COEN 275: Object Oriented Analysis, Design, and Programming
Winter 2017
Individual Assignment 2
Due: March 5th, @11:59 PM

Assignment Overview:
The objective of this assignment is to create a playable chessboard. If you are unfamiliar with the rules of chess, Appendix A contains the rules relevant to this assignment. The chessboard will restrict the user to legal chess moves, with a few exceptions explained below, and provide users a "Hint Button" to suggest the optimum move based off of a provided set of guidelines. This Assignment contains two parts:

First, create a way to calculate the allowable moves for each chess piece. Then change your chessboard so it will highlight the allowed moves whenever a piece is clicked. You can use the chessboard you created in Assignment 1 as a starting point for this assignment.

Second, using the piece's movement code, figure out the ideal chess move based off of a provided criteria, and attach it to a button handler on the board.

Part 1: Playable Chessboard Requirements
Your program must be able to figure out when a user has interacted with the board, or a chess piece. You may accomplish this in any way you choose, including by creating the chessboard with buttons, as suggested before, and capturing button click events.

Clicking on a chess piece will highlight the appropriate moves for that chess piece. Clicking on any of the highlighted squares will move the piece to that square. Clicking on any non-highlighted square will deselect the piece, and return the board to a state with no highlighted squares.

Appropriate moves for a piece should be based off the rules of chess, shown in Appendix A. You do not have to consider the rules of Check or Checkmate in your code. In other words, if your king is in Check, you may move still move other pieces. Capturing a king does not end the program. You do not have to implement code to take care of Castling or Pawn Promotion.

Your program must keep track of which player's turn it is, and it must enforce it. No player is allowed to move twice in a row. The player controlling the white pieces always moves first in a chess game.

Part 2: Chess Hint

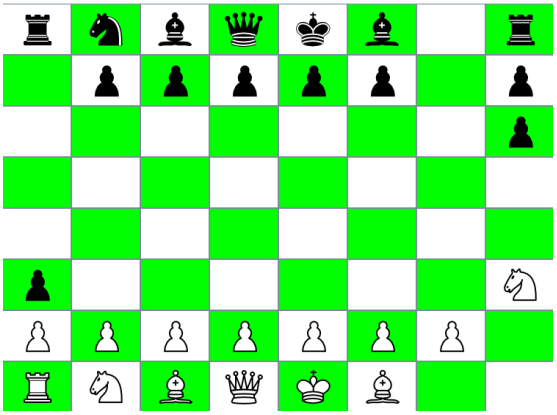
Once you have your movement code prepared, you need to add a hint button. Clicking on the hint button will cause your program to calculate the optimum move, and highlight this move on the chessboard. To choose the optimum move, use the following priority:

1) Highest priority goes to moves that capture an opposing piece
2) If two, more, or no moves capture an opposing piece, then, out of those moves whichever move who ends with a piece furthest to the right, with respect to the player. (ie Furthest Right for White, Left for Black)
3) If the first two conditions do not eliminate a tie, then from those moves choose whichever move ends with a piece furthest away from the player. (ie Furthest Up for white, Down for Black)
4) If the previous conditions still do not eliminate a tie, then from those moves choose the piece that started furthest left with respect to the player (ie Furthest Left for White, Right for Black)
5) If the previous conditions still do not eliminate a tie, then from those moves, choose the piece that started most back from the player. (Furthest Down for White, Up for Black)

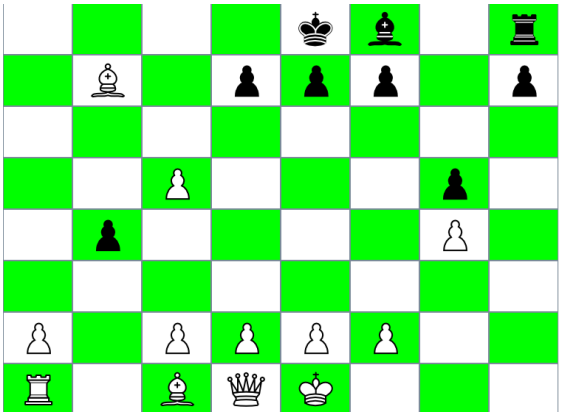
The ideal move should be highlighted and presented to the user. As before, clicking on the move moves the piece there. Clicking anywhere else cancels the highlighting.

