CSCI 3901 Assignment 1

Due date: 11:59pm Friday, January 26, 2024 in Brightspace

Problem 1

Goal

Get practice in decomposing a problem, creating a design for a program, and implementing and testing a program. Practice basic Java programming.

Background

Chess is a board game in which to players alternate in moving one of their pieces no the board to eventually capture a special piece from the opponent called the "king". A piece is captured when one of an opponent's playing pieces can move to the same square as the king.

There are several kinds of board pieces, each of which has different permissible movements on the board and different movements to "capture" another board piece. The pieces are

- Pawns
- Knights
- Bishops
- Rooks
- Queen
- King

Information on how each piece moves can be found at https://www.chess.com/terms/chess-pieces

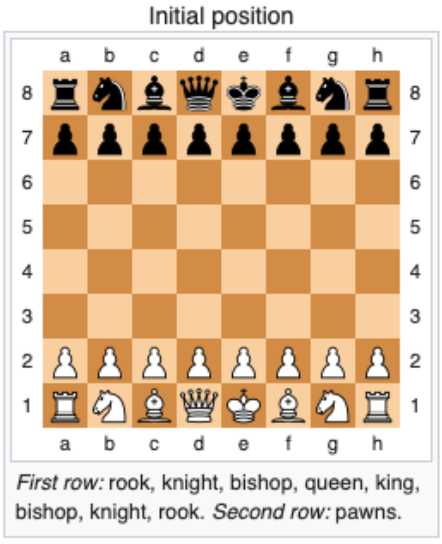

Figure 1 Standard starting chess configuration (image from https://en.wikipedia.org/wiki/Chess, January 11, 2024)

Players are identified by the colour of their playing pieces, notably "white" and "black" and, by tradition, the player with the white pieces makes the first move.

For this assignment, we want to have someone be able to follow a game being played or to be able to see what might happen with different rectangular board size or different playing piece configurations. **We are not creating something to play the game of chess nor to suggest moves.**

Specifically, we will be given a starting board configuration; the regular board and starting position is shown in Figure 1, although this assignment will let us start with different board sizes and configurations (as if we were testing new rules or watching a game part-way into play). We are then given a sequence of moves of pieces on the board will answer questions about the sequence of moves like

- Did all the moves follow the movement or capture rules for the pieces?

- Is a piece, notably a king piece, in a position where the opponent can now capture it (known as being in check)?
- Does a particular playing piece have any valid move on the board?
- What does the playing board look like after the sequence of moves is done?
- Which pieces (and in what order) have been captured by one of the players?

Again, this assignment only looks at following the play of the game. You don't make any decisions about the game.

For those who know something about chess, we will **not** be dealing with special chess moves[1] like:

- Castling
- Pawn promotion
- En-passant capture
- Pawn first move can advance two squares (to simplify the coding problem)

Problem

Write a Java class, called "FollowChess" that holds the configuration of a chess board and provides methods to answer questions about the current board configuration or to effect changes to the board (if the changes are valid).

We will refer to board locations using a letter and number combination, as in Figure 1. Rows are numbered and columns are lettered. Letters go from left-to-right on the board as seen by the player with the white pieces and begin with column 'a'. Rows are numbered starting at 1 and the bottom row, from where the player with white pieces plays, is row 1.

At all times, the game has the white pieces advancing from the bottom to the top of the board and has the black pieces advancing from the top to the bottom of the board. That movement is reflected in Figure 1.

When inputting or outputting a board configuration, follow the following conventions:

- A board square with no piece will be a period '.'
- A board square with a letter represents a piece in the square:
    - P for pawn
    - N for knight
    - B for bishop
    - R for rook
    - Q for queen
    - K for king
- White pieces will be lower case letters. Black pieces will be upper case letters.

You will also submit a small "main" method in a class called "A1" that will demonstrate a minimal use of the Java class.

