Project #5: Make a square

Project description:

This project is a **Tetris Puzzle Solver** that uses **multi-threading** to solve Tetris-like puzzles with different shapes of pieces. The goal is to place a set of Tetris pieces (L-shape, Z-shape, I-shape, etc.) on a grid and determine if they can fit without overlapping.

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 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

Team member roles:

Board: Shahd Shaban , Menna Mohammed, Nada Nabil

Pieces : Basma Mohammed, Wessam Mahmoud , Amira Ahmed

GUI : Amira Ahmed **Documentation : Nada Nabil**

What we have actually did:

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 threads equals the number of piece types the user enters. Each thread runs in parallel, starting from a different starting point to explore different solutions for the game.

Code documentation:

Our project consist of 3 main class

1. PuzzleController Class:

Description: The main controller for managing the puzzle-solving process. It connects the PuzzleGUI (user interface) and the PuzzleSolver (logic for solving the puzzle). It generates the puzzle pieces based on user input, manages the threads for solving the puzzle in parallel, and updates the GUI with the status of the puzzle solution.

Responsibilities:

Handles the puzzle-solving logic by invoking the solver in parallel threads.

Updates the GUI with the status of the puzzle-solving process.

Manages the puzzle board layout.

```
public class PuzzleController {
    private final PuzzleGUI gui;
    private final PuzzleSolver solver;
    private final ExecutorService executor;

    public PuzzleController() {...٥ lines }

    public void solvePuzzle(int[] pieceCounts) {...١٢٠ lines }

    private List<int[][]> generatePieces(int[] pieceCounts) {...١١ lines }

    private void fillBoard(int[][] board, List<int[][]> pieces, JPanel boardPanel) {...٤ lines }

    public static void main(String[] args) {...٥ lines }
}
```

1. PuzzleController()

```
public PuzzleController() {
    gui = new PuzzleGUI(controller: this);
    solver = new PuzzleSolver();
    executor = Executors.newFixedThreadPool(nThreads: 4);
}
```

Description: Constructor that initializes the GUI, solver, and an executor service with a fixed thread pool of 4 threads.

2. solvePuzzle(int[] pieceCounts)

```
public void solvePuzzle(int[] pieceCounts) {
List<int[][]> pieces = generatePieces(pieceCounts);
JPanel mainPanel = gui.getMainPanel();
mainPanel.removeAll();

// Clear previous solution panels and labels
List<JPanel> boardPanels = new ArrayList<>();
List<JLabel> solutionStatusLabels = new ArrayList<>();

// Identify distinct piece types to process
List<Integer> distinctPieceTypes = new ArrayList<>();
for (int i = 0; i < pieceCounts.length; i++) {
    if (pieceCounts[i] > 0) {
        distinctPieceTypes.add(e: i); // Store the index of selected piece types
    }
}

// If there are fewer than 4 pieces, display the "No Solution" message for each thread
if (pieces.size() < 4) {
    // Display message for insufficient pieces
    for (int pieceTypeIndex : distinctPieceTypes) {
    final int threadIndex = pieceTypeIndex;
```

```
  JPanel boardPanel = new JPanel(new GridLayout(rows: 4, cols: 4));
  boardPanel.setBackground(new Color(r: 255, g: 228, b: 225)); // Misty rose

  JLabel statusLabel = new JLabel(text: "No Solution", horizontalAlignment:JLabel.CENTER);
  statusLabel.setFont(new Font(name: "Comic Sans MS", style: Font.BOLD, size: 16));
  statusLabel.setForeground(fg: Color.WHITE);
  statusLabel.setBackground(new Color(r: 255, g: 69, b: 0)); // Red for no solution
  statusLabel.setOpaque(isOpaque: true);

  JPanel threadPanel = new JPanel(new BorderLayout());
  threadPanel.add(new JLabel("Thread " + (threadIndex + 1), horizontalAlignment:JLabel.CENTER), constraints:BorderLayo
  threadPanel.add(comp: boardPanel, constraints:BorderLayout.CENTER);
  threadPanel.add(comp: statusLabel, constraints:BorderLayout.SOUTH);

  // Update the UI with the "No Solution" message
  SwingUtilities.invokeLater(() -> {
      mainPanel.add(comp: threadPanel);
      gui.getFrame().revalidate();
      gui.getFrame().repaint();
  });
```

```
        // Submit a task for each thread to display the "No Solution" message
        executor.submit(() -> {
            int[][] board = new int[4][4];
            for (int[] row : board) {
                java.util.Arrays.fill(a: row, val:-1); // Fill board with -1 indicating empty cells
            }
            fillBoard(board, pieces, boardPanel); // You may choose to keep this method or modify it to
        });
    }
    return; // Return early since there's no solution
}
```

Description: This method handles the solving of the puzzle. It generates the pieces based on the input counts, creates UI components (panels, labels) for each piece type, and starts the puzzle-solving process using multiple threads. If there are fewer than 4 pieces, it displays a "No Solution" message for each thread.

