ASSIGNMENT ONE (CONNECT 4) REPORT:

Implemented Features: *All (Basic, Intermediate, Advanced).*

Basic Features: (features implemented: all)

The core features of the Connect 4 game are collected in the Model, TextView and Controller classes. The Model class contains the data and the behaviour required to model the state of the game (game logic, settings, internal representation, etc.), while the TextView class takes care of the visual aspects of the game (outputs to the screen). The Controller collects the methods and logic required to communicate between the Model and the View (game loop, game reset, input handling, etc.). All of the other classes found in the project were either added or modified later in the development process.

In order to model the game, the Model class defines a two-dimensional array for the internal representation of the game board. Such a data structure allows us to abstract the physical board as a matrix consisting of rows and columns, with pieces easily inserted into it through the use of valid ‘x (column) and y (row)’ indices. The two-dimensional array also provides us with an easy way to cycle through all of the positions in the board (through the use of nested for loops running from the minimum row and column position to the maximum positions, respectively). Ultimately, I believed the two-dimensional array was perhaps the most intuitive and easy-to-work-with abstraction of a matrix that we can define in native Java.

The player turn is defined as a private integer field in the Model class and takes the value 1 (when it’s player 1’s turn) and 2 (when it’s player 2’s). Switching the players is handled by the switchPlayer() function, which uses a very simple modulo operation to change from 1 to 2 and vice-versa. This method is called at the end of each player’s turn so as to implement the turn-based behaviour. Player tokens are also defined as a private field in the Model class, this time saved in a one dimensional *char* array, such that each player’s piece can be referenced through indexing this attribute (*currentPlayer - 1*), avoiding unnecessary conditional statements.

For this assignment, I didn’t automate the testing process through component or unit tests, instead recurring to manual testing throughout. A better implementation of the game would include automated tests implemented using JUnit (or equivalent) to ensure the program behaves as expected for different game states. When testing, I always started with invalid inputs, so I could handle those early on in the development process, following this up by testing for edge cases (particularly when a counter, loop or similar was involved) and finishing up with testing normal inputs and comparing the program’s output to an expected one. This involved a lot of backtracking, but once a block of related code was seemingly working fine, I would refactor this into a method and move on to the next feature, ensuring future errors wouldn't come from old code. Debugging was done through IntelliJ's built-in debugger, and was particularly useful in implementing the advanced features.

The toughest feature to implement in this section was probably the game board. Turning the two-dimensional array into a visible matrix output took some thought, but the provided utility functions proved useful.

I believe there are no remaining issues relating to the features in this section.

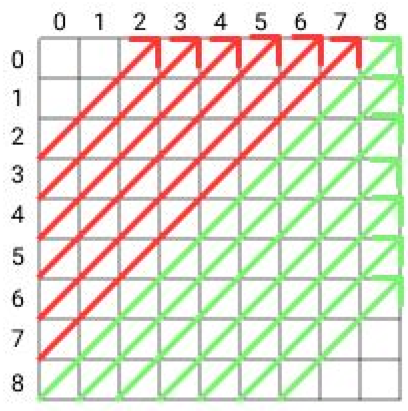
Intermediate Features: (features implemented: all)

Most of the intermediate features are implemented in the Controller and Model classes, as they relate to the flow and state of the game. Restarting the game is handled in the Controller class by the function *resetGame()* as it is related to the flow of the game. The function consists of a simple character input which uses a while loop for input validation so the program does not crash given invalid input.

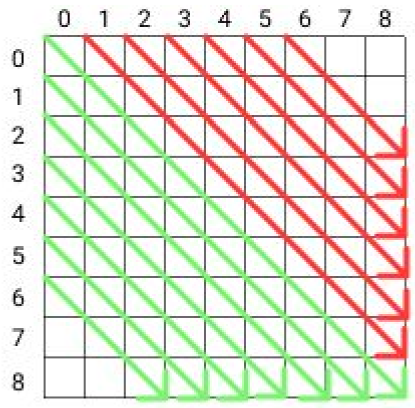
Adding variable game settings to the game is handled by the *inputSettings()* function in the Controller class. The class returns an array of integers collecting all of the relevant settings inputted by the user. This design allows for an easy and concise way to communicate these settings to the model by passing the one-dimensional array as a parameter to the method *startNewGame()* in model. The Model class can then use indexing to access each setting and initialise the game's starting attributes (ie. the attributes of the Model class). The order of the settings was left as a comment in both the relevant Model and Controller classes so that anyone reading the code understands where the indices came from, although perhaps there's a more intuitive way of doing this.

Handling invalid input was just a matter of ensuring the inputs were consistent with the Model's attributes, which are accessed by the Controller through the use of getter methods to ensure encapsulation. Whenever asking for an essential input, I used a simple while loop to validate it such that invalid data could not crash the game, always asking for further inputs until valid ones were given. When asking for the settings, I used this same while loop technique coupled with a set of conditionals which I derived for a valid combination of game settings (row and column number must be greater than 1 and number of pieces in a row less than or equal to both dimensions and not equal to 1).

