MEF ÜNİVERSİTESİ

ÖĞRENMEYE SINIRSIZ ÖZGÜRLÜK

Group Name: Comp'ilers

Project: Tetris 2048

Student Names:	***Student ID:***
Onur Keleş	042001049
Beyzanur Yıldız	042101179
Ahmet Yasir Beydili	042101071
Mert Durgun	042101180

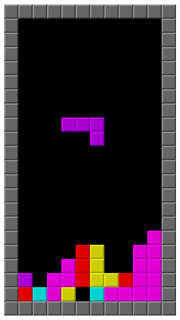
Abstract

We run our main function under our package named game in our project called *Tetris 2048* (a combination of *Tetris* and *2048* games) that we prepared using the ***StdDraw*** library in the *Java* programming language. On the gaming side, move the boxes that will do that on the grid with the help of *x* and *y* coordinates to ensure their positioning and rotation. The boxes with the same numbers on top of each other will be merged, and the sum of their numbers will be written on the merged box, which will also be moved to the lower grid, while the player can see the score on the screen.

1. Introduction

1.1 How Does *Tetris* Work?

The playing field in *Tetris* is 10x20 blocks squared. The game begins with a single block falling from the top of the screen. The falling block must be rotated and positioned properly so that it fits into the available space at the bottom of the screen.



More blocks will fall from the top of the screen faster as the game continues. The blocks are available in seven various shapes, or ***Tetrominoes*** (which is a different combination of four blocks).

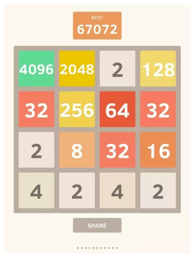
The player has to position the blocks in such a way that they form a solid horizontal line across the screen.

When a line is finished, it vanishes and the player wins points, and the game ends if the player allows the blocks to stack to the top of the screen. The game may also finish if the player runs out of time or reaches a specific score. And the challenge of the game rises as the player continues, with the blocks falling faster and in more complex patterns.

The goal is to clear as many lines as possible before the game ends to get the best possible score.

1.2 How Does *2048* Work?

The main purpose of the game is to get the number 2048 on a single tile. The game is played on a 4x4 grid, with each tile having a power of 2 (numbers from 2^0 to 2^n where n is an arbitrary number in this case).



For movement of the tiles, the player can swipe them through the rotations like left, right, up, or down. When two identical tiles collide, they merge into a single tile with the sum of the two numbers, like they (tiles) do in *Tetris* with blocks.

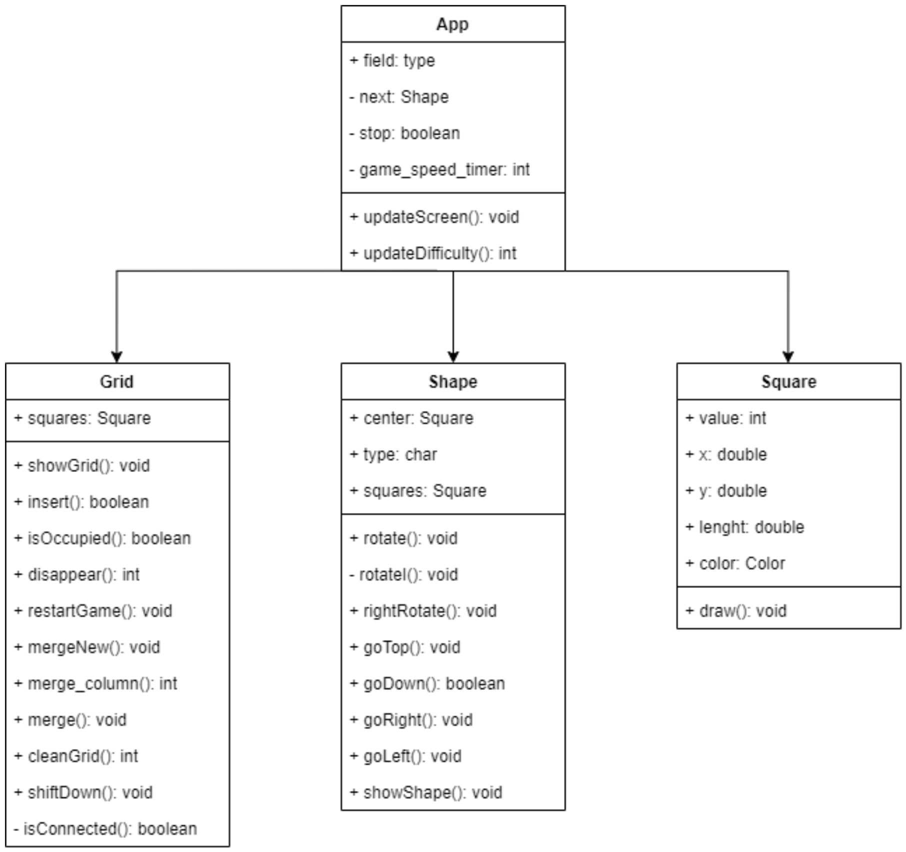
The game ends when there are no more moves available in such cases like no more free grid spaces available or if there are no adjacent tiles with the same number).

The sum of the numbers on all of the tiles on the grid is the total player score.

1.3 Purpose of this Project

The purpose of the project is to bring out the *Tetris 2048* game using the exact Java programming language and its *StdDraw* library, and more importantly and fundamentally, implementing the appropriate methods and design to perform a clean merge by taking the advantage of *Tetris* and *2048* games appearing as two suitable games.

2. Our Solution For The Project



2.1.1 Constructing the Grid

This program is the *Java* code for the *Tetris* game. It comprises a class called *Grid*, which simulates the Tetris game grid. *Grid* has one data variable for the square matrix with given row and column values. It provides methods for displaying, drawing the cells and lines, and drawing the boundaries of *Grid,* removing completed rows/adding new shapes to the grid. It also offers a method for determining whether a tile is occupying the grid cell with the given row and column indexes.

- *showGrid* defines the game grid's cells and lines. It iterates through each cell of the game grid, drawing the tile if the cell is already filled by a tile. and it also draws the grid's inner lines and has particular values to make room for a menu on the right side of the game.

- *disappear* will remove the row with the given index. After removing every square out of the grid, it will lower the current height level of the tiles in the game, then it adds the score to the *totalscore* of the game.

- *restartgame* will remove every tile currently in the game, reset every value and will bring the current tile to the top of the grid.

- *insert* creates the stationary values in the grid, it will return false if the shape of the tile is in an illegal position.

- *isOccupied* checks if the given coordinate is occupied or not.

- *isConnected* checks if the given tile in the given coordinate is faced to another tile in any other 4 faces.

