**Final Project Report for Even Semester 2020**

*Submitted as course project of*

**OBJECT ORIENTED PROGRAMMING**

**COMP6699**



*Written by:*

**Edward Mathew Kurniawan - 2440032316**

*Under the Guidance of:*

**JUDE MARTINEZ**

**Lecturer of Object-Oriented Programming**

**School of Computer Science**

**Binus International University**

**Project Requirements**

**Functional Requirements**

Specifically, the final project application MUST include examples of the following AT A MINIMUM:

* Use of primitive data
* Use of instance variables and objects
* Use of imported classes
* Use of custom-built classes & methods
* Use of Java Collection
* Use of exception handling
* Use of inheritance, polymorphism and interfaces
* Detailed Documentation Commenting
* Detailed Commenting of Methods
* Detailed Code Commenting

**Non-Functional Requirements**

Creating a working implementation is far from sufficient to achieve full marks!

* Stick strictly to the above specifications
* Don’t forget to add helpful comments
* Use (long) meaningful identifiers
* Use ample white space and a consistent indentation scheme
* Make code easy to read by keeping it simple
* Avoid duplicating similar code
* Ensure that all domain classes are modeled properly
* Use a proper access control to variables and methods

**Plagiarism/Cheating**

Binus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

**Declaration of Originality**

By signing this assignment, I understand, accept and consent to Binus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

(Name of Student)

Edward Matthew Kurniawan

1. **Program Background**

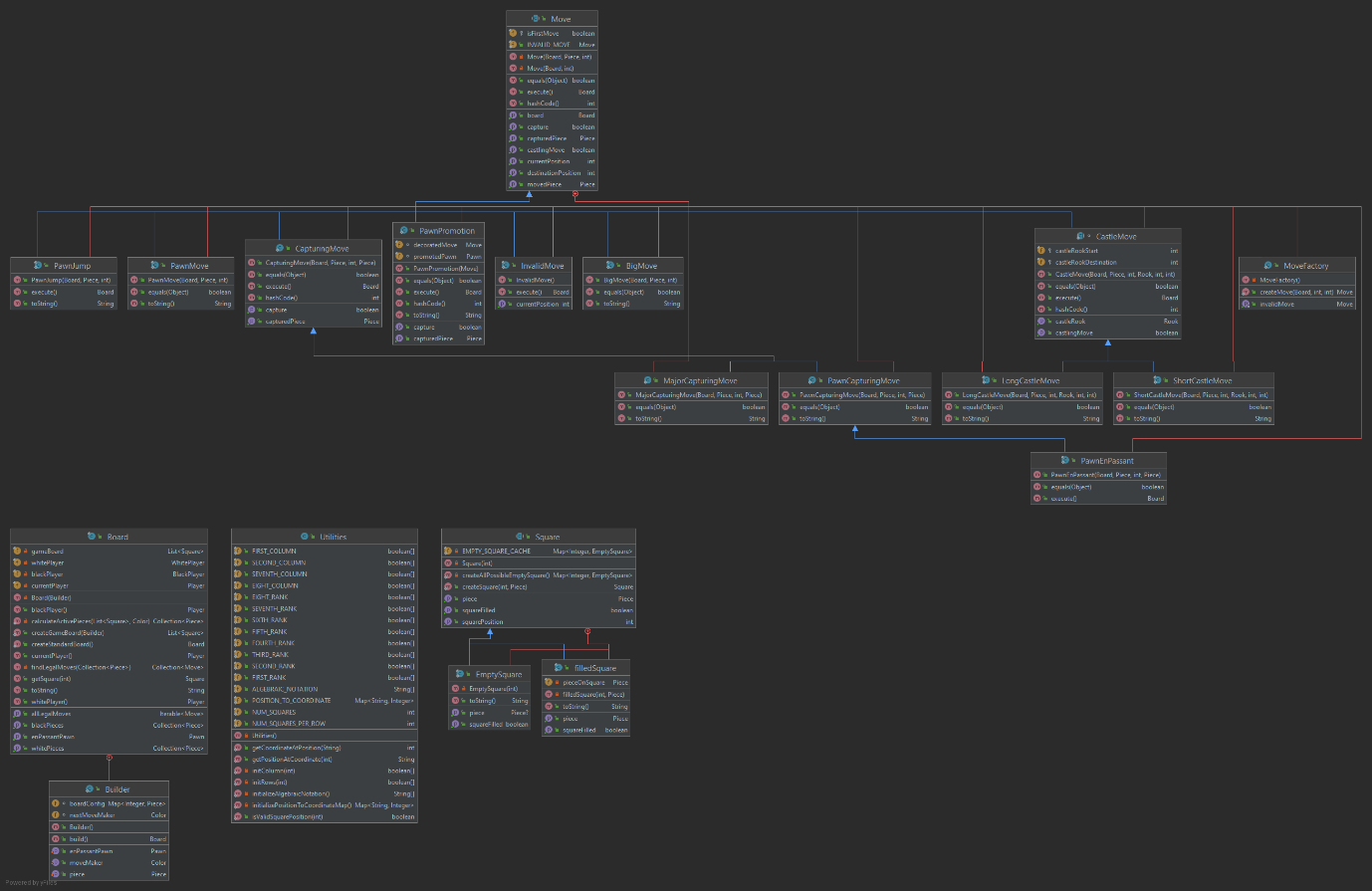
I made a chess game using Java. It follows the basic OOP principles to emulate a real-life chess game.

1. **Project Specifications**

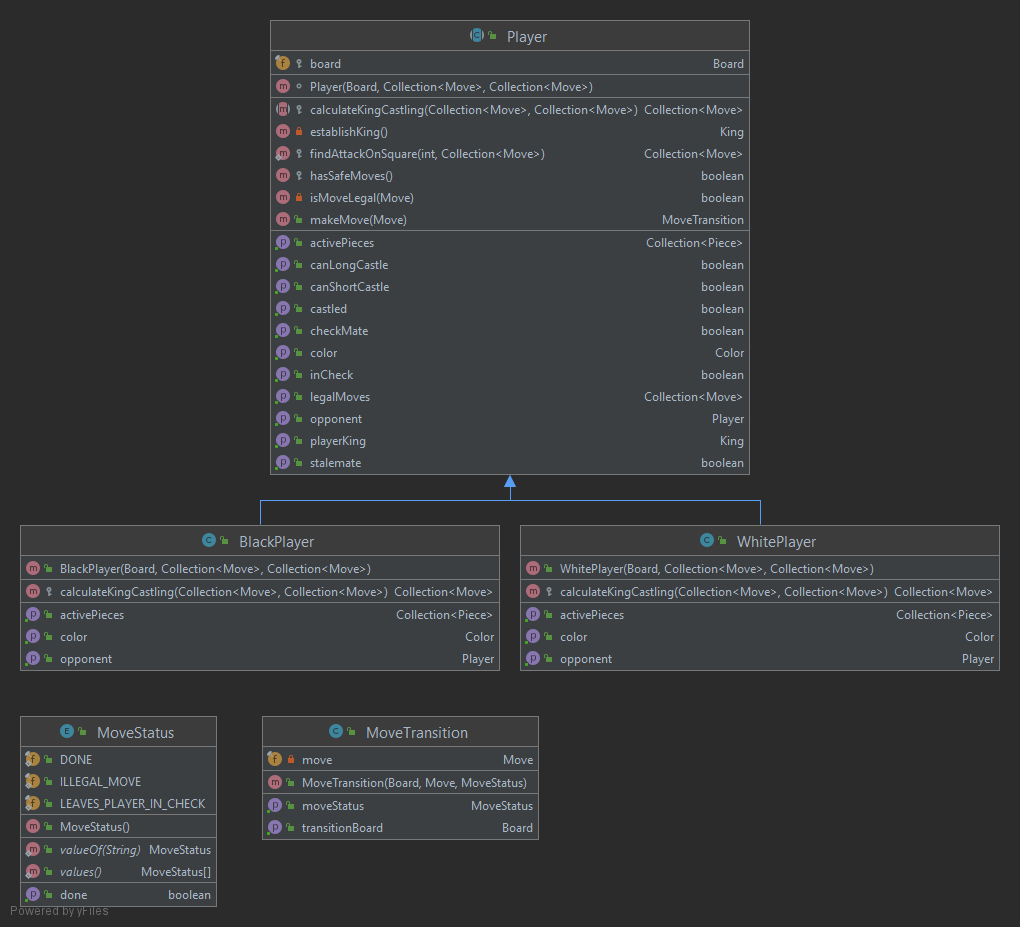
The chess game is built purely using Java 8 with IntelliJ as my IDE of choice. The GUI framework I chose to use are Java Swing and also Java AWT.

