**Make a Square Game Documentation**

**Project Description:**

Our project is based on the game of 'Tetris.' In this endeavor, we handle various pieces of different shapes, aiming to fit all of them into a 4x4 grid. The objective is to arrange the pieces without any gaps, ensuring that all the components find a suitable place within the grid.

**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

To simplify the user experience in the GUI, users can specify the quantity of each available shape they wish to use. The program will then simulate the process of placing the specified number of pieces for each shape into the grid.

**The workflow procedure:**

The core concept of the program involves implementing a backtracking algorithm using threads. Each thread manages its own 4x4 grid and begins by strategically placing valid pieces. Subsequently, it invokes another thread, acting as a child, to complete the process.

The advantage of this solution is that the parent thread does not pause or wait for its child to finish, enabling both of them to proceed with their work concurrently. Both threads operate simultaneously, continuously adding pieces and generating children in an uninterrupted manner.

In summary, the entire tree formed by the backtracking algorithm operates concurrently, leading to a faster and more efficient way to achieve an optimal result.

The program initiates by preprocessing all the valid arrangements for placing each piece, considering all possible rotations. Commencing with the first grid, which begins as an empty grid, the program iterates through all the user-inputted valid pieces to be placed.

Each thread checks for available space for a specific piece, adds it, calls another thread, passes the updated grid to it, removes the added piece, and proceeds to try with another configuration.

More briefly, each level of the tree corresponds to a number of pieces equal to the level number. Upon reaching a solution, the entire process stops, and the result is sent to the GUI to commence simulating the path, which has been saved in each grid.

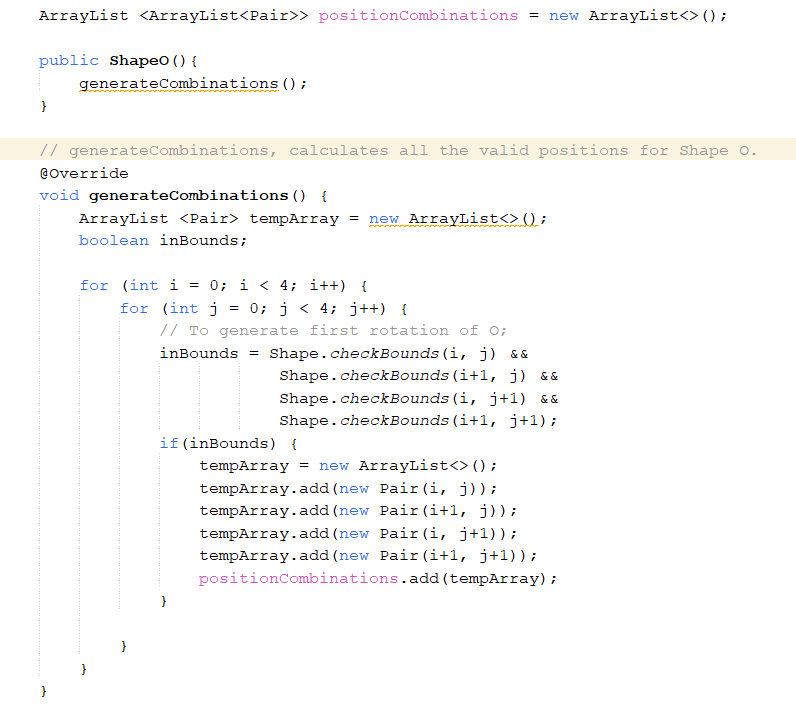
**Team Members:**

* Ahmed Bassem Ouda
* Ahmed Khaled Gomaa
* Ahmed Khaled Nasr
* Ahmed Ashraf Ahmed
* Ahmed Khaled Abd-allah
* Ahmed Gamal Adb-elhay
* Ahmed Ramadan Ragab