3. generatePieces(int[] pieceCounts)

```
private List<int[][]> generatePieces(int[] pieceCounts) {
    List<int[][]> pieces = new ArrayList<>();
    for (int i = 0; i < pieceCounts[0]; i++) pieces.add(new int[][]{{1, 1, 1}, {1, 0, 0}}); // L-shape
    for (int i = 0; i < pieceCounts[1]; i++) pieces.add(new int[][]{{1, 1, 0}, {0, 1, 1}}); // Z-shape
    for (int i = 0; i < pieceCounts[2]; i++) pieces.add(new int[][]{{1}, {1}, {1}, {1}}); // I-shape
    for (int i = 0; i < pieceCounts[3]; i++) pieces.add(new int[][]{{1, 1, 1}, {0, 0, 1}}); // J-shape
    for (int i = 0; i < pieceCounts[4]; i++) pieces.add(new int[][]{{1, 1, 1}, {0, 1, 0}}); // T-shape
    for (int i = 0; i < pieceCounts[5]; i++) pieces.add(new int[][]{{0, 1, 1}, {1, 1, 0}}); // S-shape
    for (int i = 0; i < pieceCounts[6]; i++) pieces.add(new int[][]{{1, 1}, {1, 1}}); // O-shape
    return pieces;
}
```

Description: This method generates a list of puzzle pieces based on the provided piece counts. It creates pieces like L, Z, I, J, T, S, and O shapes using the counts specified in the pieceCounts array.

4. fillBoard(int[][] board, List<int[][]> pieces, JPanel boardPanel)

```
private void fillBoard(int[][] board, List<int[][]> pieces, JPanel boardPanel) {
    PuzzleSolver tempSolver = new PuzzleSolver();
    tempSolver.solveForThread(board, pieces, threadIndex:0, new JLabel(), boardPanel);
}
```

Description: This method is used to fill the board with pieces and update the corresponding panel. It uses a temporary PuzzleSolver instance to solve for the thread and update the board.

5. main(String[] args)

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        new PuzzleController(); // Start the application
    });
}
```

Description: The main entry point of the program. It initializes the PuzzleController on the Swing Event Dispatch Thread.

2. PuzzleGUI Class:

```
public class PuzzleGUI {
    private final JFrame frame;
    private final JPanel mainPanel;
    private final JTextField[] pieceInputs;
    private final JButton solveButton;

    public PuzzleGUI(PuzzleController controller) {...٤١ lines }

    public JPanel getMainPanel() {...٣ lines }

    public JFrame getFrame() {...٣ lines }

    public int[] getPieceCounts() {...١١ lines }

}
```

Description: This class creates and manages the graphical user interface (GUI) for the puzzle solver. It displays the puzzle board and provides input fields for users to specify the number of pieces they want to use in the puzzle. The interface also contains a button to start the puzzle-solving process.

Responsibilities:

Provides a layout for the user to input the number of pieces.

Displays the puzzle board and its status.

Passes the input data to the PuzzleController to start the solving process.

1. PuzzleGUI(PuzzleController controller)

```
public PuzzleGUI(PuzzleController controller) {
    frame = new JFrame(title: "💖 Girly Puzzle Solver 💖");
    frame.setSize(width: 800, height: 600);
    frame.setDefaultCloseOperation(operation: JFrame.EXIT_ON_CLOSE);
    frame.setLayout(new BorderLayout());
    frame.getContentPane().setBackground(c: Color.PINK);

    JPanel leftPanel = new JPanel(new GridLayout(rows: 8, cols: 2));
    leftPanel.setBackground(new Color(r: 255, g: 182, b: 193));

