School of Computer Science and Engineering

Game Design Document

2048 Game

Instructor: Tran Thanh Tung

Class: Algorithms and Data Structures

Semester 2, academic year 2018 -2019

Group: Heart Hunters in Monster Kingdom

Members:

- Nguyễn Trần Chí Hiếu ITITIU17110
- Lê Anh Minh ITITWE17006
- Trần Quốc Khánh ITITWE17004
- Nguyễn Thị Minh Huệ ITITWE17012

Table of Contents

1.Gameplay

1.1. Goals

- Short-term: Move two tiles with the same numbers next to each other to merge them into one tile with doubled value.
- Long-term: Create a tile with value 2048.

1.2. User Skills

- Knowledge of Basic Math.
- Puzzle Solving
- Strategizing

1.3. Game Mechanics

General

- The gameboard is a 4x4 tiles grid.
- At the start of the game, two tiles will be spawned at two random positions on the gameboard.
- Use arrow buttons to move all the tiles on the game board.
- If there is any change in the game board (movement or merge), a new tile will appear.
- Each tile is spawned at a random unoccupied position.
- Each tile is spawned with value two or four, but mostly two (70% chance)
- Player can use undo (hotkey Z) to go back to the gameboard's previous state, undo can be used multiple times.
- When the gameboard window is opened, player can use hotkey A, B or C to quickly change to Normal Mode, Movable Obstacle Mode or Fixed Obsatcle Mode.

Normal Mode:

- Standard difficulty mode.
- Movement is not limited, player can move however they like to reach the goals.

Movable Obstacle Mode:

- Movement is slightly limited with the occurrence of an obstacle.
- This obstacle is not fixed and will move in the same direction with the tiles.
- After there has been 5 changes to the game board, the obstacle will disappear.
- However, it can be spawned again.
- The chances of obstacle appearing on the game board is 30%.
- By using undo in this mode, the player might receive a penalty, which is an appearance of a second obstacle at a random time.
- There is at most two movable obstacles on the gameboard at a time.

Fixed Obstacle Mode:

- Movement is quite limited with the occurrence of an unmovable obstacle.
- This obstacle is fixed in a random position and will NOT move in any direction.
- This obstacle will stay on the gameboard for the rest of the game.
- There is at most one fixed obstacle on the gameboard.
- Using undo will not cause any penalty, because the game is already hard as it is.

Scoring System:

- For every merge, player will get the amount of points equals to the value of the resulting tile.
  Ex: Two tiles with value 4 will give the player 8 points.
- Using the undo feature will also undo player's points by the same amount they previously received.

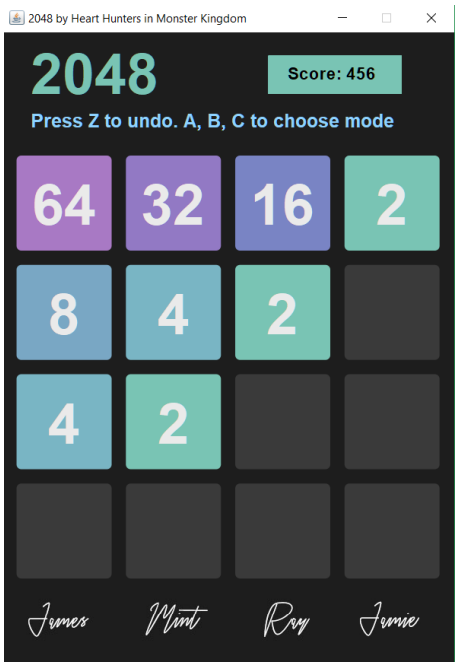


Figure 1: Normal Mode

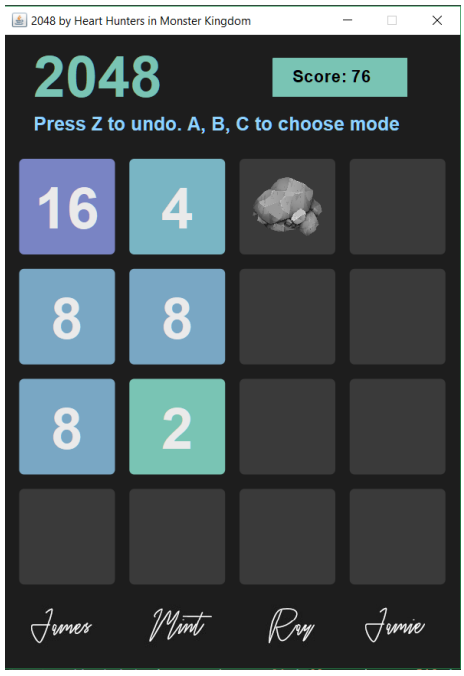


Figure 2: Movable Obstacle Mode

Figure 3: Fixed Obstacle Mode

1.4. Wining

- Player will win when a 2048 tile is produced.

1.5. Losing

- Player will lose when there can be no more available movement or merge.

2. Art Style

- Simple with cold color pallete.

3. Technical Description

- The application is compatible on Windows and Mac OS with Java Environment JDK from 1.6.0 upward.

4. Demographics

- The game is suitable for casual players of all ages.

5. Future Ideas

- Leaderboards System
- Achievements
- Increase the maximum limit score to win (E.g. 4096)
- Increase gameboard size (E.g. 5x5)

6. Classes and Methods

6.1. Tile:

- Contains one field (`value`), two constructors.
- getValue() and setValue(int value)
- `isEmpty()`: return type boolean: return true if value in a tile is zero.
- `getBackground()`: return type Color, set color for tiles based on their value.