[1] Move descriptions at https://www.chess.com/terms/special-chess-moves

FollowChess

The FollowChess class must have the following methods:

Constructor FollowChess( )
Initializes an instance of the object

boolean loadBoard(BufferedReader boardStream)
The boardStream contains the description of a board. Each line of the file represents one row. The first line represents the top row of the board, from where the player with black pieces plays. The contents of the row use the piece notation that is described before these method descriptions.

A valid playing board must be a rectangle (not necessarily 8x8) and have exactly one king piece for each player. There can be zero or more instances of all other pieces (eg. 3 queens or 5 knights or 1 rook or no bishops).

For simplicity, you may assume that there will be no spaces between board spaces in the file, nor will there be spaces before or after the characters that describe the board row. Leading or trailing blank lines should be ignored.

Return true if the board configuration has been loaded and represents a valid board. Return false otherwise and leave the objet with no board loaded.

boolean printBoard(PrintWriter outstream)
Print the board that is currently stored in the object to the PrintWriter object that is provided as a parameter. The format for printing the board is expected to match the input specification for loadBoard(). The output should contain no blank lines and no additional spaces, whether before a row, after a row, or within a row. Every line ends with a carriage return (\n character).

Return true if a board was printed to the outstream parameter. Return false if the content sent to outstream should not be trusted as a board or if an error condition was detected.

boolean applyMoveSequence(BufferedReader moveStream)
The moveStream parameter contains a sequence of piece moves to execute, with one move per line. A piece move is given as four space-separated elements:

- Starting column (letter)
- Starting row (integer)
- Ending column (letter)
- Ending row (integer)

This move designation doesn't follow the standard Chess on how to represent moves, but is chosen for a simpler implementation.

The order of the moves in the moveStream parameter is the order in which the moves should be executed on the board.

Only valid moves should be made. A move is valid if the piece at the given starting position can move or capture, in a single step, to the ending position without having to move through another pieces (except possibly to capture at the end position). The knight piece jumping over other pieces in its "L" shaped motion does not count as moving through that piece.

If all moves are valid then the result of applying this sequence of moves should be the new board configuration in the object and the method returns true. If any move is invalid then the board configuration in the object should not change and the method returns false.

Recall that play must alternate between players. The first play after loading a board must by by the player with the white pieces.

List<Character> captureOrder(int player)

As we apply move sequences to pieces, players may capture pieces of their opponent. When captureOrder() is called, you provide the list of all pieces captured by a particular player, listed in the order of capture. The list contains the single-letter representation of the piece, and maintains the upper-case / lower-case designation of the colour of the pieces.

The "player" parameter indicates whether you want the list of pieces that the player with the white pieces has captured (player = 0) or the list of pieces that the player with the black pieces has captured (player = 1).

Return null in the case of an error.

boolean inCheck(int player)

A player is "in check" if that player's king piece can be captured by the very next move of their opponent.

Method inCheck() returns true if the given player is in check and returns false if the player is not in check or if there is some error situation.

The "player" parameter indicates whether you want to check the white king (player = 0) or the blank king (player = 1).

boolean pieceCanMove(String boardPosition)

The method returns true if the chess piece located at "boardPosition" has at least one valid move available in the current board configuration.

A valid move is one in which the piece can

- Apply one motion step to end at an unoccupied board position or

- Apply one capture step to end on a board position that is occupied by an opponent's chess piece.

The boardPosition parameter identifies the location of the piece to move. The string has two space-separated values: the first is the column letter of the board position and the second is the row number of the starting board position.

Aside: The inCheck() and pieceCanMove() methods are the start towards a more complex task, which is outside this assignment. That next task would be to identify if the game is ended because of a checkmate situation. A checkmate happens when one player's king is in check, the king cannot move, and no other chess piece of the player can be moved to break the check, either by blocking the capture path to the king or by capturing the opponent's chess piece that has the king in check. This latter step, of seeing if one of a player's other pieces can break the check, is beyond the scope of this assignment. It's within your ability to do, but just doesn't add more toward the objective of the assignment.