1. **UML Diagram**

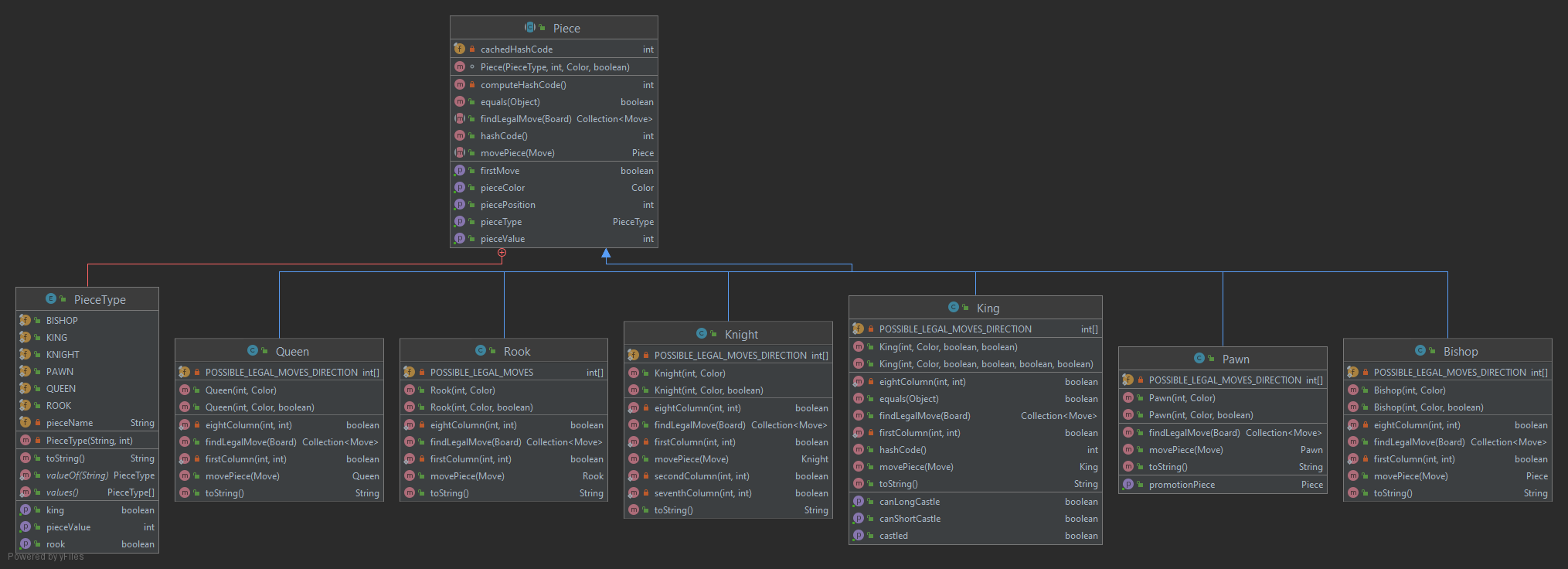
* **Board Package**

****

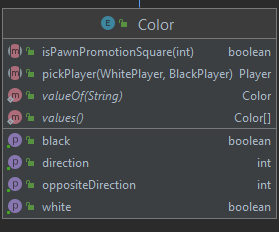
* **Player Package**

****

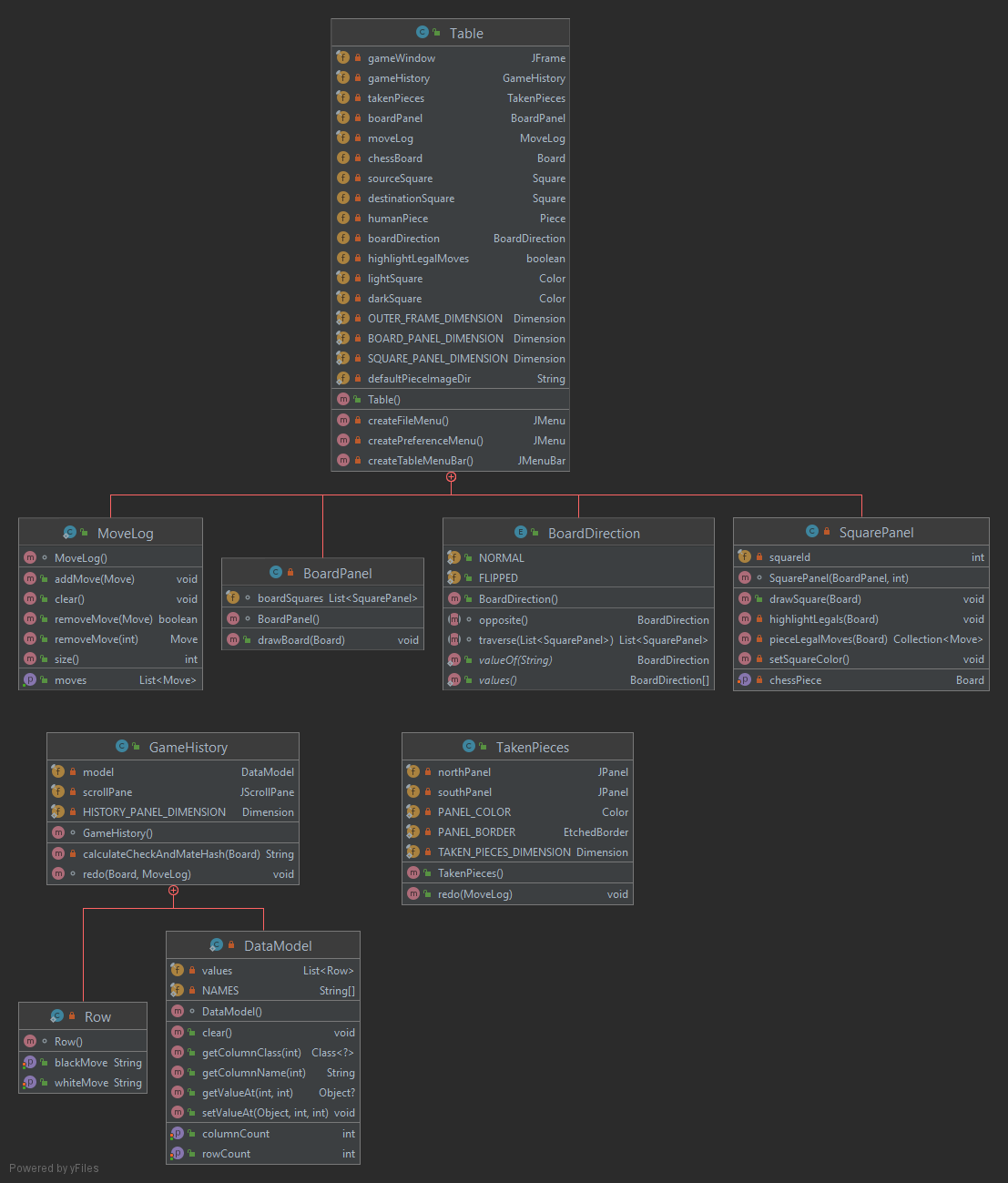
* **Piece Package**

****

* **Color Enum**

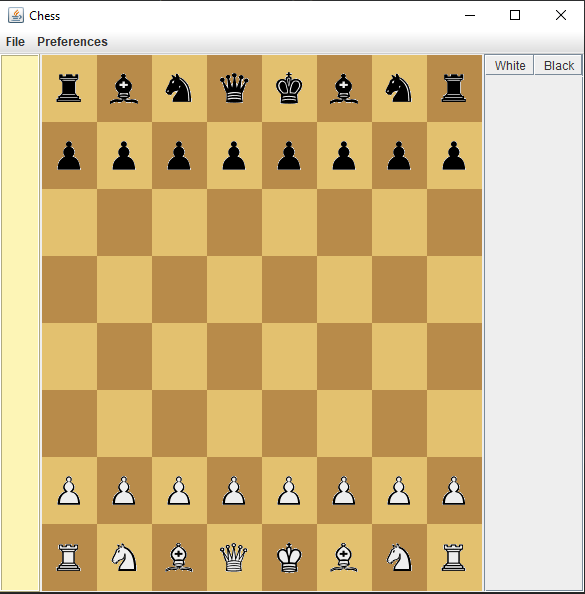
****

* **GUI package**

****

There are a total of 4 packages and 1 separate enum that I implemented. These packages correspond to the parts of a chess game. The Board Package handles the board and move generation part of the game. The Piece package handles the characteristics of each piece. Each piece also has a unique method called findLegalMove to calculate the moves each piece could make in their position. The Player Package handles the turn calculation and the interaction of the black and white players playing the game. Finally, the GUI class handles the physical GUI of the chess game implementation. The Color enum is for comparing if a piece is a certain color, black, or white.

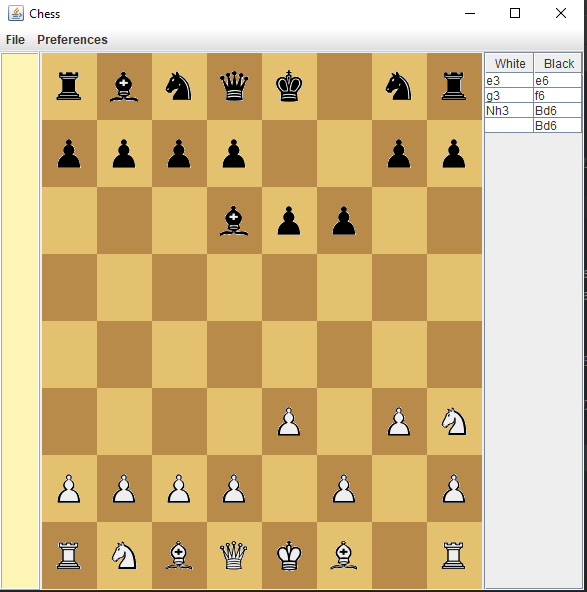
1. **Application Interface**

****

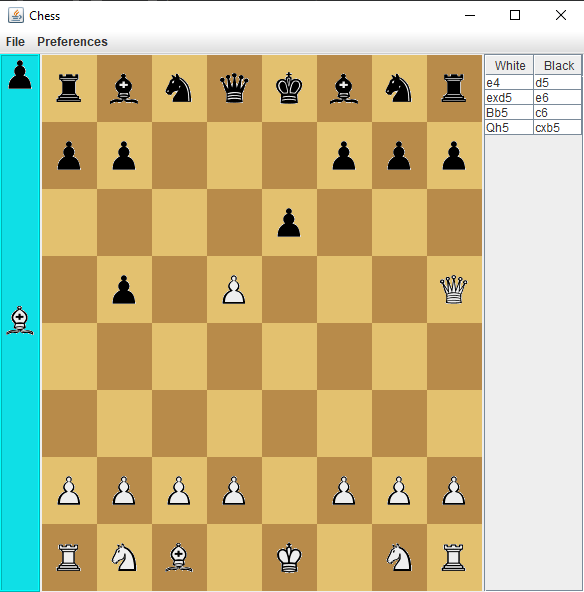
This is the interface of the chess game. The board and the pieces are exactly the same as real-life chess. I also added a move log in the right side of the interface. The left-hand side of the interfaces sports a panel to show the pieces that have been captured in the game. There are also two buttons in the top panel. The “File” panel controls the PGN import and is also used to exit the program while the “Preferences” panel sports a flip board option and also a highlight legal moves option.



When turned on, the highlight legal move option will show green dots depending on the piece selected. The user will be able to move the pieces according to one of the places specified by the green dot



The move log in the right-hand side will then keep track of all the moves played during the game. The log itself will display the moves in the most popular notation known for recording chess games, the algebraic notation.



There is also a panel displaying the taken pieces on the left-hand side of the GUI. This panel will show the pieces that have been taken by both black and white players. The panel will also sort the displayed pieces based on their value. The sorting goes from the most valuable pieces, to the least valuable.

