|  |  |  |  |
| --- | --- | --- | --- |
| **Coventry University**  **FACULITY OF Engineering and Computing**  **Coursework Cover Sheet**  *Please ensure that you complete all relevant sections legibly*  ***First Copy:*** *Attach top copy to the front of your assignment.*  ***Second Copy:*** *Keep safety as your receipt* | | | |
| **Module Code**  **210CT** | | **Student Card ID Number**  Please print in BLOCK CAPITALS  Surname………………Lin..……………………………  Other names…………Ziqiao…….………..........  Signature………………….……...…..…...  **6826257** | |
| **Module Title**  **Programming, Algorithms and Data Structures** | | | |
| **Deadline date**  **Part 1 – 27th Oct 2015**  **Part 2 – 17th Nov 2015** | **Actual word**  **Count N.A.** | | **Tutor**  **CY Cheng** |
| **Coursework Title/Number**  **Assignment Part Two** | | | |

This assessment is all

my own work and has not

been copied in part or in whole

from any other source, except for any

clearly marked up quotations. It

complies with the university regulations

on plagiarism, which I have read

and understood.

|  |  |  |
| --- | --- | --- |
| **FEEDBACK ON MARKED WORK:** *Lecturers will complete this section when work is marked.*  *NB. All marks notified during the year are provisional until confirmed by the end of year Assessment Board* | | |
| STRENGTHS  WEAKNESSES  ADVICE ON HOW WORK COULD BE IMPROVED AND FURTHER COMMENTS | | |
| If you require more feedback, please contact your tutor or see module web.  See assignment sheet for assessment criteria for this assignment. | **MARKED AWARDED** |  |
| Less any late penalty | **-** |
| Adjusted mark if penalty |  |

**Marker’s Signature** …………………………………………………………… **Date** ……………………………………………………….

|  |
| --- |
| Students are reminded that reference must be given for any previously published  work used to gather information to help write assignments, including internet  sources, but these sources should not be copied directly. |

**Second Marker Additional comments Signature** ………………Nil…………………………… **Date** …………………………

Table of Contents

[Test case 3](#_Toc435457820)

[Test case 1 3](#_Toc435457821)

[Test case 2 4](#_Toc435457822)

[Test case 3 5](#_Toc435457823)

[Test case 4 6](#_Toc435457824)

[Test case 5 7](#_Toc435457825)

[Test case 6 8](#_Toc435457826)

[Test case 7 9](#_Toc435457827)

[Test case 8 12](#_Toc435457828)

[Test case 9 13](#_Toc435457829)

[Program listing 14](#_Toc435457830)

[Game.java 14](#_Toc435457831)

[Start.java 32](#_Toc435457832)

[ArrayQueue.java 33](#_Toc435457833)

# Test case

## Test case 1

|  |  |
| --- | --- |
| Test Case ID | 1 |
| Title | User input wrong commands |
| Test Steps | 1. Enter the command which the not system printed. |
| Expected Results | * Error messages shown - "Invalid Input! Please try again." |
| Actual Results |  |

## Test case 2

|  |  |
| --- | --- |
| Test Case ID | 2 |
| Title | Slide down – basic move |
| Test Steps | 1. Enter the command ‘2’ |
| Expected Results | * Merged tiles if available. * Bring all the tiles to the bottom. * Random fill a tiles if move is valid. |
| Actual Results | Before:    After: |

|  |  |
| --- | --- |
| Test Case ID | 3 |
| Title | Slide left – basic move |
| Test Steps | 1. Enter the command ‘4’. |
| Expected Results | * Merged tiles if available. * Bring all the tiles to the left. * Random fill a tiles if move is valid. |
| Actual Results | Before:    After: |

## Test case 3

|  |  |
| --- | --- |
| Test Case ID | 4 |
| Title | Slide right – basic move |
| Test Steps | 1. Enter the command ‘6’. |
| Expected Results | * Merged tiles if available. * Bring all the tiles to the right. * Random fill a tiles if move is valid. |
| Actual Results | Before:    After: |