6.2. Game2048:

- `startGame(int keyEventCode)`:
  return type `void`, receive an integer number of a keyEvent, if the value is 65, run Normal Mode, if it is 66 or 67, run Movable Obstacle Mode or Fixed Obstacle respectively.
- `ensureSize(List<Tile> l, int s)`:
  return type `void`, to increase the size of the linked list `l` to have the size `s`.
- `getLine(int index)`:
  return an array of type Tile[] specifying all the values in a gameboard line.
- `tileAt(int x, int y)`:
  return a `Tile` at row x, column y.
- `addTile()`:
  return type `void`, this method calls the availableSpace() method to receive a list of blank spaces, it will then take a random Tile in that list and give it a value with 70% chance of creating a value 2, 30% of creating a value 4.
- `availableSpace()`:
  return type `List<Tile>`, this method will go through the `GameTiles` and check which Tile is unoccupied, then add said Tile into a list and then return said list.

- `isFull()`:
  return type `boolean`, this method returns true if there is no more blank space on the gameboard.
- `canMove()`:
  return type `boolean`, this method will check whether there is an available movement of merge after an arrow key is hit. Specifically, when the gameboard is still not empty, the player can still make a move even if there was no merge. Moreover, when the game board is full, but there are a two identical value lying next to each other vertically or horizontally, the method will return true.

8	16	16	2
8	2	4	128
2	2	8	16
2	4	4	2

Take this gameboard for example, the two 16-valued Tile at index (0,1) and (0,2) is still mergable, in terms of code, they are equivalent to `x=0, y=1` and `x=0, y=2`, through method `tileAt()`, returning indices 4 and 8 down-ward from upperleft Tile of the game board (not right-ward).

- `compare(Tile[] line1, Tile[] line2)`:
  return type boolean, this method will return true if original values of a line (`line1`) is identical to the ones of the merged line (`line2`). This method is used to decide if there are any changes in the gameboard, thus helps determine whether to spawn a new Tile.
- `setLine(int index, Tile[] re)`:
  return type `void`, this method set a line in the gameboard with values identical to those in array `re`.
- `paint(Graphics g)`:
  return type `void`, this method is used to draw a rectangle with sizes same as the game window, then, it will also draw the gameboard by using the drawTile method.
- `drawTile(Graphics g2, Tile tile, int x, int y)`:

  This method will use the `Graphics2D` library to draw content on the game window, including tiles, value in each tile, obstacles, 2048 title box, score box, score value, winning and losing screen messages and signature of our team members.
- `offsetCoors(int arg)`:
  return type `int`, this method returns the starting pixel on the gameboard to draw a tile
- `left()`:

  Since the game has 3 modes, two of which uses the same implementation (normal mode and fixed obstacle mode), we included two different cases, one is for playing either normal or fixed obstacle mode, one is for playing movable obstacle.

```
if(!playWithMovableObstacle)           // playWithMovableObstacle is false means user is play in with mod
{
    for (int i = 0; i < 4; i++) {       //move all 4 lines
        Tile[] line = getLine(i);
        Tile[] merged = fixedObstacle.mergeLineFixedObstacle(fixedObstacle.moveLineFixedObstacle(line));
        setLine(i, merged);
        if (!needAddTile && !compare(line, merged)) {
            needAddTile = true;
        }
        if(!needUpdate && !compare(line, merged))
            needUpdate = true;
    }
    if(needAddTile)
        addTile();
    if(!needUpdate){
        scoreStack.pop();
        boardStack.pop();
    }
}
```

The above code segment is executed for normal and fixed obstacle modes. We do the merging with each line of `GameTiles`. The variable line here points to the first 4 Tiles of `GameTiles`, we then move this line using the method moveLineFixedObstacle method and do the merging by using the method `mergeLineFixedObstacle()` in `FixedObstacle` class and store the result in merged. After merging, setLine is called to set the 4 Tiles in `GameTiles` to its merged version.

The boolean variable `needAddTile` is set to true when any merger is done, so that after finishing merging all lines, an arbitrary Tile spawns.

The boolean variable `needUpdate` will be best clarified when describing the procedure of undo feature.

- `undo()`:

  a. <u>The stack `boardStack`:</u>

  This stack keeps track of `GameTiles` when playing the game. In order to do so, boardStack goes through a procedure of being pushed, popped and clear:

```
switch (keyPressed.getKeyCode()) {
    case KeyEvent.VK_LEFT:
        boardStack.push(temp);
        scoreStack.push(myScore);
        left();
        break;
    case KeyEvent.VK_RIGHT:
        boardStack.push(temp);
        scoreStack.push(myScore);
        right();
        break;
    case KeyEvent.VK_DOWN:
        boardStack.push(temp);
        scoreStack.push(myScore);
        down();
        break;
    case KeyEvent.VK_UP:
        boardStack.push(temp);
        scoreStack.push(myScore);
        up();
        break;
}
```

```
for(int i=0; i<16; i++)
    temp[i] = new Tile(GameTiles[i].getValue());
```

When the game begins, as soon as the player hit an arrow key, it will be regconized by the `keyListener`, the stack will then get updated by pushing temp, a deep copy of the `GameTiles`, into the stack.

Only after being pushed, `GameTiles` gets the chance to be moved and merged in method `left()` and then waits for next movements to be made. Therefore, boardStack accumulates as the game moves on.

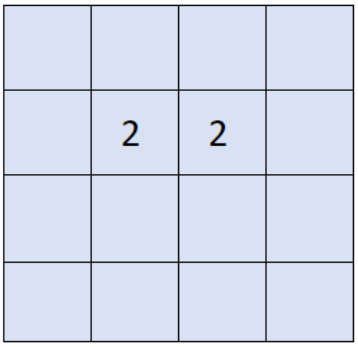
When wanting to go back, the player hits Z key on the keyboard. Once Z is hit, method `undo()`, in which `GameTiles` takes the array popped from `boardStack`, is called.

b. The stack `scoreStack`:

This stack gets pushed, popped and cleared exactly in the same way boardStack does.