1. **How The Application Works**

The application works like a how a normal chess game would. The goal is to put the enemy King in checkmate. The player who controls the white pieces moves first, then alternates with the player who controls black. The game goes back and forth until either a stalemate, or a checkmate is achieved

1. **Libraries**
2. Google Guava

Google Guava is used here to enhance the Immutability of the program. This is done to prevent the board from accidentally changing.

1. Java Swing

Java Swing used in tandem with Java AWT provides the GUI framework for the game

1. Java AWT

Java AWT is used in tandem with Java Swing to provide the GUI framework for the chess game. The input and output of the GUI is also handled by Java AWT (moving the pieces)

1. **Lessons Taken from Doing the Project**

I took on the challenge of making a chess game with another end goal in mind, which is to create a chess engine. The engine would then be capable of playing with a human player using a different moves each time that would not draw from a pool of preset moves. In the end, it proves to be much more challenging than I had anticipated, so I decided to just perfect the chess game itself and make the engine as a side project during the holidays. Java Swing and Java AWT is used to make the GUI, and it was alien to me when I started to create this project. I was a bit overwhelmed in implementing those libraries efficiently and properly. While the GUI was certainly hard to implement, the hardest part of the project was definitely making sure that all the chess moves are implemented properly. The number of bugs I encountered when creating this project is no small thing. While implementing how the pieces move wasn’t as easy as I had expected, some of the pieces had special rules and also special moves that I also need to account for. All in all, I would say that this chess game is by no means perfect, but it is working as intended. I would also say that using Java Swing and Java AWT has been interesting and I will certainly use them again in future GUI projects.

1. **Project Technical Description**
   * Why a chess game?

At first, I had not thought of a chess game at all when considering the options that I had during the beginning of this project. I settled on a chess game because not only did it hit close to home, it also had a considerable amount of OOP in the game itself and it would prove to be an interesting challenge to implement. A chess game could also be the base to coding a chess engine, something I plan to attempt in the future.

* + Making the back-end

I used only Java in implementing the back-end for the program. I stumbled a lot when I tried to generate the moves and the capture cases from the pieces. The En Passant and Castling Rule prove to be hardest for me. The castling was difficult because it was the only move that moves more than one piece, namely the King and the Rook. It also cannot be done into and to escape check, which means I had to check for those cases as well. Moreover, castling can also only be done if it was the King’s and the Rook’s first move. En Passant was particularly tricky because it is a very niche capture rule in chess. Since it can only be done when an opposing pawn moves two square and lands exactly beside a friendly pawn. Furthermore, the pawn is also the only piece whose capture is different than its movement. To make matters worse, if a pawn reaches the opposing final row (Rank in chess), it could then promote to another piece, namely a Bishop, Rook, Knight, or a Queen. I had to check so that those rules don’t break each other.

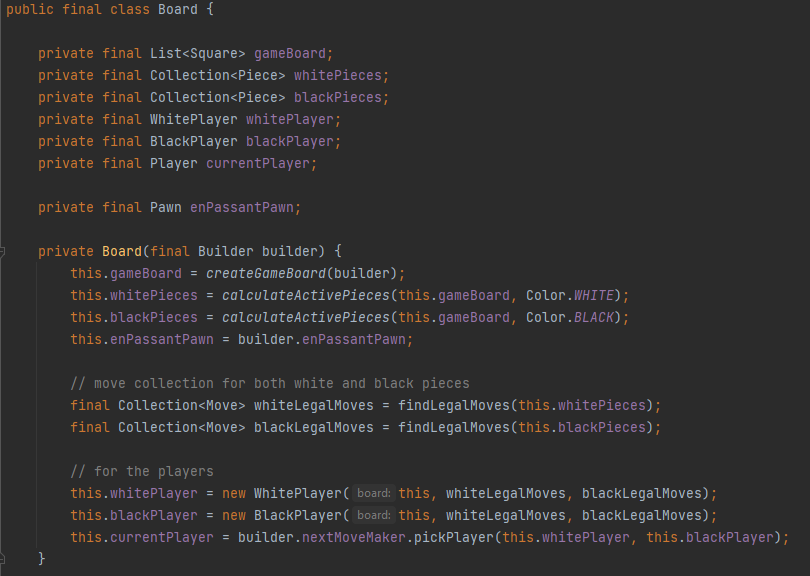
* + Implementing the UI

Because it was my first time using Java Swing and Java AWT, I had a lot of trouble when creating the project. I had to research a lot more than I had expected. The most problematic thing for me is that sometimes the GUI glitches a lot when I tried to implement it for the first time

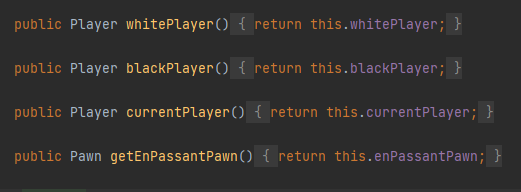
1. **Code Explanation**

The game basically works using a builder. The builder will initialize the chessboard with all of the pieces as a starting position. The pieces movement are unique to each piece, and it is added to a list called legalMoves. Every time a move is made, the board is not mutated. Instead, a new Board with the move executed is created. The GUI will then update the move history and also the taken pieces panel in case a piece was taken

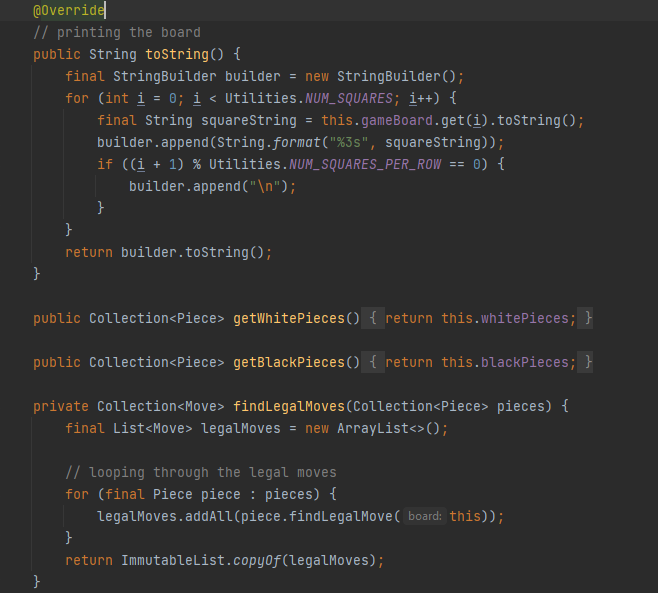
* **Board Package**
  + **Board class**



This is the constructor for the Board class. I used a builder for this chess game. The main reason I used a builder for this is to separate the object construction from its representation. That way I could construct multiple chess pieces or multiple boards step by step and the final step will return the object itself. It saves me time from having to create a new object step by step. The board here is basically used to create the chessboard itself and also initialize the pieces and the player when first initializing the game.



These four are simple getters to get the needed variable in the board class



The first method in this picture is the toString() method. It will print out the created chessboard. I then created two Collections that store a Piece object to store the white and black pieces used in the game. The final method in the picture is used to find the legal moves of the pieces in the collection. It works by looping through the pieces in the collection and then adding each piece’s legal moves into a list. I returned the list as an immutable to prevent accidental interferences



The active pieces are calculated by looping through the game board and checking for all filled square. All filled square contains a piece that is then added to an immutable list of active pieces



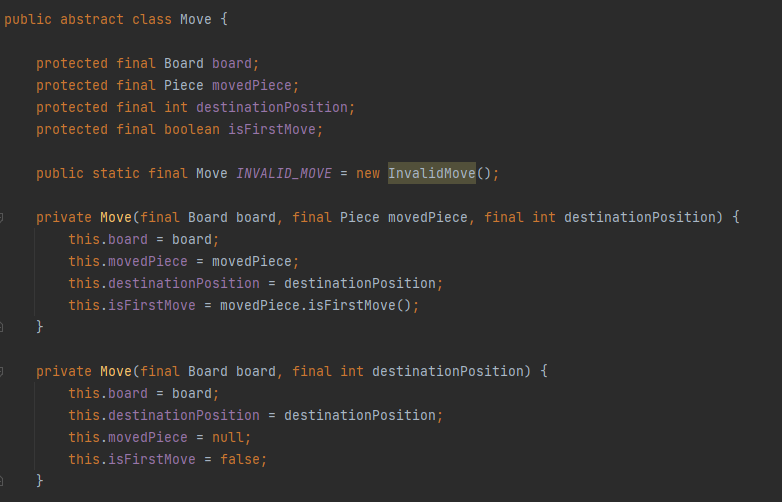


The game board is then created simply by placing all of the pieces in their respective boxes. After all of the pieces are set, the builder will then set the first move to the white player.

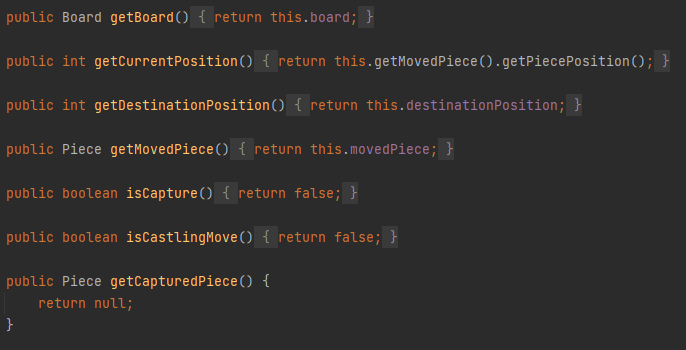
* **Move Package**

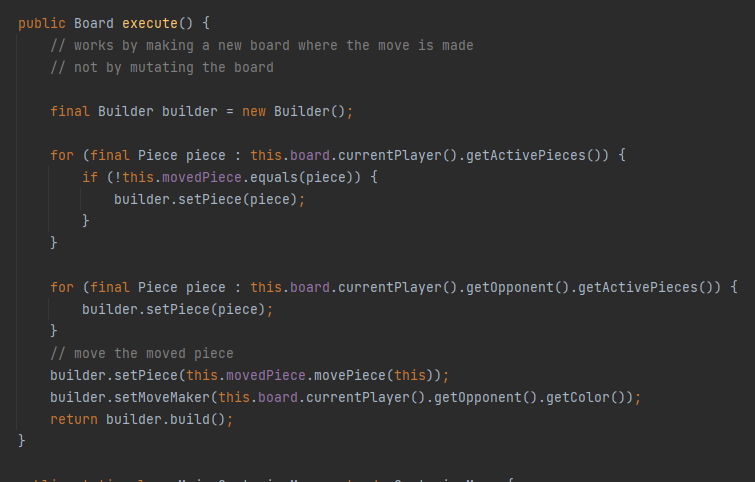
This package handles all of the move that is made by each of the pieces. Recurring methods in this package includes Overriding the toString(), equals(), and also hashCode() methods

1. **Move Class**

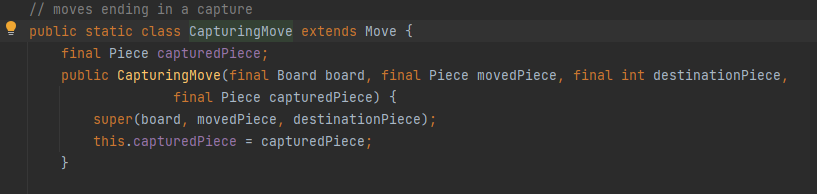


I implemented two constructors for convenience. The constructor above is for the regular move and the one below is meant for invalid or null moves.