## Test case 4

|  |  |
| --- | --- |
| Test Case ID | 5 |
| Title | Slide up – basic move |
| Test Steps | 1. Enter the command ‘8’. |
| Expected Results | * Merged tiles if available. * Bring all the tiles to the top. * Random fill a tiles if move is valid. |
| Actual Results | Before:    After: |

## Test case 5

## Test case 6

|  |  |
| --- | --- |
| Test Case ID | 6 |
| Title | Invalid move |
| Test Steps | 1. Enter the command. |
| Expected Results | * No tiles would be added. |
| Actual Results | Before:    After: |

## Test case 7

|  |  |
| --- | --- |
| Test Case ID | 7 |
| Title | Three or more in a row and Undo |
| Test Steps | 1. Enter the commands. |
| Expected Results | * Merged tiles if available. * Bring all the tiles to the input direction. * Random fill a tiles if move is valid. * Undo to the previous state. |
| Actual Results | Before:    After:    Undo:    Another move:    Another sample  Before:    After: |

|  |  |
| --- | --- |
| Test Case ID | 8 |
| Title | Game over |
| Test Steps | 1. Enter the command. |
| Expected Results | * Message shown – “Game Over! Please enter 'r' to reset or 'q' to leave.” |
| Actual Results |  |

## Test case 8

|  |  |
| --- | --- |
| Test Case ID | 9 |
| Title | Victory |
| Test Steps | 1. Enter the command. |
| Expected Results | * Message shown – “You Win! Please enter 'r' to reset or 'q' to leave..” |
| Actual Results | Before:    After: |

## Test case 9

# Program listing

## Game.java

/\*\*

\* Title: DSA Assignment 15-16

\* Description: ArrayQueue class of 2048 game

\* Student name: Lin Ziqiao

\* Student ID: 6826257

\* Date: 17 Nov 2015

\*/

import java.util.\*;

import java.io.\*;

public class Game {

private static final char null\_char = '\u0000';

private static final int grid\_width = 4;

private static final int undo\_list\_size = 20;

private static final int randomFillChar\_array\_size = 50;

private static final char[] alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();

private static char[] randomFillChar;

private static char[] clone\_randomFillChar;

private static char[][] grid;

private static char[][] clone\_grid;

private int score;

private int clone\_score;

private int counter;

private int clone\_counter;

private ArrayQueue undo\_grid;

private ArrayQueue undo\_score;

private ArrayQueue undo\_randomFillChar;

private ArrayQueue undo\_randomPosition;

private boolean gameover;

private boolean victory;

/\*\*

\* constructor for the Game class

\*/

public Game() {

undo\_grid = new ArrayQueue();

undo\_score = new ArrayQueue();

undo\_randomFillChar = new ArrayQueue();

undo\_randomPosition = new ArrayQueue();

grid = new char[grid\_width][grid\_width];

score = 0;

clone\_score = 0;

counter = 0;

clone\_counter = 0;

gameover = true;

victory = false;

generateCharForRandomFill();

}

/\*\*

\* display the grid and player's score

\*/

public void display() {

if (gridIsEmpty()) {

init();

}

System.out.println("-----------------");

for (int i = 0; i < grid.length; i++) {

System.out.print("| ");

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

if (System.getProperty("os.name").contains("Windows")) {

//for display on Window's cmd

System.out.printf("%s | ", Character.toString(grid[i][j]));

} else if (System.getProperty("os.name").contains("Mac")) {

// for display on Mac OS's terminal

if (grid[i][j] == null\_char) {

System.out.printf(" %s | ", Character.toString(grid[i][j]));

} else {

System.out.printf("%s | ", Character.toString(grid[i][j]));

}

}

}

System.out.println();

System.out.println("-----------------");

}

System.out.println("Score: " + score);

}

/\*\*

\* slide up function for the grid

\*/

public void slideUp() {

// clone the current state of grid, score, randomFillChar and randomFillChar position

cloneState();