- *merge* will connect 2 given tiles to each other and will add their score to the new one. It will remove one of the tiles if both of the tiles are connected and the values are not *null*. It will remove the tile at top, then add the value to the bottom one and it will do this until the tiles are not connected to each other.
-

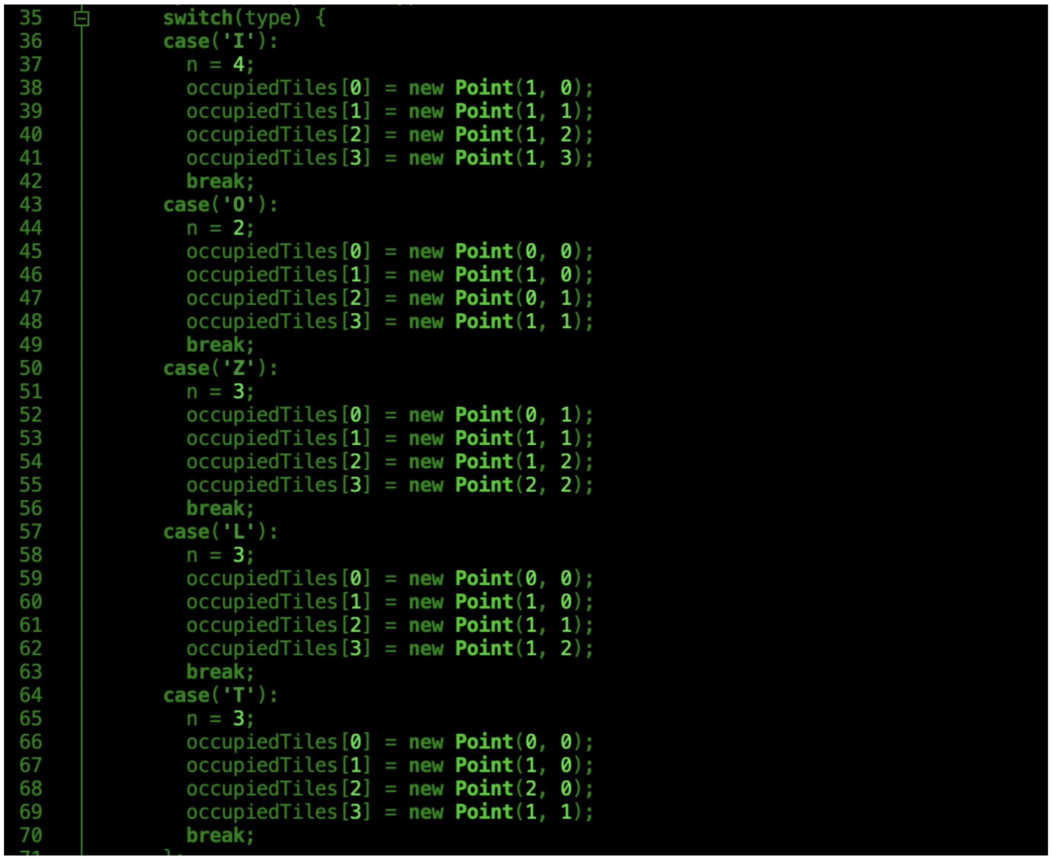
2.1.3 Constructing the skeleton of Tetris 2048

The game begins by displaying a menu with a single start button. When the game begins, tetrominoes begin to fall from the top of the screen, and the player can move them left, right, down, or rotate them with the arrow keys. The objective of the game is to form entire horizontal lines of tiles on the game grid, which will then vanish and gain points. The game is over when the tetrominoes reach the top of the grid.

The code initially defines the size of the game grid and drawing canvas, as well as the coordinate system scale used for drawing. The game menu is then displayed when the first tetromino is created. The main game loop checks for user keyboard input to move or rotate the active tetromino and also causes the tetromino to fall down one row every ten loop iterations. When the tetromino can no longer fall down, it is placed on the game grid and locked in place. When a horizontal line of tiles is finished, it is removed from the grid and the player receives points. The game is over when the game grid is filled.

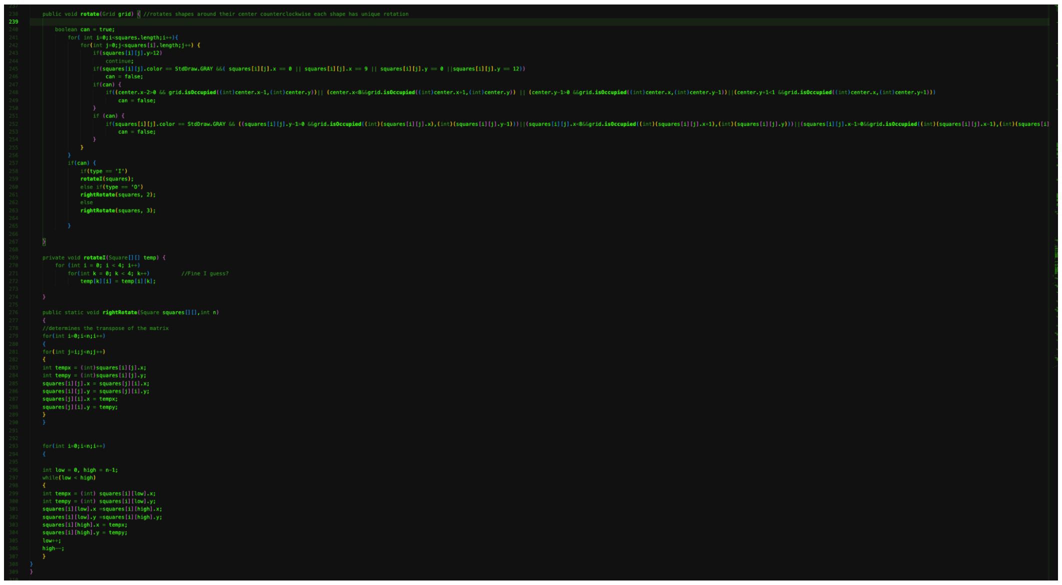
The game grid and tetrominoes are represented by two classes in the code. *Shape* class defines the tetrominos' shape, rotation, and movement. *Square* class determines the colors used for each shape and will draw the values into the square. Square has 2 different constructor, one of them is for creating a basic square, the other is for dummy square shape. *Grid* class represents the game grid and has methods for updating the grid when a tetromino is placed on it, checking for completed lines, and displaying the grid as well as the active tetromino.

