|  |  |
| --- | --- |
| **ASSIGNMENT COVER SHEET** | **ANU College**  Australian National University 95 Daley Rd, Acton ACT 2601 **www.**anucollege**.edu.au**  **+61 2 6125 6688** |
| Submission and assessment is anonymous where appropriate and possible. Please do not write your name on this coversheet.  This coversheet must be attached to the front of your assessment when submitted in hard copy.  All assessment items submitted in hard copy are due as specified in the course outline and assessment specifications. |

|  |  |  |  |
| --- | --- | --- | --- |
| Student ID  For group assignments, list each student’s ID | u6284513, u6512077, u6405512 | | |
| Course Code | ANUC1110 | | |
| Course Name | Introduction to Software Systems | | |
| Assignment number | Assignment2 | | |
| Lab group number | 16 | | |
| Tutor Name | Yujia Zhang |  |  |
| Date Submitted |  | Extension Granted |  |

I declare that this work:

* upholds the principles of academic integrity, as defined in the ANU Policy[: Code of Practice for Student Academic Integrity](https://hkxprd0610.outlook.com/owa/redir.aspx?C=pkUS4AqeVkC0OHXUsRYzk8JcJE65y9AI4r3Mqfll_bLO9DXo_dFgmbuC6N5TOcnRwCb-AmVT460.&URL=https%3a%2f%2fpolicies.anu.edu.au%2fppl%2fdocument%2fANUP_000392);
* is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the course outline and/or Wattle site;
* is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
* gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
* in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

|  |  |
| --- | --- |
| **Initials**  For group assignments, each student must initial. |  |

**ANUC1110 – Introduction to Software Systems**

**Semester 2 – 2018**

**Assignment 2 – Research Report**

Introduction.

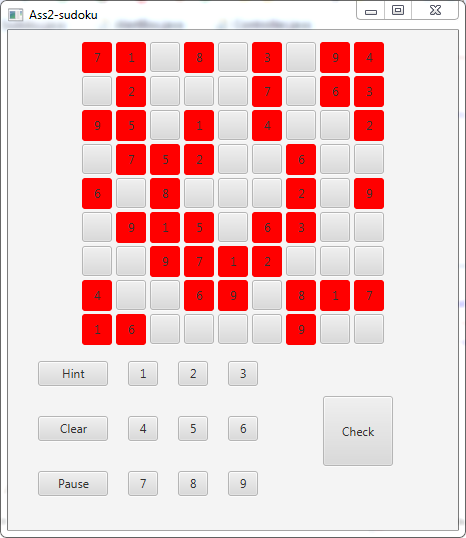
This report aims to help both students and teacher have a general idea about the code in assignment2. In particular, students will find information on how to write code for their first assignment and relevant tools they can use. How deep and broad the review should be depends on requirements stated in the assignment specification.

Running condition for each major function.

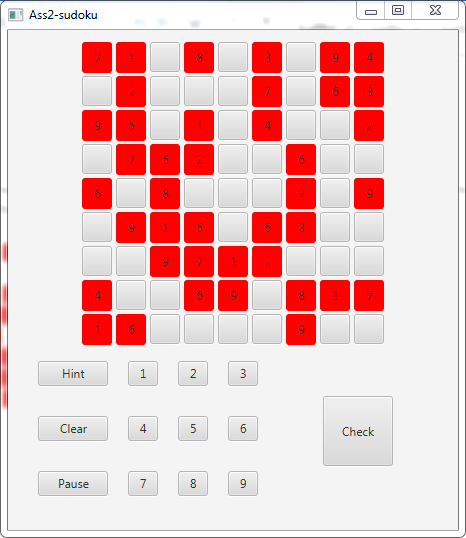
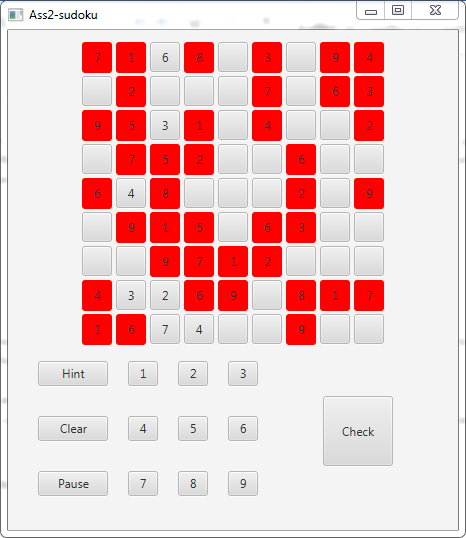
1. Main.java run the sudoku game.