The simulation bellow describes how boardStack stores game board and what happens when calling undo method through 3 simple status: game starts → `left()` → undo():left()

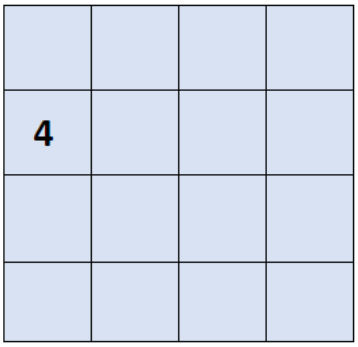
Status 1: Game starts: Score = 0



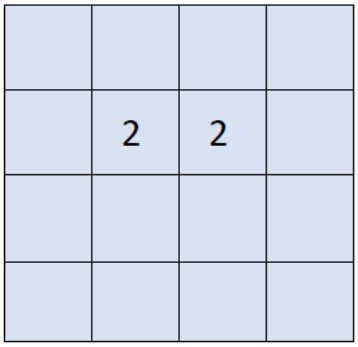
scoreStack: Empty
boardStack: Empty

`left()`

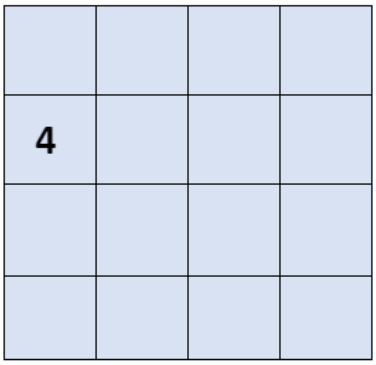
Status 2: After `left()`: Score = 4



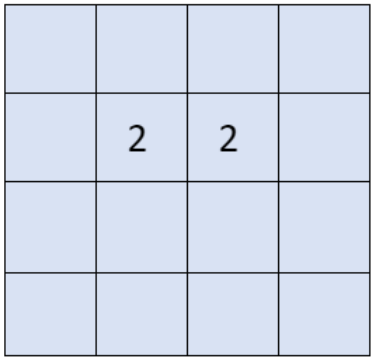
scoreStack: 0
boardStack:



Status 2: After left(): Score = 4

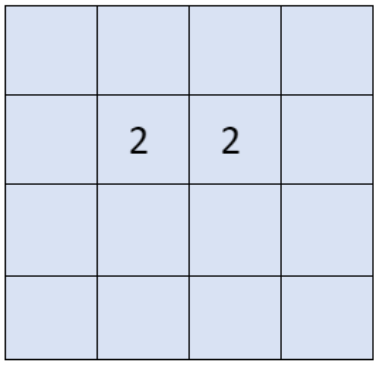


scoreStack: 0
boardStack:



undo()

Status 3: After undo():
`Score = scoreStack.pop()`
`GameTiles = boardStack.pop()`

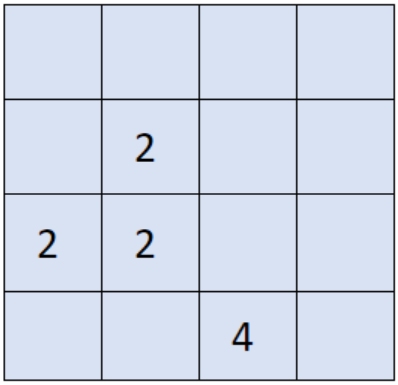


scoreStack: Empty
boardStack: Empty

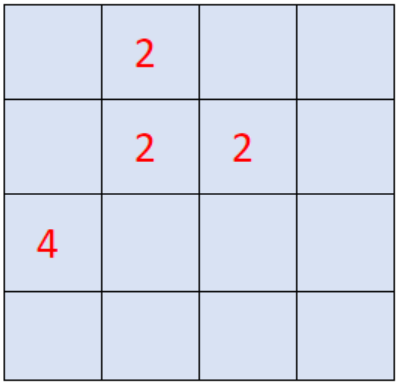
- `rotate()`:

  In order to enable non-left movements, we first apply rotation to GameTiles, then call left() method to do the moving on rotated GameTiles and rotate it back to its initial state. One special procedure to implement rotation is mathematic of rotation, which will be discussed next.

  Here is a simple rotation simulation:



rotate(90)

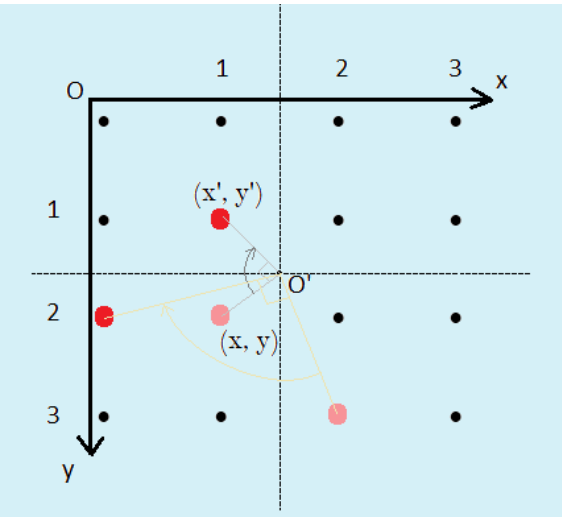


The code segment bellow shows the behind image of rotating:

```
private Tile[] rotate(int angle) {
    Tile[] newTiles = new Tile[16];
    int offsetX = 3, offsetY = 3;
    if (angle == 90) {
        offsetY = 0;
    } else if (angle == 270) {
        offsetX = 0;
    }

    double rad = Math.toRadians(angle);
    int cos = (int) Math.cos(rad);
    int sin = (int) Math.sin(rad);
    for (int x = 0; x < 4; x++) {
        for (int y = 0; y < 4; y++) {
            int newX = (x * cos) - (y * sin) + offsetX;
            int newY = (x * sin) + (y * cos) + offsetY;
            newTiles[(newX) + (newY) * 4] = tileAt(x, y);

        }
    }
    return newTiles;
}
```

a. Mathematic of rotation:

Imagine that *GameTiles* is nothing but a set of dots in an Oxy graph:



The image above shows the illustration how the pink Tiles are rotated by an angle of 90 degree, which generates a new game board with moved Tiles (red).