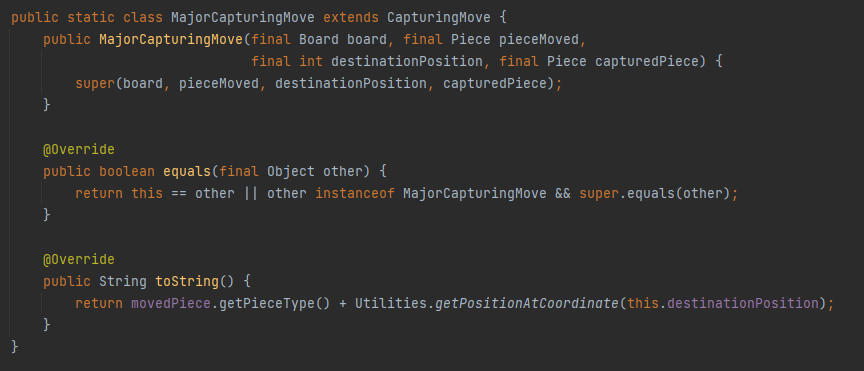
For this class I overrode the equals, hashCode, and the toString method to return an accurate representation of the moves made. The equals method is used so that it will return true if the other object has the same position, destinationPosition, and the same piece type as the object. 

These are the setter and getter methods to get the required variables

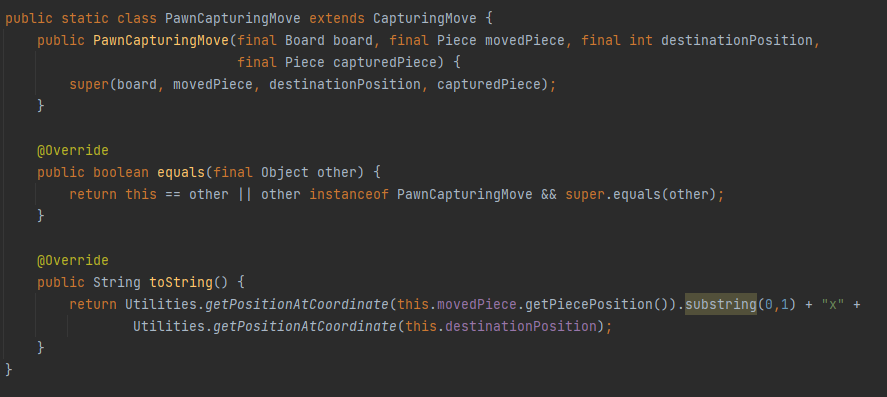
The most important method in the Move class however is the execute method. This method will execute the type of move the player wants. This method serves as a template. This method will later be overridden when the execute method fails to compute for the new move class. One important note is the classes that extends from move in general all implement the equals(), hashCode(), and toString() method



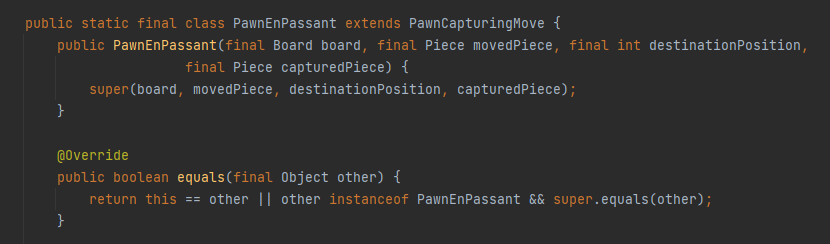
The first move subclass of the move class is the CapturingMove class. This class will handle all of the moves ending in a capture. The constructor follows the same pattern as the move class with an extra parameter, capturedPiece. This is to record the captured pieces when the game is played. This class has two subclasses:



The first class is the MajorCapturingMove class. This class takes the exact same parameters as the capturing move class and is mainly used for executing a capture move using any pieces other than the pawn.

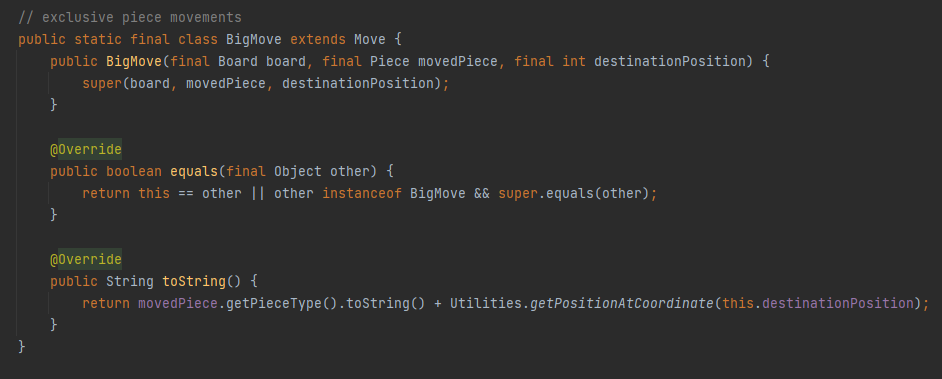


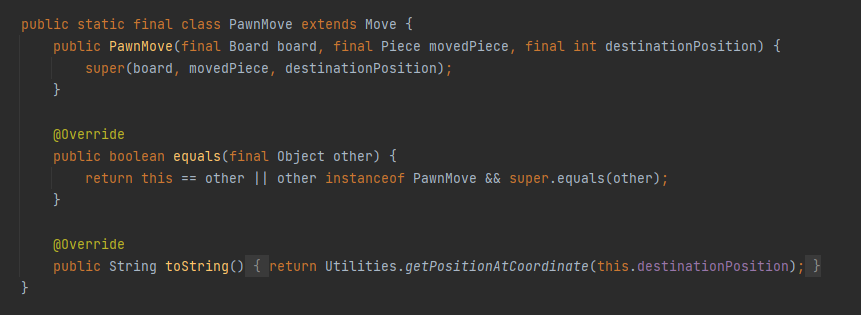
The PawnCapturingMove class is used for anything involving pawn captures. This move also has one subclass, which is for executing En Passant, a special pawn capture rule. Below is the implementation of the class





The PawnCapturingMove class is used for anything involving pawn captures. This move also has one subclass, which is for executing En Passant, a special pawn capture rule. Below is the implementation of the class

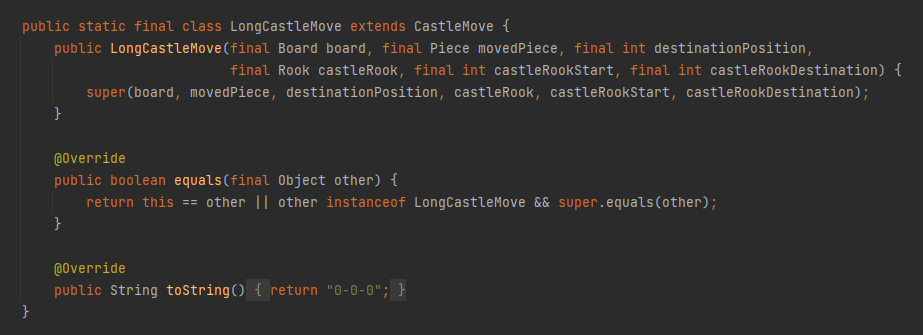
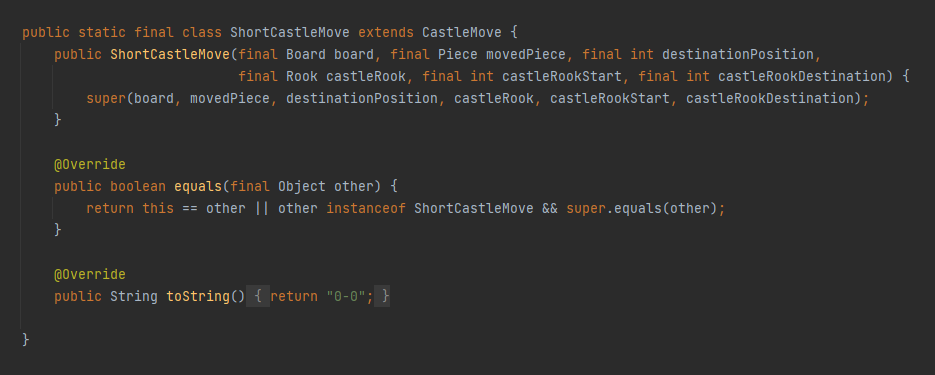
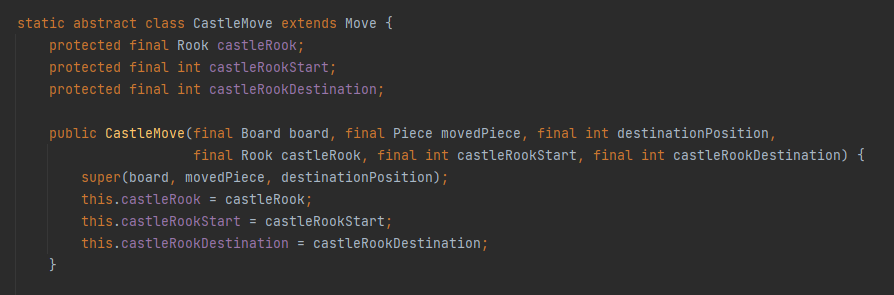
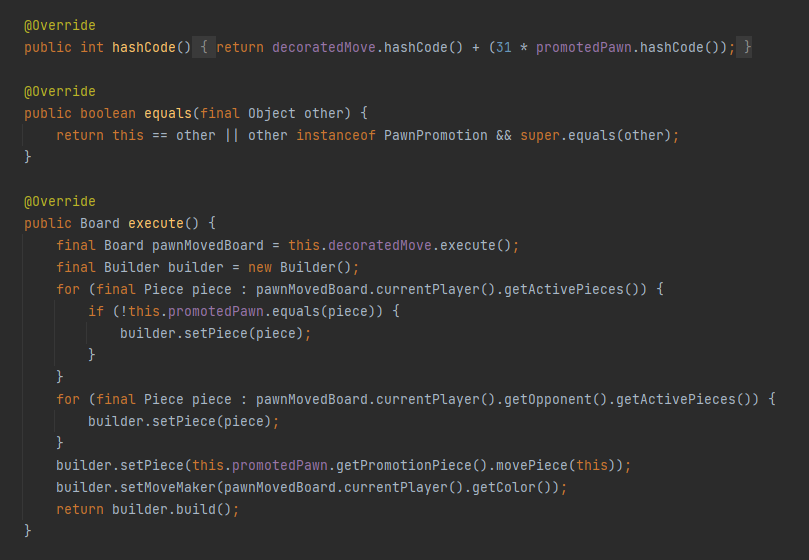
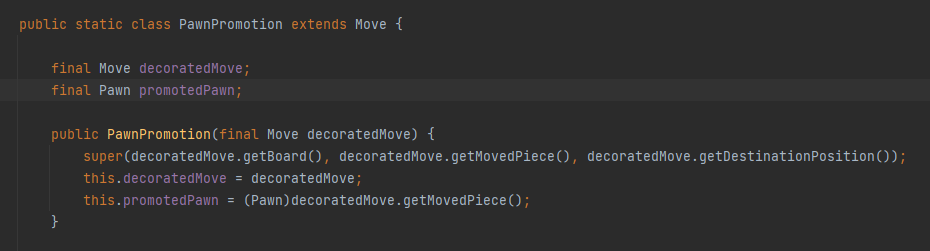