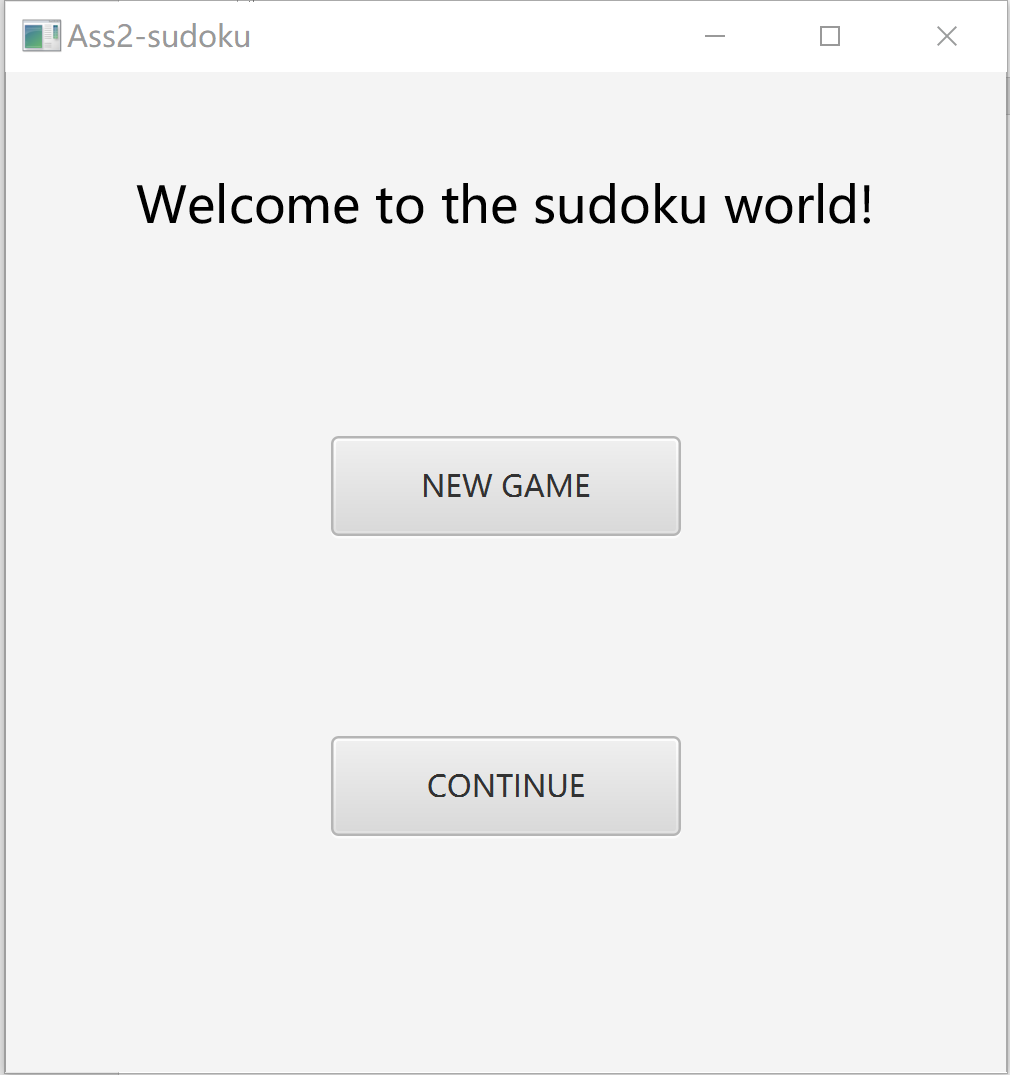
1. Enter the game and we can see an interface (in Fig.1) with “Welcome to the sudoku world!” and two buttons with “NEW GAME” and “CONTINUE” on it. For now, the “CONTINUE” button is dark (It will be discussed latter.)
2. Click “NEW GAME” to enter the game.



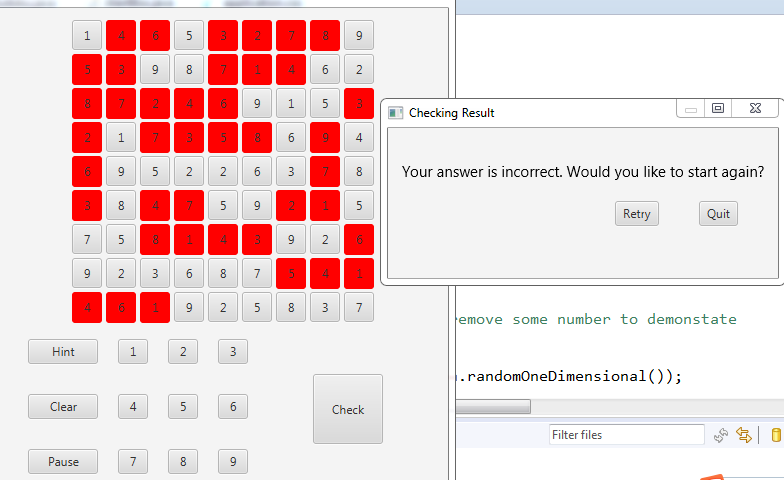
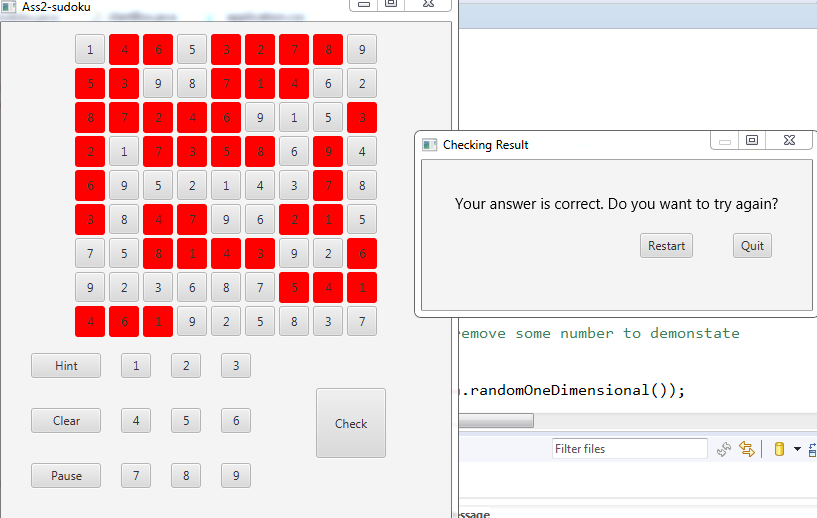
1. At the interface of sudoku game (in Fig.2). There is a 9 \* 9 gameboard on the top with partial solution and the currently selected cell is highlighted. A number button pane consisting nine buttons with nine numbers on them is located at the bottom of the gameboard.
2. To play the game, user should select a cell on the gameboard and press a number button for the answer. Then the number for the pressed button will appear in the selected cell.