2.1.4 Constructing the Tetrominoes



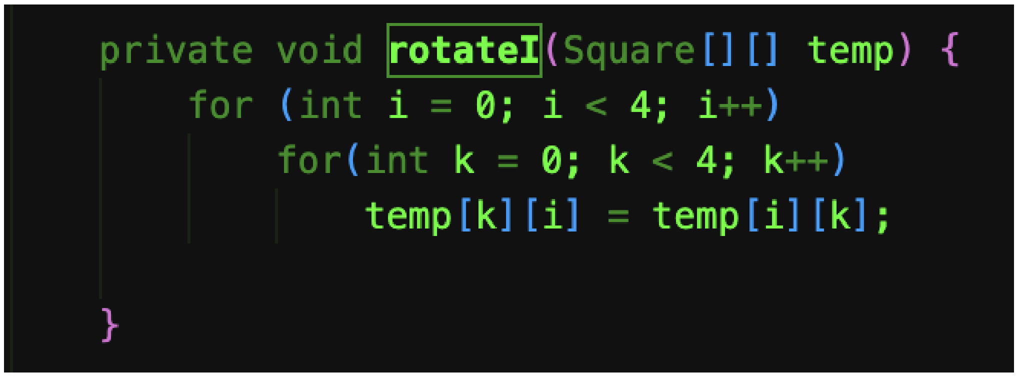
```
        switch(type) {
        case('I'):
          n = 4;
          occupiedTiles[0] = new Point(1, 0);
          occupiedTiles[1] = new Point(1, 1);
          occupiedTiles[2] = new Point(1, 2);
          occupiedTiles[3] = new Point(1, 3);
          break;
        case('O'):
          n = 2;
          occupiedTiles[0] = new Point(0, 0);
          occupiedTiles[1] = new Point(1, 0);
          occupiedTiles[2] = new Point(0, 1);
          occupiedTiles[3] = new Point(1, 1);
          break;
        case('Z'):
          n = 3;
          occupiedTiles[0] = new Point(0, 1);
          occupiedTiles[1] = new Point(1, 1);
          occupiedTiles[2] = new Point(1, 2);
          occupiedTiles[3] = new Point(2, 2);
          break;
        case('L'):
          n = 3;
          occupiedTiles[0] = new Point(0, 0);
          occupiedTiles[1] = new Point(1, 0);
          occupiedTiles[2] = new Point(1, 1);
          occupiedTiles[3] = new Point(1, 2);
          break;
        case('T'):
          n = 3;
          occupiedTiles[0] = new Point(0, 0);
          occupiedTiles[1] = new Point(1, 0);
          occupiedTiles[2] = new Point(2, 0);
          occupiedTiles[3] = new Point(1, 1);
          break;
```

- The *switch* statement is used to determine the occupied tiles in the tetromino's tile matrix based on its kind. The constructor method's type parameter determines the type of the tetromino, which can be one of seven distinct types: 'I', 'O', 'Z', 'L', 'J', 'T', and 'S'. Each type also has a smaller version for the next tetramonio and each type corresponds to a particular tetromino shape, which is represented by a nxn tile grid which n can be 2, 3 or 4. The *switch* statement examines the type parameter against each of the possible scenarios as its argument. If the type matches a specific case, the associated block of code is executed to identify whether tiles in the tetromino's tile matrix are occupied.



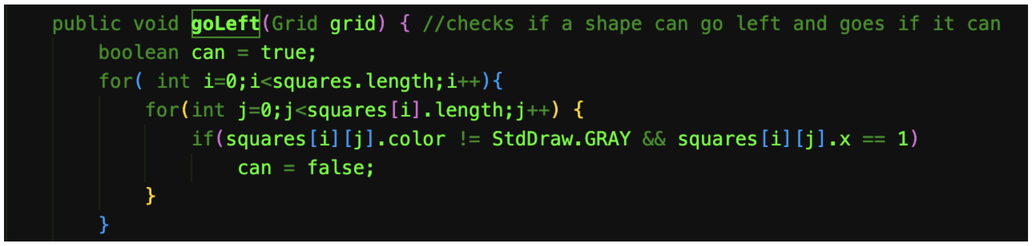
```
public void rotate(Grid grid) { //rotates shapes around their center counterclockwise each shape has unique rotation

    boolean can = true;
        for( int i=0;i<squares.length;i++){
            for(int j=0;j<squares[i].length;j++) {
                if(squares[i][j].y>12)
                    continue;
                if(squares[i][j].color == StdDraw.GRAY &&( squares[i][j].x == 0 || squares[i][j].x == 9 || squares[i][j].y == 0 ||squares[i][j].y == 12))
                    can = false;
                if(can) {
                    if((center.x-2>0 && grid.isOccupied((int)center.x-1,(int)center.y))|| (center.x<8&&grid.isOccupied((int)center.x+1,(int)center.y)) || (center.y-1>0 &&grid.isOccupied((int)center.x,(int)center.y-1))||(center.y+1<1 &&grid.isOccupied((int)center.x,(int)center.y+1)))
                        can = false;
                }
                if (can) {
                    if(squares[i][j].color == StdDraw.GRAY && ((squares[i][j].y-1>0 &&grid.isOccupied((int)(squares[i][j].x),(int)(squares[i][j].y-1)))||(squares[i][j].x<8&&grid.isOccupied((int)(squares[i][j].x+1),(int)(squares[i][j].y)))||(squares[i][j].x-1>0&&grid.isOccupied((int)(squares[i][j].x-1),(int)(squares[i]
                        can = false;
                }
            }
        }
        if(can) {
            if(type == 'I')
            rotateI(squares);
            else if(type == 'O')
            rightRotate(squares, 2);
            else
            rightRotate(squares, 3);

        }

}

private void rotateI(Square[][] temp) {
    for (int i = 0; i < 4; i++)
        for(int k = 0; k < 4; k++)          //Fine I guess?
            temp[k][i] = temp[i][k];

}

public static void rightRotate(Square squares[][],int n)
{
//determines the transpose of the matrix
for(int i=0;i<n;i++)
{
for(int j=i;j<n;j++)
{
int tempx = (int)squares[i][j].x;
int tempy = (int)squares[i][j].y;
squares[i][j].x = squares[j][i].x;
squares[i][j].y = squares[j][i].y;
squares[j][i].x = tempx;
squares[j][i].y = tempy;
}
}


for(int i=0;i<n;i++)
{

int low = 0, high = n-1;
while(low < high)
{
int tempx = (int) squares[i][low].x;
int tempy = (int) squares[i][low].y;
squares[i][low].x =squares[i][high].x;
squares[i][low].y =squares[i][high].y;
squares[i][high].x = tempx;
squares[i][high].y = tempy;
low++;
high--;
}
}
}
```

- The method *rotate* in the class *Tile* is in the role of turning the current tile counterclockwise. First, the method determines whether the direction string is valid counterclockwise. If the rotation can't be done because the direction is invalid, the method returns false. If the direction is correct, the method rotates the tile. The rotation is done by constructing a new 2D array with the same dimensions as the current tile but with the rows and columns swapped, resulting in a 90-degree rotation of the tile. The method then checks to see if the rotated tile would overlap with any occupied cells on the game grid. If there is no overlap, the method changes the tile's state to the rotated tile and returns true to indicate that the rotation was successful. If there is overlap, the method returns false, indicating that the rotation failed.

