**Project 9 – Concentration**

**(20 points)**

**Names:** Ozaner Hansha

**Due Date: Thursday, February 25, 2016**

**Description:**

In this partner project, you will implement a GUI for the famous matching game, Concentration. The project will use Model-View-Controller (MVC) pattern. It will incorporate interfaces and abstract classes. It will give you practice using one-dimensional and two-dimensional arrays. The rules for Concentration are given in the appendix.

We will program our Concentration game using our playing card classes, but we will write our code in a general way so that it would be easy to substitute any other collection of graphical objects that provided a core set of methods such as getWidth, turnFaceUp, equals, etc. The way to accomplish this in Java is to use an ***interface***. We will define an interface called Cell, which will be the reference type for our generically written code.

The game model will be programmed in the abstract class ConcentrationModel. The class ConcentrationCardModel will extend the abstract class and provide implementations for the abstract methods that must deal with the particulars of using our now-familiar playing card classes for the objects.

The Concentration class will contain the main method and will implement the GUI. The principal actions to be conveyed to the model are:

* startGame(int rows, int cols, int players)
* choose(Cell cell)

The notifications that the model will need to make to the GUI include:

* gameStartedNotification(Cell[][] board)
* selectCellNotification(Cell cell)
* removeCellNotification(Cell cell1, Cell cell2, int player, int score
* deselectCellsNotification(Cell cell1, Cell cell2, int nextPlayer)
* gameOverNotification(int[] winners)

You may notice that one of the abstract methods in ConcentrationModel has an interesting signature:

public ListIterator<Cell> getCellIterator()

This permits the generic code in ConcentrationModel to use inheritance to get an appropriate type of Cell on demand. (So, for example, it can fill up a concentration array!) The use of iterators is tested on the AP exam, so this will be a good example of using iterators.

In this project, I am giving you clean versions of the supporting classes that we developed earlier for writing card games. I am also providing the Cell interface, as mentioned above. You may alter it if necessary. You will also need to slightly alter the GCard class so that it implements the Cell interface.

1. In Eclipse, import the file APCS2015Proj09Concentration.zip.
2. Study the supplied classes carefully so that you understand how they work. The code should be familiar from our earlier project. Re-use as much of the code as possible.
3. Complete the project by finishing the classes Concentration, ConcentrationModel and ConcentrationCardModel
4. Start your coding with a fixed number of rows and columns and only one player. Progressively, add more features to:

* permit from 1 to 4 players (show the running scores for each player)
* permit a variable number of row and columns
* check for errors in the inputs for the players, rows and columns

1. Submit this Word document to your shared Google Drive folder with

* your name filled out above
* the source listings for the classes Concentration, ConcentrationModel, ConcentrationCardModel and GCard.
* key screen captures of your Concentration GUI interface

**Extensions:**

* allow resizing of the window with rescaling of the objects on the canvas
* implement a timer, so that the game must be completed in a given time
* implement a control (e.g., a slider) that varies the time that the card face is shown before turning it back over✓

play some audio when key events occur in the game (winning, losing, a match, etc.)✓

**Appendix**

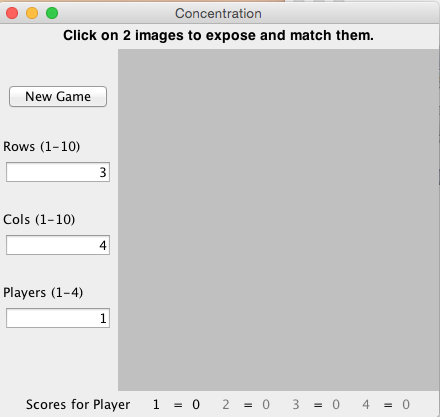
**Here are the rules for Concentration**:

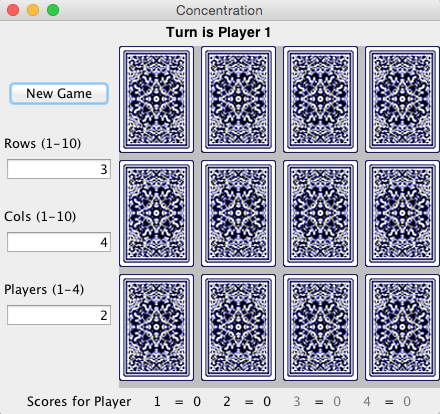
There are from one to four players. There may be any even number of cells, each of which can be shown face up or face down. The back of the cells must be identical. The cells are arranged in a 2D, rectangular fashion, initially face down. Since the number of cells is even, at least one of the row or column count must be even. Each unique face must occur an even number of times so that pairing may occur. It is typical for only one pair of each face to appear in the game, but that is not necessary. You may put some practical limits on the number of rows or columns that your GUI will permit.