1. “Hint” button can be used to get the correct number of the sudoku game
2. “Clear” button can be used to clear all the numbers that has been filled in the gameboard.

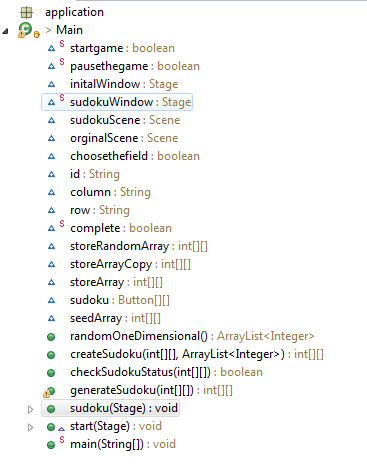


1. When you click the “Pause” button, the game will be stopped, and you will back to the start interface. At the meantime, ”CONTINUE” button which is dark before will turn to light.
2. Current game condition will be saved. After going back to the start page, Click the “CONTINUE” button and you will be back to the game.

1. On finishing the game, user could press “Check” button to check his answer. Then a dialog with checking result should appear like in Fig.6 and Fig.7, for incorrect checking result and correct checking result respectively.
2. For the dialog in Fig.7 (which means user has incorrect answer), start page should appear again just like pausing happens if user presses “Quit” button. And when user presses “Retry” button, your program should return to the current gameboard and game can be continued.
3. If user gets correct answer, dialog like Fig.6 should appear. When user presses “Restart” button, a new gameboard should turn up just like starting a new game. And start page should turn up again if user presses “Quit” on the dialog.

How to solve the problem and explanation to code.



Code:

**public** **class** Main **extends** Application {

**static** **boolean** *startgame* = **false**;

**static** **boolean** *pausethegame* = **false**;

Stage initalWindow = **new** Stage();

**static** Stage *sudokuWindow* = **new** Stage();

Scene sudokuScene;

Scene orginalScene;

**boolean** choosethefield = **false**;

String id;

String column;

String row;

Explanation:

First, extends the Application to get some common features about this game like the color of the button and information about background. Then, define some field that will be used in the next codding step.

Code:

// check if sudoku is complete

**static** **boolean** *complete* = **true**;

**int**[][] storeRandomArray = **new** **int**[9][9];

// use two dimensional storeRandomArray to store the generation of random array

// made by createSudoku

**int**[][] storeArrayCopy = **new** **int** [9][9];

**int**[][] storeArray = **new** **int**[9][9];

//storearray is to store the sudoku game array

Button[][] sudoku = **new** Button[9][9];

Explanation:

Create complete to check if the sudoku game is complete. use two dimensional storeRandomArray to store the generation of random array made by createSudoku.

Code:

**int**[][] seedArray = {{ 5, 3, 4, 6, 7, 8, 9, 1, 2 }, { 6, 7, 2, 1, 9, 5, 3, 4, 8 }, { 1, 9, 8, 3, 4, 2, 5, 6, 7 }, { 8, 5, 9, 7, 6, 1, 4, 2, 3 }, { 4, 2, 6, 8, 5, 3, 7, 9, 1 }, { 7, 1, 3, 9, 2, 4, 8, 5, 6},{9, 6, 1, 5, 3, 7, 2, 8, 4 }, { 2, 8, 7, 4, 1, 9, 6, 3, 5 }, { 3, 4, 5, 2, 8, 6, 1, 7, 9 }};

**public** ArrayList<Integer> randomOneDimensional() {

ArrayList<Integer> oneDimensionalarray = **new** ArrayList<Integer>();

**for** (**int** i = 0; i < 9; i++) {

oneDimensionalarray.add(i + 1);

}

Collections.*shuffle*(oneDimensionalarray);

**return** oneDimensionalarray;

} }

Explanation:

Generate a random Sudoku matrix from a one-dimensional array and the original array. Traverse the data in the two-dimensional array, find the position of the current value in a one-dimensional array, and assign the current position of the one-dimensional array to the current two-dimensional position in the array. The purpose is to cyclically exchange the 9 data in a random order according to the one-dimensional array which is, in this file called seedArray to generate a random Sudoku matrix.

