INF1B Assignment 1

s2085530

Jack MCPAKE

2021

# Overview

My connect four program works by modelling a connect 4 board, manipulating the board through a controller class, and displaying it via a view class.

The users will play against each other by placing pieces on a board. The game will end when one of the players concedes or if the board is full.

I will be using the object-oriented language java to implement my solution.

# Basic Features

## Representing the Board

I will be modelling the state of the game through a board, which I have chosen to be as a 2d array of characters. With ‘a’ representing the character of one player, and ‘b’ representing the character of another player.

## Playing Pieces

To play a piece, the player must choose a column to ‘drop’ their piece into. If they chose a column with free space in it. The player’s piece will be dropped into the column one slot above the last played piece in that column. If the column is empty, the piece is dropped into the bottom of that column (or the highest index of that column). If the column is full, the player will be asked to put their piece into a different column.

To check if the player can play a piece, I created a module gameTermConditions() that would return one of two characters. ‘g’ for ‘go ahead’, meaning that the game should go ahead as normal, and ‘f’ for ‘full’ for when the board is full. I chose char for this so that I could display different messages based on the state of the board once the game is over.

## Game Loop

The game will continually ask the player to insert a piece, if they conceded, check if that piece is valid, and then play that piece until the board is full, or the player concedes. I chose a while loop to implement this, because I would have to loop the instructions based on if gameTermConditions() would return ‘g’.

## Game Over

Once the game is over (i.e., gameTermConditions() returns ‘f’, or move is put as 0 which means concede), the user will be prompted that either the current player has conceded or that the board is full. Based on which one of these are true, the message will be different.

## Testing

I tested this part of the program by playing some games against myself, filling up the board, and conceding against myself. Early on I came across and error where the program would crash if the user put a value that wasn’t one of the columns from the game board, I fixed this by adding a check when a user plays a move called isMoveValid() which checks if the move is within the permissible range of column values.

# Intermediate Features

## Start New Game

I added a new method in the TextView class which would ask the user if they wanted to start the game again called askAgain(), and a new method in the Model class called setBoard() which would reset the board. I called these into a new section at the end of the game loop, so after the game is over, the user is asked if they want to start the game again. If they do, then the board is reset, and startSession() is called.

## Variable Game Settings

For the setNrRows() and setNrCols() methods in Model class, I rewrote them so that they will read integers from the user, and check if the input integer is less than or equal to 4. While this is true, the user is once again asked to input an integer. I didn’t need to change any code anywehere else because the setNrRows and setNrCols were referenced anywhere column and row length was refrecned.

## Enhanced Input Validation

The inputUtil class checked if my inputs were of the correct type already, so I just had to make sure that inputs were in the correct range. For example, in my variable game settings, I made sure that the input for rows and columns was less than or equal to 4.

## Automatic Win Detection

I split the automatic win detection into 4 parts. Every time a piece was played, the game checks if there is a piece on the board, using the gameTermConditions() method. Then it goes through the following checks.

### 4 in a Column

If the row index is less than or equal to 3, it can’t go up 3 times, so it goes down. If it detects that the next three in that column are the same, then it returns the player who’s row it detected. Signalling a win for that player. Otherwise, if the row index is more than or equal to the number of rows – 3, it will go up the column, checking the next 3 positions if they are the same. If this is true, then it returns the player who’s row it detected. Signalling a win for that player.

### 4 in a Row

If the column index is less than or equal to 3, it can’t go back 3 times, so it goes across. If it detects that the next three in that column are the same, then it returns the player who’s row it detected. Signalling a win for that player. Otherwise, if the column index is more than or equal to the number of column – 3, it will go backwards within the row, checking the next 3 positions if they are the same. If this is true, then it returns the player who’s row it detected. Signalling a win for that player.

### 4 in a Diagonal Line

How I figured out diagonal in detection is through deducing that a diagonal line for a win in connect four can only go one of four ways, down and left, down and right, up and left, and up and right. And each of these directions have a terminating condition, they can’t start if the point lies in the opposite corner of them. (e.g. down and left cant start at the bottom left corner). So how I did my diagonal win detection was:

*If the point doesn’t lie in the bottom right corner (column and row indexes are not more than the number of rows or columns - 4), then check the next 3 left and down points if they are the same. If they are, signal a victory for that player.*

*If the point doesn’t lie in the top right corner (column and row indexes are not less than 3), then check the next 3 left and up points if they are the same. If they are, signal a victory for that player.*

*If the point doesn’t lie in the bottom left corner (column index is not less than 3 or row index is not more than the number of rows - 4), then check the next 3 right and down points if they are the same. If they are, signal a victory for that player.*

*If the point doesn’t lie in the top left corner (row index is not less than 3 or column index is not more than the number of columns - 4), then check the next 3 right and up points if they are the same. If they are, signal a victory for that player.*

# Evaluation

In hindsight, I should have planned out my time more efficiently so that I could do some of the advanced features.

As well as this, I ran into a problem where I lost a lot of my progress due to not backing up my work. In my next assignment I’ll use backups more often.

My diagonal win detection breaks if the board is 4x4, so I had to make the user not be able to enter 4 as height or width for the board.

I had also planned for the player to be able to pick the character they play with, as well as the colour of their piece.

There are also quite a few inefficient points of my program, especially with how win detection is handled, but I chose readability and self-explaining code over efficiency.