**Code Documentation**

**Generating the valid pieces:**

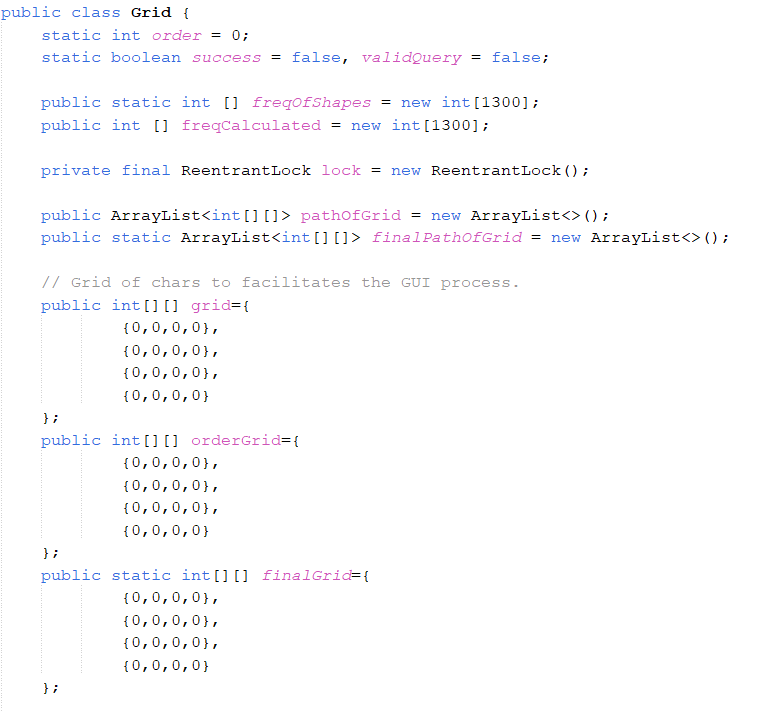
We begin by preprocessing all the valid placements for a single type of shape and store them in an array of arrays of pairs. Each array of pairs represents a distinct way to place a shape, as shown below.



Iterating over the array list "positionCombinations" simply facilitates and enhances the manipulation of the pieces, making the process more efficient and straightforward.

**Manipulating a Grid:**

Any instance of the Grid Class will contain the following:

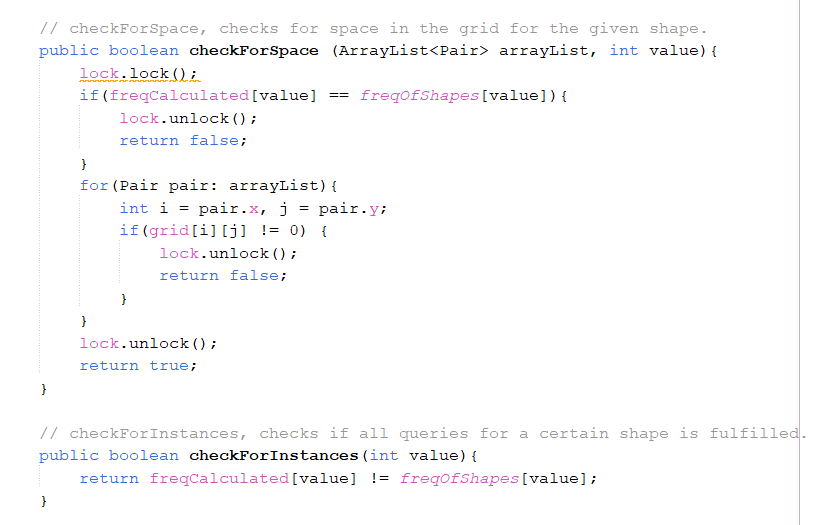


First of all, **success** is used to determine if a solution is found or not. **freqOfShapes** is a frequency array that stores the input query entered by the user. **freqOfCalculated** is a frequency array that keeps track of all the pieces placed in a single grid in order to compare it later with the **freqOfShapes** to determine if a solution is reached.

Then we have **pathOfGrid** that saves the path of a single grid so far. When reaching a solution the **pathOfGrid** is copied into the **finalPathOfGrid**, which will be visualized by the GUI.