// slide up

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

for (int i = 0; i < grid.length; i++) {

if (grid[i][j] != null\_char) { // check if cell is empty or not

if ((i + 1) < grid.length) { // check if array is last cell or not

for (int k = i + 1; k < grid.length; k++) {

if (grid[k][j] != null\_char) { // check if cell is empty or not

if (grid[i][j] == grid[k][j]) { // merge if two tiles are the same

grid[i][j] = mergeTile(grid[k][j]);

grid[k][j] = null\_char;

break; // break for finished one mapping

} else {

break; //break for jump to compare next cell

}

}

}

}

}

}

}

// bring all tiles to the user's input direction after merging tiles

for (int i = 1; i < grid.length; i++) {

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

int k = i;

while (grid[k - 1][j] == null\_char) {

grid[k - 1][j] = grid[k][j];

grid[k][j] = null\_char;

k--;

if (k == 0) {

break;

}

}

}

}

// validating move, random fill a cell and store the state to the undo list

if (moveValidate()) {

randomFill();

addToUndoList();

}

}

/\*\*

\* slide down function for the grid

\*/

public void slideDown() {

// clone the current state of grid, score, randomFillChar and randomFillChar position

cloneState();

// slide down

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

for (int i = grid.length - 1; i >= 0; i--) {

if (grid[i][j] != null\_char) { // check if cell is empty or not

if ((i - 1) >= 0) { // check if array is last cell or not

for (int k = i - 1; k >= 0; k--) {

if (grid[k][j] != null\_char) { // check if cell is empty or not

if (grid[i][j] == grid[k][j]) { // merge if two tiles are the same

grid[i][j] = mergeTile(grid[k][j]);

grid[k][j] = null\_char;

break; // break for finished one mapping

} else {

break; //break for jump to compare next cell

}

}

}

}

}

}

}

// bring all tiles to the user's input direction after merging tiles

for (int i = grid.length - 2; i >= 0; i--) {

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

int k = i;

while (grid[k + 1][j] == null\_char) {

grid[k + 1][j] = grid[k][j];

grid[k][j] = null\_char;

k++;

if (k == 3) {

break;

}

}

}

}

// validating move, random fill a cell and store the state to the undo list

if (moveValidate()) {

randomFill();

addToUndoList();

}

}

/\*\*

\* slide left function for the grid

\*/

public void slideLeft() {

// clone the current state of grid, score, randomFillChar and randomFillChar position

cloneState();

// slide left

for (int i = 0; i < grid.length; i++) {

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

if (grid[i][j] != null\_char) { // check if cell is empty or not

if ((j + 1) < grid.length) { // check if array is last cell or not

for (int k = j + 1; k < grid.length; k++) {

if (grid[i][k] != null\_char) { // check if cell is empty or not

if (grid[i][j] == grid[i][k]) { // merge if two tiles are the same

grid[i][j] = mergeTile(grid[i][k]);

grid[i][k] = null\_char;

break; // break for finished one mapping

} else {

break; //break for jump to compare next cell

}

}

}

}

}

}

}

// bring all tiles to the user's input direction after merging tiles

for (int i = 0; i < grid.length; i++) {

for (int j = 1; j < grid.length; j++) {

int k = j;

while (grid[i][k - 1] == null\_char) {

grid[i][k - 1] = grid[i][k];

grid[i][k] = null\_char;

k--;

if (k == 0) {

break;

}

}

}

}

// validating move, random fill a cell and store the state to the undo list

if (moveValidate()) {

randomFill();

addToUndoList();

}

}

/\*\*

\* slide right function for the grid

\*/

public void slideRight() {

// clone the current state of grid, score, randomFillChar and randomFillChar position

cloneState();