A1

The A1 class has the minimal code to show that you can load a board and apply some piece moves to the board configuration. The program will ask the user for two file names from the keyboard. The first file will contain the board configuration. The second file contains a sequence of moves to apply to the board. Given these two inputs, your program will print the final board configuration to the screen.

Example

Sample input for loadBoard, that matches the configuration in Figure 1:

```
RNBQKBNR
PPPPPPPP
........
........
........
........
pppppppp
rnbqkbnr
```

Sample input for applyMoveSequence (noting our constraint that pawns cannot start with a two-cell step):

```
e 2 e 3
g 8 f 6
d 2 d 3
e 7 e 6
e 3 e 4
e 6 e 5
```

Output from printBoard() after making the sequence of moves above:

```
RNBQKB.R
PPPP.PPP
.....N..
....P...
....p...
...p....
Ppp..ppp
rnbqkbnr
```

Assumptions

You may assume that

- All board configuration strings that we provide will have the letter column precede the row number and that there will be at least one space between the letter column and the row number.
- Any column letter will be provided in lower case.
- There will not be more than 26 columns.

Constraints

- We are **not** using the following chess moves:
    - Pawns being able to move 2 squares as their first move
    - Pawn promotion
    - En-passant captures
    - Castling
- If in doubt for testing, I will be running your program on timberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- Make a plan for your solution before starting to code. Write that plan and the parts as part of your external documentation.
- You can create more than one class for your solution. Group information as it makes the most sense.
- Pick a few aspects of the solution to implement at a time rather than try to write code that solves everything in one pass of coding. Use the marking scheme to guide what parts could look like and/or where to prioritize your efforts.
- This assignment does not use exceptions to simplify the coding for anyone who is just learning Java. You may _not_ allow your public methods of FollowChess throw exceptions.
- There are no marks for efficiency in this assignment. The focus is on getting something working, not on being the most clever person in the class. If you find this assignment easy then feel free to add the challenge of efficiency to your design and implementation.
- Do not handle exceptions by printing a stack trace or printing to the screen. Neither of these approaches are appropriate in large-scale software so get used to not relying on them now.

Marking scheme

- Documentation (internal and external) – 3 marks
- Program organization, clarity, modularity, style – 4 marks
- Load and print a board – 6 marks
- Apply a move sequence to the board (or recognize it is invalid) – 6 marks
- Report on piece capture orders – 2 marks
- Report on the state of the board (in check, pieces can move) – 6 marks
- Demonstrate the implementation of FollowChess with your main() method – 2 marks

The majority of the functional testing will be done with an automated script or JUnit test cases

Test cases for FollowChess class

The lists below appear to have a lot of cases, and they're listed as trying for each piece type. That's because testing at this stage doesn't know if you're writing separate code to move each piece type or if you have a common move algorithm.

loadBoard

Input validation

- Null parameter
- Empty board stream

Boundary cases

- Read in a 1x1 board
- Read in a 1x2 board with two kings
- Read in a 2x1 board with two kings
- Read in a board with one of the kings missing
- Read in a board with two kings of the same colour and one of the other colour

Control flow

- Read in a larger board with just the two kings and empty spaces
- Read in a larger board with the two kings and one copy of each pieces, both black and white
- Read in a regular chess board
- Read in a board with more instances of pieces than is configured in a regular chess board, like having three rooks
- Read in a board that is bigger than a standard chess board
- Read in a board with all pieces on the edge of the board
- Read in a board with all pieces away from the edge of the board
- Read a board with leading blank lines
- Read a board with trailing blank lines
- Read a board with a blank line in the middle of the board (shouldn't crash)
- Read a board where the second line is shorter than the first line
- Read a board where the second line is longer than the first line

Data flow

- Call loadBoard twice in a row with different boards

printBoard

Input validation

- Null output stream

Boundary cases

- Print a 1x2 board
- Print a 2x1 board

- Print a board with pieces at the edge of the board
- Print a board with pieces away from the edge of the board