To help you test your code, these are the positions of the pieces after following the moves suggestions from the hints button 10 and 30 times:

10 Hints



30 Hints



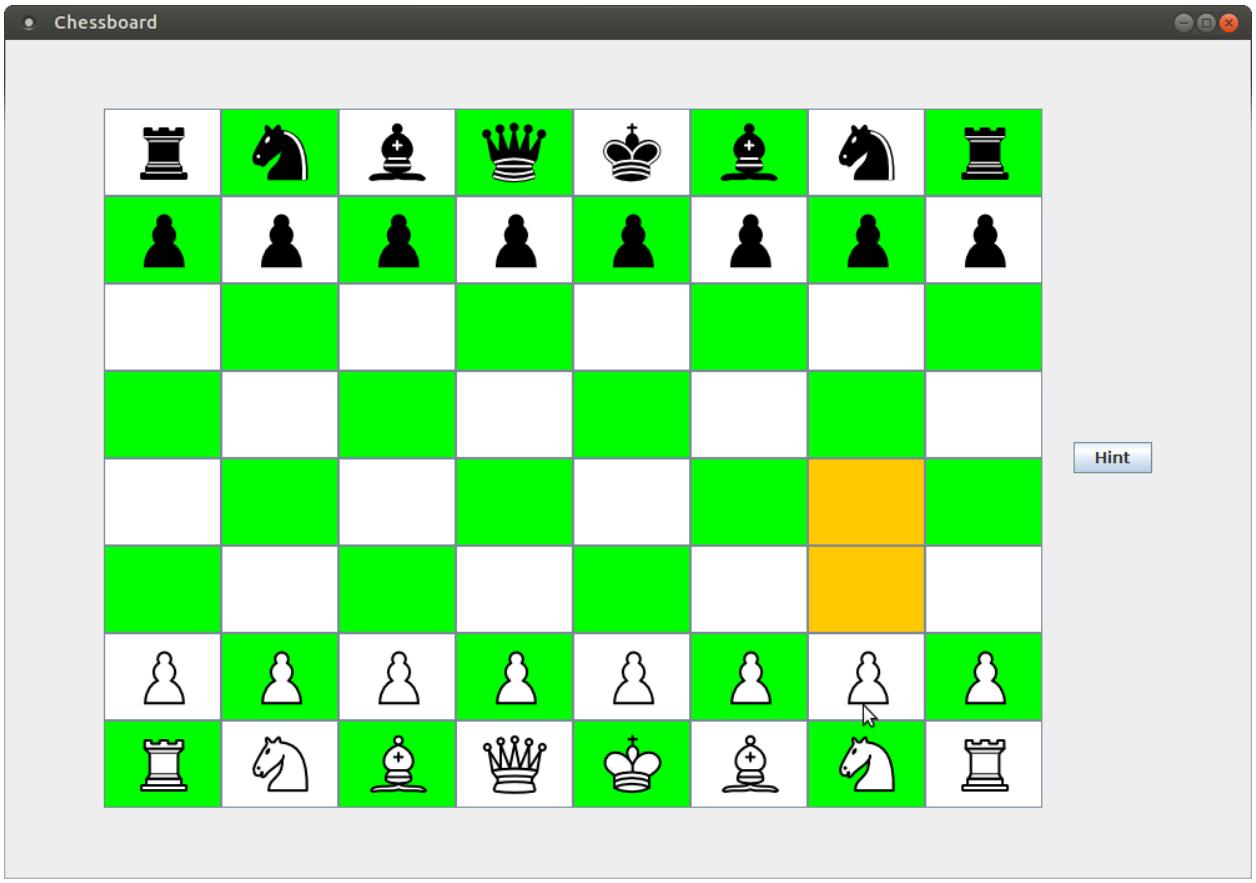
To calculate the move, your program must create one thread for each piece allowed to move that turn. All of these threads will calculate all the possible moves for their piece using the code implemented in part 1. Each thread will add their results to a Java Container of your choice. This container must be shared by all threads, and access to it must be controlled by a lock. You may forgo the lock IF you can prove your code does not need it using *Oracle Documentation*. *Stackoverflow threads, or similar unofficial sources do not count*. The

container must be sorted to find the optimum move. Your reply should be placed in your README file, and clearly marked.