// slide right

for (int i = 0; i < grid.length; i++) {

for (int j = grid.length - 1; j >= 0; j--) {

if (grid[i][j] != null\_char) { // check if cell is empty or not

if ((j - 1) >= 0) { // check if array is last cell or not

for (int k = j - 1; k >= 0; k--) {

if (grid[i][k] != null\_char) { // check if cell is empty or not

if (grid[i][j] == grid[i][k]) { // merge if two tiles are the same

grid[i][j] = mergeTile(grid[i][k]);

grid[i][k] = null\_char;

break; // break for finished one mapping

} else {

break; //break for jump to compare next cell

}

}

}

}

}

}

}

// bring all tiles to the user's input direction after merging tiles

for (int i = 0; i < grid.length; i++) {

for (int j = grid.length - 2; j >= 0; j--) {

int k = j;

while (grid[i][k + 1] == null\_char) {

grid[i][k + 1] = grid[i][k];

grid[i][k] = null\_char;

k++;

if (k == 3) {

break;

}

}

}

}

// validating move, random fill a cell and store the state to the undo list

if (moveValidate()) {

randomFill();

addToUndoList();

}

}

/\*\*

\* random fill tile function for the grid

\*/

public void randomFill() {

boolean filled = true;

int x, y;

char tile;

Random rd1 = new Random();

Random rd2 = new Random();

// re-generate a new random fill array and set the counter(index) to 0 if used all elements in the array

if (counter == 50) {

generateCharForRandomFill();

counter = 0;

}

tile = randomFillChar[counter];

while (filled) {

x = rd1.nextInt(4);

y = rd2.nextInt(4);

if (grid[x][y] == null\_char) {

grid[x][y] = tile;

filled = false;

counter++;

}

}

}

/\*\*

\* game over function for the grid

\*

\* @return boolean gameover

\*/

public boolean gameover() {

gameover = true;

// validate available moves for slide up

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

for (int i = 0; i < grid.length; i++) {

if (grid[i][j] != null\_char) { // check if cell will empty

if ((i + 1) < grid.length) { // check if array will outbound

for (int k = i + 1; k < grid.length; k++) {

if (grid[k][j] != null\_char) { // check if cell will empty

if (grid[i][j] == grid[k][j]) { // check if two tiles are the same

gameover = false;

if (!gameover)

return gameover;

break; // break for finished one mapping

} else {

break; //break for jump to next cell

}

}

}

}

}

}

}

// validate avaliable moves for slide down

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

for (int i = grid.length - 1; i >= 0; i--) {

if (grid[i][j] != null\_char) { // check if cell will empty

if ((i - 1) >= 0) { // check if array will outbound

for (int k = i - 1; k >= 0; k--) {

if (grid[k][j] != null\_char) { // check if cell will empty

if (grid[i][j] == grid[k][j]) { // check if two tiles are the same

gameover = false;

if (!gameover)

return gameover;

break; // break for finished one mapping

} else {

break; //break for jump to next cell

}

}

}

}

}

}

}

// validate avaliable moves for slide left

for (int i = 0; i < grid.length; i++) {

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

if (grid[i][j] != null\_char) { // check if cell will empty

if ((j + 1) < grid.length) { // check if array will outbound

for (int k = j + 1; k < grid.length; k++) {

if (grid[i][k] != null\_char) { // check if cell will empty

if (grid[i][j] == grid[i][k]) { // check if two tiles are the same

gameover = false;

if (!gameover)

return gameover;

break; // break for finished one mapping

} else {

break; //break for jump to next cell

}

}

}

}

}

}

}

// validate avaliable moves for slide right

for (int i = 0; i < grid.length; i++) {

for (int j = grid.length - 1; j >= 0; j--) {

if (grid[i][j] != null\_char) { // check if cell will empty

if ((j - 1) >= 0) { // check if array will outbound

for (int k = j - 1; k >= 0; k--) {

if (grid[i][k] != null\_char) { // check if cell will empty

if (grid[i][j] == grid[i][k]) { // check if two tiles are the same

gameover = false;

if (!gameover)

return gameover;

break; // break for finished one mapping

} else {

break; //break for jump to next cell

}

}

}

}

}

}

}

return gameover;