The BigMove class is for calculating the moves of the pieces that does not end in a capture. There is also a pawn variant of this move that records the pawn move

**BELOW ARE CLASSES WITH THE SAME IMPLEMENTATION PATTERN THAT RECORDS DIFFERENT MOVES**

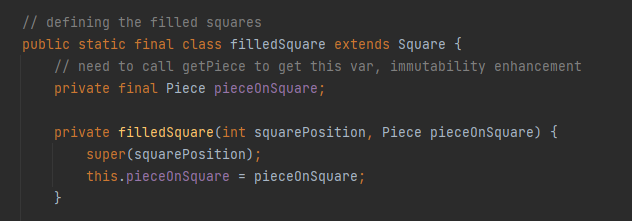
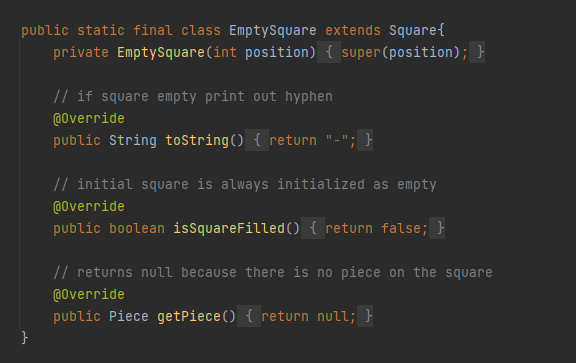
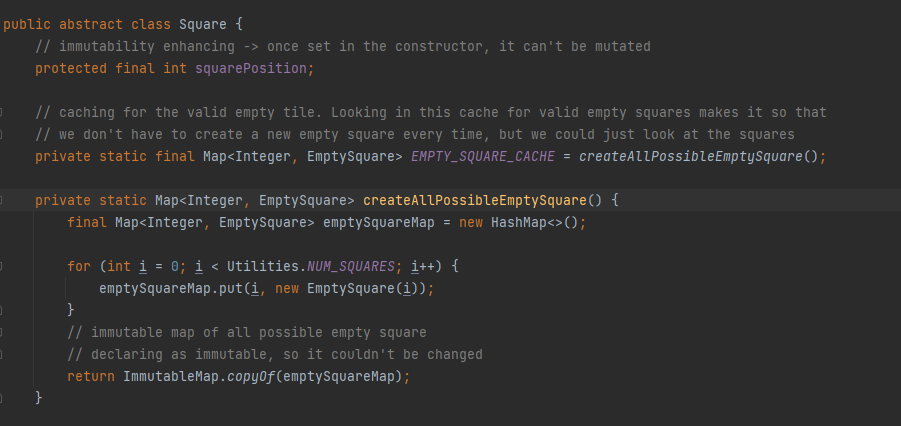
****

****

The last major convenience class in this move class is the MoveFactory class

This class loops through each legal moves currently available on the board and will then create a move corresponding to one of the available legal moves currently on the board.

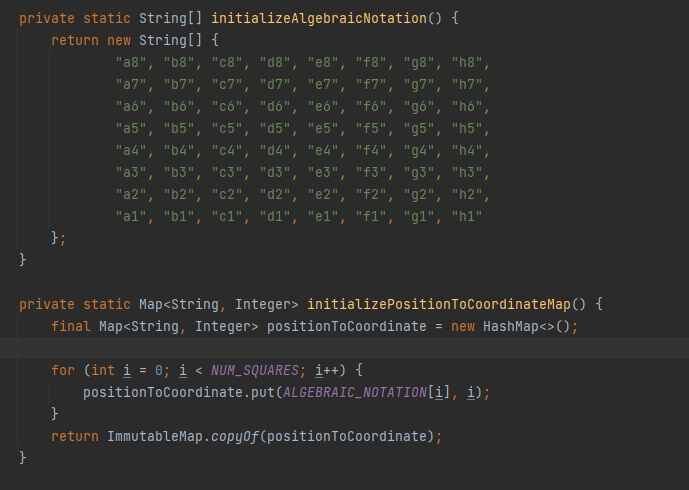
**Square Class**

This class deals with the properties of a square in the chessboard

The square class is implemented as a way to generate the squares that makes up the chessboard. It will create all possible empty square by mapping an empty square to a position on the board. The square class also has a filledSquare method, and it serves as a checker to see whether a particular square is occupied by a piece or not.

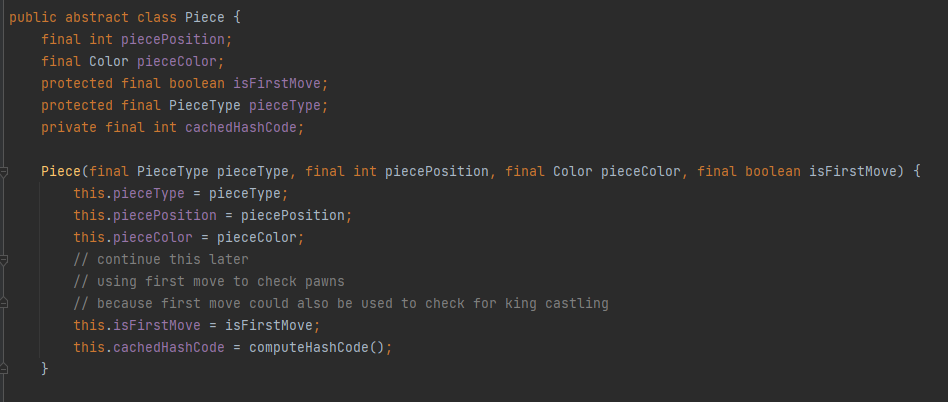
**Utilities Class**

The utilities class is used mainly for initializing special rows and columns that will check with a move algorithm to see whether a particular move will break down or not. The two most important methods in this class are the methods that will initialize the algebraic notation, and initialize position to coordinate map



These two methods work in tandem. The top method initializes the algebraic notation for a chess game while the bottom method maps those position to each of the squares in the board.

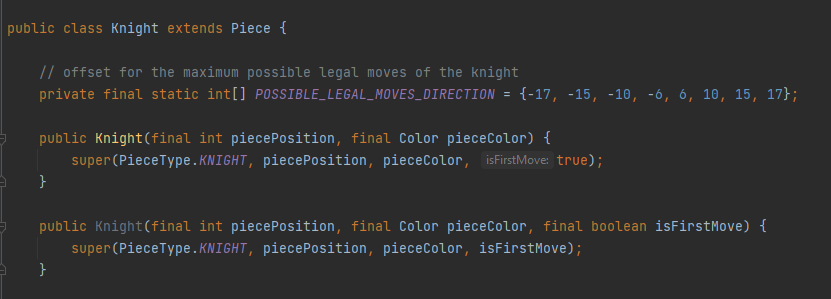
* **Piece Package**

This package contains an abstract piece class which sets the template and will set the template for all of the pieces. Below is the Piece abstract class

This class mostly just sets the attributes for each of the pieces. It contains an abstract move which functions as a legal move calculating algorithm which will be overridden by each piece of the game. It also contains a piece type enum to discern the important pieces from each other and to also to store the value of each piece one another. Another important note to mention is that each piece has two constructor methods, one if it is the first move, and the other one if it is not the first move

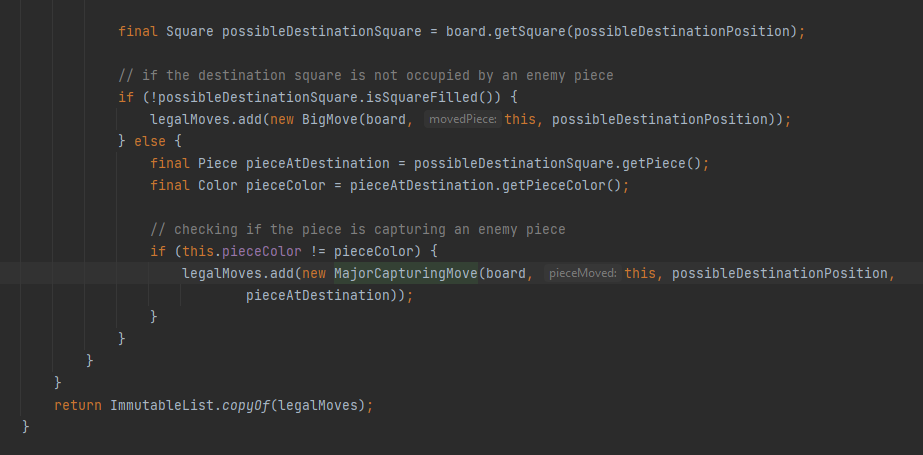
There are 6 types of pieces in the game of chess:

1. The Knight



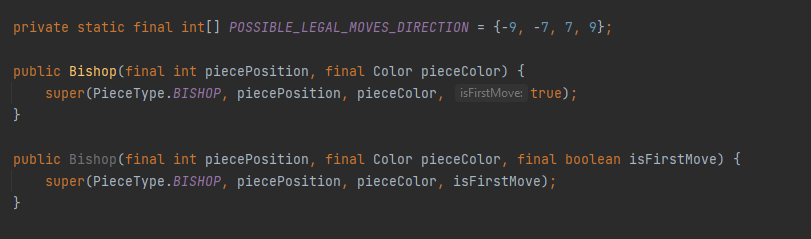
The constructor for the pieces will be the same with the other pieces. It takes the piece position, color, and also the first move (second constructor) of the piece in question. In this class, there is an array of integers which direct the offset of the knight’s movement. The way the offset is calculated for each piece is by assigning each square of the board a number from 0 to 63, from the top-left to the bottom-right. Minus means going back a certain number of integers and plus means going a certain number of integers. This will result in the unique L-move of the Knight.

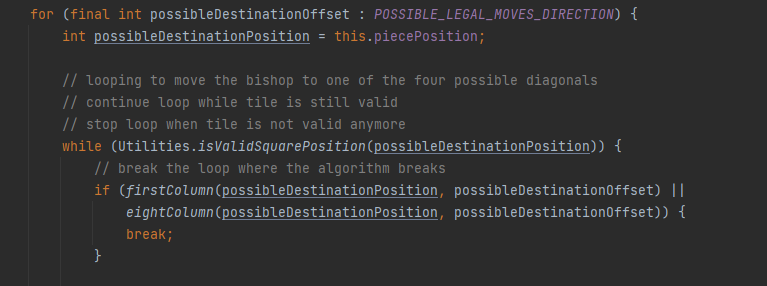
The legal moves of the pieces are implemented as an list of moves which will then be used to determine the moves of all of the pieces in the board. For the knight, the legal move calculation starts by looping through all of the possible legal direction/offset for the knight, and then adding the knight’s position with the offset integer to make it go to a certain square. There is also a checker to continue without adding the offsets in certain columns to prevent the algorithm from breaking



After adding the offsets to the knight’s position, the algorithm then checks whether or not the destination square is occupied by a piece. If it was occupied by a piece, it checks the color of said piece. If the colors are different from one another, we add the move as a major capture move, else we add the move as a big move to the list of legal moves

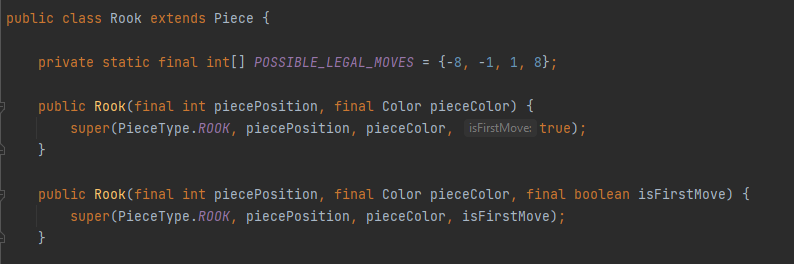
1. The Bishop

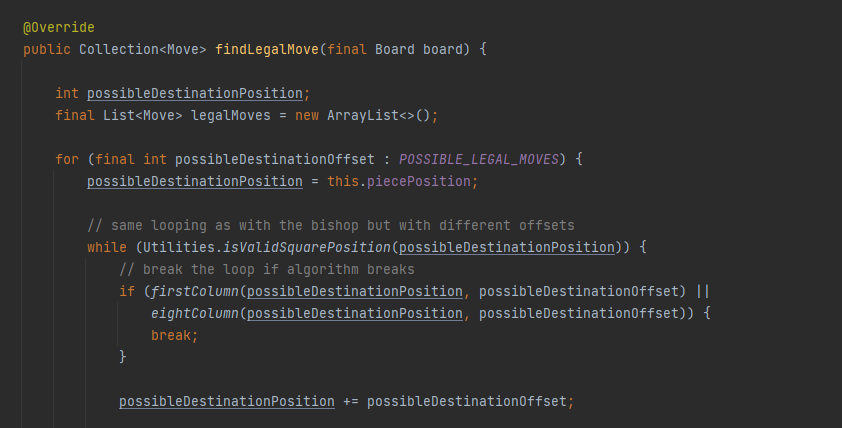


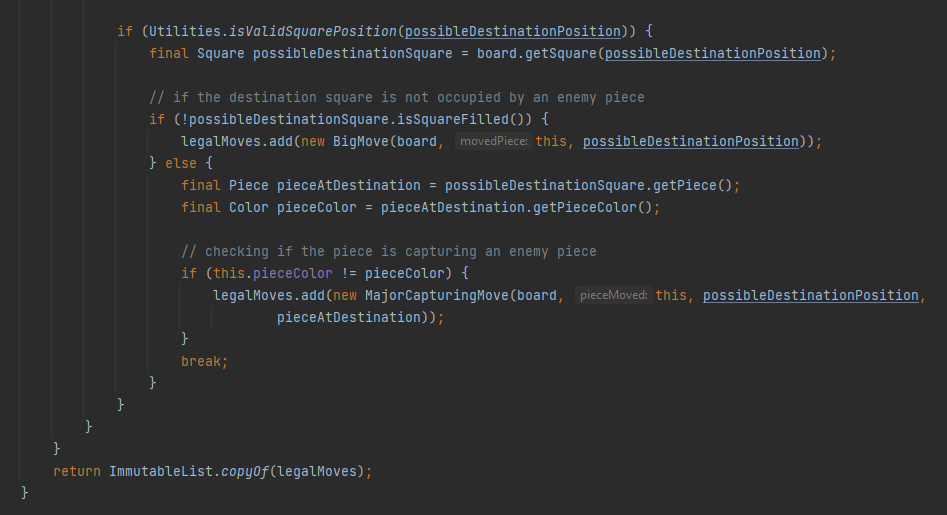
The bishop is a piece that could only move diagonally. The offset of this piece is to go to the four diagonals adjacent to the bishop.

The movement of the bishop works by looping the offsets so that the bishops could move to the desired diagonal. The algorithm checks for cases where the algorithm breaks and then skips them. The rest of the implementation follows the same rule as the knight, where a major capturing move is added to the list when it captures a move, and a big move is added instead when the move does not end up in a capture

1. The Rook

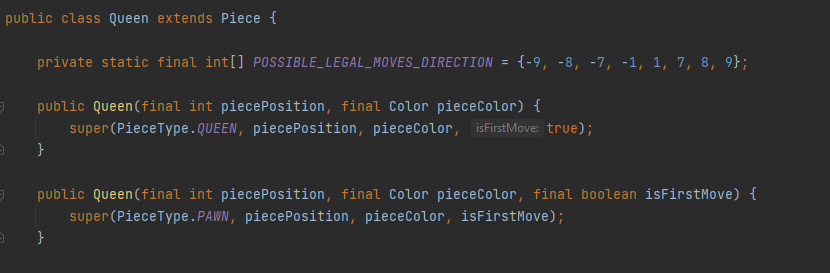


The Rook follows the same design pattern as the bishop. It has four offset position which are the vertical/horizontal squares adjacent to it. 

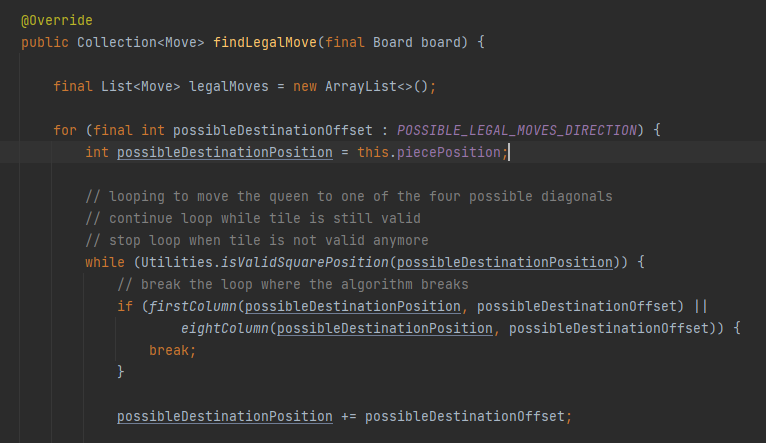
The legal move calculation is exactly the same implementation as the Bishop. It loops through all of the possible offsets and checks for places where the algorithm breaks, which the code will then skip

And then it follows the same design principle as the Bishop. The algorithm loops the movement until the piece reaches the desired square position. Then the algorithm checks if the piece captures an enemy piece and adds the move as a major capturing move if it does, and adds the move as a big move if the move does not end in a capture.

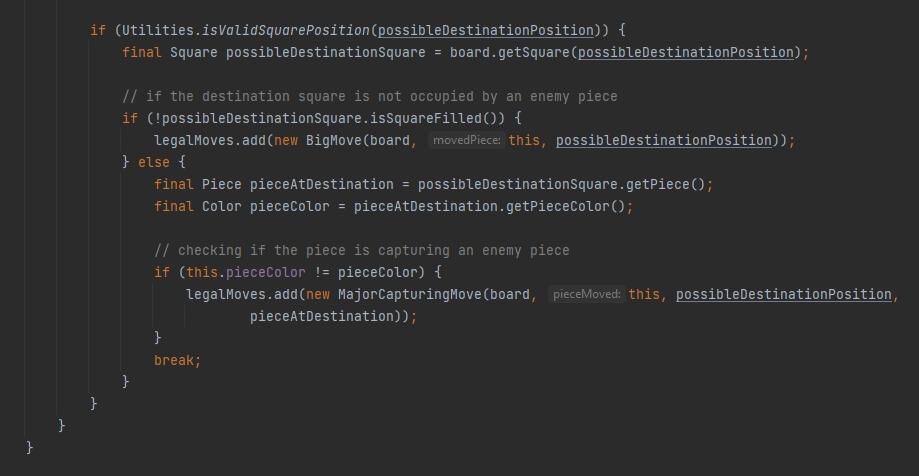
1. The Queen



The Queen is the most powerful piece in chess. It moves like a Rook and a Bishop put together. The offsets for the queen are all of the 8 squares adjacent to the Queen itself.

