Matric no. 40430736

SET11102

4 December 2019

Coursework: Connect 4

A simple implementation of the game Connect 4 was developed using Java 8. The design for the game is fairly simple and uses only one class (App) with a variety of methods. This is because of the use of JavaFX for the GUI and the difficulty in creating a controller to use an MVC implementation. The class diagram is shown below.

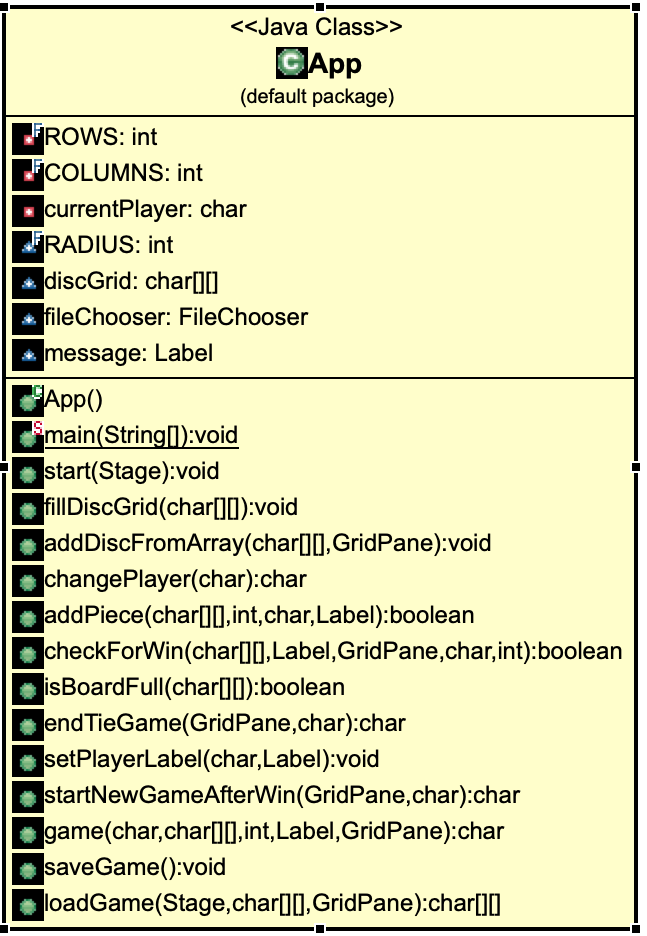


Figure 1. Class diagram

The main method launches the application, and the start method sets up the stage and creates the GUI. The other methods implement different functions in the game such as adding pieces, starting a new game, saving, etc. Ideally, at least the grid and its associated methods should have been placed into a different class, but this was not possible.

The logic for the grid uses a 2D char array. It is initially created with the array filled with ‘w’ representing white or empty spaces. When a player adds a disc, the next available white disc will be changed to ‘r’ or ‘y’ depending on whose turn it is.

The grid is displayed using JavaFX. A GridPane was added to the AnchorPane to represent the grid of the game, and it was given a blue background to represent the real life game. A loop was created to go through the 2D array generated previously. If the character in the array is a ‘w,’ a white circle is placed into the GridPane. If it is ‘r’ or ‘y,’ then a red or yellow circle is placed, respectively. Each time a disc is added, the GridPane is cleared and circles are generated from the updated array.

There are four different ways to win this game. You can have four circles of the same colour next to each other horizontally, vertically, diagonally pointing up to the right, and diagonally pointing up to the left. One method checkForWin checks for each of these each time a player takes their turn. First, it checks horizontally by checking each row and the first four columns. Only the first four columns are checked because after this point there aren’t enough columns left, so if the others are checked, it will generate an error. It adds one then two then three to each column so that it will check through the row. Checking vertically follows the same logic as checking horizontally only it reverses rows with columns.

Checking diagonally is somewhat more complicated. In the first direction, you check the first four rows and all columns. But then instead of adding/subtracting one to the rows or the columns, both must be modified. So in this direction, you would add one to the row and subtract one from the column, and continue up to three. In the second direction, you only check the first four rows and four columns. Then you add one (and up to three) to both the column and the row to check for the win.

It was also necessary to only allow players to add a piece to a column if that column had space left. This was accomplished by checking if the top space in the column selected was set to ‘w’ or not. If it’s set to ‘w,’ there is still space, and a loop will place a disc in the next available space. If the top space is not set to ‘w,’ an alert will appear asking them to choose a different column.

Lastly, the facility to create a save file and load it later was also added. This was done using Output and Input Streams. If the players choose to save their game upon exit, the current 2D array and the current player will both be saved to a location of the players’ choosing (using a fileChooser). Upon reopening the game, it will ask if they want to load a save file. If they say yes, it will open the fileChooser, so they may select their save file. It will then replace the all ‘w’ in the starting array with the saved array and set the current player to the player that was saved. If they choose to open a save file but are unable or change their mind, it will ask if they’d like to play anyway and start a new game.