    String[] pieceLabels = {"L", "Z", "I", "J", "T", "S", "O"};
    pieceInputs = new JTextField[pieceLabels.length];
    for (int i = 0; i < pieceLabels.length; i++) {
        JLabel label = new JLabel(pieceLabels[i], horizontalAlignment:JLabel.CENTER);
        label.setFont(new Font(name: "Comic Sans MS", style: Font.BOLD, size: 16));
        label.setForeground(fg: Color.WHITE);
        leftPanel.add(comp: label);

        pieceInputs[i] = new JTextField(text: "0");
        pieceInputs[i].setHorizontalAlignment(alignment: JTextField.CENTER);
        pieceInputs[i].setBackground(bg: Color.WHITE);
        pieceInputs[i].setForeground(fg: Color.PINK.darker());
        pieceInputs[i].setFont(new Font(name: "Comic Sans MS", style: Font.PLAIN, size: 14));
        leftPanel.add(pieceInputs[i]);
```

```
        pieceInputs[i].setForeground(fg: Color.PINK.darker());
        pieceInputs[i].setFont(new Font(name: "Comic Sans MS", style: Font.PLAIN, size: 14));
        leftPanel.add(pieceInputs[i]);
    }

    solveButton = new JButton(text: "💖 Solve 💖");
    solveButton.setFont(new Font(name: "Comic Sans MS", style: Font.BOLD, size: 18));
    solveButton.setBackground(new Color(r: 255, g: 105, b: 180));
    solveButton.setForeground(fg: Color.WHITE);
    solveButton.addActionListener(e -> controller.solvePuzzle(pieceCounts:getPieceCounts()));
    leftPanel.add(new JLabel());
    leftPanel.add(comp: solveButton);
    frame.add(comp: leftPanel, constraints:BorderLayout.WEST);

    mainPanel = new JPanel();
    mainPanel.setLayout(new GridLayout(rows: 2, cols: 3));
    frame.add(comp: mainPanel, constraints:BorderLayout.CENTER);

    frame.setVisible(b: true);
}
```

Description: Constructor that initializes the graphical user interface (GUI) for the puzzle solver. It creates the main frame, left panel (with input fields for piece counts), and a solve button. It also sets up the layout and adds components like labels and text fields for user input.

2. getMainPanel()

```
public JPanel getMainPanel() {
    return mainPanel;
}
```

Description: Returns the main panel where the puzzle's board and status will be displayed.

3. getFrame()

```
public JFrame getFrame() {
    return frame;
}
```

Description: Returns the main JFrame (window) of the application, which contains all GUI components.

4. getPieceCounts()

```
public int[] getPieceCounts() {
    int[] pieceCounts = new int[pieceInputs.length];
    for (int i = 0; i < pieceInputs.length; i++) {
        try {
            pieceCounts[i] = Integer.parseInt(s: pieceInputs[i].getText().trim());
        } catch (NumberFormatException ex) {
            pieceCounts[i] = 0;
        }
    }
    return pieceCounts;
}
```

Description: Retrieves the number of pieces for each shape (L, Z, I, J, T, S, O) entered by the user in the text fields. It converts the input text into an integer array, defaulting to 0 if the input is invalid.

-

3. PuzzleSolver Class:

```
public class PuzzleSolver {
    private final int BOARD_SIZE = 4;
    private final int DELAY = 300;
    private final Color[] pieceColors = {
        Color.PINK, Color.MAGENTA, Color.LIGHT_GRAY, Color.ORANGE,
        Color.YELLOW, Color.CYAN, new Color(r: 230, g: 230, b: 250)
    };

    public boolean solveForThread(int[][] board, List<int[][]> pieces, int threadIndex, JLabel statusLabel, JPanel boardP

    private boolean backtrack(int[][] board, List<int[][]> pieces, int pieceIndex, JLabel statusLabel, JPanel boardPanel)

    private void displayBoard(int[][] board, JPanel boardPanel) { ...١٨ lines }

    private boolean canPlace(int[][] board, int[][] piece, int startRow, int startCol) { ...١١ lines }

    private void placePiece(int[][] board, int[][] piece, int startRow, int startCol, int id) { ...٩ lines }

    private void removePiece(int[][] board, int[][] piece, int startRow, int startCol) { ...٩ lines }

    private int[][] rotatePiece(int[][] piece, int times) { ...٧ lines }

    private int[][] rotateOnce(int[][] piece) { ...١٠ lines }
}
```

Description: This class contains the core logic for solving the puzzle. It uses backtracking to attempt placing puzzle pieces on a 4x4 board, checking for valid placements, and rotating pieces as needed. It also handles the updating of the board visually.

Responsibilities:

Solves the puzzle using a backtracking algorithm.

Handles piece placement, removal, and rotation.

Updates the GUI with the current state of the puzzle as it tries different configurations.

Manages the delay between different puzzle-solving steps to visually show the process.

1. solveForThread(int[][] board, List<int[][]> pieces, int threadIndex, JLabel statusLabel, JPanel boardPanel)