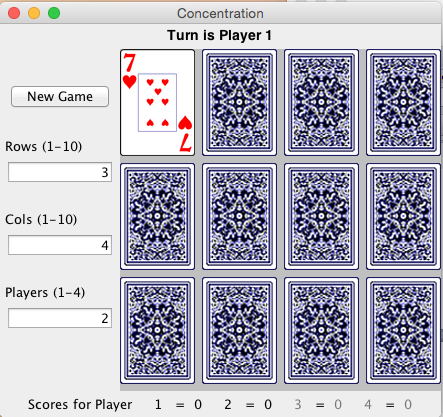
The order of play:

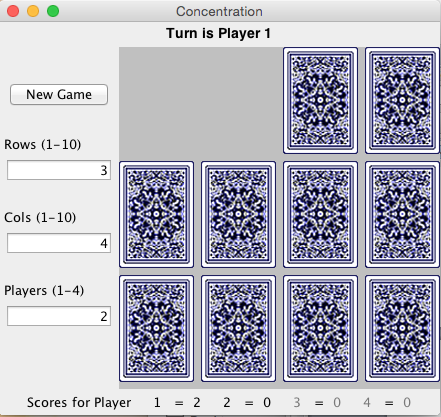
* Each player plays in turn.
* A player turns two cards (cells) face up. If the cards match, the player receives 2 points, the cards are removed, and the player continues play. If the cards do not match, then the cards are turned face down again and the next player resumes play.
* When all of the cards have been removed, play finishes. Any player having the highest number of points wins the game. There may be multiple winners.

**GUI Examples**









**Your Source Code**

**ConcentrationModel**

**package** unit7.concentration;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.ListIterator;

/\*\*

\* The generic model of a game of Concentration with,<br>

\* a table of {@link Cell}s, multiple {@link #players},<br>

\* {@link #scores}.<br>

\* **@author** Ozaner Hansha

\*/

**public** **abstract** **class** ConcentrationModel {

/\*\*

\* A reference to this {@link ConcentrationModel}'s

\* Corresponding View for callbacks.

\*/

**private** Concentration app;

/\*\*

\* 1 of 3 basic settings for a game.

\*/

**private** **int** rows, cols, players;

/\*\*

\* Keeps track of which player's turn it is.

\*/

**private** **int** currentPlayer;

/\*\*

\* A 2D array of cells representing

\* the concentration board.

\*/

**private** Cell[][] board;

/\*\*

\* An array of the scores of the players.

\*/

**private** **int**[] scores;

/\*\*

\* How many cards have been removed (matched) so far.

\*/

**private** **int** matchedCards;

/\*\*

\* A buffer for the 2 currently chosen cells.

\*/

**private** Cell[] currentCells = **new** Cell[2];

/\*\*

\* Constructs a new model with a view reference.

\* **@param** app - View for callbacks

\*/

**public** ConcentrationModel(Concentration app) {

**this**.app = app;

}

/\*\*

\* **@return** The width in pixels of any given cell.

\*/

**public** **abstract** **double** getCellWidth();

/\*\*

\* **@return** The height in pixels of any given cell.

\*/

**public** **abstract** **double** getCellHeight();

/\*\*

\* **@return** A listIterator of the Cell objects to be placed on the board.

\*/

**public** **abstract** ListIterator<Cell> getCellIterator();

/\*\*

\* Starts a new round by resetting all old values,<br>

\* creating a new board and sending it to the {@link #app}.

\* **@param** rows - Rows of board.

\* **@param** cols - Columns of board.

\* **@param** players - How many players.

\*/

**public** **void** startGame(**int** rows, **int** cols, **int** players) {

**this**.rows = rows;

**this**.cols = cols;

**this**.players = players;

currentPlayer = 1;

matchedCards = 0;

scores = **new** **int**[players+1]; //Init Scores

//Creates and fills board

board = **new** Cell[rows][cols];

fillBoard();

app.gameStartedNotification(board, players); //Starts game

}

/\*\*

\* Changes {@link #currentPlayer} to the next player.

\* **@return** The currentPlayer

\*/

**public** **int** nextPlayer() {

**if**(currentPlayer < players)

currentPlayer++;

**else**

currentPlayer = 1;

**return** currentPlayer;

}

/\*\*

\* Fills the 2D board with Cells. At least one of rows and cols must be even.

