Project: Chess

My project consists of a Chess game that can be played with two players. This game features a GUI, with which the users can interact to move their pieces during the game. The controls are to select a piece of the current player's colour, then to select the square that the piece should move to. The game ends once one player is in a checkmated position (they cannot escape their king being captured on the next move), or in a stalemated position (they have no legal moves, but their king is not currently in danger).

Implementation

Chess has a lot of rules that I do understand so here are the ones that are implemented, and how they were implemented (yes, I took my chess advice from Wikipedia). Test files are also included to test some of these features under the 'Load Game' button.

Setup: https://en.wikipedia.org/wiki/Chess#Setup

Movement: https://en.wikipedia.org/wiki/Chess#Movement

Castling: https://en.wikipedia.org/wiki/Chess#Castling

The CastleTest.txt file allows for testing of this. The white king may currently castle right, but not left because the black queen challenges one of the squares the king will move through. If the black queen moves one square to the right, the white king may castle left. This is allowed even though a Black bishop challenges a square involved in castling because only the white rook moves through that square (not the king). If the black queen moves down five squares, white may no longer castle because the king is in check. If the white queen captures the black queen the turn after, white may once again castle.

En Passant: https://en.wikipedia.org/wiki/Chess#En_passant

The EnPassantTest.txt file allows for testing of this. The white pawn which is moved up may capture the black pawn to its left with en passant. If white chooses not to do this, it loses this opportunity on the next turn.

Promotion: https://en.wikipedia.org/wiki/Chess#Promotion

The PromotionTest.txt file allows for testing of this. The white pawn which is moved up may move forward one square and be promoted. If the same situation is created with a black pawn, colours are adjusted accordingly on the board as well as the pop-up window.

Check: https://en.wikipedia.org/wiki/Chess#Check

Checkmate: https://en.wikipedia.org/wiki/Checkmate

The CheckmateTest.txt file allows for testing of this. If the white queen moves up and to the right four squares black in checkmate and the game is ended.

Stalemate: https://en.wikipedia.org/wiki/Stalemate

The StalemateTest.txt file allows for testing of this. If the white queen moves down and right two squares, black is in a stalemate position.

Below is a list of classes and descriptions of how their state and behavior is used in the game of chess.

Pieces classes:

The Piece class is an abstract parent class that outlines the state and behavior that every type of piece needs to inherit. All pieces must have a get moves function which takes the board and returns the squares that this particular piece could move to on the next turn.

All pieces must also be able to return the local variables which are stored in this class. These include an image of the piece, the location of the piece, the previous location of the piece, and the colour of the piece. The getLetter method was used to display the board in the console but its functionality has since been commented out, I chose to leave the code in because it still may be useful for debugging purposes.

This class also implements concrete methods which include a moveTo method which changes the location of the piece, a compareTo method which sorts the pieces by importance to be displayed in that order once captured. This implements the 'Comparable' interface. It finally also reads and writes its information to saved files.

Main Game Classes:

These classes are organized and inspired by the code written in the tutorial practice problems and assignments. All variables and methods are implemented using appropriate encapsulation principles. The only public variables are the final int Gameboard.Width, Height variables, which are simply used for code readability throughout the program, rather than using arrays and loops of an arbitrary size 8 everywhere.

Point2D class:

This class is a copy-paste of the same class used in the tutorials which effectively demonstrates the modularity and reusability of object-oriented programming.

GameBoard class:

This class was inspired by the GameBoard class from the moving ball and player game tutorial. This class stores information about the state of the board (where the pieces are, which are captured etc.) and updates itself using the display method. This class is also responsible for promoting pawns for other pieces.

Game class:

This class was also inspired by the Game class from the moving ball and player game tutorial. This class stores information about the entire game including all pieces (separated by captured or not), the board, the previously selected piece. It allows for pieces to be added, for example in the case of a promotion, removed, also like in the case of a promotion, and captured. It also controls whose turn it is.

ChessModel class:

This class represents the model part of the MVC paradigm. It holds a game and is responsible for adding pieces to the game as they should be in the default starting layout of chess. This allows for new games to be started by simply calling model = new ChessModel();

IllegalMoveException class:

This class extends exception and is thrown whenever a player tries to move themself into check. When caught. This exception uses its 'handle' method to handle itself using a similar approach to the one used in the tutorials.

ChessApp class:

This class represents the controller part of the MVC paradigm. It is responsible for gathering user inputs and translating them into commands for the model to interpret based on the rules of chess. This class uses its various methods to implement all of the rules mentioned above. It is also responsible for creating multiple views upon clicking certain buttons that demand a new Scene to be used.