```
public boolean solveForThread(int[][] board, List<int[][]> pieces, int threadIndex, JLabel
    List<int[][]> threadPieces = new ArrayList<>(c: pieces);
    Collections.shuffle(list: threadPieces);
    return backtrack(board, pieces: threadPieces, pieceIndex: 0, statusLabel, boardPanel);
}
```

Description: Shuffles the pieces, then attempts to solve the puzzle by invoking the backtrack method. Returns true if the puzzle is solved, otherwise false.

2. backtrack(int[][] board, List<int[][]> pieces, int pieceIndex, JLabel statusLabel, JPanel boardPanel)

```
private boolean backtrack(int[][] board, List<int[][]> pieces, int pieceIndex, JLabel statusLabel, JPanel boardPanel)
    if (pieceIndex == pieces.size()) {
        SwingUtilities.invokeLater(() -> displayBoard(board, boardPanel));
        return true;
    }

    for (int rotation = 0; rotation < 4; rotation++) {
        int[][] rotatedPiece = rotatePiece(piece: pieces.get(index: pieceIndex), times: rotation);

        for (int row = 0; row < BOARD_SIZE; row++) {
            for (int col = 0; col < BOARD_SIZE; col++) {
                if (canPlace(board, piece: rotatedPiece, startRow: row, startCol: col)) {
                    placePiece(board, piece: rotatedPiece, startRow: row, startCol: col, pieceIndex + 1);
                    SwingUtilities.invokeLater(() -> displayBoard(board, boardPanel));

                    try {
                        Thread.sleep(millis: DELAY);
                    } catch (InterruptedException ignored) {}

                    if (backtrack(board, pieces, pieceIndex + 1, statusLabel, boardPanel)) {
                        return true;
                    }
                    removePiece(board, piece: rotatedPiece, startRow: row, startCol: col);
```

Description: Recursively attempts to place pieces on the board using backtracking. It rotates each piece, checks if it can be placed, and then places it. If the board is filled correctly, it displays the board; otherwise, it removes the piece and tries again.

3. displayBoard(int[][] board, JPanel boardPanel)

```
private void displayBoard(int[][] board, JPanel boardPanel) {
    boardPanel.removeAll();
    for (int r = 0; r < BOARD_SIZE; r++) {
        for (int c = 0; c < BOARD_SIZE; c++) {
            JLabel cell = new JLabel(text: "", horizontalAlignment:JLabel.CENTER);
            cell.setOpaque(isOpaque: true);
            if (board[r][c] == -1) {
                cell.setBackground(new Color(r: 255, g: 240, b: 245));
            } else {
                cell.setBackground(pieceColors[board[r][c] - 1]);
            }
            cell.setBorder(border: BorderFactory.createLineBorder(color: Color.BLACK));
            boardPanel.add(comp: cell);
        }
    }
    boardPanel.revalidate();
    boardPanel.repaint();
}
```

Description: Updates the board display on the GUI by adding colored labels to represent placed pieces, and refreshing the display to reflect the current state of the board.

4. canPlace(int[][] board, int[][] piece, int startRow, int startCol)

```
private boolean canPlace(int[][] board, int[][] piece, int startRow, int startCol) {
    for (int r = 0; r < piece.length; r++) {
        for (int c = 0; c < piece[0].length; c++) {
            if (piece[r][c] == 1) {
                int row = startRow + r, col = startCol + c;
                if (row >= BOARD_SIZE || col >= BOARD_SIZE || board[row][col] != -1) return false;
            }
        }
    }
    return true;
}
```

Description: Checks if a given piece can be placed on the board at a specified position, ensuring no overlap with already placed pieces and that the piece fits within the board.

5. placePiece(int[][] board, int[][] piece, int startRow, int startCol, int id)

```
private void placePiece(int[][] board, int[][] piece, int startRow, int startCol, int id) {
    for (int r = 0; r < piece.length; r++) {
        for (int c = 0; c < piece[0].length; c++) {
            if (piece[r][c] == 1) {
                board[startRow + r][startCol + c] = id;
            }
        }
    }
}
```

Description: Places a given piece on the board at a specified position, updating the board with the piece's ID.

6. removePiece(int[][] board, int[][] piece, int startRow, int startCol)