\*/

**private** **void** fillBoard() {

**int** numCellsNeeded = rows \* cols;

ArrayList<Cell> cells = **new** ArrayList<Cell>(numCellsNeeded);

ListIterator<Cell> cellIterator = **null**;

// create a list of the cells to use for our board

// we use the abstract method getCellIterator() to generate the cells

**while** (cells.size() < numCellsNeeded) {

cellIterator = getCellIterator(); // may get called again if we run out of cells

// get the next cell, as long as we still need one and it is available

**for** (Cell cell = cellIterator.next();

cells.size() < numCellsNeeded && cellIterator.hasNext();

cell = cellIterator.next()) {

// add the cell and a copy of it to the list of cells to use for our board

cells.add(cell);

cells.add(cell.copy());

}

}

Collections.*shuffle*(cells); // randomize the list

**for**(**int** i = 0; i < numCellsNeeded; i++)

board[i/cols][i%cols] = cells.get(i);

}

/\*\*

\* Handles the action of a user choosing a cell.

\* **@param** cell the chosen cell

\*/

**public** **void** choose(Cell cell) {

**if**(currentCells[0] == **null**) {

currentCells[0] = cell;

app.selectCellNotification(cell);

}

**else** **if**(currentCells[1] == **null** && cell != currentCells[0]){

currentCells[1] = cell;

app.selectCellNotification(cell);

**if**(cell.equals(currentCells[0]))

match();

**else** {

app.deselectCellsNotification(currentCells[0], currentCells[1], nextPlayer());

currentCells = **new** Cell[2];

}

}

}

/\*\*

\* Increases the {@link #matchedCards} and {@link #scores}

\* of a player by 2,<br> informs {@link #app} and calls

\* {@link #gameOver()} if all matches found.

\*/

**public** **void** match() {

scores[currentPlayer]+=2;

matchedCards+=2;

app.removeCellsNotification(currentCells[0],currentCells[1],currentPlayer,scores[currentPlayer]);

currentCells = **new** Cell[2]; //Resets cell buffer

**if**(matchedCards >= rows\*cols)

gameOver();

}

/\*\*

\* Called when all matches are done.<br>

\* Finds the list of all winners and sends it back to the {@link #app}.

\* **@see** #match()

\*/

**public** **void** gameOver() {

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

**int** max = scores[0];

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

**if**(scores[i] > max) {

max = scores[i];

winners.clear();

winners.add(i);

}

**else** **if**(scores[i] == max) winners.add(i);

}

**int**[] arr = **new int**[winners.size()];

**for**(**int** i = 0; i < winners.size(); i++)

arr[i] = (**int**)winners.toArray()[i];

app.gameOverNotification(arr);

}

}

**ConcentrationCardModel**

**package** unit7.concentration;

**import** java.util.ArrayList;

**import** java.util.ListIterator;

/\*\*

\* The Concentration Model based off of {@link GCard}.

\* **@author** Ozaner Hansha

\*/

**public** **class** ConcentrationCardModel **extends** ConcentrationModel {

/\*\*

\* Creates a new Concentration Model based off of cards.

\* **@param** app - The Concentration application

\*/

**public** ConcentrationCardModel(Concentration app) {

**super**(app);

}

/\*\*

\* Returns the width of a cell (Card).

\* **@see** unit7.concentration.ConcentrationModel#getCellWidth()

\*/

@Override

**public** **double** getCellWidth() {

**return** GCard.*cardWidth*();

}

/\*\*

\* Returns the height of a cell (Card).

\* **@see** unit7.concentration.ConcentrationModel#getCellHeight()

\*/

@Override

**public** **double** getCellHeight() {

**return** GCard.*cardHeight*();

}

/\*\*

\* Returns a listIterator of a GCard deck made from {@link GCard#makeDeck()}.