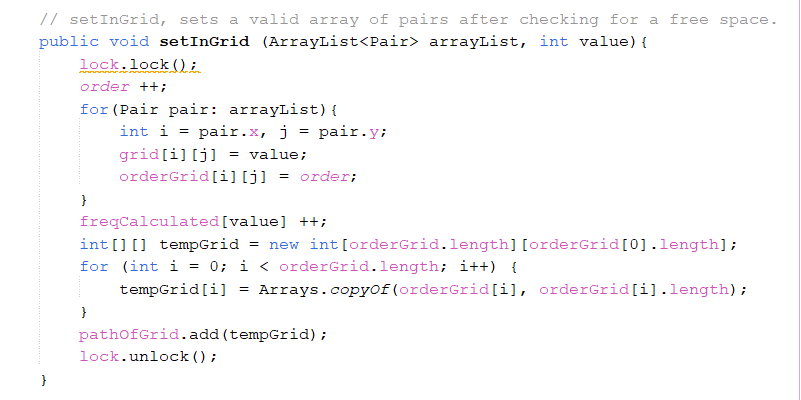
Three grids are defined, **grid** for storing the pieces as letters, **orderGrid** to store the pieces by their order of entering and a **finalGrid** that includes the solution when reached.

Then for the needed methods we have:



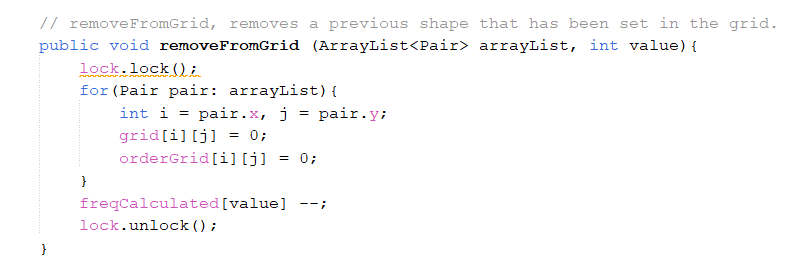
In method **checkForSpace**, given a pieces represented as array of pairs, will determine if the needed space for placing this piece is available or not.

In method **checkForInstances**, given a shape type, it determines if it’s available to add another pieces of the same kind by checking the **freqOfCalculated**.

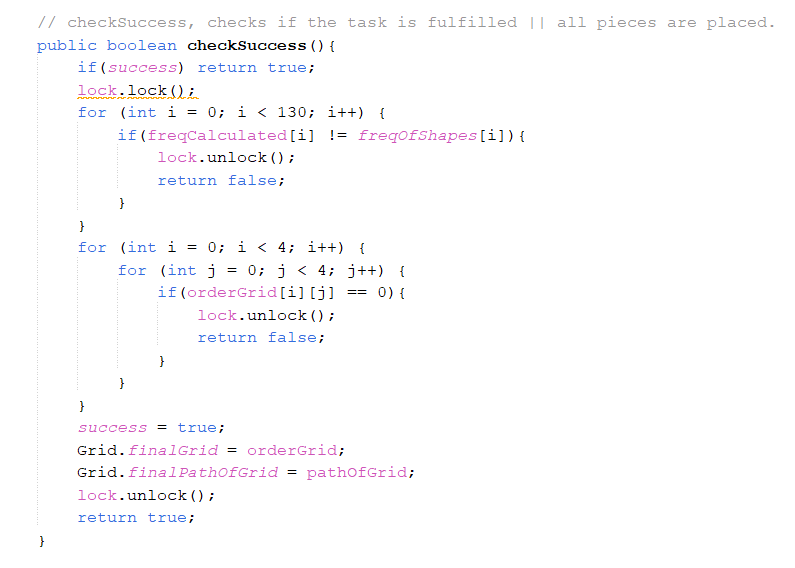


In method **setInGrid**, a piece is set and the involved grids and the **freqOfCalculated** are updated and the **orderGrid** is saved in the **pathOfGrid**.