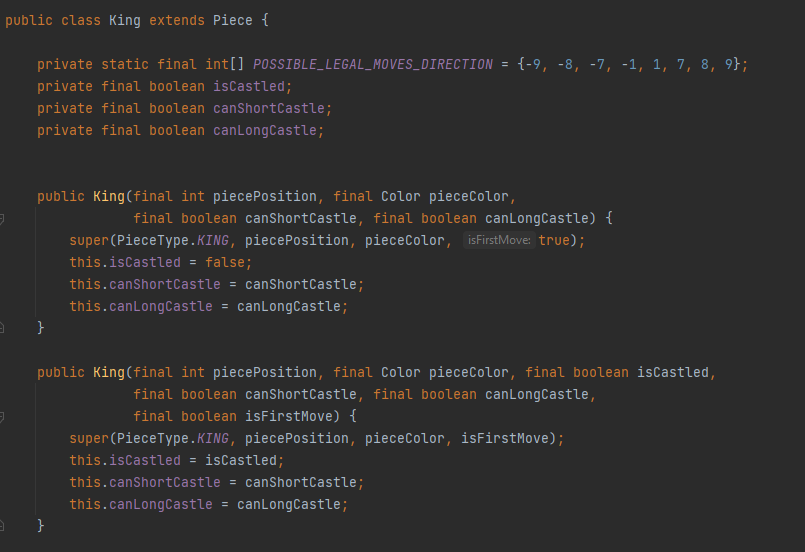


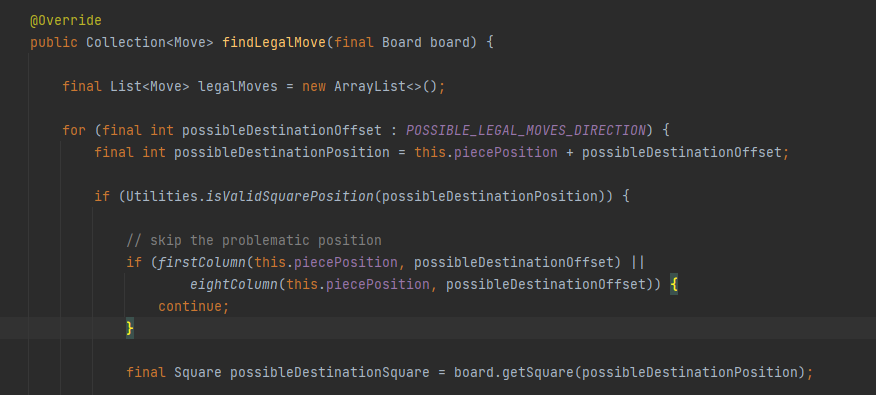
The legal move generation is a combination of the legal move calculation for the Bishop and the Rook. The algorithm loops to check the possible diagonals and adjacent squares and will stop in places in which the algorithm breaks or when the desired square is reached. It will then update the destination square, adding the offsets until the desired squares is reached.

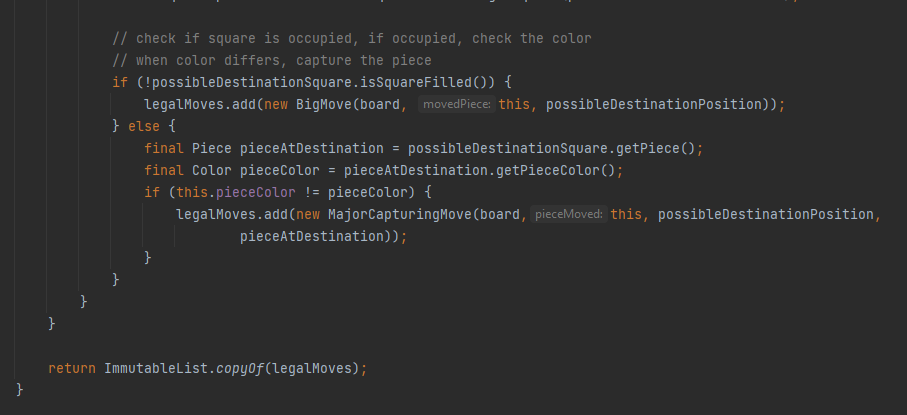


The final part of the legal move calculation is the same as with the other pieces thus far. The algorithm checks the pathing of the pieces. It will then return the move as a major capture move if it captures a piece and it will return the move as a big move, which is then added to the immutable list of legal moves

1. The King

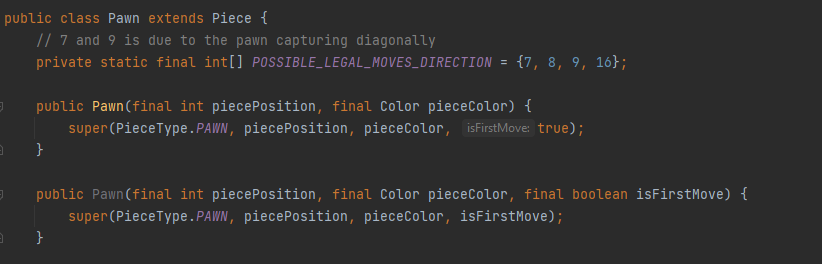


The King is the most important piece in chess. It moves like a Queen that is only limited to one square only, and cannot move to a checked/attacked position. The King also has a special two-piece move with the Rook called Castling. Castling is a move where the King moves two squares to the direction of the Rook, and the Rook jumps over the King. It is also the only two-piece move in chess. The caveat of Castling is that the move can only be done if it was the King’s and the Rook’s first move in that game. Because of that, the King piece has a special constructor that checks whether it is castled, and is able to perform castling or not. Other than that, the King has the exact same offset as the Queen does.

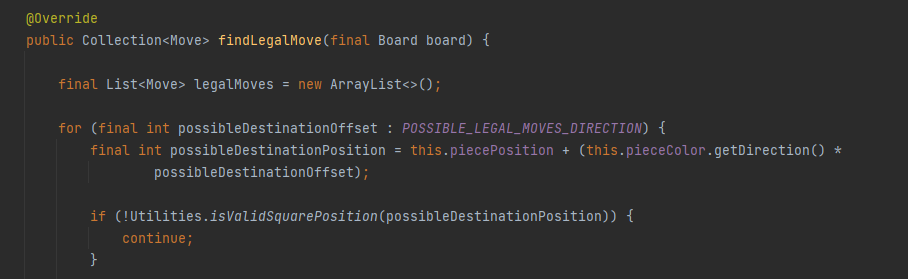
The legal move generation is almost exactly the same with the Queen, the only difference is that the move generation for the King is not looped, due to the King’s ability to only move one square at a time. The destination position of the King is not the pieces position + the destination offset, but rather the offset itself. 

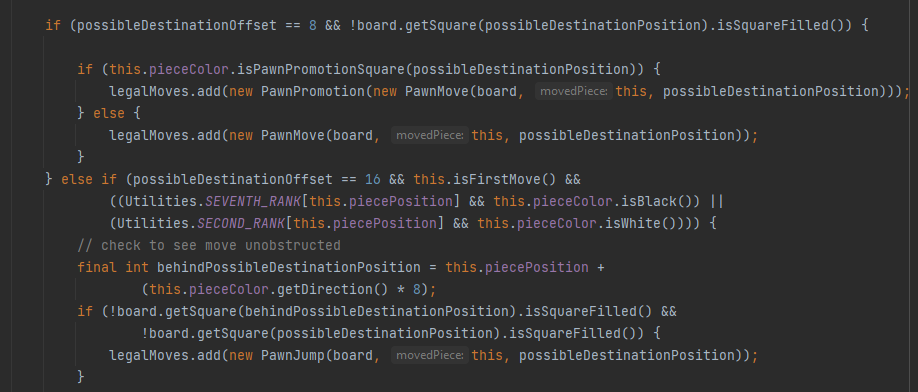
The algorithm then checks for captures and non-captures, which will then return the appropriate move which gets added to the immutable list.

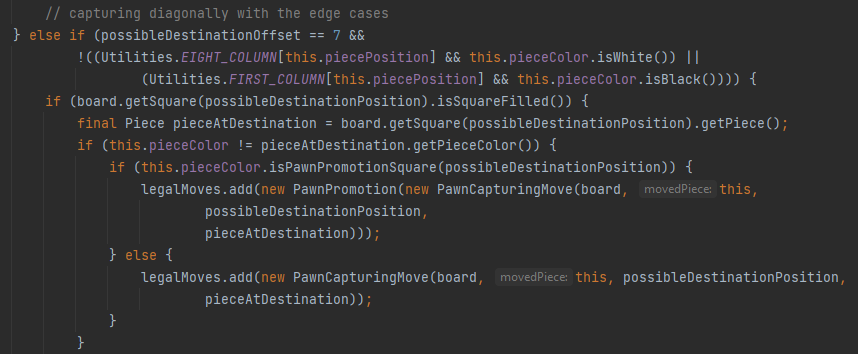
1. The Pawn

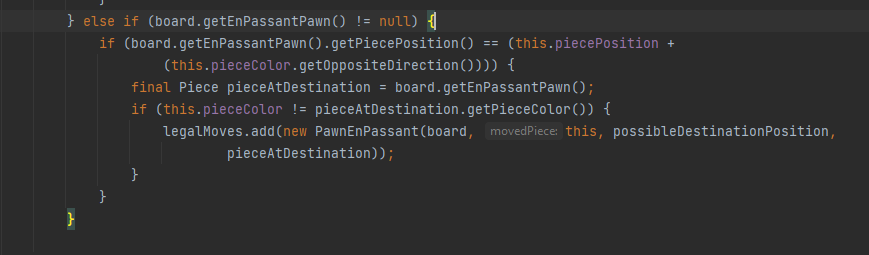


The Pawn is a unique piece in chess due to it being the only piece whose capture move is different to its non-capture move. The pawn captures diagonally, and moves one square to the opposing rank at a time. If it is the first move made by that pawn, the pawn could do a jump, where it moves two squares ahead instead of one. Another uniqueness of the pawn is that the pawn has two special additional move rules. The first one is called En Passant. The En Passant capture happens when an opposing pawn jumps and lands directly to the left or right beside on of our own pawn. Our pawn could then capture that opposing pawn by moving one square diagonally to that pawn’s position had it advanced only one square. The second special attribute of the pawn is promotion. A pawn could promote into any other piece provided that it has reached the eighth rank of their respective positions. 96 % of the time, the Pawn would promote into a Queen because it has the largest value and usability out of all of the pieces. Promotion to any other piece in chess is commonly called underpromoting. In this implementation, unfortunately the payer does not have a choice between the pieces that they are able to promote to and the pawn is automatically promoted into a Queen provided it had reached the eighth rank.