The automatic win detection was definitely the hardest and longest part of this section. The code implementing this feature originally resided within the Model class, but later in the development stage, I decided to refactor it into its own GameRules class because of how long and elaborate it was, breaking it up into individual methods to check each of the possible winning states in the board. The internal representation of the board allowed for a simple nested for loop structure to be used to cycle through all of the columns and rows in the board and count the number of consecutive pieces. In order to achieve this, a counter variable is used, which is incremented every time the same piece is encountered and reset if the opponent's piece is found instead. If the counter reaches the winning requirement, the function will return true and the Controller will apply the appropriate logic in the game loop to output the corresponding game over message. The hardest thing about the automatic win detection, however, were the diagonal checks. These are separated into two methods for the sake of concision: one detecting ascending diagonals (positive slope) and another detecting descending diagonals (negative slope). These functions in turn have two separate checks within them as they check for diagonals in both halves of the board, as illustrated in the diagrams below.



Ascending Diagonals



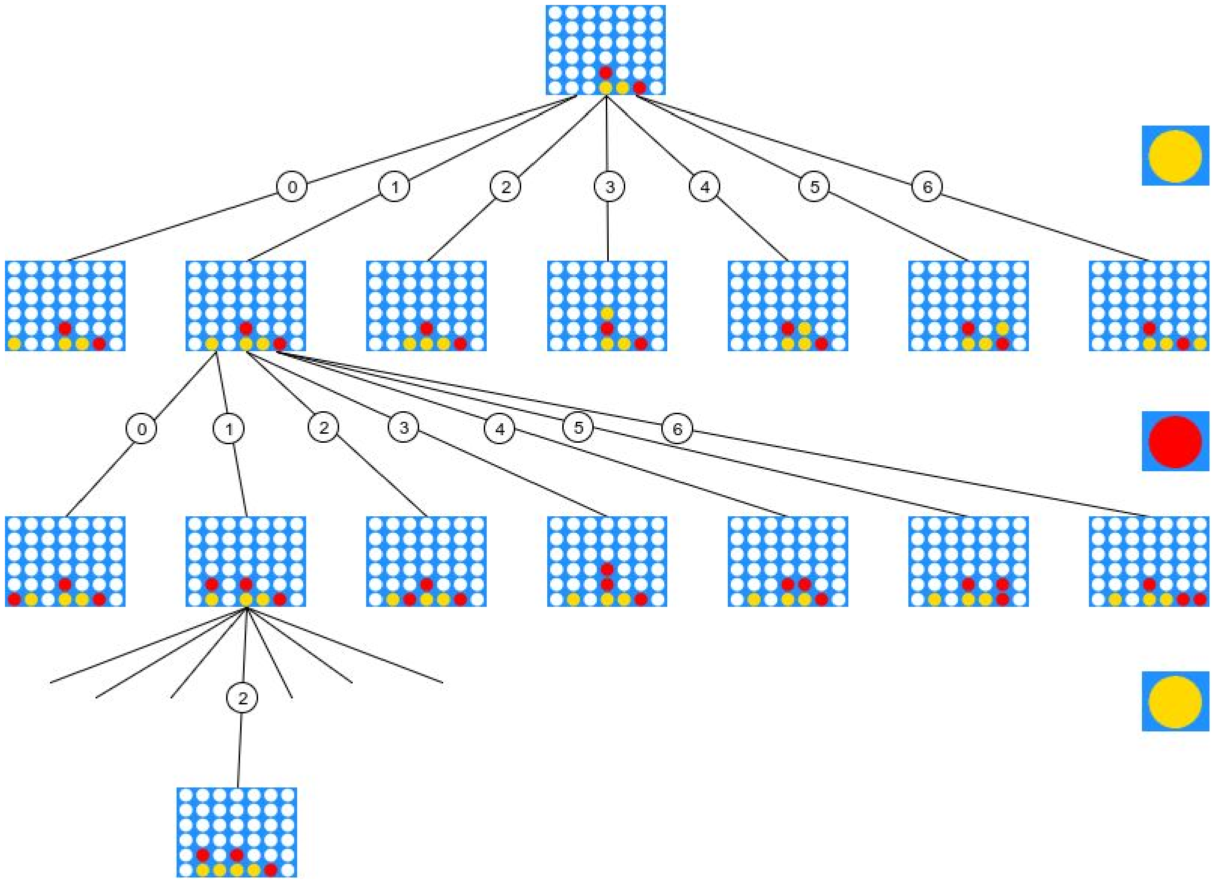
Descending Diagonals

In order to cycle through the board diagonally, I tried many different approaches including the use of different counters and incrementers or extracting diagonals in a different method. Ultimately, however, I chose to use a for loop with two different counters, each initialised at different points in the board and incremented by different intervals so as to achieve this motion. This turned out to be the most efficient and concise way to implement these diagonal checks.