Control flow
- Print a board that just has the two kings
- Print a board with a copy of every type of playing piece, both white and black
- Print a board that is larger than a regular chess board

Data flow
- Print a board before calling loadBoard
- Print a board after having successfully applied moves
- Print a board after applyMoveSequence failed

applyMoveSequence

Input vaidation
- Null stream of moves
- Empty stream of moves

Boundary cases
- Apply a single move to the board
- Apply a move that starts on the top edge of the board
- Apply a move that starts on the bottom edge of the board
- Apply a move that ends on the right edge of the board
- Apply a move that ends on the left edge of the board
- Apply a move that ends on the top edge of the board
- Apply a move that ends on the bottom edge of the board
- Apply a move that ends on the right edge of the board
- Apply a move that ends on the left edge of the board
- Apply a move that ends above the top edge of the board
- Apply a move that ends below the bottom edge of the board
- Apply a move that ends to the left of the left edge of the board
- Apply a move that ends to the right of the right edge of the board

Control flow
- Read a move sequence with fewer than 4 entries in a line
- Read a move sequence with more than 4 entries in a line
- There is a blank line among the move sequence lines
- For each piece type, try to move the piece one cell in each possible move direction for the piece when nothing is stopping the move
- For each piece type, try to move the piece one cell in each possible move direction for the piece when there is an opposing pieces blocking the move
- For each piece type try to move the piece one cell I each possible move direction for the piece when there is a blocking pieces to the move of the same colour as the moving piece

- For each piece type, try to do a capture move of the pieces by one cell move in each possible capture direction for the piece
- For each piece that can move an arbitrary number of board positions in one direction (rook, bishop, queen)
    - Move the piece all the way to the edge of the board (top, bottom, left, right) when the path is clear
    - Move the piece all the way to the edge of the board when the path is blocked by an opposing piece
    - Move the piece all the way to the edge of the board when the path is blocked by a piece of the same colour
    - Do a capture of a piece all the way at the edge of the board (top, bottom, left, right) when the path is clear
- For each piece type, try to move the piece in a direction that is not allowed for the piece, using a pattern that is valid for another piece (eg. a bishop move for a rook)
    - Include trying to move a white pawn down the board and a black pawn up the board
- For each piece type, try to capture another piece in a direction that is not allowed for the piece
- For each piece that can move just one pattern at a time (pawn, king, knight)
    - Move the piece two cells along a valid move direction when the path is clear
- Designate a move where the starting cell has no piece in it
- Ask to move two white pieces in a row
- Ask to move two black pieces in a row

Data flow

- Do a move sequence where the first call to make a move isn't for a white piece
- Do two calls to applyMoveSequence where
    - the first call ends with a white move and the second call begins with a white move
    - the first call ends with a black move and the second call begins with a white move
    - the first call ends with a white move and the second call begins with a white move
    - the first call ends with a black move and the second call begins with a black move
- Make moves before a board is loaded

captureOrder

Input validation

- Ask for invalid player number
    - -1
    - 2

Boundary cases

- Call for a player that has captured no pieces

Control flow
- Call for a player that has captured one piece
- Call for a player who has captured multiple distinct piece types
- Call for a player who has captured multiple pieces, in sequence, of the same type
- Call for a player who has captured multiple pieces of the same type intermixed with capturing pieces of other types
- Call when each piece type has been captured
- Call for the player with white pieces having captured pieces
- Call for the player with black pieces having captured pieces

Data flow
- Call before a board is loaded
- Call before any move is applied
- Call after moves are applied but no pieces were captured in the moves
- Call after two separate sequences of applyMoveSequence calls are done onto the same board

inCheck

Input validation
- Ask for invalid player number
    - -1
    - 2

Boundary cases
- Call for a player with a single king on the board
- Call for a player whose king has already been captured on the board

Control flow
- Call when the king is not in check
    - Call for each player
- For each piece type, trying each player, call when that piece type alone is set to capture the king
    - With one cell move
    - With several cell moves, if allowed by the piece type (rook, bishop, queen)
- For pieces that can move several cells (rook, bishop, queen), call when the king is in a capture path for the piece, but there is another piece on the board along the capture path that stops the capture
- Call when the king can be captured by more than one piece on the board