}

/\*\*

\* undo function for the grid

\*/

public void undo() {

if (undo\_grid.length() > 0 && undo\_score.length() > 0 && undo\_randomFillChar.length() > 0 && undo\_randomPosition.length() > 0) {

char[][] undo = new char[grid\_width][grid\_width]; // initial a new char 2d array

undo = (char[][])undo\_grid.pop(); // assign the previous state to the new char 2d array

// use nested loop to write into the grid array

for (int i = 0; i < undo.length; i++) {

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

grid[i][j] = undo[i][j];

}

}

score = (int)undo\_score.pop(); // replace the score to the previous one

char[] undo1 = new char[randomFillChar\_array\_size]; // initial a new char array

undo1 = (char[])undo\_randomFillChar.pop(); // assign the orevious state to the new char array

// use loop to write into the randomFillChar array

for (int i = 0; i < undo1.length; i++) {

randomFillChar[i] = undo1[i];

}

counter = (int)undo\_randomPosition.pop(); // replace the counter to the previous one

} else {

// message for empty undo lists

System.out.println("No moves can undo!");

}

}

/\*\*

\* reset the game and the variables it needs

\*/

public void reset() {

undo\_grid = new ArrayQueue();

undo\_score = new ArrayQueue();

undo\_randomFillChar = new ArrayQueue();

undo\_randomPosition = new ArrayQueue();

grid = new char[grid\_width][grid\_width];

clone\_grid = new char[grid\_width][grid\_width];

score = 0;

clone\_score = 0;

counter = 0;

clone\_counter = 0;

gameover = true;

victory = false;

generateCharForRandomFill();

}

/\*\*

\* check the grid is empty or not

\*

\* @return boolean

\*/

public boolean gridIsEmpty() {

boolean empty = false;

for (int i = 0; i < grid.length; i++) {

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

if (grid[i][j] != null\_char) {

empty = true;

break;

}

}

}

if (empty) {

return false;

} else {

return true;

}

}

/\*\*

\* check the grid is full of tiles

\*

\* @return boolean full

\*/

public boolean gridIsFull() {

boolean full = false;

for (int i = 0; i < grid.length; i++) {

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

if (grid[i][j] != null\_char) {

full = true;

} else {

full = false;

if (!full)

return full;

}

}

}

return full;

}

/\*\*

\* initial the grid with two random tiles

\*/

public void init() {

randomFill();

randomFill();

// grid[0][0] = 'A';

// grid[0][1] = 'B';

// grid[0][2] = 'C';

// grid[0][3] = 'D';

// grid[1][0] = 'E';

// grid[1][1] = 'F';

// grid[1][2] = 'G';

// grid[1][3] = 'H';

// grid[2][0] = 'I';

// grid[2][1] = 'J';

// grid[2][2] = 'K';

// grid[2][3] = 'L';

// grid[3][0] = 'M';

// grid[3][1] = 'N';

// grid[3][2] = 'Z';

// grid[3][3] = 'Z';

}

/\*\*

\* using number to represent the letters, return the next position of current char

\* used to calculate the score and game over if two 'Z' merged

\*

\* @param char c

\* @return int x

\*/

public int convertLetter(char c) {

int x = 0;

for (int i = 0; i < alphabet.length; i++) {

if (c == 'Z') {

victory = true;

return alphabet.length - 1;

} else {

if (c == alphabet[i]) {

x = i + 1;

break;

}

}

}

return x;

}

/\*\*

\* return next letter of current one for merging tile

\*

\* @param char c

\* @return char

\*/

public char mergeTile(char c) {

score += Math.pow(2, convertLetter(c)); // calculate score

return alphabet[convertLetter(c)];

}

/\*\*

\* clone grid array, score, randomFillChar array and randomFillChar array counter

\*/

public void cloneState() {

clone\_grid = new char[grid\_width][grid\_width]; // everytime initial with new char 2d array to avoid duplicate memory address

