**Date Started:** 11/22/2021

**Note**: I’m taking forever trying to pick and start a programming project in Java so I’m just going to make myself pick one for now. Because, I think it would be good for me to start flexing my at home programming muscles outside of work.

<http://www.sakkpalota.hu/index.php/en/chess/rules>

<https://www.chess.com/terms/chess-knight>

Chess Piece Pictures: <https://opengameart.org/content/pixel-chess>

**Object Types**

Pieces:

(and their unique, peculiar rules. Some rules might be common. Some rules depend on 2 pieces and board position like the rook and horseman maneuver):

1. Pawns
2. Rooks
3. Bishops
4. Horsemen
5. King
6. Queen

Players:

* 1 on 1.

The Board:

1. Size is 8x8. There are 64 squares total. **DONE**
2. Black and white squares. Opposite colors are adjacent, and same colors are diagonal to each other. Corner diagonals are the same color; half black, half white, symmetrically on the board. **DONE**
3. Each game piece has its own unique starting positions. **DONE**
4. (Optional) Squares could glow special colors with effects based on available moves the player can make based on the game pieces.
5. Grid pane style. Will need to stack pictures on top. Maybe gridpane with stackpanes on top? **DONE**

User (s) Input:

* Probably mouse clicks. Or, could be keyboard… **DONE**
* For movement for starters could be a simple click. If the position player is trying to move to is valid based on the game piece’s rules then move is allowed. **DONE**

Game Manager:

* Handles determining which team goes next. **DONE**
* Determines if game has ended. Either by a win, loss, or player exits the game.
* Keeps count of game pieces lost maybe.
* Creates new pieces when Pawns reach opposite side.

Extras:

1. Move Timer? :
2. Piece Animation cutscenes? :
3. Game AI.

Unit Tests:

1. Test gameboard boundaries.
2. Maybe Test movement patterns of pieces.
3. Test game board pieces are removed when destroyed.

Class Diagrams:

Game Piece:

Methods:

1. Move.
2. Attack.
3. (Passive) Be destroyed.
4. Check for available moves. (Might have an obstacle.)

Possible Phases of Development:

1. Chess piece rules/behavior with unit tests. Mostly **DONE**
2. The board. **DONE**
3. Click input. **DONE**
4. Game manager to determine player turns. **DONE**

That might be enough for a rough-draft first. Get that done, and then you can try other stuff.

Pseudo Code:

Movement: 2D array[y][x] Probably an ArrayList so we can get index of object for convenience.

1. Diagonal –
   1. UpLeft [+1][-1]
   2. UpRight [+1][+1]
   3. DownLeft [-1][-1]
   4. DownRight [-1][+1]
2. Adjacent movement
   1. Up [+1][+0]
   2. Down [+1][+0]
   3. Left [+0][-1]
   4. Right [+0][+1]
3. canMove() method

Basic Movement rules:

1. Diagonal only
2. Adjacent only
3. Adjacent only, but with directional change (Knight)
4. Can or cannot jump piece.
5. Diagonal and adjacent (King)
6. Distance traveled
   1. Some limited number, edge of the board, or by other pieces.
   2. Some limited only by board edges, or by other pieces.
   3. Some can jump pieces.

Note: Some Movement shortcuts maybe could be performed by gameboard tile color checks before calculation.

Maybe precalculated a full move before changing the board model. If you can do it after being commanded then update the board model.

Possible chess interfaces:

Pawn.move(y, x)

Date: 12/10/2021

**Status**: I think I’ve implemented and tested most of the low-level movement logic, and the piece-specific basic moves which rely on movement.

However, I can still later add the special moves like

1. the rook and king swap,
2. the pawn swap-out for any requested piece,
3. may that weird pawn capture move I had never heard of (move name is French).