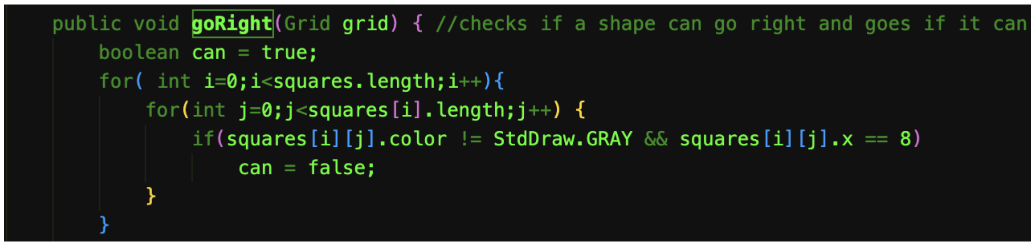


```
private void rotateI(Square[][] temp) {
    for (int i = 0; i < 4; i++)
        for(int k = 0; k < 4; k++)
            temp[k][i] = temp[i][k];

}
```

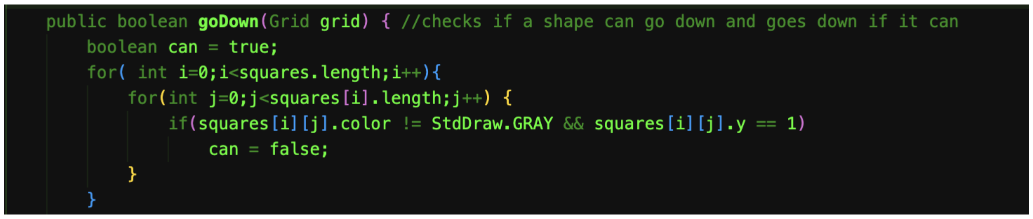
- The method *rotateI* is used by the *Tile* class' *rotate* method. It rotates an I-shaped tile 90 degrees counterclockwise. As an argument, a 2D array of *Tile* objects is passed to the method. The array represents the game grid's current state, and the method returns a new 2D array of *Tile* objects with the same dimensions as the previous array, with all of its elements set to null. The method then iterates through the original array, determining its new location in the rotated array, and copying the *Tile* object from the old place to the new position. To rotate the I-shaped tile, the process swaps the top and bottom square positions with the left and right square positions. After all elements in the rotated array have been copied to their new positions, the method assigns the rotated array to the original array so that the rotated tile is reflected in the game grid.



```
public void goLeft(Grid grid) { //checks if a shape can go left and goes if it can
    boolean can = true;
    for( int i=0;i<squares.length;i++){
        for(int j=0;j<squares[i].length;j++) {
            if(squares[i][j].color != StdDraw.GRAY && squares[i][j].x == 1)
                can = false;
        }
    }
```

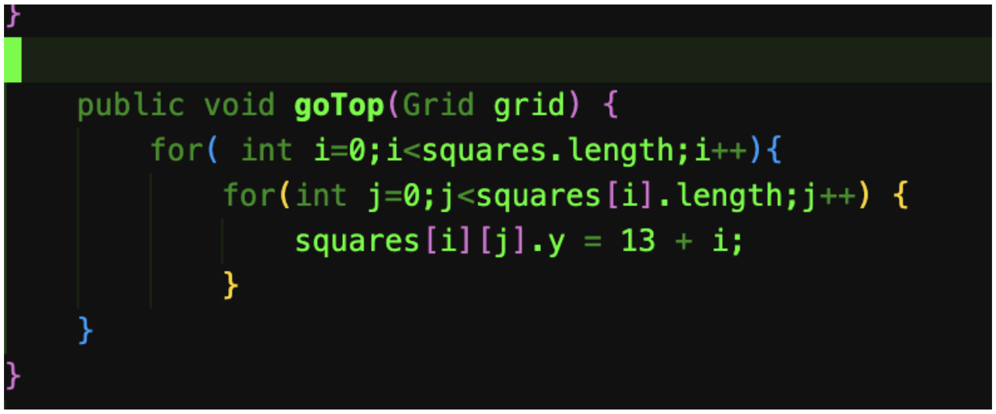


```
public void goRight(Grid grid) { //checks if a shape can go right and goes if it can
    boolean can = true;
    for( int i=0;i<squares.length;i++){
        for(int j=0;j<squares[i].length;j++) {
            if(squares[i][j].color != StdDraw.GRAY && squares[i][j].x == 8)
                can = false;
        }
    }
```



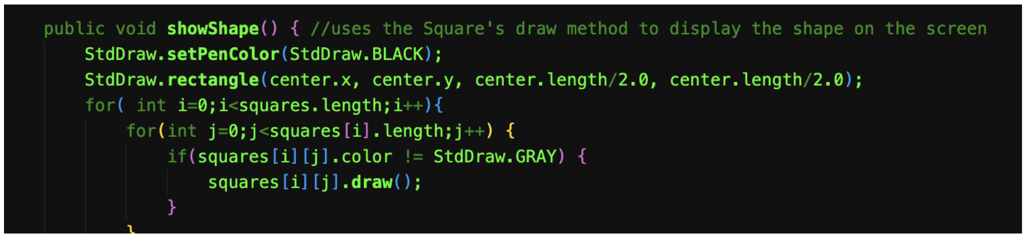
```
public boolean goDown(Grid grid) { //checks if a shape can go down and goes down if it can
    boolean can = true;
    for( int i=0;i<squares.length;i++){
        for(int j=0;j<squares[i].length;j++) {
            if(squares[i][j].color != StdDraw.GRAY && squares[i][j].y == 1)
                can = false;
        }
    }
```

- *goLeft*, *goRight and goDown* is carrying the current shape to the desired direction by one block, to do this the shape should meet the conditions.



```
}

    public void goTop(Grid grid) {
        for( int i=0;i<squares.length;i++){
            for(int j=0;j<squares[i].length;j++) {
                squares[i][j].y = 13 + i;
            }
        }
}
```