// clone grid array

for (int i = 0; i < grid.length; i++) {

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

clone\_grid[i][j] = grid[i][j];

}

}

clone\_score = score; // clone score

clone\_randomFillChar = new char[randomFillChar\_array\_size]; // everytime initial with new char array to avoid duplicate memory address

// clone randomFillChar array

for (int i = 0; i < randomFillChar\_array\_size; i++) {

clone\_randomFillChar[i] = randomFillChar[i];

}

clone\_counter = counter; // clone the array counter

}

/\*\*

\* moves validation

\*

\* @return boolean

\*/

public boolean moveValidate() {

return !(Arrays.deepEquals(grid, clone\_grid));

}

/\*\*

\* add the grid and score to the undo list

\*/

public void addToUndoList() {

if (undo\_grid.length() == undo\_list\_size && undo\_score.length() == undo\_list\_size && undo\_randomFillChar.length() == undo\_list\_size && undo\_randomPosition.length() == undo\_list\_size) {

undo\_grid.dequeue(); // remove the last grid

undo\_grid.enqueue(clone\_grid); // add grid in to undo list

undo\_score.dequeue(); // remove the last score and add the new score

undo\_score.enqueue(clone\_score); // add score in to undo list

undo\_randomFillChar.dequeue(); // remove the last array

undo\_randomFillChar.enqueue(clone\_randomFillChar); // add the array to undo list

undo\_randomPosition.dequeue(); // remove the last random position

undo\_randomPosition.enqueue(clone\_counter); // add the random position to undo list

} else {

undo\_grid.enqueue(clone\_grid);

undo\_score.enqueue(clone\_score);

undo\_randomFillChar.enqueue(clone\_randomFillChar);

undo\_randomPosition.enqueue(clone\_counter);

}

}

/\*\*

\* generate random fill number for the randomFill function

\*/

public void generateCharForRandomFill() {

randomFillChar = new char[randomFillChar\_array\_size];

// fill the '0' for 40 times to the array

for (int i = 0; i < 40; i++) {

randomFillChar[i] = '0';

}

// fill the 'A' for 9 times to the array

for (int i = 40; i < 49; i++) {

randomFillChar[i] = 'A';

}

// fill the 'B' for 1 times to the array

randomFillChar[49] = 'B';

// shuffle the array

shuffle(randomFillChar);

}

/\*\*

\* shuffle the array

\*

\* @param char[] c

\*/

public void shuffle(char[] c) {

int len = c.length;

for (int i = 0; i < c.length; i++) {

// get the random index from past i

int random = i + (int)(Math.random() \* len - i);

// swap the element between present one and the random one

char tmp = randomFillChar[random];

randomFillChar[random] = randomFillChar[i];

randomFillChar[i] = tmp;

}

}

/\*\*

\* start game function and menu with logic

\*/

public void start() {

// read user's input

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

// control the system input life cycle

boolean cont = true;

// print the menu and grid

try {

while(cont) {

System.out.println();

System.out.println(" 2048");

display();

if (!victory) { // menu for not yet victory

System.out.println("(2) Down");

System.out.println("(4) Left");

System.out.println("(6) Right");

System.out.println("(8) Up");

System.out.println("(0) Undo");

}

System.out.println("(R) Reset");

System.out.println("(Q) Quit");

if ((gameover() && gridIsFull() && !victory)) { // 'gamover' message print out here

System.out.println("Game Over! Please enter 'r' to reset or 'q' to leave.");

}

if (victory) { // 'victory' message print out here

System.out.println("You Win! Please enter 'r' to reset or 'q' to leave.");

}

System.out.print("Which Move: ");

String cmd = br.readLine();

if (!victory) {

// menu logic for not yet victory

if (cmd.equals("2")) {

slideDown();

} else if (cmd.equals("4")) {

slideLeft();

} else if (cmd.equals("6")) {

slideRight();

} else if (cmd.equals("8")) {

slideUp();