\* **@see** unit7.concentration.ConcentrationModel#getCellIterator()

\*/

@Override

**public** ListIterator<Cell> getCellIterator() {

Deck deck = GCard.*makeDeck*();

deck.shuffle();

ArrayList<Cell> cells = **new** ArrayList<Cell>();

**for**(Card c: deck) {

cells.add((GCard)c);

}

**return** cells.listIterator();

}

}

**Concentration**

**package** unit7.concentration;

**import** java.awt.Color;

**import** java.awt.Font;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.event.MouseEvent;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** javax.sound.sampled.AudioInputStream;

**import** javax.sound.sampled.AudioSystem;

**import** javax.sound.sampled.Clip;

**import** javax.sound.sampled.LineUnavailableException;

**import** javax.sound.sampled.UnsupportedAudioFileException;

**import** javax.swing.ButtonGroup;

**import** javax.swing.JButton;

**import** javax.swing.JLabel;

**import** javax.swing.JRadioButton;

**import** javax.swing.JSlider;

**import** javax.swing.JTextField;

**import** javax.swing.Timer;

**import** javax.swing.event.ChangeEvent;

**import** javax.swing.event.ChangeListener;

**import** acm.graphics.GPoint;

**import** acm.program.GraphicsProgram;

/\*\*

\* An app for the Concentration game. This version uses playing cards via the

\* ConcentrationCardModel class. The game may use images other than cards by

\* extending the ConcentrationModel abstract class with a different implementation.

\*

\* The GCard class only required a couple of methods to implement the Cell interface.

\* Thus, for the most part, the classes used in earlier playing cards apps were able

\* to be reused without modification.

\*

\* **@author** Ozaner Hansha

\* Dr. Jones<br>

\* AP Computer Science<br>

\* February 25th, 2016<br>

\*/

@SuppressWarnings("serial")

**public** **class** Concentration **extends** GraphicsProgram **implements** ChangeListener {

/\*\*

\* The maximum amount of players.

\*/

**public** **static** **final** **int** ***MAX\_PLAYERS*** = 4;

/\*\*

\* The maximum size of the rows/columns.

\*/

**public** **static** **final** **int** ***MAX\_SIZE*** = 10;

/\*\*

\* The minimum amount of players.

\*/

**public** **static** **final** **int** ***MIN\_PLAYERS*** = 1;

/\*\*

\* The minimum size of the rows/columns.

\*/

**public** **static** **final** **int** ***MIN\_SIZE*** = 1;

/\*\*

\* How many spaces (" ") between each {@link #scores}.

\*/

**public** **static** **final** **int** ***SCORE\_SPACING*** = 8;

/\*\*

\* Space (in pixels) between each card, both x and y.

\*/

**public** **static** **final** **int** ***CARD\_SPACING*** = 5;

/\*\*

\* Labels above {@link #options} boxes.

\*/

**public** **static** **final** JLabel[] ***OPTION\_LABELS*** =

{**new** JLabel("Columns (1-10)"), **new** JLabel("Rows (1-10)"),

**new** JLabel("Players (1-4)"), **new** JLabel("Memory Time (0-5 Sec)")};

/\*\*

\* The font of the labels.

\*/

**public** **static** **final** Font ***NAME\_FONT*** = **new** Font("Ariel", Font.***BOLD***, 13);

/\*\*

\* Path of the tie sound effect.

\* **@see** #playSound(String)

\*/

**public** **static** **final** String ***TIE\_SOUND*** = "/concentrationTie.wav";

/\*\*

\* Path of the win sound effect.

\* **@see** #playSound(String)

\*/

**public** **static** **final** String ***WIN\_SOUND*** = "/concentrationWin.wav";

/\*\*

\* A reference to the {@link ConcentrationCardModel} for callbacks.

\*/

**private** ConcentrationModel model;

/\*\*

\* Displays who's turn it is and who won.

\*/

**private** JLabel notifications = **new** JLabel();

/\*\*

\* This displays the players' scores.

\* **@see** #updateScoreboard()

\*/

**private** JLabel scoreText = **new** JLabel("Scores for Player");

/\*\*

\* Displays the scores of the players.

\*/

**private** JLabel[] scores = **new** JLabel[4];

/\*\*

\* This button starts a new game when pressed.

\* **@see** #newGame()

\*/

**private** JButton newGame = **new** JButton("New Game");

/\*\*

\* The rows, columns and player options.

\*/

**private** JTextField[] options = **new** JTextField[3];

/\*\*

\* A slider that changes {@link #memoryTime}.

\*/

**private** JSlider timerSlider = **new** JSlider(0,5000,1000);

/\*\*

\* The amount of time to memorize the cells (in milliseconds).

\* **@see** #cellTimer

\*/

**private** **int** memoryTime = 1000;

/\*\*

\* Whether or not the game is currently paused due to a timer.

\*/

**private** **boolean** paused;

/\*\*

\* The main method, starts graphics program

\* **@param** args - Not expecting any args.

\*/

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

**new** Concentration().start(args);

}

/\*\*

\* Initializes GUI and adds event listeners.