GUI Minimum Requirements:

The GUI must contain at minimum a chessboard that registers clicks, and a hint button. As with before, coloring, style, etc is up to you, as long as the required elements are easily identifiable. The highlight color should stand out from your board.

At minimum your GUI, after the Pawn at G2 is clicked, should look like the following:



Code Requirements:

Your code must adhere to proper Java Style. You are welcome to use a Java Style you are comfortable with, as long as you are ***consistent*** in your code. If you do not have a Java Style, a good starting point is the Google Java Style Guide. Ample commenting is welcome and encouraged. As a general rule of thumb, someone reading your code should be able to determine the function of any method within 30 seconds of reading it.

Submission Guidelines:

Your submission must contain the following files:

- Java Source Files
- Combined Source File
- README.txt
- Any other files required to run your program not listed above

Your submission should not contain files or folders created by your IDE. Only the contents of the src/ directory are required.

Combined Source Files:

You must include with your submission a file that contains the contents of all your source files concatenated together. The file should be named firstInitialLastName.txt

README.txt File:

You must also include with your submission a README file. This file is used to help debug setup related issues. The README file should use the following format:

Name: <Your first and last name>
JDK Used: <JDK Provider> <Version>
IDE Used: <IDE and Version>
Main File: <Filename of file that should be run>
Other Instructions: <Optional special instructions specific to your program>

All files EXCEPT the Combined Source file should be compressed into a .zip file. This file should be named firstInitialLastName.zip. There will be two Assignments created in Camino, one for your .zip file, one for your Combined Source file. Submit both the .zip file and the Combined Source File to their respective DropBoxes.

Grade Breakdown:

Functionality	60%
GUI	15%
Style	25%

Miscellaneous / Hints:

- You should focus on implementing part 1 before you begin implementing part 2.
- In order to sort an object in a Java container, you need to implement a comparator. Using a Lambda function is also acceptable.
- Try to reuse code as much as possibly by combining classes that share commonalities under a single parent class, interface, or abstract class.
- Separating your code into smaller classes that each implement one chess piece will help you keep your code organized, and easier to read.

Appendix A: Chess Basics

A turn in chess consists of one piece moving from their current location to another location. At least one piece must move, and no more than one piece may move. Once one player moves, their turn is over, and the opposing player must move.

No piece is allowed to travel over another piece, with the exception of a Knight. No piece is allowed to occupy the same location as another piece. If a piece of the same color is in the location in which you would like to move to, you may not move there. If however a piece of a different color is located on the square you would like to go to, you may capture that piece. Capturing a piece, for our purposes means remove that piece from the board, and have your piece occupy the space it originally held.

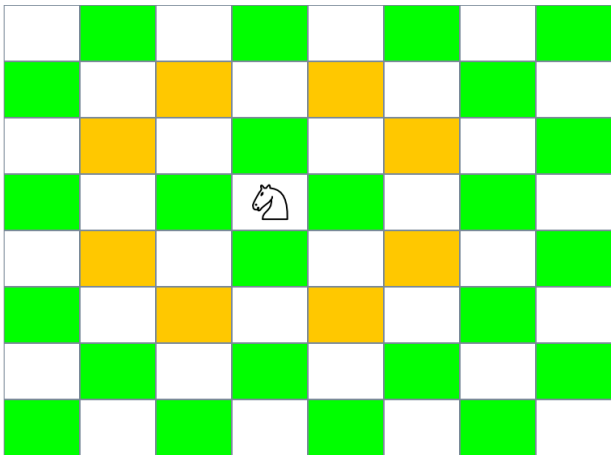
Each chess piece has their own unique pattern of movement:

Knight

A knight moves in an L shaped pattern. It moves 2 spaces in one direction, and 1 space in an orthogonal direction.