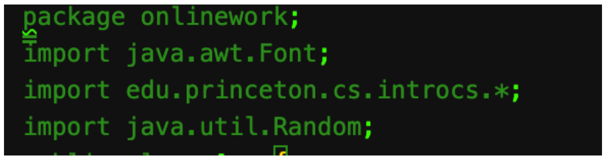
- *goTop* is a special method for restarting the game. It will carry the current tile to the top of the game.



```
public void showShape() { //uses the Square's draw method to display the shape on the screen
    StdDraw.setPenColor(StdDraw.BLACK);
    StdDraw.rectangle(center.x, center.y, center.length/2.0, center.length/2.0);
    for( int i=0;i<squares.length;i++){
        for(int j=0;j<squares[i].length;j++) {
            if(squares[i][j].color != StdDraw.GRAY) {
                squares[i][j].draw();
            }
```

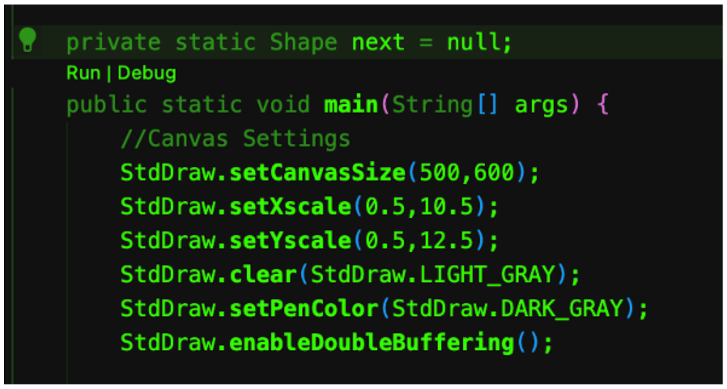
- *showShape* is drawing the shape for displaying.

2.1.5 Constructing the App



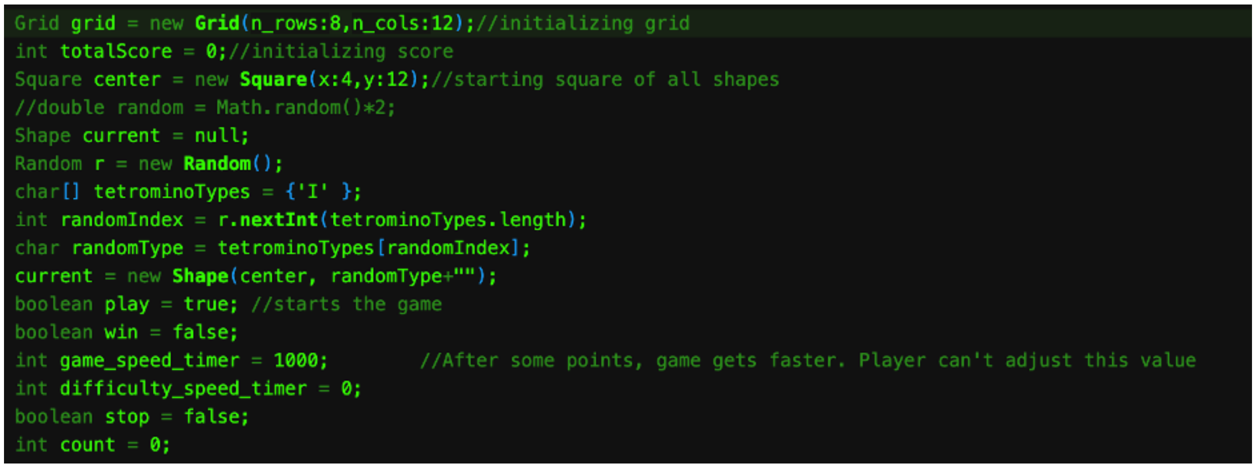
```
package onlinework;
import java.awt.Font;
import edu.princeton.cs.introcs.*;
import java.util.Random;
```

- Importing the necessary libraries for the game like *StdDraw* for graphics, *Random* for generating random numbers etc.



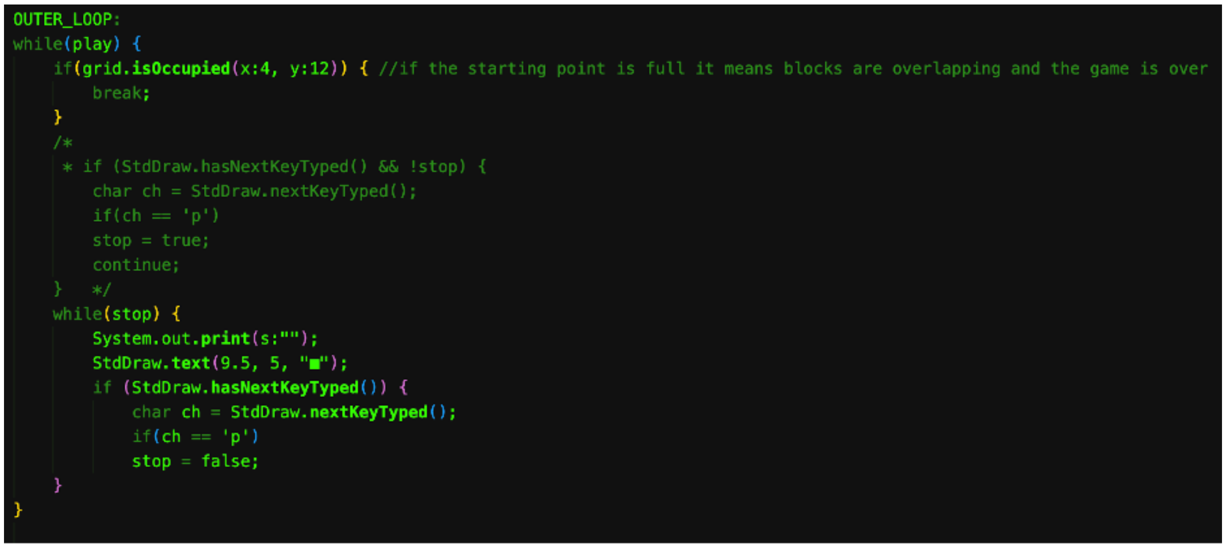
```
private static Shape next = null;
Run | Debug
public static void main(String[] args) {
    //Canvas Settings
    StdDraw.setCanvasSize(500,600);
    StdDraw.setXscale(0.5,10.5);
    StdDraw.setYscale(0.5,12.5);
    StdDraw.clear(StdDraw.LIGHT_GRAY);
    StdDraw.setPenColor(StdDraw.DARK_GRAY);
    StdDraw.enableDoubleBuffering();
```

- Initializing the canvas for the game, setting the size and scale of the canvas, and also enabling double buffering by using *StdDraw* library.