In order to get new coordinates (x', y') after rotating by an angle of θ around the origin O, Mathematic of rotation is applied:

$$x' = xcos(\theta) - ysin(\theta)$$

$$y' = xsin(\theta) + ycos(\theta) \qquad (1)$$

However, for the case of game board, the whole board is rotated around O'(1.5, 1.5), which is at the center of the board, so we once again apply a different procedure of getting (x', y') as below:

First subtract the original Tile (x, y) with the new origin O', we get a new coordinate: (x-1.5, y-1.5)

Secondly, we apply (1) with x = x-1.5; y=y-1.5:

$$x' = (x - 1.5)cos(\theta) - (y - 1.5)sin(\theta)$$
$$= xcos(\theta) - ysin(\theta) - 1.5(\cos(\theta) - \sin(\theta))$$
$$y' = (x - 1.5)sin(\theta) + (y - 1.5)cos(\theta)$$
$$= xsin(\theta) + ycos(\theta) - 1.5(\cos(\theta) + \sin(\theta))$$

Finally, we add (x', y') with (1.5, 1.5), which has been substracted previously, yielding:

$$x' = xcos(\theta) - ysin(\theta) - 1.5(\cos(\theta) - \sin(\theta) - 1)$$
$$y' = xsin(\theta) + ycos(\theta) - 1.5(\cos(\theta) + \sin(\theta) - 1)$$

And the result above will be used as a new coordinate for one Tile after being rotated.

b. Looking at the code:

Moreover, since right(), up(), down() only deal with the rotation of either 90, 180 or 270 degree, the constant $-1.5(\cos(\theta) - \sin(\theta) - 1)$ will be shorted to offsetX and offsetY, depending on each kind of angle passed to rotate method.

```
for (int x = 0; x < 4; x++) {
    for (int y = 0; y < 4; y++) {
        int newX = (x * cos) - (y * sin) + offsetX;
        int newY = (x * sin) + (y * cos) + offsetY;
        newTiles[(newX) + (newY) * 4] = tileAt(x, y);

    }
}
```

In specific:

- When an angle of 180 degree is passed to the method:

offsetX = $-1.5(\cos(\theta) - \sin(\theta) - 1) = 3$

offsetY = $-1.5(\cos(\theta) + \sin(\theta) - 1) = 3$

- When an angle of 90 degree is passed:

offsetX = $-1.5(\cos(\theta) - \sin(\theta) - 1) = 3$

offsetY = $-1.5(\cos(\theta) + \sin(\theta) - 1) = 0$

- When an angle of 270 degree is passed:

offsetX = $-1.5(\cos(\theta) - \sin(\theta) - 1) = 0$

offsetY = $-1.5(\cos(\theta) + \sin(\theta) - 1) = 3$

Yielding:

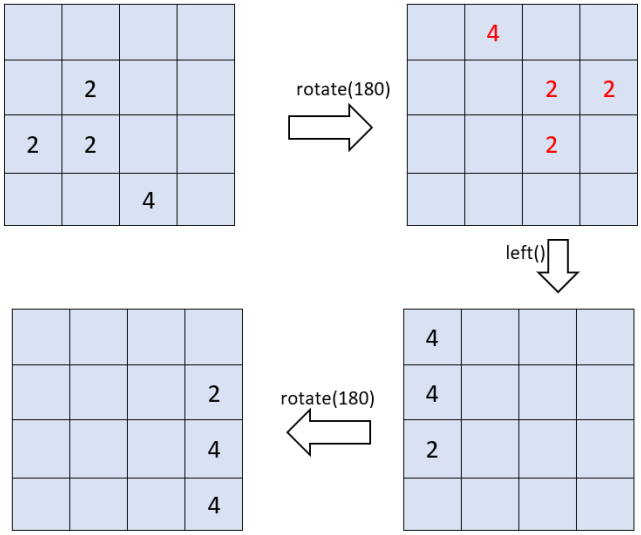
```
int offsetX = 3, offsetY = 3;
if (angle == 90) {
    offsetY = 0;
} else if (angle == 270) {
    offsetX = 0;
}
```

`right(), up(), down()` methods are pretty short, since each only uses rotate() method and left() method for completing the movement.

- `right()`:

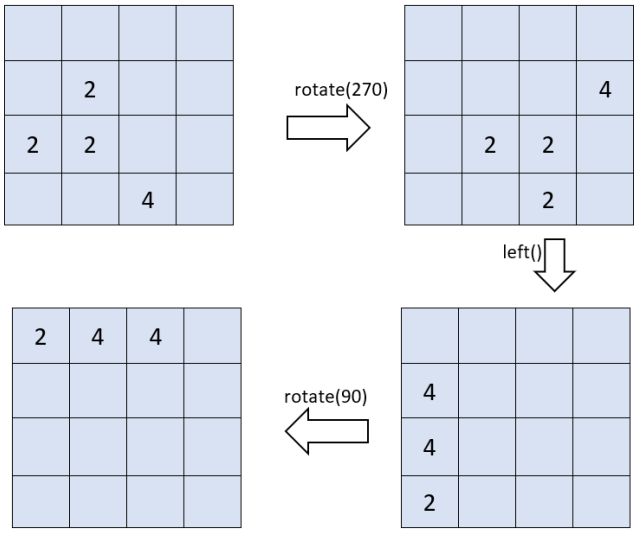
```
private void right() {
    GameTiles = rotate( angle: 180);
    left();
    GameTiles = rotate( angle: 180);
}
```

After rotating the map, `left()` is called to move it to the left, then rotate it once again, giving us a 'virtual' movement to the right.