\* **@see** acm.program.GraphicsProgram#init()

\*/

**public** **void** init() {

model = **new** ConcentrationCardModel(**this**);

setBackground(Color.***LIGHT\_GRAY***);

//Adds notification bar

notifications.setFont(***NAME\_FONT***);

add(notifications, ***NORTH***);

notifications.setText("Welcome to Concentration!");

//Scoreboard

scoreText.setFont(***NAME\_FONT***);

add(scoreText, ***SOUTH***);

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

add(**new** JLabel(**new** String(**new** **char**[***SCORE\_SPACING***]).replace("\0", " ")),***SOUTH***);

scores[i] = **new** JLabel("Player " + (i+1) + ": 0");

scores[i].setFont(***NAME\_FONT***);

add(scores[i], ***SOUTH***);

}

newGame.setFont(***NAME\_FONT***);

add(newGame, ***WEST***); //Adds new game button

//Rows, Columns & Player options

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

options[i] = **new** JTextField("2");

options[i].addActionListener(**this**);

add(**new** JLabel(" "),***WEST***);

***OPTION\_LABELS***[i].setFont(***NAME\_FONT***);

add(***OPTION\_LABELS***[i], ***WEST***);

options[i].setFont(***NAME\_FONT***);

add(options[i], ***WEST***);

}

//Adds Timer slider

add(**new** JLabel(" "),***WEST***);

add(***OPTION\_LABELS***[3],***WEST***);

timerSlider.addChangeListener(**this**);

add(timerSlider,***WEST***);

addActionListeners();

addMouseListeners();

}

/\*\*

\* Checks to see if a cell was chosen and informs {@link #model}.

\*/

@Override

**public** **void** mouseClicked(MouseEvent e) {

**if**(!paused) {

Object obj = getElementAt(**new** GPoint(e.getPoint()));

**if**(obj **instanceof** GCard)

model.choose((Cell)obj);

}

}

/\*\*

\* Handles New Game Button input.

\* **@see** acm.program.Program#actionPerformed(java.awt.event.ActionEvent)

\*/

@Override

**public** **void** actionPerformed(ActionEvent e) {

**if**(e.getActionCommand().equals("New Game"))

newGame();

}

/\*\*

\* Handles Slider Input

\* **@see** javax.swing.event.ChangeListener#stateChanged(javax.swing.event.ChangeEvent)

\*/

@Override

**public** **void** stateChanged(ChangeEvent e) {

**if**(e.getSource().equals(timerSlider))

memoryTime = timerSlider.getValue();

}

/\*\*

\* Adds the board and adds player scores to the GUI.

\* **@param** board - 2D Array of cells.

\* **@param** player - number of players.

\*/

**public** **void** gameStartedNotification(Cell[][] board, **int** players) {

//Grays out names of non-Players

**for**(**int** i = 0; i < ***MAX\_PLAYERS***; i++)

scores[i].setForeground(Color.***LIGHT\_GRAY***);

**for**(**int** i = 0; i < players; i++)

scores[i].setForeground(Color.***BLACK***);

//Adds board to screen

**int** rows = board.length;

**int** cols = board[0].length;

**for**(**int** i = 0; i < rows\*cols; i++) {

**int** x = i/cols, y = i%cols;

add((GCard)board[x][y],

x \* (model.getCellWidth() + ***CARD\_SPACING***),y \* (model.getCellHeight()+***CARD\_SPACING***));

}

}

/\*\*

\* Returns whether or not a string can be parsed to an int.

\* **@param** string - The string to check

\* **@return** True if string can be parsed to an int, else false.

\*/

**public** **boolean** isInteger(String string) {

**try** {

Integer.*valueOf*(string);

**return** **true**;

} **catch** (NumberFormatException e) {

**return** **false**;

}

}

/\*\*

\* Starts a new game with the given parameters ({@link #options}).

\*/

**public** **void** newGame() {

//Removes all cards from canvas.

**for**(**int** x = getGCanvas().getElementCount()-1; x >= 0; x--) {

**if**(getGCanvas().getElement(x) **instanceof** GCard) {

getGCanvas().remove(getGCanvas().getElement(x));

}

}

**for**(**int** i = 0; i < scores.length; i++)

scores[i].setText("Player " + (i+1) + ": 0");

**int** rows,cols,players;

//Checks if options are invalid

rows = isInteger(options[0].getText()) ?

Integer.*parseInt*(options[0].getText()): ***MIN\_SIZE***;

cols = isInteger(options[1].getText()) ?