This class demonstrates the largest use of OOP principles throughout this project. It uses all three components of the MVC paradigm which requires it to utilize all classes mentioned above through the ChessModel class and also utilizes all of the view classes mentioned below. This class is a subclass of Application which demonstrates the power of abstraction through classes for the simple fact that I have not created this class but am still able to understand how to use it for my program. Many of the Instances of other classes held in this class also implement other interfaces such as Serializable for saving objects to .txt files and Comparable for sorting Piece objects.

The Piece classes which are held in the model of this class are set in a hierarchy where all pieces extend the Piece Class. This allows for the use of polymorphism throughout this controller class by using abstract methods to represent all subclasses of Piece, as a Piece. This ultimately reduces code duplication in allowing all different pieces to often be represented as one.

This class also implements several ArrayLists which are ADTs. Although I did not implement the arrayList myself, I understand how ArrayList is its own class which allows me to use OOP principles on it and have yet another advantage in simplifying my code.

View classes:

All view classes use essentially the same concepts from the MVC paradigm. All of them extend Pane and use JavaFX components to create a simple GUI for the user to interact with. Each of them is slightly different to serve their different purposes, but all of them share the same simple idea of adding components such as buttons or labels to themselves, resizing and coloring themselves, and being able to be instantiated through their constructors. Since their components are private, they all use get methods so the user can interact with their components using encapsulation. All of them are instantiated in the ChessApp class directly as the scene of a Stage (usually named secondaryStage).

Trade-offs and Optimizations

One of the parts of my design that is not the best is how the event listeners in the ChessApp class are implemented. I don't know too much about how initializing event listeners works so I don't understand entirely how the code on lines 60 and 61 affects time complexity, but if it has the effect of making everything at least n^2 by default it is not good. Moving forward if I am to use JavaFX again in one of my projects, I would like to read more about EventListeners and how they work.

For all my algorithms, time complexity was taken into account and optimized even though it is not explicitly stated. For some calls, code is being repeated and could optionally be stored in memory for a faster algorithm, but since no methods were taking a noticeably long time to execute, this was not often implemented. In some cases I did chose to store elements such as model.getGame().getGameBoard .getPieces() in a variable to improve on this a little, but cases like this are often better represented as upgrades in readability than time complexity.

One of my favourite types of improvements to implement does not necessarily change time complexity but makes it more likely to execute in times closer to the best-case complexity than the worst-case complexity. I learned the importance of this while implementing alpha-beta pruning into my Connect 4 AI in 1405 and was sure to implement something like this here again. When the game searches for checkmate or stalemate, it searches through every piece and checks if any of its moves would remove the player from triggering one of these situations. Since pieces like Queens and Rooks have more moves, they are more likely to be able to be the piece that removes a player from one of these situations. Based on this observation, I have deliberately chosen to add them to the game's pieces ArrayList first (in the ChessModel class) to increase the likelihood that these algorithms will finish executing sooner.

Final Notes

I was sure to implement my algorithms in a way that covered every course concept at least once, except for recursion. I really could not find a use for recursion in my program that would present a more elegant solution than my iterative algorithms. I think that the best use for recursion in a game of chess would be to implement an AI, like the one I used during the 1405 course that would simulate future moves and iterate over these moves in a recursive fashion using a tree. The reason I chose not to do an AI this time around is that in chess there are many more moves to search on every turn and this would make for a much less effective AI using the methods I implemented last class. If I wanted to do a project like this, I would likely need to learn more about Ais. This is something to keep in the back of my mind as I progress in computer science because it is something that I would be interested in trying out.

I was able to implement everything that I outlined in my proposal for this project, including some of the additional notes. Instead of implementing a replay function, I decided to take your advice and implement a save function which I think proved to be more useful and allowed me to save 'buggy' situations and test them more efficiently. I chose not to implement a move hint option because I decided that was too much like cheating and was not something that a proper chess app would allow for if playing against other real players (like in my case). I was also able to implement the multiple view classes that I wanted to which helped to develop my understanding of the usefulness of the MVC paradigm. I was also able to implement an invalid move exception as outlined above. Initially, I was not going to implement en passant, castling, or stalemate since they were rules that I didn't understand too well but after reading about them they seemed like quite crucial rules for chess, and given that I had time, I chose to implement them.

Useful Sources

https://en.wikipedia.org/wiki/Chess

https://openjfx.io/javadoc/15/