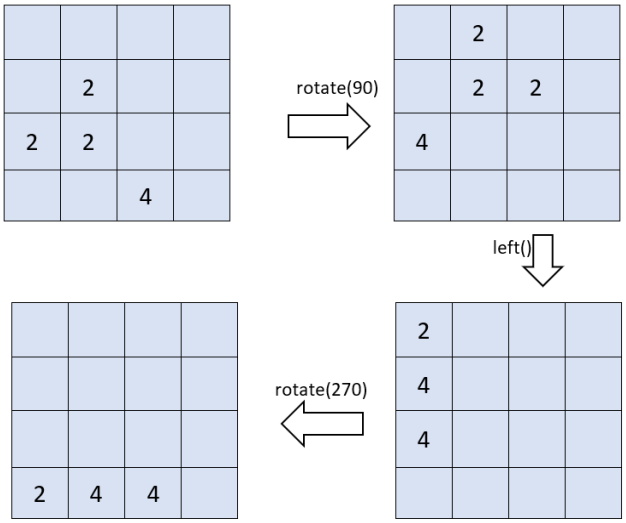
- `up()`:

```
private void up() {
    GameTiles = rotate( angle: 270);
    left();
    GameTiles = rotate( angle: 90);
}
```



- `down()`:

```
private void down() {
    GameTiles = rotate( angle: 90);
    left();
    GameTiles = rotate( angle: 270);
}
```



6.3. MovableObstacle:

- `add()`: return type void, check if a movable obstacle is already on the gameboard, if not, add a new one.
- `killObstacle()`: return type void, slowly kill off the obstacle if there is any change on the gameboard.
- `moveLineMovableObstacle(Tile[])`: return type `Tile[]`, makes all the tiles stack to one direction for the Movable Obstacle Mode.
- `mergeLineMovableObstacle(Tile[])`: return type `Tile[]`, correctly merges lines that has already moved from the moveLineMovableObstacle methodand check whether a new tile should be spawned for the Movable Obstacle Mode.

6.4. FixedObstacle:

This is when the code gets more complicated and trickier. Since there is an obstacle staying at one fixed index, `moveLine()` and `mergeLine()` must be modified accordingly.

- `moveLineFixedObstacle()`:

  We devided this into 4 sub-tasks corresponding to 4 possible indices of a fixed obstacle.

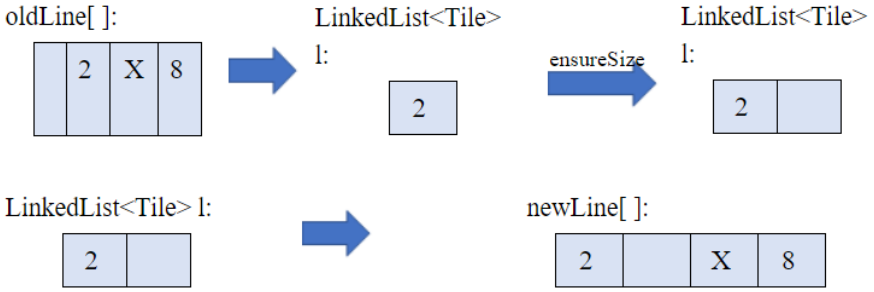
```
if(oldLine[1].getValue()==-6){          // If the fixed obstacle is at index 1 in a line, element at index
    for(int i=2; i<4; i++){             // 0 stays unchanged, but at indices 2, 3 is moved
        if(!oldLine[i].isEmpty())
            l.addLast(oldLine[i]);      // add into the temporary stacked list l values in indices 2, 3 of oldLine
    }
    if (l.size() == 0) {                // If there's no tiles nor obstacle, then just return the empty oldLine
        return oldLine;
    } else {
        Game2048.ensureSize(l, s: 2);           //Ensure that the new stacked line must have 2 elements
        newLine[0] = oldLine[0];
        newLine[1] = oldLine[1];
        for (int i = 2; i < 4; i++) {
            newLine[i] = l.removeFirst();   // newLine gets popped value from l
        }
    }
}
```

a. Obstacle is at index 1 (value at `oldLine[1] = -6`)

In this case, the two Tiles to the right of the obstacle are the only two that needs moving. Therefore, `newLine` is mannually assigned `oldLine[0]` and `oldLine[1]` since they weren't changed, and the moved part being stored and then ensured with a length of 2 in l is assigned to `newLine`.

b. Obstacle is at index 2 (value at `oldLine[2] = -6`)

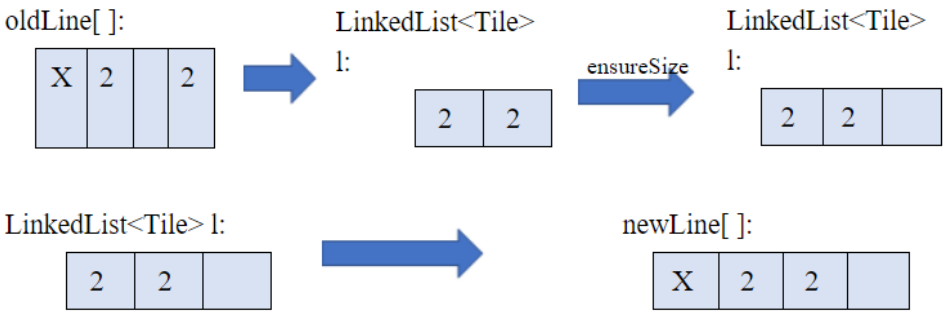
The same thing takes place, but the first half of the line will be moving while the later stays the same.