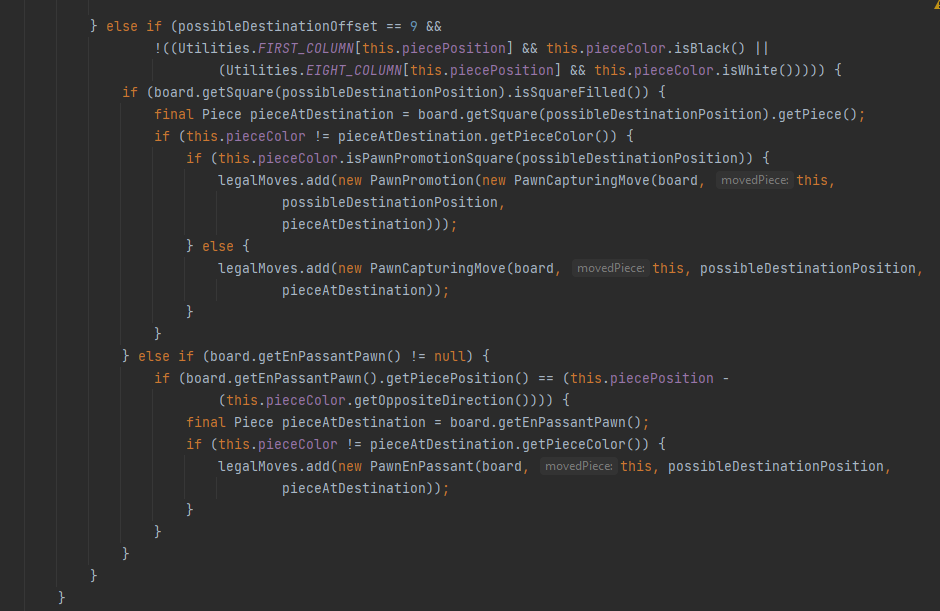


The legal move calculation of the pawn starts just like any other pieces. It works by adding the offset to the original piece position so that it moves to the desired spot.

First, the algorithm checks for the one square ahead move, in which it immediately gets added as a pawn move if it moves to any other rank other than the eighth rank provided that the square it is moving into is not occupied by any other pieces. If it does reach the eighth rank, the added move is regarded as a pawn promotion that accepts that pawn move as its parameter instead. The algorithm then checks for a possible pawn jump. It does so by checking if both the black and white pawn are still in its starting position. If they are is still on their starting position, the algorithm then checks the destination square and the square behind the destination square for any possible occupied square. If both of the squares are not occupied, the new move is added to the list of legal moves as a new pawn jump move.

The capture for the pawn is calculated differently than the other pieces due to the nature of it. The algorithm will first check if the offset is the two diagonals adjacent to the pawn in question and will then check where the algorithm breaks and not take that into account. Then it checks whether those squares have a piece in them or not. The algorithm will then move the pawn to the designated square and add the move as a pawn capture if the destination square is not in the eighth rank, and will promote the pawn to a queen and add a pawn promotion move along with a pawn capture move if it does. 

This else statement checks whether an En Passant move is available. The algorithm checks the board for an En Passant pawn (a pawn that had just done a jump) that is horizontally adjacent to an opposing piece. If that is the case, the move is added as an En Passant capture to the legal move list.

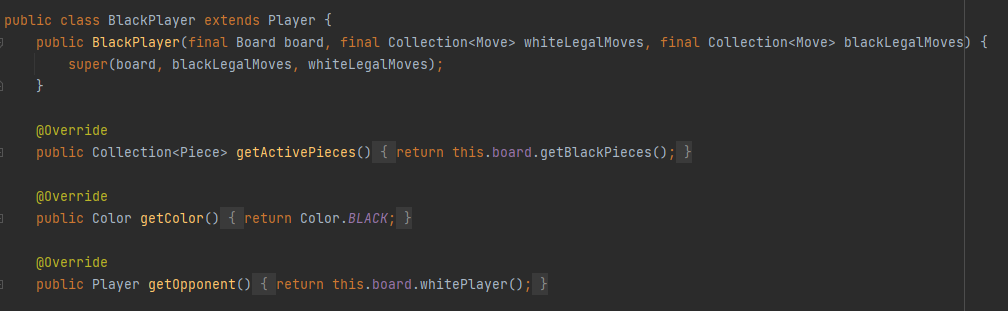


This is the same implementation for the other diagonal of the pawn.

* **Player Package**

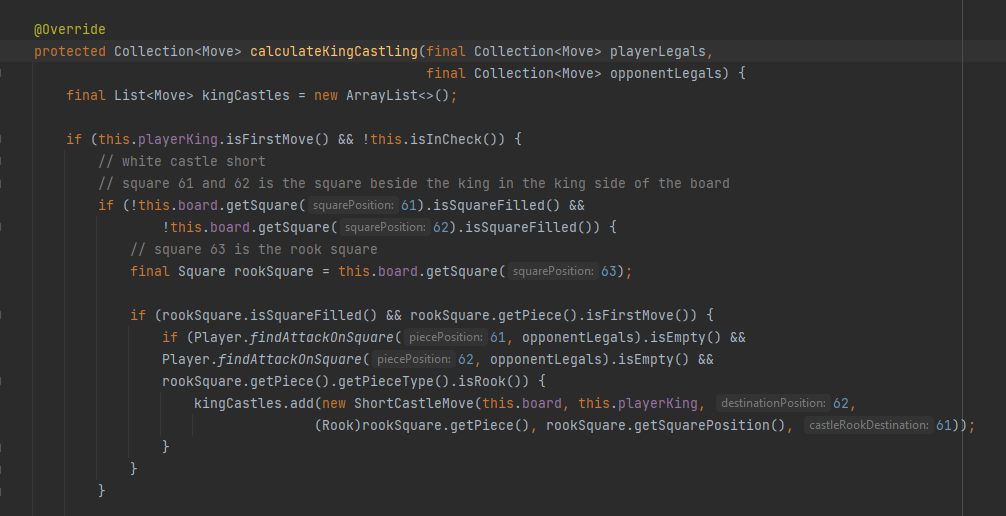
This package contains 5 classes. All in all, this package handles the interaction between the white player and the black player, the move status, and also the move transition between the players.

* + **Black Player and White Player class**



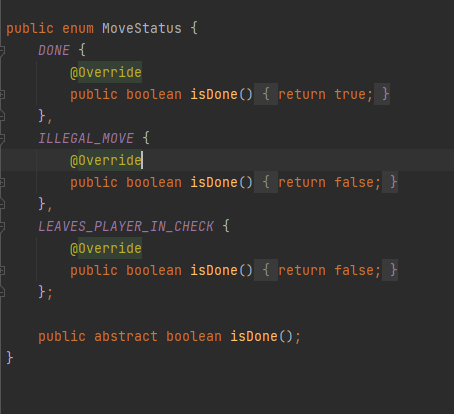


The black player and white player class has the same exact implementations and methods. The constructor of the class takes in the board, and the legal moves for both white and black. The class has some getter methods that returns the color appropriate variable for each class.



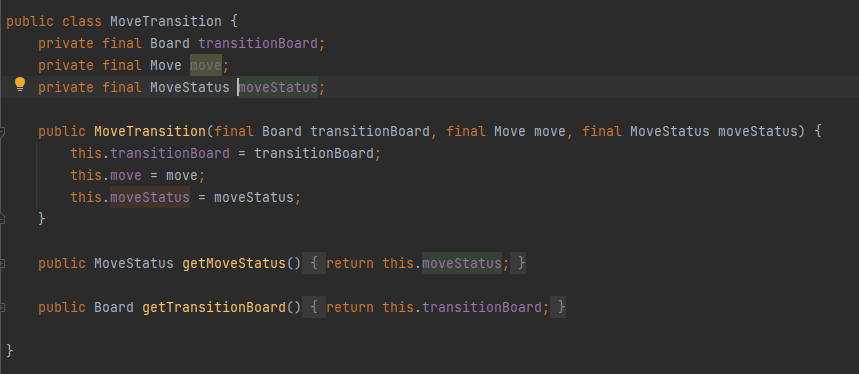
This method calculates the king castling for the players. The way it works is by initializing a new array list of king castles. Then the algorithm checks if the King has moved before, and if it is in check. If both of the condition is satisfied, then further check if the square between the rook and the king is empty or not. If it is, then perform castling and add the castling move to the list. The top method is for generating short castling, and the bottom method is for generating long castling. The method then simply returns an immutable list of the castle moves generated.

* + **MoveStatus enum**



This enum is used to get the status of the move that is being done. If the move is successfully executed, we return true. If the move leaves the player in check, then it will return false. So does the illegal move. It returns false because the move fails to execute.

* + **Move Transition**



The MoveTransition class is used to handle the transition from one board to the new board. The constructor takes in the board to transition, the move being made, and the move status of the move.

1. **Project and Video Link**

Github link: https://github.com/EdwardMatthew/Chessbot.git

Video link:

1. **References**

https://www.youtube.com/watch?v=h8fSdSUKttk&list=PLOJzCFLZdG4zk5d-1\_ah2B4kqZSeIlWtt

https://www.chessprogramming.org/Main\_Page