Integer.*parseInt*(options[1].getText()): ***MIN\_SIZE***;

players = isInteger(options[2].getText()) ?

Integer.*parseInt*(options[2].getText()): ***MIN\_PLAYERS***;

//Checks if options are over/under the limit.

**if**(rows > ***MAX\_SIZE***) rows = ***MAX\_SIZE***;

**if**(rows < ***MIN\_SIZE***) rows = ***MIN\_SIZE***;

**if**(cols > ***MAX\_SIZE***) cols = ***MAX\_SIZE***;

**if**(cols < ***MIN\_SIZE***) cols = ***MIN\_SIZE***;

**if**(players > ***MAX\_PLAYERS***) players = ***MAX\_PLAYERS***;

**if**(players < ***MIN\_PLAYERS***) players = ***MIN\_PLAYERS***;

//Checks if there is an even # of cards.

**if**(rows\*cols % 2 == 1)

rows = rows == ***MAX\_SIZE*** ? rows-1:rows+1;

//Updates text fields to reflect changes (if any)

options[0].setText(""+rows);

options[1].setText(""+cols);

options[2].setText(""+players);

setSizeCells(rows, cols);

model.startGame(rows,cols,players);

notifications.setText("Player 1's Turn");

}

/\*\*

\* Sets the size of the window based of of the table size.

\* **@param** rows - Rows of board.

\* **@param** cols - Columns of board.

\*/

**public** **void** setSizeCells(**int** rows, **int** cols) {

**int** xOffset = 152, yOffset = 117;

**int** minWidth = 580, minHeight = 440;

**int** windowWidth = (**int**)(model.getCellWidth()+***CARD\_SPACING***)\*rows + xOffset;

**int** windowHeight = (**int**)(model.getCellHeight()+***CARD\_SPACING***)\*cols + yOffset;

setSize(windowWidth < minWidth ? minWidth:windowWidth,

windowHeight < minHeight ? minHeight:windowHeight);

}

/\*\*

\* Turn the cell face up.

\* **@param** cell the chosen cell

\*/

**public** **void** selectCellNotification(Cell cell) {

cell.turnFaceUp();

}

/\*\*

\* Update the player's score and prepare to remove the cells.

\* **@param** cell1 cell to be removed (after a brief pause)

\* **@param** cell2 cell to be removed (after a brief pause)

\* **@param** player the player who scored

\* **@param** score the player's new score

\*/

**public** **void** removeCellsNotification(Cell cell1, Cell cell2, **int** player, **int** score) {

paused = **true**;

**new** Timer(memoryTime, **new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

remove((GCard)cell1);

remove((GCard)cell2);

scores[player-1].setText("Player " + (player) + ": " + score);

paused = **false**;

((Timer)e.getSource()).stop();

}

}).start();

}

/\*\*

\* Prepare to turn the selected cells face down again, and pass play to the next player.

\* **@param** cell1 cell to be turned over (after a brief pause)

\* **@param** cell2 cell to be turned over (after a brief pause)

\* **@param** nextPlayer the next player to play

\*/

**public** **void** deselectCellsNotification(Cell cell1, Cell cell2, **int** nextPlayer) {

paused = **true**;

**new** Timer(memoryTime, **new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

cell1.turnFaceDown();

cell2.turnFaceDown();

notifications.setText("Player " + nextPlayer + "'s Turn");

paused = **false**;

((Timer)e.getSource()).stop();

}

}).start();

}

/\*\*

\* Notification that the game is over, and which players had the highest score.

\* **@param** winners the winning players

\*/

**public** **void** gameOverNotification(**int**[] winners) {

String str;

**if**(winners.length == 1) {//in the event of 1 winner

str = "Player " + winners[0] + " wins!";

playSound(***WIN\_SOUND***);

}

**else** {//in the event of multiple winners

str = "Players " + winners[0];

**for**(**int** i = 1; i < winners.length; i++)

str += " and " + winners[i];

str += " win!";

playSound(***TIE\_SOUND***);

}

notifications.setText(str);

}

/\*\*

\* Plays the given .wav file.

\* **@param** URL - Path of the sound in project.

\*/

**public** **void** playSound(String URL) {

AudioInputStream audioIn;

**try** {

audioIn = AudioSystem.*getAudioInputStream*(Concentration.**class**.getResource(URL));

Clip clip = AudioSystem.*getClip*();

clip.open(audioIn);

clip.start();

}

**catch**(UnsupportedAudioFileException | IOException | LineUnavailableException e) {

e.printStackTrace();

}

}

}

**Your Screen Shots**