Code:

**public** **int**[][] createSudoku(**int**[][] seedArray, ArrayList<Integer> randomList) {

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

**for** (**int** k = 0; k < 9; k++) {

**if** (seedArray[i][j] == randomList.get(k)) {

seedArray[i][j] = randomList.get((k + 1) % 9);

**break**;

}

}

}

}

**return** seedArray;

}

Explanation:

We create sudoku list by using the seedarray and randomlist generated above.

Code:

**public** **boolean** checkSudokuStatus(**int**[][] sudoku) {

**boolean** flag = **true**;

**for**(**int** i=0;i<9;i++) {

**for**(**int** j=0;j<9;j++) {

**if**(sudoku[i][j] !=Math.*abs*(storeArray[i][j])) {

flag = **false**;

}

}

}

**return** flag;

}

Explanation:

checksudokustatus by checking users' list is equal to the generated list or not

Code:

**public** **int**[][] generateSudoku(**int**[][] originalSudoku) {

ArrayList<String> removenumberarray = **new** ArrayList<String>();

ArrayList<Integer> columnArray = **new** ArrayList<Integer>();

ArrayList<Integer> rowArray = **new** ArrayList<Integer>();

**int**[][] changedSudoku = **new** **int** [9][9];

**for**(**int** i=0;i<9;i++) {

**for**(**int** j=0;j<9;j++) {

changedSudoku [i][j] = originalSudoku[i][j];

}

}

**int** column = 0;

**int** row = 0;

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

// randomly generate the row and column which is wanted to be removed

Random generate = **new** Random();

**if** (removenumberarray != **null**) {

// check the removearray if it has the same element in the removenumberarray

**while** (removenumberarray.contains(row + "," + column)) {

column = generate.nextInt(9);

row = generate.nextInt(9);

}//randomly generate the column and row elements divided by comma

removenumberarray.add(row + "," + column);

} **else** {

column = generate.nextInt(9);

row = generate.nextInt(9);

removenumberarray.add(row + "," + column);

}

}//now we use 'split' to split the row and column array

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

rowArray.add(Integer.*parseInt*(removenumberarray.get(i).split(",")[0]));

columnArray.add(Integer.*parseInt*(removenumberarray.get(i).split(",")[1]));

}

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

**for** (**int** k = 0; k < 40; k++) {

**if** (rowArray.get(k) == i && columnArray.get(k) == j) {

// if the element is more than 0, change elements to the opposite number

changedSudoku[i][j] = -originalSudoku[i][j];

}

}

}

}

**return** changedSudoku;

}

Explanation:

The information we get from the internet that the least number of the sudoku which contains only one solution is 17,so we randomly remove 40 number of sudoku and generate the sudoku by removing some numbers and after the number has been generated then use the boolean flag which is defined in the stage s to check the sudoku has a unique solution or not.

Code:

**public** **void** sudoku(Stage s) **throws** Exception {

**try** {

VBox sudoku\_vb = **new** VBox();

sudokuScene = **new** Scene(sudoku\_vb, 450, 500); // the next scene

HBox setOfSudoku = **new** HBox();

HBox[] hbSudoku = **new** HBox[9]; // 9 row of the sudoku

**for** (**int** i = 0; i < 9; i++) {

hbSudoku[i] = **new** HBox();

}

// set a 9-3\*3 sudoku regions

Button[][] sudoku = **new** Button[9][9];// set size of rectangle

**int**[][] storeArrayCopy = **new** **int**[9][9];

// at first we generate a complete sudoku then remove some number to demonstate to the users

Main ma = **new** Main();

storeRandomArray = ma.createSudoku(seedArray, ma.randomOneDimensional());

// activate the randomlymovethenumber function

// generate storearray which is directly show the sudoku to users

storeArrayCopy = generateSudoku(storeRandomArray);

/\* if the sudoku solution doesn't fit the answer we get from the

'solve'(in the Sudoku java file) then it will frequently random the number

because it definitely doesn't have a unique solution. However, if the count

of the sudoku solution is more than 1 then again random the number

\*/

**boolean** flag = Sudoku.*isEqualsToSudokuSolution*(storeArrayCopy) &&

Sudoku.*countOfSudoku*(storeArrayCopy)==1;

**while**(!flag) {

storeArrayCopy = generateSudoku(storeRandomArray);

flag = Sudoku.*isEqualsToSudokuSolution*(storeArrayCopy) &&

Sudoku.*countOfSudoku*(storeArrayCopy)==1;

}

storeArray = storeArrayCopy.clone();

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

sudoku[i][j] = **new** Button();// set the size of every Button 30\*30

sudoku[i][j].setPrefSize(30, 30);

HBox.*setMargin*(sudoku[i][j], **new** Insets(2)); // set margin 2

// set id which is "row" + "," + "column"

sudoku[i][j].setId(Integer.*toString*(i) + "," + Integer.*toString*(j));

**if** ((storeArray[i][j]) < 0) {

sudoku[i][j].setText("");

