Application Overview

This project involves creating a Tic Tac Toe Game as the Final Project. The user will enter a number of games to play, and choose to either play as X or O.

The user will then play the games taking turns to fill in the 3 x 3 squares. Play continues until one person has O's or X's three in a row in a straight line. If there are no rows after all squares filled, it is a tie.

During the process, a history of wins and losses are recorded for each game and the board history. The user has the option to view this history after completing all games.

The user at the end will have an option to play a new set of games or end the program.

Program Logic

Input

* Prompt the user to enter the number of games you want to play.
* Prompt the user whether you want to use X or O as their marker.
* Prompt the user where you want to place the X or O in a square of the Tic-Tac-Toe board.
* Then it would be the second player’s turn in prompting to place a marker on a blank square of the Tic-Tac-Toe board.
* Continue to prompt the user to place the markers on the board until it makes a line of three X’s or O’s sideways, up and down or diagonally.
* Prompt the user if you would like to view the game history.
* Prompt the user if you want to start a new game or end the program.

Processing

* Upon choosing a marker for yourself, the second player will have the other marker from what you chose.
* Ensure that whenever the user places the marker on the space, it is empty. If not, try again until the user places a marker on an empty square.
* After the user places a marker on a valid empty space, the second player would then make its move.
* Store the locations of each of the markers on the board.
* When a line of three X’s or O’s line up sideways, up and down or diagonally, end the current game. Whatever marker made this line wins this particular game.
* If there are no lines of X’s or O’s in any order, after all 9 squares are occupied, end the game and declare it as a tie.
* After declaring a winner or a tie, proceed on to the next game if any.
* Store information of the game board after winner or tie declaration.
* Store statistical data of the wins and loss of X and O, game ties and games played.
* Clear the board of markers upon the next game if any.

Output

* Display a prompt to enter the number of games to play.
* Display option to play as X or O as their markers.
* Display the 3 x 3 board resembling much as a Tic-Tac-Toe board.
* Display information of whose turn is it and asks the player to place a marker on the board.
* Display the markers placed on the Tic-Tac-Toe board after the user places its markers.
* Display results if a particular user wins in a three consecutive X’s or O’s.
* Display results if the game play results in a tie and that there is no empty squares.
* Display an option to view the history of games played after all games played.
* Display the board of each particular game and the winner of that game if any.
* Display an option if the user wants to start a new game session or end the program.

Flowchart

A flowchart is a great way to determine the layout of the game. This is one of the first steps in determining how the game should function and work upon multiple conditions, functions and actions. A thorough flow chart will allow easy coding when it is time to code the entire program.

Pseudo Code

Markers Class:

This class initializes a list of markers X and O plus instance variables for assigning markers to player 1 and player 2.

One method assigns the marker of player 1 to the marker that the user entered.

The other method assigns the marker of player 2 according to the marker that the user did not choose. If player 1 chose X, then player 2 will have O, while player 1 having O will have player 2 having X.

GridlineInterface Class:

This class contains variables in initializing a static ArrayList, which will comprise the layout of the Tic-Tac-Toe game board.

The instance variable is in protected access to allow limited but no value changing for other classes inheriting this class.

The constructor initializes the ArrayList and its instance variables in regards to accessing different indexes of the list.

One method sets the ArrayList by assigning each of its indexes a blank string instead of null. A loop will assign the blank strings according to the indexes by the instance variables.

Another method will set the game markers according to a certain index based on user responses. Since the Tic-Tac-Toe game board will follow the 3 x 3 style, providing user responses in selecting column and row is necessary to provide proper placement of the game markers.

Another method will check the board to see if the board is full of markers. A loop will iterate through the ArrayList checking for any values of X or O. If all 9 cells of the ArrayList contain either X’s or O’s, then it will return a Boolean value to indicate that there are no more pieces to add and the game is considered a tie.

Another method will simply clear the board by removing all markers on the ArrayList.

A method toString() will return a unique character of strings that will print out the correct format of the Tic-Tac-Toe game. Indexes 2, 5 and 8 will start a new line to conform to the correct 3 x 3 format. The format will include the bracket squares and spaces between to insert the marker characters and establish realism.

Availability Class:

This class would inherit methods from the GridlineInterface class.

This class will only have one method to check to see if a particular cell has a game marker present in that cell, based on user inputs of columns and rows. By providing conditions for all cells and determine whether the cell as either an X or an O, will determine whether or not a cell is available for placing a game marker. This is to ensure that no particular game marker will overlap or replace a marker placed by another player, avoiding duplication of marker placement at same locations and avoiding unnecessary turn wasting. If there is a marker present in a cell according to user return a Boolean value of true to indicate that a marker exists on that cell. The user will later then have to enter a new value. If it returns false, then it will allow the marker placement on that particular cell.

RowCompletion Class:

This class would inherit methods from the GridlineInterface class.