In method **removeFromGrid**, given the location of wanted piece, it’s place in the involved grids will be erased and the **freqOfCalculated** is reduced.

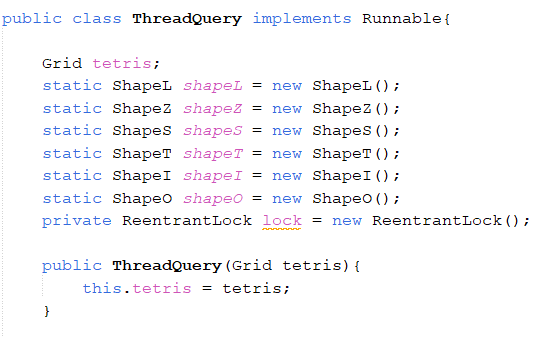


Whenever a solution is found, by comparing **freqOfCalculated** with **freqOfShapes** to ensures that all the needed pieces all placed, **success**, **finalGrid** and the **finalPathOfGrid** are set.



**Threading Process:**

Starting by identifying instances of all the shapes and a grid which the thread will work on, as shown, each thread receives its initial thread within the constructor.



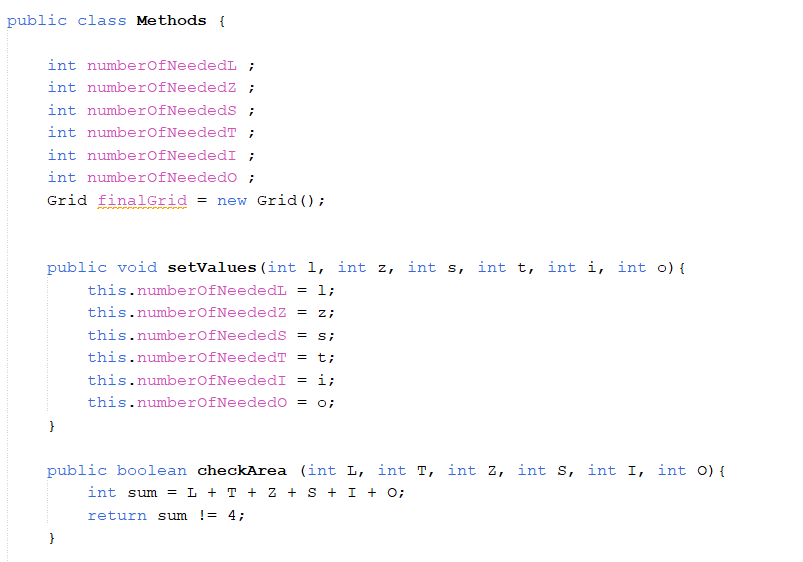
Then each thread starts working throw iterating on all ways of placing a single shape, shape by shape. In each iteration, it is checked if there is a space for the piece or not, if yes it is placed, sent to another thread, removed, and the loop continues.

Whenever **success** is true, the treads terminates.



**Receiving the input and starting the threading:**

Starting by placed the input values in containers and checking the validity.