} else if (cmd.equalsIgnoreCase("q")) {

System.out.println();

System.out.println("Leave Game. Good Bye!!!");

System.out.println();

//end the system input life cycle

cont = false;

} else if (cmd.equals("0")) {

undo();

} else if (cmd.equalsIgnoreCase("r")) {

reset();

} else {

System.out.println();

System.out.println("Invalid Input! Please try again.");

}

} else {

// menu logic for victory

if (cmd.equalsIgnoreCase("r")) {

reset();

} else if (cmd.equalsIgnoreCase("q")) {

System.out.println();

System.out.println("Leave Game. Good Bye!!!");

System.out.println();

//end the system input life cycle

cont = false;

} else {

System.out.println();

System.out.println("Invalid Input! Please try again.");

}

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

## Start.java

/\*\*

\* Title: DSA Assignment 15-16

\* Description: ArrayQueue class of 2048 game

\* Student name: Lin Ziqiao

\* Student ID: 6826257

\* Date: 17 Nov 2015

\*/

public class Start {

public static void main(String[] args) {

Game game = new Game(); // instantiate the Game class

game.start(); // invoke the Gmae

}

}

## ArrayQueue.java

/\*\*

\* Title: DSA Assignment 15-16

\* Description: ArrayQueue class of 2048 game

\* Student name: Lin Ziqiao

\* Student ID: 6826257

\* Date: 17 Nov 2015

\*/

public class ArrayQueue {

private int maxSize;

private Object[] undo\_list;

private int front;

private int rear;

private int count;

/\*\*

\* constructor method for ArrayQueeue

\*

\* @param int size

\*/

public ArrayQueue(int size) {

maxSize = size;

undo\_list = new Object[maxSize];

count = 0;

front = 0;

rear = -1;

}

/\*\*

\* constructor method for ArrayQueeue

\* initial the size

\*/

public ArrayQueue() {

this(20);

}

/\*\*

\* check the array is empty or not

\*

\* @return boolean

\*/

public boolean empty() {

return count <= 0;

}

/\*\*

\* check the array is full or not

\*

\* @return boolean

\*/

public boolean full() {

return count == maxSize;

}

/\*\*

\* get the array length

\*

\* @return int count

\*/

public int length() {

return count;

}

/\*\*

\* add object to the rear of the queue

\*

\* @param object item

\* @throws QueueFullException

\*/

public void enqueue(Object item) throws QueueFullException {

if (count < maxSize) {

count++;

if (rear < maxSize - 1)

rear++;

else

rear = 0;

undo\_list[rear] = item;

} else

throw new QueueFullException();

}

/\*\*

\* remove object from front of the queue

\*

\* @return object temp

\* @throws EmptyQueueException

\*/

public Object dequeue() throws EmptyQueueException {

if (count <= 0)

throw new EmptyQueueException();

count--;

Object temp = undo\_list[front];

if (front < maxSize - 1)

front++;

else

front = 0;

return temp;

}

/\*\*

\* remove and return the last object from the queue

\* @return object temp

\* @throws EmptyQueueException

\*/

public Object pop() throws EmptyQueueException {

if (count <= 0)

throw new EmptyQueueException();

count--;

Object temp = undo\_list[rear];

if (rear < front)

if (rear != 0)

rear--;

else

rear = maxSize - 1;

else

rear--;

return temp;

}

/\*\*

\* override toString method

\*

\* @return String s

\*/

public String toString() {

String s = "front [ ";

int index = front;

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

s = s + undo\_list[index] + " ";

if (index < maxSize - 1)

index++;

else

index = 0;

}

s = s + "] rear";

return s;

}

}

class EmptyQueueException extends RuntimeException {

/\*\*

\* customized EmptyQueueException

\*/

public EmptyQueueException () {

super("Queue is empty");

}

}

class QueueFullException extends RuntimeException {

/\*\*

\* customized QueueFullException

\*/

public QueueFullException () {

super("Queue is full");

}

}