If we assume the ChessBoard is an XY Coordinate plane, and the Knight starts at position (X,Y), it can move to the following locations:
(X + 2, Y + 1), (X + 2, Y - 1), (X - 2, Y + 1), (X - 2, Y - 1),
(X + 1, Y + 2), (X + 1, Y - 2), (X - 1, Y + 2), (X - 1, Y - 2)

Graphically:



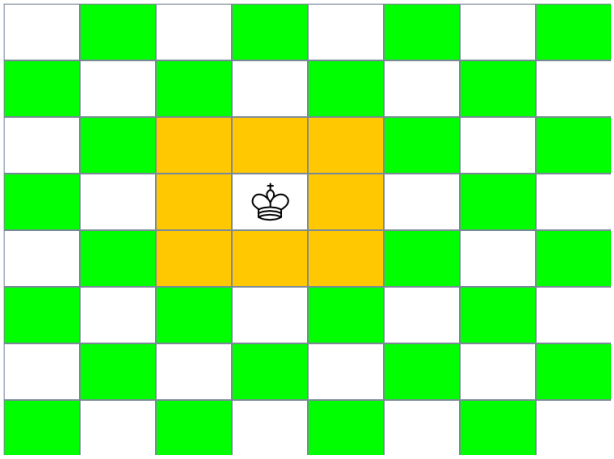
The knight is allowed to jump over other pieces to get to it's location. Pieces that are jumped over are not captured. Only pieces that are on the space the knight lands on are captured.

King

The King can only move 1 square, but can move in any direction.

Using the XY Coordinate notation introduced in the Knight section, the King is allowed to move to these combinations of squares:
(X + 1, Y + 1), (X + 1, Y - 1), (X - 1, Y + 1), (X - 1, Y - 1),
(X + 1, Y), (X - 1, Y), (X, Y + 1), (X, Y - 1)

Graphically:

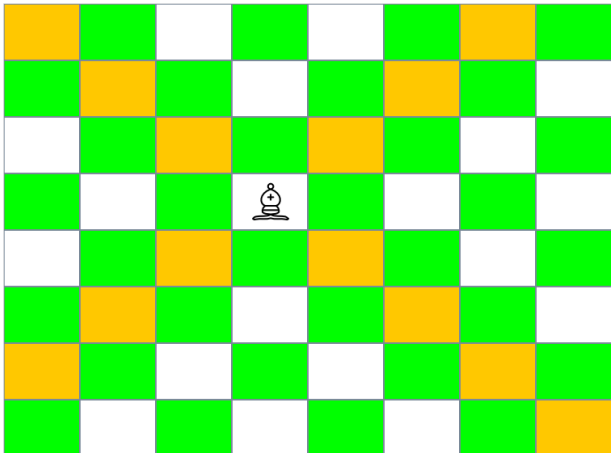


Bishop

The Bishop can only move diagonally. If a Bishop started on a white square, then any move it makes must result in it staying on a white square, and vice versa. It can move any number of spaces.

In XY Coordinates:
(X + n, Y + n), (X + n, Y - n), (X - n, Y + n), (X - n, Y - n), where n is any positive integer ≠ 0

Graphically:



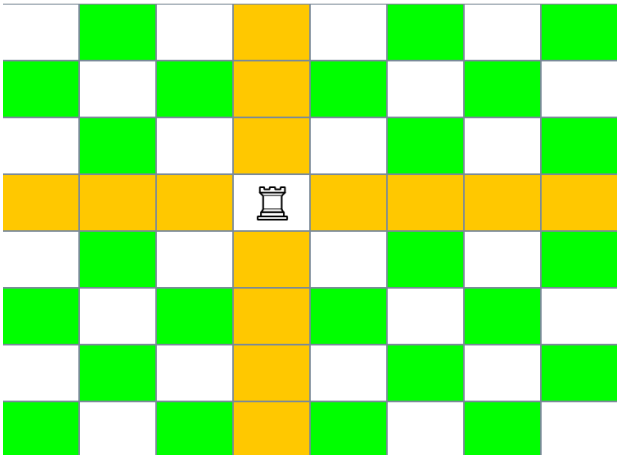
Rook

The rook can move up, down, left, or right

In XY Coordinates:

(X + n, Y), (X - n, Y), (X, Y + n), (X, Y - n), where n is any positive integer ≠ 0

Graphically:



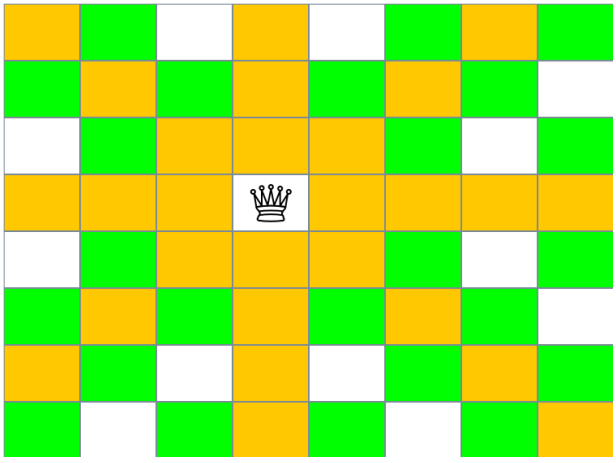
Queen

The queen's movement pattern is the same as the King, but it can move any number of spaces. It is also identical to the movement patterns of the Bishop and the Rook combined

In XY Coordinates:

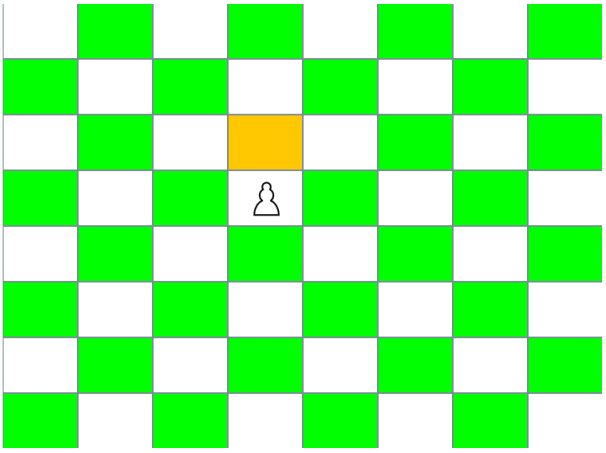
(X + n, Y + n), (X + n, Y - n), (X - n, Y + n), (X - n, Y - n),

(X + n, Y), (X - n, Y), (X, Y + n), (X, Y - n), where n is any positive integer ≠ 0

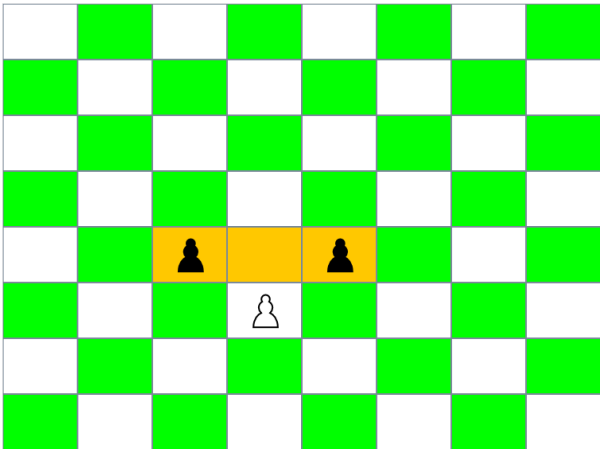


Pawn

A Pawn has the most complicated movement pattern. Normally the pawn can only move one square up.



However, if there are opposing pieces one square on top, and one square to the left or right, the pawn may move to capture those pieces.



If the pawn has not moved yet, then it may move up two squares instead of one.