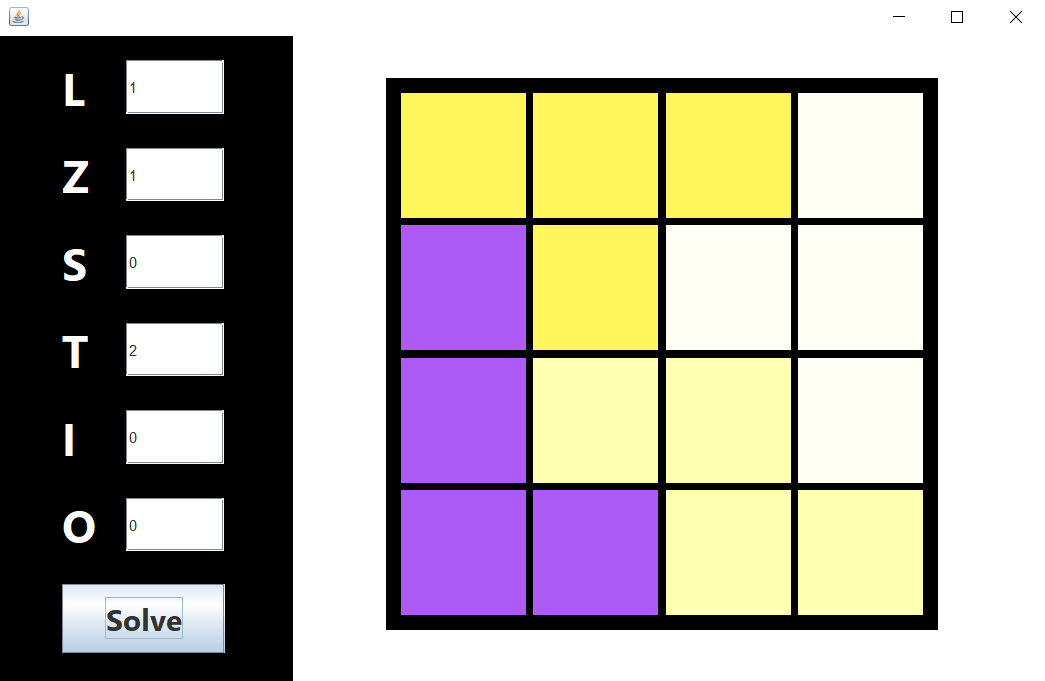


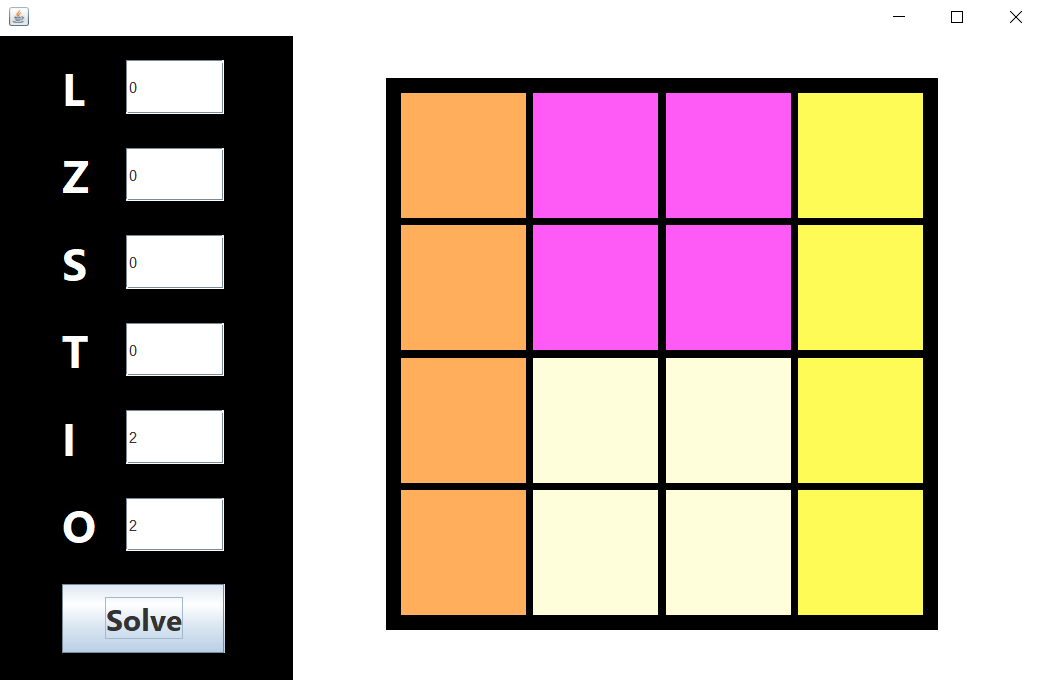
Then a frequency array is made where is later sent to the Grid Class.

Then the first thread is declared with an empty initial grid and starts working.



**Examples:**





If no solution is found :