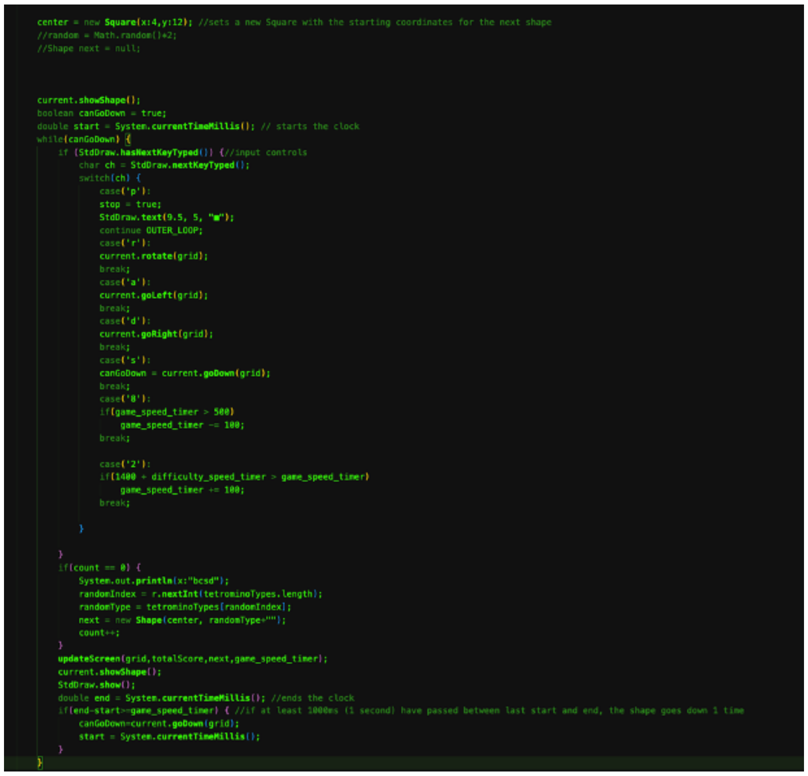
```
Grid grid = new Grid(n_rows:8,n_cols:12);//initializing grid
int totalScore = 0;//initializing score
Square center = new Square(x:4,y:12);//starting square of all shapes
//double random = Math.random()*2;
Shape current = null;
Random r = new Random();
char[] tetrominoTypes = {'I' };
int randomIndex = r.nextInt(tetrominoTypes.length);
char randomType = tetrominoTypes[randomIndex];
current = new Shape(center, randomType+"");
boolean play = true; //starts the game
boolean win = false;
int game_speed_timer = 1000;        //After some points, game gets faster. Player can't adjust this value
int difficulty_speed_timer = 0;
boolean stop = false;
int count = 0;
```

- Creating a *grid* object representing the game board, setting the starting score to zero, and initializing the first shape to fall from the top center of the board.



```
OUTER_LOOP:
while(play) {
    if(grid.isOccupied(x:4, y:12)) { //if the starting point is full it means blocks are overlapping and the game is over
        break;
    }
    /*
     * if (StdDraw.hasNextKeyTyped() && !stop) {
        char ch = StdDraw.nextKeyTyped();
        if(ch == 'p')
        stop = true;
        continue;
    }   */
    while(stop) {
        System.out.print(s:"");
        StdDraw.text(9.5, 5, "■");
        if (StdDraw.hasNextKeyTyped()) {
            char ch = StdDraw.nextKeyTyped();
            if(ch == 'p')
            stop = false;
    }
}
```

- Running the game loop, which executes as long as the player has not lost or won the game. Within this loop, the code checks if the starting point for the shape is occupied, indicating the game is over, and it also checks for any input from the player to move the shape left, right, down, or rotate it.



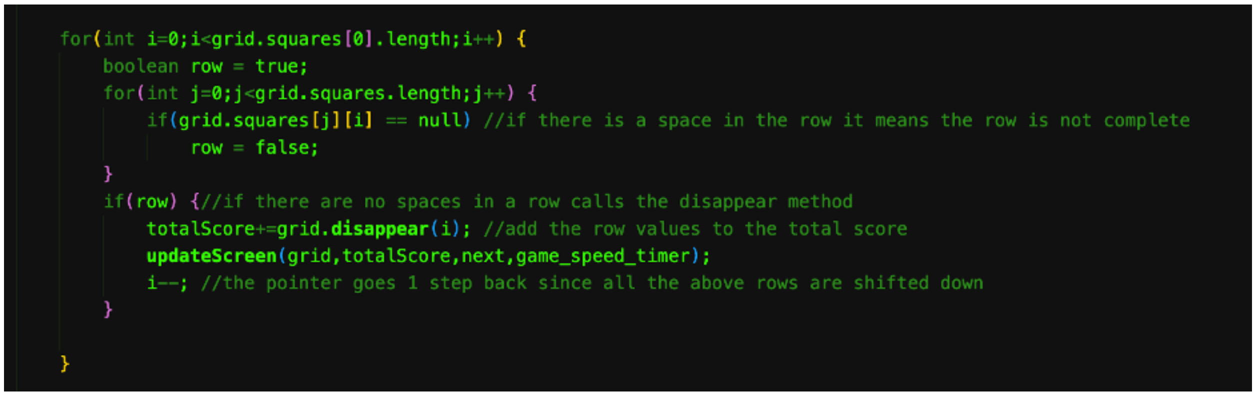
```
center = new Square(x:4,y:12); //sets a new Square with the starting coordinates for the next shape
//random = Math.random()*2;
//Shape next = null;



current.showShape();
boolean canGoDown = true;
double start = System.currentTimeMillis(); // starts the clock
while(canGoDown) {
    if (StdDraw.hasNextKeyTyped()) {//input controls
        char ch = StdDraw.nextKeyTyped();
        switch(ch) {
            case('p'):
            stop = true;
            StdDraw.text(9.5, 5, "■");
            continue OUTER_LOOP;
            case('r'):
            current.rotate(grid);
            break;
            case('a'):
            current.goLeft(grid);
            break;
            case('d'):
            current.goRight(grid);
            break;
            case('s'):
            canGoDown = current.goDown(grid);
            break;
            case('8'):
            if(game_speed_timer > 500)
                game_speed_timer -= 100;
            break;

            case('2'):
            if(1400 + difficulty_speed_timer > game_speed_timer)
                game_speed_timer += 100;
            break;

        }

    }
    if(count == 0) {
        System.out.println(x:"bcsd");
        randomIndex = r.nextInt(tetrominoTypes.length);
        randomType = tetrominoTypes[randomIndex];
        next = new Shape(center, randomType+"");
        count++;
    }
    updateScreen(grid,totalScore,next,game_speed_timer);
    current.showShape();
    StdDraw.show();
    double end = System.currentTimeMillis(); //ends the clock
    if(end-start>=game_speed_timer) { //if at least 1000ms (1 second) have passed between last start and end, the shape goes down 1 time
        canGoDown=current.goDown(grid);
        start = System.currentTimeMillis();
    }
}
```

- Basically, the loop also updates the screen to show the current state of the game board, the player's score, and the next shape to fall. It also calculates the time between each falling movement of the shape, making the game faster as the player scores more points. In a more detailed manner, the variable *canGoDown* is a boolean that indicates whether or not the current shape is able to go downwards. The loop continues to execute as long as *canGoDown* is true.