c. Obstacle is at index 0 (value at `oldLine[0] = -6`):

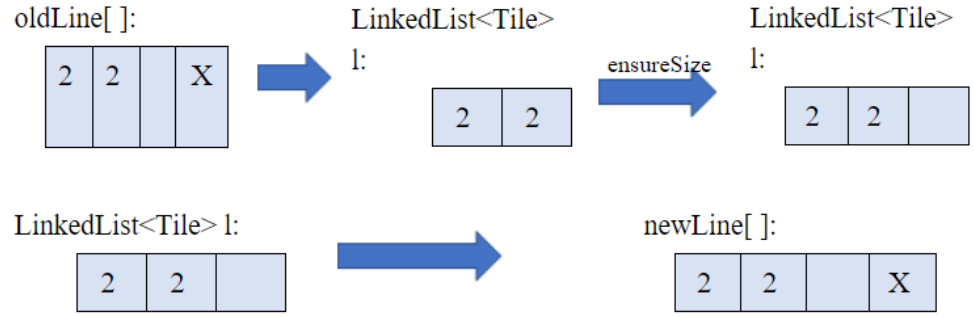
```
if(oldLine[0].getValue()==-6){          // If obstacle is at index 0, move line from index 1 to index 3
    for (int i = 1; i < 4; i++) {
        if (!oldLine[i].isEmpty())
            l.addLast(oldLine[i]);
    }
    if (l.size() == 0) {
        return oldLine;
    } else {
        Game2048.ensureSize(l, s: 3);           //Ensure that the new stacked to the
        newLine[0] = oldLine[0];                //right of the obstacle line must have 3 elements
        for (int i = 1; i < 4; i++) {
            newLine[i] = l.removeFirst();
        }
    }
}
```

When obstacle is at index 0, the 3 later Tiles will be moving. What is different here is that l is ensured to a size of 3, then attached to `newLine`.



d. Obstacle is at index 3 (value at `oldLine[3] = -6`):

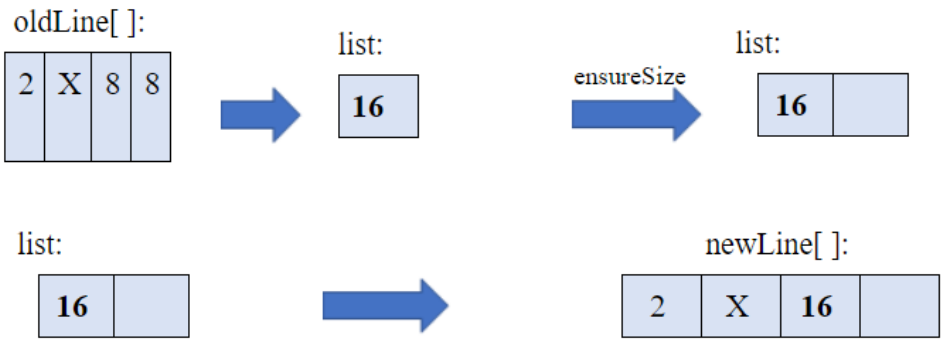
The same thing takes place, with obstacle is fixed at index 3, the first 3 Tiles move.



- `moveLineFixedObstacle()`:
  a. Obstacle is at index 1 (value at `oldLine[1] = -6`)

```
if(oldTile[1].getValue()==-6){      // if the obstacle is in index 1, then we
                                    // just need to merge the two Tiles to the right of the obstacle
    if(oldTile[2].getValue()==oldTile[3].getValue()){   // do the merging for the two last
        int num = oldTile[2].getValue();                // tiles to the right of the obstacle
        num *=2;
        Game2048.myScore += num;
        if (num == 2048) {
            Game2048.isWon = true;
        }
        list.add(new Tile(num));
    }
    if (list.size() == 0) {
        return oldTile;
    } else {
        Game2048.ensureSize(list, s: 2);                // ensure list is having 2 Tiles, which
        newLine[0] = oldTile[0];
        newLine[1] = oldTile[1];
        newLine[2] = list.remove();
        newLine[3] = list.remove();
    }
}
```

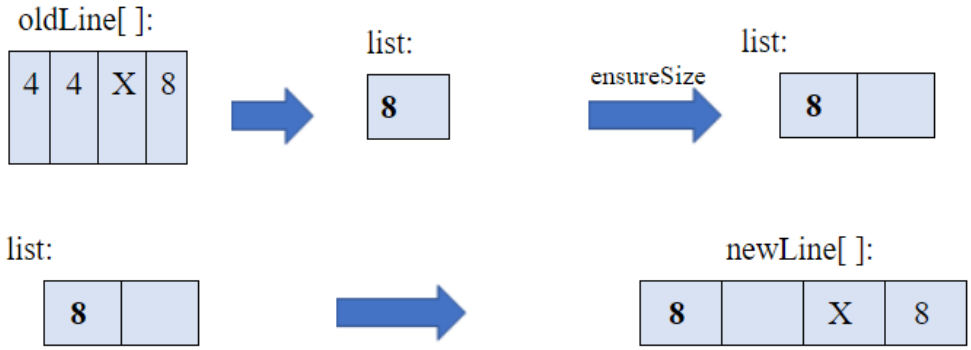
We check whether the two Tiles to the right of the obstacle is the same or not, if they are the same, then merge them and push to list. Next, we ensure that list has two Tiles (since it



will have one if any merge is made) and assign to the last two Tiles of `newLine`. The first two Tiles of `newLine` is plugged in using those of oldLine, since they were not changed

b. Obstacle is at index 2 (value at `oldLine[2] = -6`)

