/\*\*

\* Class of individual Tile in 2048 game. Value of zero implies an empty tile.

\*

\* @author amritak077

\*/

**Class Tile**

public class Tile {

private int value;

/\*\*

\* Instantiate tile with a value of zero (empty).

\*/

public Tile(){

this(0);

}

/\*\*

\* Instantiate tile with a specific value.

\*

\* @param value

\*/

public Tile(int value){

this.value = value;

}

public void setValue(int value){

this.value = value;

}

public int getValue(){

return value;

}

/\*\*

\* Two tiles are the same if they have the same value. (Useful for merging tile)

\*

\* @param tile

\* @return true if two tiles are equal, false otherwise

\*/

public boolean equals(Tile tile){

return tile.getValue() == this.getValue();

}

/\*\*

\* Add the value of this tile by the value of the tile in the parameter.

\*

\* @param tile

\*/

public void merge(Tile tile){

this.setValue(value + tile.getValue());

}

/\*\*

\* Set the value to zero. In other words, delete / empty the tile.

\*

\*/

public void clear(){

this.setValue(0);

}

public String toString(){

return (Integer.toString(this.getValue()));

}

}

**Class Grid**

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

/\*\*

\* The main game algorithm. The grid contains n x n tiles. Tiles with a value of zero implies an empty tile.

\* The algorithm operate by passing value to other grid (without moving the object itself.

\*

\* @author amritak077

\*

\*/

public class Grid {

//size of the grid

private static final int SIZE = 4;

private Tile[][] tiles = new Tile[SIZE][SIZE]; // for m\*n, provide the row and col no as Tile[row][column]

/\*\*

\* Instantiate n x n grid with all zero values (grid with empty tile).

\*/

public Grid(){

for (int i = 0; i < tiles[0].length; i++){

for (int j = 0; j < tiles.length; j++){

tiles[i][j] = new Tile();

}

}

}

/\*\*

\* Generate a tile with a random value of 2 or 4 in a random position.

\*

\* @return true if successfully placed a new tile, false if there is no empty tile left.

\*

\*/

public boolean generateNewTile(){

if (!(hasEmptyTile())){

return false;

}

Random random = new Random();

//iterate until an empty tile if found

while (true){

int x = random.nextInt(SIZE);

int y = random.nextInt(SIZE);

if (tiles[x][y].getValue() == 0){

tiles[x][y].setValue(getNewTileValue());

return true;

}

}

}

//get tile value of either 2 or 4

private int getNewTileValue(){

Random random = new Random();

int rng = random.nextInt(2) + 1;

return (rng \* 2);

}

public void move(Direction direction){

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

//group of tile

List<Tile> tileSet = new ArrayList<Tile>();

for (int j = 0; j < SIZE; j++){

switch(direction){

case LEFT: tileSet.add(tiles[i][j]); break;

case RIGHT: tileSet.add(tiles[i][SIZE - j - 1]); break;

case UP: tileSet.add(tiles[j][i]); break;

case DOWN: tileSet.add(tiles[SIZE - j - 1][i]); break;

default: break;

}

}

if (!(isEmptyTile(tileSet))){

slide(tileSet); //main tile group algorithm

}

}

}

private boolean isEmptyTile(List<Tile> tileSet) {

for (Tile tile: tileSet){

if (tile.getValue() != 0){

return false;

}

}

return true;

}

//main tile group algorithm

private void slide(List<Tile> tileSet){

slideToEdge(tileSet);

mergeTile(tileSet);

}

//slide all tile into the edge, in case there is a zero in between

private void slideToEdge(List<Tile> tileSet){

for (int i = 0; i < tileSet.size(); i++){

if (remainingIsZero(tileSet, i)){

return;

}

while (tileSet.get(i).getValue() == 0){

slideTo(tileSet, i);

}

}

}

private boolean remainingIsZero(List<Tile> tileSet, int i) {

List<Tile> remainingTile = new ArrayList<Tile>();

for (int j = i; j < tileSet.size(); j++){

remainingTile.add(tileSet.get(j));

}

return (isEmptyTile(remainingTile));

}

private void slideTo(List<Tile> tileSet, int index){

for (int j = index; j < tileSet.size() - 1; j++){

tileSet.get(j).setValue(tileSet.get(j + 1).getValue());

}

tileSet.get(tileSet.size() - 1).clear();

}

//Merge tile, if tile in the direction has the same value.

private void mergeTile(List<Tile> tileSet){

for (int i = 0; i < tileSet.size() - 1; i++){

if (tileSet.get(i).equals(tileSet.get(i + 1))){

tileSet.get(i).merge(tileSet.get(i + 1));

tileSet.get(i + 1).clear();

slideTo(tileSet, i + 1);

}

}

}

/\*\*

\* Check for losing condition. Losing implies no possible move can be made to change the tile.

\*

\* @return true, if no possible move left

\*/

public boolean noPossibleMove(){

if (hasEmptyTile()){

return false;

}

return !(hasEqualNeighbour());

}

private boolean hasEmptyTile(){

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

for (int j = 0; j < SIZE; j++){

if (tiles[i][j].getValue() == 0){

return true;

}

}

}

return false;

}

private boolean hasEqualNeighbour() {

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

for (int j = 0; j < SIZE; j++){

//check the tile in the right of the chosen tile. Ignore last column.

if (j < SIZE - 1){

if (tiles[i][j].equals(tiles[i][j + 1])){

return true;

}

}

//check the tile below the chosen tile. Ignore last row.

if (i < SIZE - 1){

if (tiles[i][j].equals(tiles[i + 1][j])){

return true;

}

}

}

}

return false;

}

public String toString(){

StringBuilder sb = new StringBuilder();

for (Tile[] tileRow: tiles){

for (Tile tile: tileRow){

sb.append(tile);

sb.append(" ");

}

sb.append("\n");

}

return sb.toString();

}

}

public enum Direction {

UP, DOWN, LEFT, RIGHT;

}