  Within the loop, the application uses *StdDraw.hasNextKeyTyped*() to determine whether any keys have been pressed by the user. If a key is pressed, the program responds to the input by doing the relevant action, such as moving or rotating the shape.

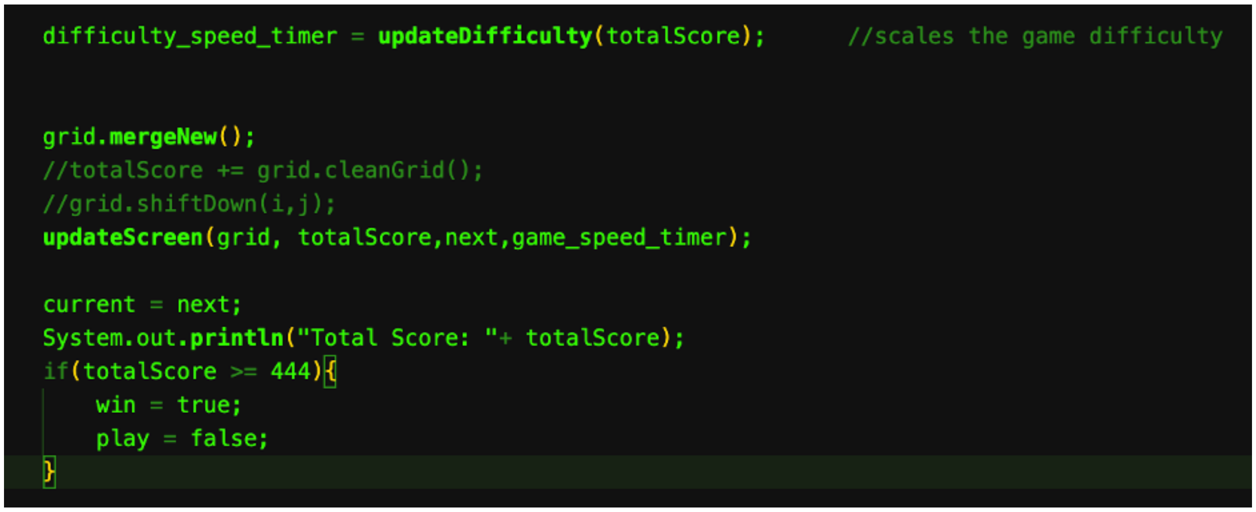
  The program then calls the method *updateScreen*, which redraws the game board, the score, and the next shape on the screen. It then uses *StdDraw.show*() to display the updated drawing on the screen.

  Finally, the program determines whether enough time has passed for the current shape to move downwards once more. If this is the case, it calls the current to move the form down one place.*canGoDown* is set to the result of the method *goDown*(*grid*) call. If the shape can no longer go downward, *canGoDown* is set to false, and the loop is terminated.



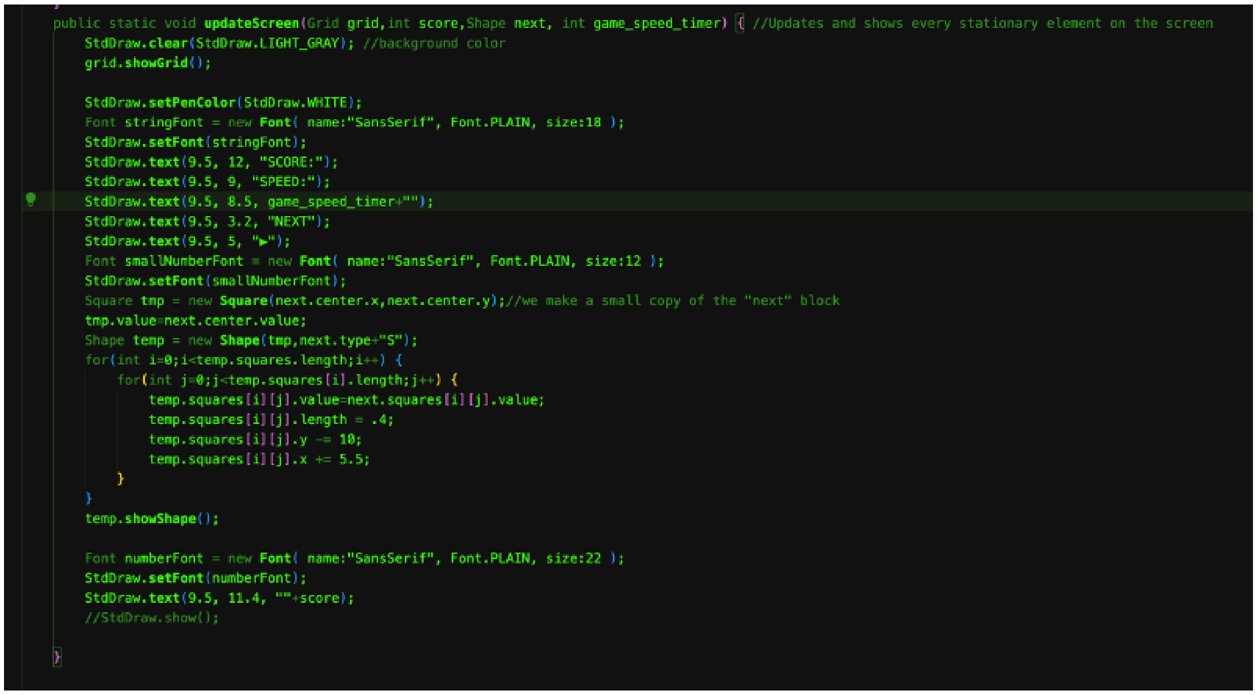
```
for(int i=0;i<grid.squares[0].length;i++) {
    boolean row = true;
    for(int j=0;j<grid.squares.length;j++) {
        if(grid.squares[j][i] == null) //if there is a space in the row it means the row is not complete
            row = false;
    }
    if(row) {//if there are no spaces in a row calls the disappear method
        totalScore+=grid.disappear(i); //add the row values to the total score
        updateScreen(grid,totalScore,next,game_speed_timer);
        i--; //the pointer goes 1 step back since all the above rows are shifted down
    }

}
```

- This loop iterates through the columns of *grid*. It then loops through the grid's rows, checking whether each square in the current column is occupied. *row* is set to false if any of the squares in the column are null. *row* will still be true at the end of the loop if all of the squares in the column are occupied. If *row* is true, it indicates that the entire row is occupied, and the disappear method of the *grid* is called on that row. The result given by disappear is added to *totalScore*, and the method *updateScreen* is invoked to update the graphics on the screen. Finally, *i* is decremented so that the loop can check the row that was just shifted down (since all of the rows above it were eliminated).