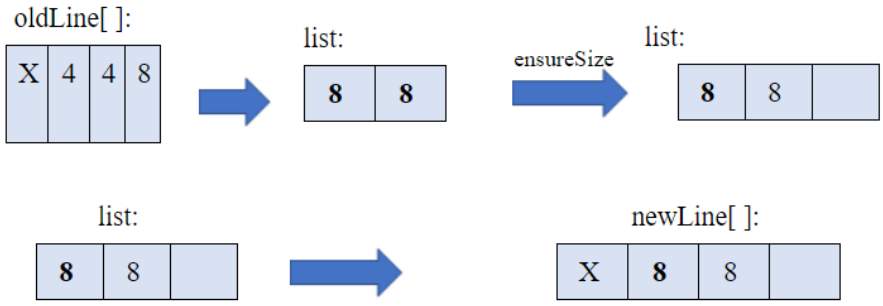
The same thing takes place, but the first half of the line will be merged while the later stays the same.



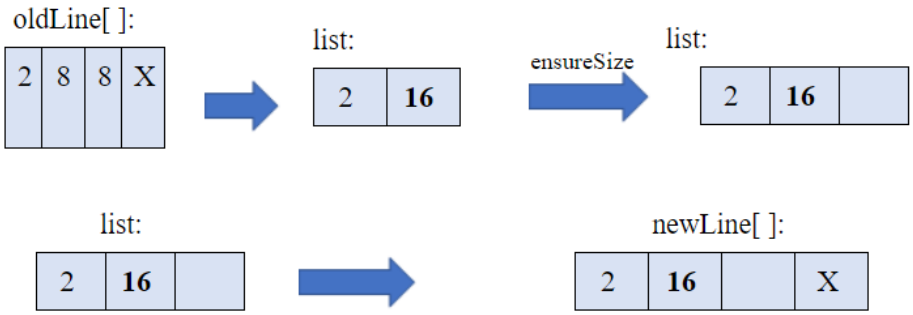
c. Obstacle is at index 0 (value at `oldLine[0] = -6`)

```
if(oldTile[0].getValue()==-6){    //obstacle in index 0
    for(int i=1; i<4 && !oldTile[i].isEmpty();i++){
        int num = oldTile[i].getValue();                    // current value of current Tile
        if ((i < 3) && (oldTile[i].getValue() == oldTile[i + 1].getValue())) {
            num *= 2;                       // num is now doubled
            Game2048.myScore += num;                // update score

            if (num == 2048) {
                Game2048.isWon = true;
            }

            i++;
        }
        list.add(new Tile(num));
    }
    if (list.size() == 0) {
        return oldTile;
    } else {
        Game2048.ensureSize(list, s: 3);
        newLine[0] = oldTile[0];
        for(int i=1;i<4;i++)
            newLine[i]=list.remove();
    }
}
```

The for loop starts considering Tiles from index 1 to 3 to find tiles that can be merged, then merge them if possible.



d. Obstale is at index 3 (value at `oldLine[3] = -6`)



Important observation: the cases where we move/ merge one certain segment of a line and keep the others unchanged only happen when there is a fixed obstacle, or, when there is a Tile that has the value of -6 (add method in `FixedObstacle` class claims this). Therefore, Normal and Movable obstacle mode can be implemented using fixed obstacle procedure, this is kind of a special case where there is no value -6 in the lines. Therefore, move line and merge line implementation will be the same for the two modes:

Move line:

```
for (int i = 0; i < 4; i++) {
    if (!oldLine[i].isEmpty())
        l.addLast(oldLine[i]);
}
if (l.size() == 0) {
    return oldLine;
} else {
    Game2048.ensureSize(l, 4); //Ensure that the new stacked line must have 4 elements
    for (int i = 0; i < 4; i++) {
        newLine[i] = l.removeFirst();
    }
}
```

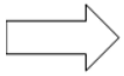
The method takes one line of `GameTiles` as its parameter: oldLine.
The first loop adds Tiles having values into the l LinkedList:

Then, if l contains no Tiles (means the taken line is an empty line with no numbers), then no movement is made. Otherwise, `ensureSize` is called to ensure that l has 4 Tiles:

8	8	8	

A for loop is next included to "assign" l to newLine as an array of Tiles: `Tile[] newLine`. As a result, `newLine` being returned is the moved version of `oldLine`.
On the scope of the entire game board, we have:

2	4		16
2	2		2
8	8		
	32	64	64



2	4	16	
2	2	2	
8	8		
32	64	64	

Merge line: to merge two lefmost consecutive tiles if their values are the same

```
for (int i = 0; i < 4 && !(oldTile[i].isEmpty()); i++) {        // oldLine is NOT empty
    int num = oldTile[i].getValue();                            // current value of current Tile
    if ((i < 3) && (oldTile[i].getValue() == oldTile[i + 1].getValue())) {  //if current Tile and next Tile is equal
        num *= 2;                       // num is now doubled
        Game2048.myScore += num;                // update score

        if (num == 2048) {
            Game2048.isWon = true;
        }

        i++;
    }
    list.add(new Tile(num));
}
if (list.size() == 0) {
    return oldTile;
} else {
    Game2048.ensureSize(list, 4);
    for(int i=0;i<4;i++)
        newLine[i]=list.remove();
}
```

The for loop helps identify Tiles that can be merged and merge them together: num *=2.
Also, the score is updated by adding num in. The line of code: `list.add(new Tile(num));` adds the Tile being considered into the `LinkedList<Tile> list`.
Subsequently, once again the Tile array `newLine` is "assigned" by list, yielding a merged version of `oldLine`.
The whole procedure gives us the bellow transformation:

2	4	16	
2	2	2	
8	8		
32	64	64	

2	4	16	
4	2		
16			
32	**128**		

6.5. ResourceLoader:

Since we need to not only use additional images in our program, but also include said images in our jar file to make it runnable, a normal code like:

```
Image image = ImageIO.read(new File( pathname: "resources\\Cone.png"));
```

is not enough to satisfy the second requirement.