} **else** {

sudoku[i][j].setText(Integer.*toString*(storeArray[i][j]));

sudoku[i][j].getStyleClass().add("given");

sudoku[i][j].setDisable(**true**);

}

// set css style hover and focused

sudoku[i][j].getStyleClass().add("sudokubutton");

hbSudoku[i].getChildren().add(sudoku[i][j]);

}

}

Explanation:

First, create stage of sudoku. we generate a complete sudoku then remove some number to demonstrate to the users. Then, set a 9-3\*3 sudoku regions ,set size of button. Activate the randomlymovethenumber function, generate storearray which is directly show the sudoku to users center the hbox of sudoku. What is more, in this part, it also allows to check whether the generated sudoku is legal or not. if the sudoku solution doesn't fit the answer we get from the 'solve'(in the Sudoku java file) then it will frequently random the number because it doesn’t have a unique solution definitely. However, if the count

of the sudoku solution is more than 1, then again random the number.

Code:

**for** (**int** i = 0; i < 9; i++) {

hbSudoku[i].setAlignment(Pos.***CENTER***);

}

HBox[] hbtnset = **new** HBox[3];

**for** (**int** i = 0; i < 3; i++) {

hbtnset[i] = **new** HBox();

hbtnset[i].setAlignment(Pos.***CENTER***);

hbtnset[i].setPadding(**new** Insets(5));

}

HBox hbtncheck = **new** HBox();

VBox vbtnleft = **new** VBox();

Button hint = **new** Button("Hint");

hint.setPrefSize(70, 15);

// set the size of hint clear pause button are all 50\*15

Button clear = **new** Button("Clear");

clear.setPrefSize(70, 15);

Button pause = **new** Button("Pause");

pause.setPrefSize(70, 15);

Button[] number = **new** Button[9];

HBox.*setMargin*(hint, **new** Insets(10));

HBox.*setMargin*(clear, **new** Insets(10));

HBox.*setMargin*(pause, **new** Insets(10));

Explanation:

set the size of hint clear pause button are all 50\*15.

Code:

**for** (**int** i = 0; i < 9; i++) {

number[i] = **new** Button(Integer.*toString*(i + 1));

number[i].setPrefSize(30, 25);

HBox.*setMargin*(number[i], **new** Insets(10));

}

Button btncheck = **new** Button("Check");

btncheck.setPrefSize(70, 70);

hbtncheck.getChildren().add(btncheck);

HBox.*setMargin*(btncheck, **new** Insets(50));

setOfSudoku.setAlignment(Pos.***CENTER***);

Code: hbtnset[0].getChildren().addAll(hint, number[0], number[1], number[2]);

hbtnset[1].getChildren().addAll(clear, number[3], number[4], number[5]);

hbtnset[2].getChildren().addAll(pause, number[6], number[7], number[8]);

vbtnleft.getChildren().addAll(hbtnset[0], hbtnset[1], hbtnset[2]);

setOfSudoku.getChildren().addAll(vbtnleft, hbtncheck);

Explanation:

Use the method to set children.

Code:

sudoku\_vb.getChildren().addAll(hbSudoku[0], hbSudoku[1], hbSudoku[2], hbSudoku[3], hbSudoku[4], hbSudoku[5],hbSudoku[6], hbSudoku[7], hbSudoku[8], setOfSudoku);

Explanation:

add every row of Button and the setofsudoku.

Code:

sudoku\_vb.setPadding(**new** Insets(10));

Explanation:

set padding insets of 10.

Code:

**final** EventHandler<ActionEvent> getHint = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(**final** ActionEvent event) {

**if** (choosethefield) {

String value = Integer

.*toString*(Math.*abs*(storeRandomArray[Integer.*parseInt*(row)][Integer.*parseInt*(column)]));

sudoku[Integer.*parseInt*(row)][Integer.*parseInt*(column)].setText(value);

}

}

};

Explanation:

if one of the 9\*9 button has been chosen, then locate the button and get the button number corresponding to the original sudoku.

Code:

//here you get the coordinate of the button location

**final** EventHandler<ActionEvent> chooseItem = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(**final** ActionEvent event) {

choosethefield = **true**;

Button btn = (Button) event.getSource();

id = btn.getId();

row = id.split(",")[0];

// split the row and column by the ","

column = id.split(",")[1];

}

};

**final** EventHandler<ActionEvent> addnumber = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(ActionEvent event) {

// get the value from the button and and the number to place which is chosen

**if** (choosethefield) {

String value = ((Button) event.getSource()).getText().toString();

sudoku[Integer.*parseInt*(row)][Integer.*parseInt*(column)].setText(value);

} **else** {

**return**;

}

}

};

Explanation:

get the value from the button and the number to place which is chosen. For at the design of button, we have set every button for specified id as “row” + “,” + “column”, and now we just need to split them by using “,”.