Data flow
- Call before a board is loaded
- Call after moves are applied to the board

pieceCanMove

Input validation

- Provide null for the board position string
- Provide an empty string for the board position string

Boundary cases

- Call with a board position above the top edge
- Call with a board position below the bottom edge
- Call with a board position to the left of the left edge
- Call with a board position to the right of the right edge
- Call with a board position that has no piece in it

Control flow

- For each piece type, call when
    - the piece has a valid one-step move in each valid direction (all others blocked)
    - the piece has more than one one-step move in a valid direction
    - the piece has all move directions blocked by pieces of the same colour
    - the piece has all move directions blocked by pieces of the same colour except one piece in a capture direction is of the opposing colour
    - the pieces has more than one one-step capture moves available
- A pawn cannot move forward but has a capture direction that is free, but no piece to capture in that direction
- A white pawn has an open move down the board
- A black pawn has an open move up the board
- A piece that can move multiple cells (rook, bishop, queen) has a space further on the board available for a move in a valid direction, but it would involve going through board position that have blocking pieces

Data flow

- Call before a board is loaded
- Call after moves are applied to the board

Example of identifying the outcomes of each test case to know what your code should do

The tables below repeat the lists of tests above, but now think through what the outcome of the test should be relative to the requirements or to how we've decided to handle odd cases.

loadBoard

Test	Outcome	Note
Null parameter	False	Expecting an object as parameter, but nothing there
Empty board stream	False	Invalid board
Read in a 1x1 board	False	Not enough space to have two kings on the board
Read in a 1x2 board with two kings	True	Minimal board
Read in a 2x1 board with two kings	True	Minimal board
Read in a board with one of the kings missing	False	Not enough kings
Read in a board with two kings of the same colour and one of the other colour	False	Too many kings
Read in a larger board with just the two kings and empty spaces	True	Minimal valid board configuration
Read in a larger board with the two kings and one copy of each pieces, both black and white	True	Nothing says you need multiple copies of pieces
Read in a regular chess board	True	Nominal case
Read in a board with more instances of pieces than is configured in a regular chess board, like having three rooks	True	Other than a restriction on kings, the problem has no limits on the number of other pieces.
Read in a board that is bigger than a standard chess board	True	Not confined to chess-sized boards
Read in a board with all pieces on the edge of the board	True	Extreme condition for pieces
Read in a board with all pieces away from the edge of the board	True	Safest condition for pieces
Read a board with leading blank lines	True	Requirement says to ignore leading lines
Read a board with trailing blank lines	True	Requirement says to ignore trailing lines
Read a board with a blank line in the middle of the board (shouldn't crash)	Your choice	Requirement doesn't say what to do and doesn't exclude it. Just shouldn't crash.

Read a board where the second line is shorter than the first line	False	A board that isn't rectangular
Read a board where the second line is longer than the first line	False	A board that isn't rectangular
Call loadBoard twice in a row with different boards	True	Second board should replace first one

printBoard

Test	Outcome	Note
Null output stream	False	Expecting an object as parameter, but nothing is there
Print a 1x2 board	True	Minimal board to print
Print a 2x1 board	True	Minimal board to print
Print a board with pieces at the edge of the board	True	Extreme condition for pieces
Print a board with pieces away from the edge of the board	True	Safe condition for pieces
Print a board that just has the two kings	True	Simplest board you can have
Print a board with a copy of every type of playing piece, both white and black	True	All pieces should be there with the correct capitalizations
Print a board that is larger than a regular chess board	True	Large boards are ok
Print a board before calling loadBoard	True	Print no output as there is no board…you've printed what you know of the board
Print a board after having successfully applied moves	True	Board after moves should be printed
Print a board after applyMoveSequence failed	True	Board before moves should be printed as applyMoveSequence is supposed to leave the board unchanged.