Advanced Features: (features implemented: all)

Adding a non-playable character (NPC) to the game proved to be the most complicated task in this assignment. In order to include the NPC, I decided to prompt the user for the game mode at the start of the game. The *selectGameMode()* method in the Controller will hence ask the user if they want to start a 2 Player game or play against the computer, their choice will then be assigned to the appropriate Model field to inform the Model class of their decision. The rest of the implementation simply involved updating the main game loop with a conditional that wouldn't ask for a prompt if it was player 2's turn and the user was playing against the computer. The real trouble with implementing this feature came with the logic used to compute the AI's moves. After some research online, I decided I would try to implement a simple version of the Minimax algorithm for the AI logic. The result isn't optimal and the AI has many flaws, but I believe it is working as expected.

The Minimax algorithm works by computing all of the possible game states after a specific number of moves (referred to as the depth of the algorithm) and evaluating each game state according to some heuristic. In order to calculate the best outcome, the algorithm considers 2 agents: the maximising player (AI) and the minimising player (human). When it's the maximising player's turn, the AI will choose the move which yields the highest score for itself; if it's the minimising player's turn, the AI will assume the move with the lowest score for itself (highest score for the human player). The algorithm will then make the move which yielded the highest overall score for the AI, assuming the human player plays optimally in each turn. The diagram below represents the process as a tree.



(source: *https://towardsdatascience.com/creating-the-perfect-connect-four-ai-bot-c165115557b0*).

The biggest challenge with this implementation came with the heuristic function. Calculating a score that would be suitable for every board with variable game settings proved quite complicated. My final implementation simply looks at columns and rows, extracting sublists of length *connectX* (the number of pieces that need to be connected in a row to win) and checking how many ai and player pieces as well as spaces there are. For connect 4, for example, if there are 4 AI pieces in a row of a potential board, that board will get a very high score. If there are 3 AI pieces and 1 space, then it will get a slightly lower score, etcetera. Analogously, if it is the player who is about to get a 4-in-a-row in a particular board, it will receive a very low (negative) score. This solution is far from optimal. For starters, the AI is completely blind to diagonal moves, and so this is a very easy way to consistently beat it. Moreover, the heuristic function does not work as expected for every possible board for reasons I cannot explain. This is particularly the case in smaller sized boards (ie. 3x3) where the AI performs better with a smaller depth. Adding a bias for the center column as done in the code (ie. counting the number of AI pieces in centre column, multiplying by 3 and adding this to the score) is a partial, but not ideal, solution. If I had had more time to work on the code, I would have perhaps considered calculating the score of each board differently (ie. counting the number of possible x-in-a-row in each board) to see if they worked better. While the solution works, it is more than likely that I made some mistakes implementing it which are causing these errors, but after extensive debugging, I couldn't find anything I didn't fix in this final version.

Saving and loading the game involved using the java API for the first time in the development of this application. Saving the game occurs in the Model class (since it's where all of the game state data is found), whereas loading happens in the Connect4 class (since it's when the game starts). It might have been best to write these methods in an alternative location, but the current one made sense to me. In order to perform the actual IO, I decided it would be cleanest to simply save the whole game state (Model) object into a file and then load it back up. To do this, I used the Object Output and Input Stream functions provided by the API. These require the classes being saved to implement 'Serializable' and to have a (unique) private, final, static long type field called serialVersionUID. After these changes were made, it was simply a matter of saving the object into a file in the src directory. Loading the object involved overriding the Model constructor so as to allow for some loaded settings to be passed and assigned to the fields in Model upon creating the object. With this code in place, all that was left was adding a way for the user to save the game state, and I decided to assign the input 0 when making a move to such a purpose. In the initial prompt, the user is informed of this, but I am aware that it is bad UX and maybe a more user-friendly message and input should be used instead.

Finally, in terms of the application's architecture, the provided code used the Model-View-Controller (MVC) design pattern. As suggested by the acronym, this design pattern separates the application into 3 separate classes: the model (which takes care of the object and its data), the view (which is responsible for outputting the data contained by the model) and the controller (which allows the two to communicate, controlling the flow of data from the model to the view). This pattern is useful because it allows multiple programmers to work simultaneously on each of the 3 components making development faster. Additionally, the application will be easier to update, easier to debug, and will allow for multiple views to exist for the same model (if there are multiple ways to output the same data). On the other hand, disadvantages of the MVC design include a less intuitive project structure (multiple files, more abstraction and a requirement to be familiar with the separation of tasks), but overall, this design pattern is quite widely accepted.

Evaluation: *For this assignment, I believe I have successfully implemented all of the suggested features. Even though some of the advanced ones (particularly the AI) aren't without flaws, the code is still entirely functional and the game is playable.*