```
private void removePiece(int[][] board, int[][] piece, int startRow, int startCol) {
    for (int r = 0; r < piece.length; r++) {
        for (int c = 0; c < piece[0].length; c++) {
            if (piece[r][c] == 1) {
                board[startRow + r][startCol + c] = -1;
            }
        }
    }
}
```

Description: Removes a piece from the board at a specified position, setting the affected cells back to the initial empty value (-1).

7. rotatePiece(int[][] piece, int times)

```
private int[][] rotatePiece(int[][] piece, int times) {
    int[][] rotated = piece;
    for (int i = 0; i < times; i++) {
        rotated = rotateOnce(piece: rotated);
    }
    return rotated;
}
```

Description: Rotates the given piece a specified number of times (0-3 rotations) and returns the resulting rotated piece.

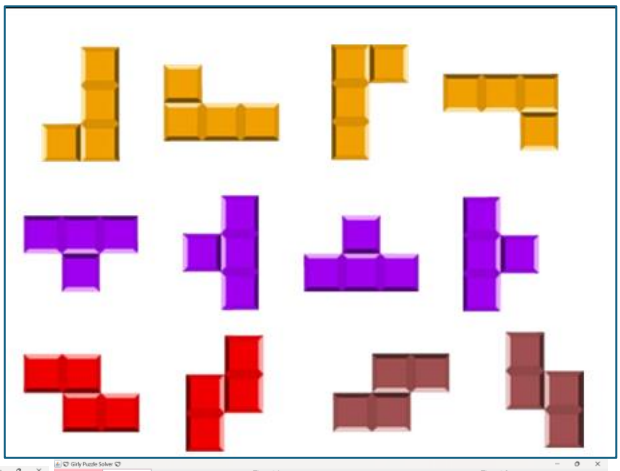
8. rotateOnce(int[][] piece)

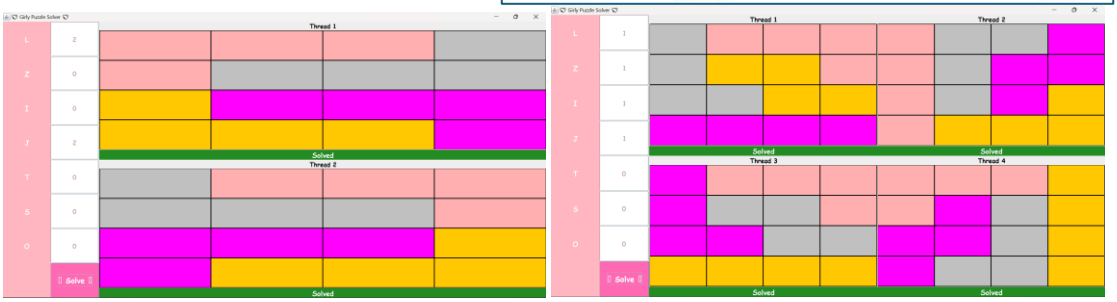
```
    private int[][] rotateOnce(int[][] piece) {
        int rows = piece.length, cols = piece[0].length;
        int[][] rotated = new int[cols][rows];
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                rotated[c][rows - 1 - r] = piece[r][c];
            }
        }
        return rotated;
    }
}
```

Description: Rotates the piece 90 degrees clockwise once and returns the rotated piece.

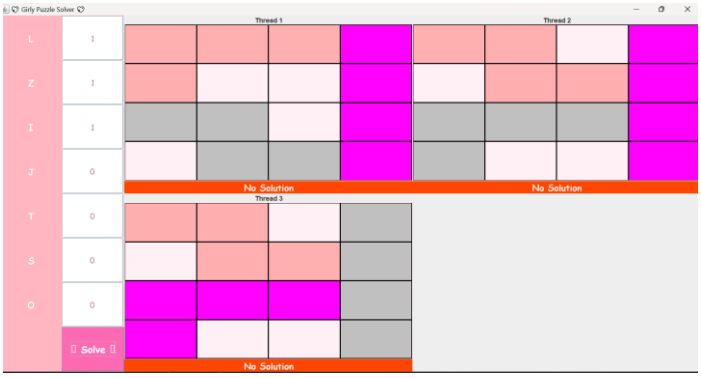
Graphical user interface:

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