**What’s left?**

1. I need to set the board game correctly. **DONE**
2. I need to implement turn-based logic for 2 players. **DONE**
3. I need to get images for each type of piece, and somehow bind them to the pieces. **DONE**
4. I would like to refactor any movement (or other) methods to classes that are logically related instead of having functions in unrelated classes. \*\***Basically Done**\*\*
5. \*\*\*I need logic to determine if the game has ended, aka check and checkmate. \*\*\*
6. Implement Stalemate and draw behavior as well.
7. I need to start implementing the barebones gui interface. **DONE**
8. (optionally) could implement a turn timer.
9. (optionally) make the game visually look appealing.
10. (future feature) I could implement some opponent AI so you could play against the computer.
11. (future feature) implement a save game feature.

SCRATCH NOTES:

Clean up for the Model-View integration rough draft code:

Note: I might have tried to do much in this one commit.

**Main Components:**

1. ChessPieceView - extends ImageView to store a reference to the respective ChessPiece. This is needed so that I can easily identify which ChessPiece the user selects in the view.

**Areas to clean up:**

1. Messy ChessPieceView class relationship with ChessPiece that allows for an ImageView to be accessed inside a ChessPiece, but a ChessPieceView allows access to a ChessPiece. And, this is probably unnecessary and not completely used.
   1. Is this truly necessary? And on a sidenote, is this ever a useful feature?
   2. **ANSWER: useful for now.**
2. Confusing relationship between the BoardView and BoardModel.
   1. Note: Confusing relationship in general between my View and Model. Maybe I could implement an observer relationship?
   2. Maybe I can try to keep the view code outside of the model classes, or I can create integrated control methods that operate on both simultaneously.
   3. **DONE. Basically did A and B at once by updating view to everything model did after a move.**
3. Cleanly destroying a game piece has not been implemented yet. **DONE, handled by observer update.**
4. Inspect the number of places you have to do (Y,X) coordinate conversions. Try to simplify and reduce this. **DONE**
5. Simplify the EventHandler code if possible.
6. Plan UI code that helps a user know what’s going on behind the scenes. Or, have very good console print code until you get to it. **DONE, but could still improve.**

**Bugs**:

#1

**Problem**: Pawns are destroying pieces that are right in front of them. I think this is either due to a bug in the ‘path’ code, or duplication of destroy behavior implemented in the canMove() pawn function and the path code. Or, I might not even be aware of what piece of code handles the removal of other pieces. **DONE**

**Investigation**:

1. ChessBoard.putPieceHere() appears to be what removes destroyed pieces from the model.
2. ChessBoard.isPieceBlockingPath() appears to work for everything except Pawn since pawns are one of the pieces (The other is the knight) that cannot attack head on.

**Solution**:

Pawn’s movement logic rules are a bit different than other pieces that can move long distances linearly. It is the only linear-moving piece that cannot attack enemy pieces head on. However, it can still rely on the ChessBoard.isPieceBlockingPath() for detecting friendly pieces that are blocking its path.

**Note**: maybe I should start doing these kinds of projects in GitHub or other management system to better keep track of bugs.

**Algorithms**:

**Check and Checkmate**:

Note: this must include the behavior to prevent the player from putting themselves into check!

So, this requires logic for both friendly and enemy turns, for each teams’ respective Kings.

Options:

1. Allow the player to just attack the King. (not a part of Chess rules)
2. Somehow calculate if a king is in Check and then later in checkmate.
   1. Check for any linear attacks (Queen, Bishop, Rook, and King).
   2. Check for any knights within a set radius of the king. (Could use Knight’s canMove() logic, but for other pieces be aware that some rely on piece state, such as Pawns)
   3. Check for any Pawn diagonal attacks?

Note: King has multiple ways to break check. Could attack, move away, or move a piece to block the line of attack.

Note2: need to read up on how Check and checkmate is suppose to work in the game.

Maybe could use a state machine here?...

canMove

x x x i x x

x x x i x b

x x x x x x

2, 6 queen can attack 7, 3 king

7 – 2, 3 – 6 = (5, -3) = diff