Code:

**final** EventHandler<ActionEvent> checkSudoku = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(ActionEvent event) {

**int**[][] sudokuCheck = **new** **int**[9][9];

/\* initialize the complete situation, if the situation is changed, then complete becomes false \*/

*complete* = **true**;

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

**if** (sudoku[i][j].getText().equals("")) {

*complete* = **false**;

}

}

}

**if** (*complete*) {

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

sudokuCheck[i][j] = Math.*abs*(Integer.*parseInt*(sudoku[i][j].getText()));

}

}

}

/\* print out the sudoku status

if the button has not filled completely then give the retrybox otherwise check the status\*/

**if** (*complete*) {

**if** (checkSudokuStatus(sudokuCheck)) {

AlertBox.*display*("Checking Result", "Your answer is correct. Do you want to try again?");

} **else** {

RetryBox.*display*("Checking Result",

"Your answer is incorrect. Would you like to start again?");

}

} **else** {

RetryBox.*display*("Checking Result", "Your answer is incorrect. Would you like to start again?");

}

}

};

Explanation:

initialize the complete situation, if the situation is changed, then complete becomes false. If the sudoku is not completed, then return false.

Code:

// if users click the clear button then initialize the situation

**final** EventHandler<ActionEvent> clearnumber = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(ActionEvent event) {

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

//if the number is less then 0 then set the button ""

**if** ((storeArray[i][j]) < 0) {

sudoku[i][j].setText("");

} **else** {

sudoku[i][j].setText(Integer.*toString*(storeArray[i][j]));

}

}

}

}

};

Code:

/\*when you clicked the pause button then you can't click the new game button

after go back to the menu\*/

**final** EventHandler<ActionEvent> pauseTheGame = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(ActionEvent event) {

*startgame* = **true**;

*pausethegame* = **true**;

*sudokuWindow*.close();

**try** {

ma.start(initalWindow);

} **catch** (Exception e) {

e.printStackTrace();

}

initalWindow.show();

}

};

Explanation:

when you clicked the pause button then you can't click the new game button after going back to the menu.

Code:

//add eventhandle of keyevent so that the users can type in the number from key board

**final** EventHandler<KeyEvent> typeTheNum = **new** EventHandler<KeyEvent>() {

**public** **void** handle(KeyEvent event) {

event.getCode();

**if**(event.getCode().isDigitKey() && //exclude 0 number

event.getCode()!=KeyCode.***DIGIT0***) {

**if**(choosethefield) {

String str = event.getCode().getName();

sudoku[Integer.*parseInt*(row)][Integer.*parseInt*(column)].setText(str);

}

}

}

};

Explanation:

It allows users to input numbers from the keyboard. And that time we ignore inputing the keyboard from number 0.

Code:

**for** (**int** i = 0; i < 9; i++) {

number[i].setOnAction(addnumber);

}

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

sudoku[i][j].setOnAction(chooseItem);

sudoku[i][j].setOnKeyPressed(typeTheNum);

}

}

clear.setOnAction(clearnumber);

hint.setOnAction(getHint);

btncheck.setOnAction(checkSudoku);

pause.setOnAction(pauseTheGame);

// set application.css

sudokuScene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

s.setScene(sudokuScene);

s.setTitle("Ass2-sudoku");

s.show();

*sudokuWindow* = s;

} **catch** (Exception e) {

e.printStackTrace();

}

};

Explanation:

1. set actions on the 1-9 buttons

2. set actions on the 9\*9 button at the above

3. set actions on clear, hint, btncheck, pause

Code:

@Override

**public** **void** start(Stage primaryStage) **throws** Exception {

**try** {

Main main = **new** Main();

VBox vb = **new** VBox();

// vb is the start stage

HBox hbtext = **new** HBox();

Text welcome = **new** Text("Welcome to the sudoku world!");

welcome.setFont(**new** Font(20));

hbtext.getChildren().add(welcome);

hbtext.setAlignment(Pos.***CENTER***);

// let the hbtext hbox align in the centre

HBox.*setMargin*(welcome, **new** Insets(40));

// set margin of hbox

HBox hbbtnew = **new** HBox();

HBox hbbtncont = **new** HBox();

Button new\_game = **new** Button("NEW GAME");

Button cont = **new** Button("CONTINUE");

new\_game.setPrefSize(140, 40);

// set size of the button

hbbtnew.setAlignment(Pos.***CENTER***);

HBox.*setMargin*(new\_game, **new** Insets(40));

cont.setPrefSize(140, 40);

// set size of the button

hbbtncont.setAlignment(Pos.***CENTER***);

HBox.*setMargin*(cont, **new** Insets(40));

hbbtnew.getChildren().add(new\_game);

hbbtncont.getChildren().add(cont);

vb.getChildren().addAll(hbtext, hbbtnew, hbbtncont);

orginalScene = **new** Scene(vb, 400, 400);

// the first scene

primaryStage.setScene(orginalScene);

primaryStage.setTitle("Ass2-sudoku");

primaryStage.show();

initalWindow = primaryStage;

**final** EventHandler<ActionEvent> newgame = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(**final** ActionEvent event) {

**try** {

main.sudoku(*sudokuWindow*);

} **catch** (Exception e) {

e.printStackTrace();

}

primaryStage.close();

*sudokuWindow*.show();

}

};

**final** EventHandler<ActionEvent> resume = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(**final** ActionEvent event) {

**if** (*startgame*) {

*sudokuWindow*.show();

}

primaryStage.close();

*sudokuWindow*.show();

}

};

Explanation:

Use javaFX operation to set the interface like the size of button and world that you want to show.