This class will only have one method to check to see if there are markers filled in a way that would follow a horizontal, vertical or diagonal row sequence. By providing conditions for all cells, each cell will have conditions unique to its location to ensure correct cell inspection. Each cell will check adjacent cells in a 3 x 3 format to see if there are game markers found in those cells. If there are game markers in an adjacent cell, it will also check the farthest cell that follows a particular row sequence from the original cell of inspection. An example is the cell at the upper right corner. The adjacent cells for this cell include the top middle cell, the middle right cell and the center cell. If at least one of these adjacent cells has a game marker, then it will check the other corresponding cells, which are the top left cell, the bottom right cell or the bottom left cell.

If a particular row sequence has markers present in all of those cells, then it will return a Boolean value of true to indicate that there is a row to check for a possible winning. If it returns false, then there is no winnings possible and the game continues. This class is important to ensure that the AcceptableWins class will function properly.

AcceptableWins Class:

This class would inherit methods form the GridlineInterface class.

This class would also have one method to check to see if the markers following a particular row sequence contain the same marker type for all three of those cells. After verifying that there are complete rows in the RowCompletion class, this class will then inspect each of the cells for presence of a particular game marker. Upon finding a particular marker, it will then look for adjacent cells in a 3 x 3 format to see if those cells contain the same marker as the original cell of inspection. An example is the bottom left cell, which contains adjacent cells in the middle left, bottom middle, and center cell. If at least one of the three adjacent cells have the same marker as the bottom left cell, then it will check the farthest cell that correspond to the row sequence. In this case, it would be top left, top right and bottom right cells. If any of these cells have the same markers, as the previous two, then it will return a Boolean value of true to indicate that there is a winning pair. As a result, that particular marker with the winning pair will win this game; therefore will end the current game. If it returns false, then there is no winning pair and the game continues as normal.

TicTacToeGameRun Class:

This class is the main program that houses all the print statements and makes the Tic-Tac-Toe program user interactive and friendly whenever possible.

* Begin the game program loop to allow a possibility of new game sessions after this one ends.
* A settings loop to allow possible configuration to the Tic-Tac-Toe game.
* The settings of the Tic-Tac-Toe game include the number of games to play and marker selection, including a way to verify your choices before closing the loop.
* Ensure that the number of games to play is valid and that the marker choice is both X or O. Catch any exceptions when necessary and yield messages to ask user if needed to try again.
* Create a simple Heads and Tails game to determine who will go first, generate a random number and assign 1 of the 2 numbers as heads and the other as tails.
* Prompt the user to guess and provide only one chance to guess it right. If guessed correctly, player 1 goes first; otherwise, player 2 goes first if wrong. Prompt the user to reenter if the entry they type is invalid.
* Begin the game play loop to start playing the Tic-Tac-Toe game. This would require generating the Tic-Tac-Toe game board interface and setting the priority of who is going first according to the heads and tails game.
* Begin the game play and would involve either player 1 or 2 going first. Player 1 and 2 will receive prompts to enter a number for a column in the Tic-Tac-Toe board, including a row. Catch any exceptions when necessary and yield messages to ask user if needed to try again. Once a column and row is set, then call the Availability class to determine if the row and column set does not contain any existing game markers. If there is a marker present, re-prompt the user to enter another location, otherwise set the game marker to that location.
* After marker placement, we need to check for rows for completion if any. Using RowCompletion class, check the board for game markers that follow a particular row sequence such as horizontal, vertical or diagonal row. If there is a completed row, then call on AcceptableWins class to check the rows further. By inspecting the rows again for markers, check the rows for markers following a particular row sequence. Look at adjacent cells to see if the markers are the same as the original cell of inspection. If successful, look at the third most cells within the row sequence to see if it has the same markers as before. If all three conditions pass, then a winner is declared and the player that houses the winning markers is the winner for that game. Record game board information and clear the board to get ready for the next game.
* If there is no winning pair or completed rows, ensure to check to see if all the cells in the Tic-Tac-Toe board have markers. If so, declare game as tie, record game info and clear the board for the next game if any.
* Process repeats for the next player and so on until a winner or a tie is declared. This remains the same regardless of whether player 1 or 2 is going first.
* Increment game counter for every game completed. After playing all games, exit the game play loop.
* Prompt the user if he or she wants to view the game history. If yes, display the game board data from first to last and display statistical information. If no, skip the history compilation.
* Prompt the user if he or she wants to start a new game session. If yes, restart the game session by clearing the game board data and all statistics before establishing new game settings. If no, farewell the user and exit the program.

Testing

Just like any other Java program, many errors can possibly go wrong in any program. Extensive testing such as boundary cases, extraneous inputs, invalid inputs and correct error response is important to tackle all the bugs in a program. Testing is also important to ensure that the program performs all possible features without crashing horribly. A consumer entering an invalid response, only to result in a horrible game crash is not the way to go. Tackling the errors is a time-consuming process, but it is better to create a functional program that takes a long time to make perfect than making a program that is short to make but multiple bugs present. Money, productivity, work effort and personal motivation make a huge factor in the difference between a good and a bad program.