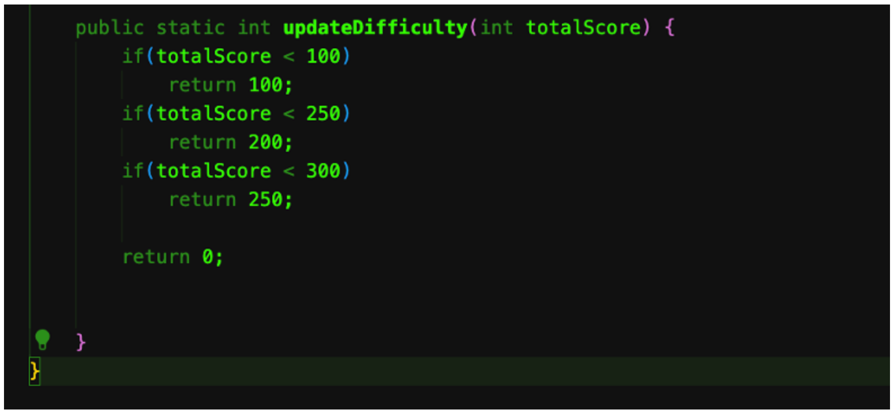
```
difficulty_speed_timer = updateDifficulty(totalScore);      //scales the game difficulty


grid.mergeNew();
//totalScore += grid.cleanGrid();
//grid.shiftDown(i,j);
updateScreen(grid, totalScore,next,game_speed_timer);

current = next;
System.out.println("Total Score: "+ totalScore);
if(totalScore >= 444){
    win = true;
    play = false;
}
```

- *difficulty_speed_timer* and *updateDifficulty*(*totalScore*) updates the game's difficulty based on the player's total score so far. It takes the current total score as an input and returns a new difficulty speed timer value, which will affect the speed at which the tetrominoes fall on the game grid. *grid.mergeNew*() creates a new tetromino and adds it to the game grid. *updateScreen*(*grid, totalScore, next, game_speed_timer*) updates the game screen to reflect the current state of the game. It takes the game grid, the player's total score, the next game piece, and the current game speed timer as inputs. Initializing *next* to the variable *current*, sets the current game piece to be the same as the next game piece that was generated.



```java
public static void updateScreen(Grid grid,int score,Shape next, int game_speed_timer) { //Updates and shows every stationary element on the screen
    StdDraw.clear(StdDraw.LIGHT_GRAY); //background color
    grid.showGrid();

    StdDraw.setPenColor(StdDraw.WHITE);
    Font stringFont = new Font( name:"SansSerif", Font.PLAIN, size:18 );
    StdDraw.setFont(stringFont);
    StdDraw.text(9.5, 12, "SCORE:");
    StdDraw.text(9.5, 9, "SPEED:");
    StdDraw.text(9.5, 8.5, game_speed_timer+"");
    StdDraw.text(9.5, 3.2, "NEXT");
    StdDraw.text(9.5, 5, "▸");
    Font smallNumberFont = new Font( name:"SansSerif", Font.PLAIN, size:12 );
    StdDraw.setFont(smallNumberFont);
    Square tmp = new Square(next.center.x,next.center.y);//we make a small copy of the "next" block
    tmp.value=next.center.value;
    Shape temp = new Shape(tmp,next.type+"S");
    for(int i=0;i<temp.squares.length;i++) {
        for(int j=0;j<temp.squares[i].length;j++) {
            temp.squares[i][j].value=next.squares[i][j].value;
            temp.squares[i][j].length = .4;
            temp.squares[i][j].y -= 10;
            temp.squares[i][j].x += 5.5;
        }
    }
    temp.showShape();

    Font numberFont = new Font( name:"SansSerif", Font.PLAIN, size:22 );
    StdDraw.setFont(numberFont);
    StdDraw.text(9.5, 11.4, ""+score);
    //StdDraw.show();

}
```

- This section of code is to update the game screen with the current state of the game. This includes updating the display of the game board, updating the score display, and updating the display of the next shape to appear on the board.



```java
    public static int updateDifficulty(int totalScore) {
        if(totalScore < 100)
            return 100;
        if(totalScore < 250)
            return 200;
        if(totalScore < 300)
            return 250;

        return 0;


    }
}
```

- This method updates the difficulty of the game by changing the speed of the tetrominoes based on the total score. As the total score increases, the difficulty level is increased, which will cause a faster falling speed of the tetrominoes.

2.1.6 Constructing the Tiles

This code defines the class *Tile*, which is used to simulate numbered tiles. The number on the tile, background color, foreground color, and boundary color are all instance variables in the class defined. The thickness of the boundary around the tile and the font used to display the tile number are also defined. The default constructor of the Tile class creates a tile with a randomly determined number of either 2 or 4, then sets the tile's colors. If the tile is merged to another tile it will also change the tile's color from here.

3. Discussion

In this project, one of the problematic parts was how to decide rotation’s operation logic. At first we tried to specify each rotation to each tetromino but we ended up thinking it would be inefficient and not proper to implement so we ended up on choosing the other logic and it worked fine. We tried different ways to rotate for ‘I’ but we couldn’t find anything, it was always giving tile in a different shape at first 3 tries and giving the original tile at the 4th one. We thought of different ways to add extra features and we also added the difficulty system in our game. The original game’s speed is based on 1 move per 1 second, but if the user gets higher scores before winning the game, the game will force the user to speed up their game. Both speeding and slowing are limited because it would be very difficult for the user to play in 1 move per 0.1 second. Restarting the game was also another obstacle because clearing the next and current tetrominoes are not very viable in our code so we just cleared the current game grid and carried the tetromino to the top, reseted every value as if it was the first tetromino. We still couldn’t solve the issue of “mid air” blocks. We tried to eliminate them or bring the blocks to a lower level but none of them worked as intended or didn’t work at all.

4. Conclusion

In this project we combined 2 popular games (*Tetris* and *2048*) with Java using the library *StdDraw*. This project led our group working in different sections of the code and gave us a hint to understand how teamwork and brainstorming is working in a real life project.

References:

1. Bidragsgivare till Wikimedia-projekten. (2022). Tetris. sv.wikipedia.org. https://sv.wikipedia.org/wiki/Tetris

2. How do I win the game “2048”? (n.d.). Quora.

   https://www.quora.com/How-do-I-win-the-game-%E2%80%9C2048%E2%80%9D