Code:

cont.setDisable(!*pausethegame*);

// disable the button

cont.setOnAction(resume);

// set an action so that to open a sudoku stage

new\_game.setOnAction(newgame);

} **catch** (Exception e) {

e.printStackTrace();

}

}

**public** **static** **void** main(String[] args) {

*launch*(args);

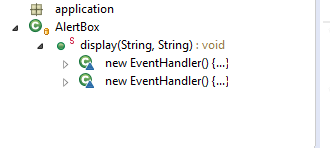
}

}

Explanation:

Setdisable function is used to disable the button. Boolean value pausethegame is controlled by pause button. If the game is paused, then the value becomes true.

1. Alertbox.java & RetryBox.java pop up when check the sudoku game



Code:

**public** **static** **void** display(String title, String message) {

Main main = **new** Main();

Stage window = **new** Stage();

window.initModality(Modality.***APPLICATION\_MODAL***);

window.setTitle(title);

VBox vb = **new** VBox();

Text txt = **new** Text(message);

txt.setFont(**new** Font(15));

HBox hbtxt = **new** HBox();

hbtxt.getChildren().add(txt);

HBox.*setMargin*(hbtxt, **new** Insets(20,20,20,20));

hbtxt.setAlignment(Pos.***CENTER***);

HBox hbox = **new** HBox();

Button restart = **new** Button("Restart");

Button quit = **new** Button("Quit");

hbox.setAlignment(Pos.***BOTTOM\_RIGHT***);

HBox.*setMargin*(quit,**new** Insets(20,20,20,20));

HBox.*setMargin*(restart,**new** Insets(20,20,20,20));

hbox.getChildren().addAll(restart,quit);

vb.getChildren().addAll(hbtxt,hbox);

window.setScene(**new** Scene(vb, 390, 150));

vb.setAlignment(Pos.***CENTER***);

vb.setPadding(**new** Insets(20));

window.show();

Explanation:

Defines a modal window that blocks events from being delivered to any other application window.

Code:

**final** EventHandler<ActionEvent> backToMenu = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(**final** ActionEvent event) {

window.close();

Main.*pausethegame* = **false**;

Main.*sudokuWindow*.close();

**try** {

main.start(main.initalWindow);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

};

**final** EventHandler<ActionEvent> newTheGame = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(**final** ActionEvent event) {

window.close();

Main.*sudokuWindow*.close();

//back to menu

**try** {

main.sudoku(Main.*sudokuWindow*);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

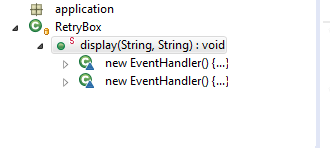
}

}

};

restart.setOnAction(newTheGame);

quit.setOnAction(backToMenu);



Code:

**public** **static** **void** display(String title, String message) {

Main main = **new** Main();

Stage window = **new** Stage();

window.initModality(Modality.***APPLICATION\_MODAL***);

//Defines a modal window that blocks events from being delivered to any other application window.

window.setTitle(title);

VBox vb = **new** VBox();

Text txt = **new** Text(message);

txt.setFont(**new** Font(15));

HBox hbtxt = **new** HBox();

hbtxt.getChildren().add(txt);

HBox.*setMargin*(hbtxt, **new** Insets(20,20,20,20));

hbtxt.setAlignment(Pos.***CENTER***);

HBox hbox = **new** HBox();

Button retry = **new** Button("Retry");

Button quit = **new** Button("Quit");

hbox.setAlignment(Pos.***BOTTOM\_RIGHT***);

HBox.*setMargin*(quit,**new** Insets(20,20,20,20));

HBox.*setMargin*(retry,**new** Insets(20,20,20,20));

hbox.getChildren().addAll(retry,quit);

vb.getChildren().addAll(hbtxt,hbox);

window.setScene(**new** Scene(vb, 390, 150));

vb.setAlignment(Pos.***CENTER***);

vb.setPadding(**new** Insets(20));

window.show();

**final** EventHandler<ActionEvent> backToMenu = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(**final** ActionEvent event) {

window.close();

Main.*pausethegame* = **true**;

Main.*sudokuWindow*.close();

//back to menu

**try** {

main.start(main.initalWindow);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

};

**final** EventHandler<ActionEvent> restartTheGame = **new** EventHandler<ActionEvent>() {

@Override

**public** **void** handle(**final** ActionEvent event) {

window.close();

window.setScene(main.sudokuScene);

}

};

retry.setOnAction(restartTheGame);

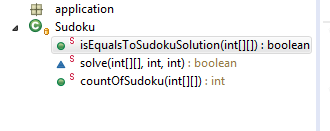
quit.setOnAction(backToMenu);

}

Explanation:

This box has a lot of similarity to the alertbox.

