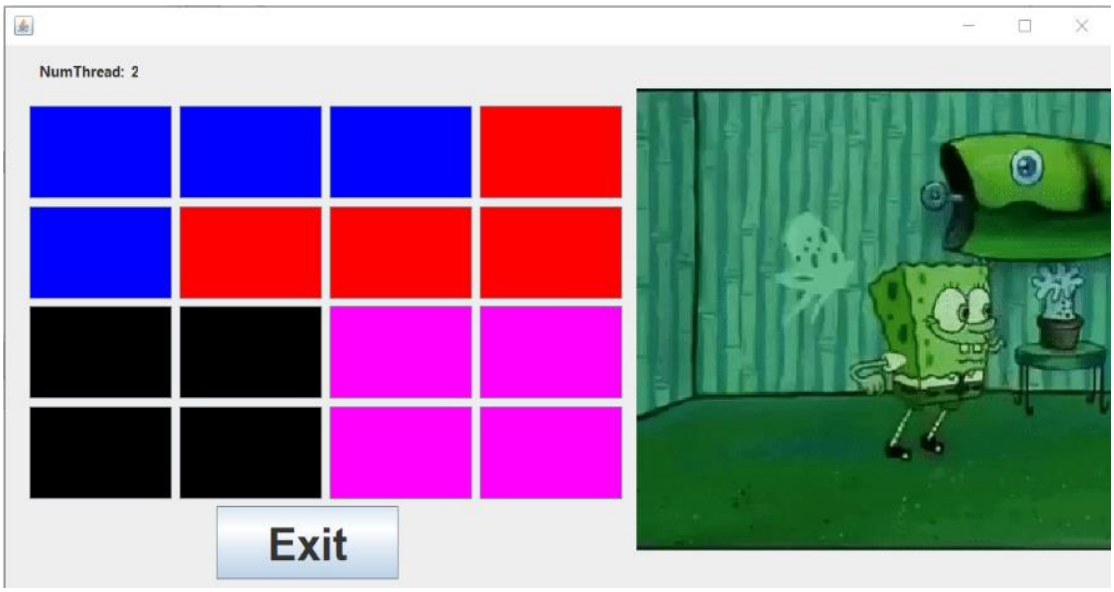
This is an example of pieces we have used in our project; each piece is given to the program by writing how many pieces of the same shape we need in the text box beside the shape of the piece in the GUI.



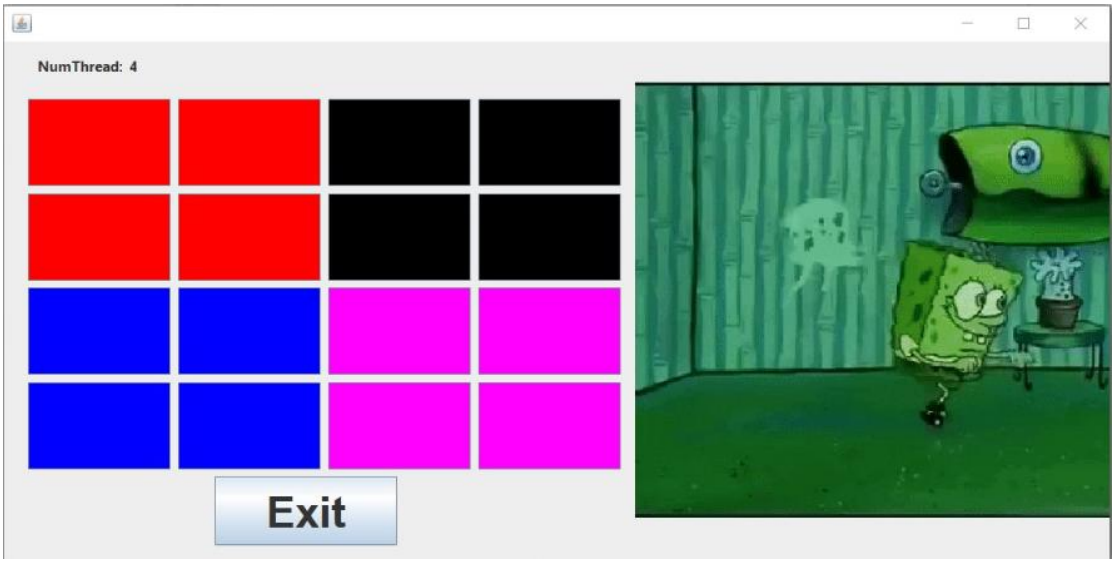
Output

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




NumThread: 2
Exit


NumThread: 3
Exit


NumThread: 4
Exit

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