Instead, we mark the *resources* folder as *Sources Root,* and rather than using a URL in the `read()` method, we will use an `InputStream`. The `ResourceLoader` class with only method `load(String path)` and return type `InputStream` will help us achieve both of our goals.

- `load(String path):`

the `getResourceAsStream()` method returns an input stream for reading the specified resource.

```
final public class ResourceLoader {
    public static InputStream load(String path){
        InputStream input = ResourceLoader.class.getResourceAsStream(path);
        System.out.println(input);
        if(input == null){
            input = ResourceLoader.class.getResourceAsStream( name: "/"+path); // "/" means absolute path
            System.out.println(input);
        }
        return input;
    }
}
```

6.6. Menu:

a. Constructor:
   - Initialize a JFrame which contains the Game UI.
   - Add on-click event listeners on individual buttons, including normal mode button, fixed obstacle button and movable obstacle button, each corresponding to a different game mode.

   For example , the code of fixed obstacle button:

```
fixedObstacleButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            if (logic != null) {
                game.remove(logic);
            }
            logic = new Game2048( playWithMovableObstacle: false,  playWithFixedObstacle: true);
            game.add(logic);
            game.setVisible(true);
        } catch (IOException err) {
            System.exit( status: 1);
        }
    }
});
```

- Every time these buttons are clicked, the JFrame will remove any existing game panel and replace it with a new game mode.

```
try {
    if (logic != null) {
        game.remove(logic);
    }
```

b. <u>main</u>

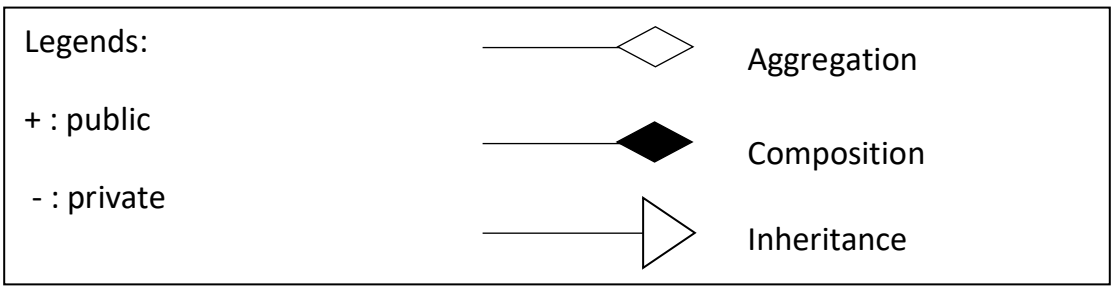
- Initialize a JFrame which contains a menu with 3 buttons

```
frame.setContentPane(new Menu().menuView);
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
frame.setPreferredSize(new Dimension( width: 450,  height: 450));
frame.pack();
frame.setVisible(true);
frame.setResizable(false);
```

```
private JButton movableObstacleButton;
private JButton fixedObstacleButton;
private JButton normalModeButton;
private JPanel menuView;
```

7. Class Diagram

Below the the class diagram of the program:




Menu
+ movableObstacleButton: Jbutton
+ fixedObstacleButton: JButton
+ normalModeButton: JButton
+ menuView: JPanel
+ game: JFrame
+ logic: Game2048
+ Menu()
+ main(Strings): void
Game2048
- BACKGROUND: Color(0x1D1D1D)
- FONT_NAME: String = "Arial"
- TILE_SIZE: int = 100
- TILE_MARGIN: int = 15
+ isObsatcleExist: boolean
- playWithMovabaleObstacle: boolean
- movableObstacle: MovableObstacle
- fixedObstacle: FixedObstacle
+ BOSSHEALTH: int = 5
- boardStack: Stack<Tile[]>
- scoreStack: Stack<Integer>
+ GameTiles: Tile[]
+ isWon: boolean
- isLost: boolean
+ myScore: int
- image, ob5, ob4, ob3, ob2, ob1, Jamie, Hieu, Khanh, Mint : Image
+ startGame(int): void
+ ensureSize(List<Tile>, int): void
- tileAt(int, int): Tile
- addTile(): void
+availableSpace(): List<Tile>
- isFull(): boolean
- canMove(): boolean
- setLine(int, Tile[]): void
+ paint(Graphics): void
- drawTile(Graphics g2, Tile, int, int): void
- offsetCoors(int): int
- left(): void
- right(): void
- up(): void
- down(): void
- rotate(int): Tile[]
- getLine(int): Tile[]
- compare(Tile[], Tile[]): boolean
JPanel
ResourceLoader
+ load(String): InputStream
Tile
- value: int
+ Tile(int)
+ Tile()
+ setValue(int): void
+ getValue(): int
+ getBackground(): Color
MovableObstacle
+ add(): void
+ killObstacle(): void
+ moveLineMovableObstacle(Tile[]): Tile[]
+ mergeLineMovableObstacle(Tile[]): Tile[]
FixedObstacle
+ add(): void
+ killObstacle(): void
+ moveLineFixedObstacle(Tile[]): Tile[]
+ mergeLineFixedObstacle(Tile[]): Tile[]

9. References

- github.com/bulenkov/2048
- "How add images to runnable jar using Eclipse 4.2"
- "Rotation around a point"
- "Pixar in a box" – Khan Academy