1. Sudoku.java check whether the sudoku has a unique solution or not



Code:

**public** **class** Sudoku {

**public** **static** **boolean** isEqualsToSudokuSolution(**int**[][] storeArray) {

**boolean** flag = **true**;

**int**[][] sk = **new** **int**[9][9];

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

**if** (storeArray[i][j] < 0) {

sk[i][j] = 0;

} **else** {

sk[i][j] = storeArray[i][j];

}

}

}

**if** (*solve*(sk, 0, 0)) {

**for** (**int** x = 0; x < 9; x++) {

**for** (**int** y = 0; y < 9; y++) {

**if** (Math.*abs*(storeArray[x][y])!= sk[x][y]) {

flag = **false**;

}

}

}

}

**return** flag;

}

Explanation:

This isEqualsToSudokuSolution method is to check whether the solution got from sudoku is same or not equal to the generated sudoku answer. If it is false, then rapidly random the sudoku game.

Code:

**static** **boolean** solve(**int**[][] sk, **int** row, **int** col) {

**if** (col > 8) {

col = 0;

row++;

}

**if** (row > 8) {

**return** **true**;

}

/\*If the current box has a solution,

then the operation of next box is performed.\*/

**if** (sk[row][col] > 0) {

**return** *solve*(sk, row, ++col);

}

F: **for** (**int** i = 1; i <= 9; i++) { //start test numbers

// If the current row and column have duplicate numbers

**for** (**int** temp = 0; temp < 9; temp++) {

**if** (sk[row][temp] == i || sk[temp][col] == i) {

**continue** F; // went directly to the next loop

}

}

// If the current 3\*3 box has duplicate numbers, then the next loop

**int** xMax = row / 3 \* 3 + 3;

**int** yMax = col / 3 \* 3 + 3;

**for** (**int** x = row / 3 \* 3; x < xMax; x++)

**for** (**int** y = col / 3 \* 3; y < yMax; y++)

**if** (sk[x][y] == i) {

**continue** F;

}

// After all the measurements, fill the i value into the array.

sk[row][col] = i;

**if** (*solve*(sk, row, col + 1)) {

**return** **true**;

} **else** {

sk[row][col] = 0;

}

/\*If you can go to this step, it means that the next number has failed to fill,

so you should reset the current array and continue the loop measurement.\*/

}

**return** **false**;

}

Explanation:

We create a *solve*method to implement this algorithm. In our implementation, we will stop the algorithm after one solution is found. It is good because a correct Sudoku grid has a unique solution.

We iterate on the grid, and then, we try to assign a value on an empty cell. If the grid is correct after this assignment, we call recursively the *solve*method and we return true. Otherwise, we return false and then, the algorithm can try another assignment for the current cell.

Code:

**public** **static** **int** countOfSudoku(**int**[][] sudoku) {

**int** count = 0;

**int** row = 0;

**int** col = 0;

//initialize count, row, col to be 0

**int**[][] sk = **new** **int**[9][9];

**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

sk[i][j] = Math.*abs*(sudoku[i][j]) ;

//Assign the absolute value of sudoku array to the sk array

}

}

//count is the solution of the countOfSudoku

F:**for** (**int** i = 0; i < 9; i++) {

**for** (**int** j = 0; j < 9; j++) {

**if**(sk[i][j]==0) {

row = i;

col = j;

}

**break** F;

}

}

//get the 0 value of the button row = i,col = j

**if** (row < 9 && col < 9) {

//recursion of solve of sudoku

**if** (*solve*(sk, row, col)) {

count++;

}

row++;

**if**(row==8) {

col++;

}

}

**return** count;

}

Explanation:

count how many solutions we get from the sudoku. If the solution is more than one then it means that sudoku game should be generated another time. First, it will check the isEqualsToSudokuSolution then if it is false then rapidedly random the array until it becomes true. And then start to check whether it has one solution or more.

1. Application.css decorate the sudoku game

Code:

/\* JavaFX CSS - Leave this comment until you have at least create one rule which uses -fx-Property \*/

.sudokubutton:hover{

-fx-border-color:orange;

-fx-border-width:2px;

}

.sudokubutton:focused{

-fx-border-color:orange;

-fx-border-width:2px;

}

.given{

-fx-background-color: red;

}

.given:disabled{

-fx-opacity:1;

}

Explanation:

1. Focus selector is used to select the element that has focus.
2. Hover selector is used to select the element that has hovered.
3. change the opacity of the button with given sudoku number to 1.

Contribution of Assignment2:

u6284513: 50% Doing report and sudoku stage

u6405512: 25% Doing the start stage of java and the report of the start stage and write the function of checking whether the sudoku has a unique solution or not.

u6512077: 25% Doing the report and the structure of report and screen shot of the result of every situation.

For u6512077, he also has planned when we start doing the report and the code.

External Resource:

Cite from:

1. https://medium.com/@rossharrison/generating-sudoku-boards-pt-1-structures-algorithms-a1e62feeb32
2. <http://www.geometer.org/mathcircles/sudoku.pdf>

From these two resources, I learnt that the least sudoku that has unique solution just has 17 numbers. We decide to generate 50 numbers in the sudoku. What is more, I learnt how to generate a random sudoku by seed array and the backtracking algorithm.

Assignment submission

The report will be submitted through the ANU Gitlab. The deadline for this

Assignment is Wednesday, 31 October 2018, 7:00 PM