Due:

Friday March 22nd, 2024, 4:30 pm

Assignment dropboxes will be open on UM Learn until 6:00 AM in the morning following the assignment due date. You can submit until this time without penalty. Note that you should not expect to get last-minute help if you are working late.

The dropbox closing time is firm. Late assignments will generally not be accepted, and after 6:00 AM no submissions will be accepted without prior arrangement (e.g. submission of self-declaration form & email to your instructor).

Notes

• **All students in this course must read and meet the expectations described in the Expectations for Individual Work in Computer Science (follow the link and read all information provided).**

• Follow both the "Assignment Guidelines" and "Programming Standards" for all work you submit. Instructions given in this document take priority over instructions in the guidelines/standards. **Marks will be deducted for not following the programming standards and the guidelines described in this document.**

• You must complete the honesty declaration in UM Learn (in Assessments – Quizzes) before the assignment submission folder will be available to you.

• Hand-in will be via the UM Learn Assignments facility. Make sure you leave enough time before the deadline to ensure your hand-in works properly. The assignments are submitted using UM Learn's time (not the time on your computer).

• Questions about the assignment should be posted on the discussion forum, Piazza. You are also responsible for reading the answers and clarifications posted in the discussion forum.

Assignment 3: A Simplified Game of Chess

Description

In this assignment, you will write **Java** classes to enable a user to play a simplified version of the game of Chess. Chess is played on an 8 by 8 board, with two players each controlling a set of 16 pieces. The goal of the game is to capture your opponent's King. Here is a video that describes the game: https://www.youtube.com/watch?v=fKxG8KjH1Qg but note that you will not implement all rules (see below).

In your game, a human player will play against the computer. The human player always plays first. The human player will be able to play again after a match is finished and play as many matches as they want.

You will be responsible for 1) implementing the game logic (controlling turns, determining if someone has won the game, etc.), 2) controlling input and displaying the game board, and 3) two or more game AIs (automated players) to play against the human.

The game logic is responsible for maintaining the current status of the board (i.e. the location of each players' pieces) as well as determining if a player has won.

Displaying the game board and prompting the human player for input will be done via simple text input/output in this assignment. However, you should follow good practice and keep your code modular so that a future coder could replace this portion of your code with a nice-looking GUI.

You are responsible for implementing at least two different game AIs (automated players). How the AIs work is up to you, but you should develop two different approaches to playing the game. You don't need to program an AI that wins all the time but your AIs should at least be ranked (one harder than the other) and the more sophisticated approach **should involve some logical move choice** by the game AI. The reward for developing a very sophisticated game AI is the possible admiration of your classmates, and hours of entertaining game play.

The logic of your game should proceed roughly as follows:

- The game starts, and the human player is prompted for the difficulty of opponent they want.
- The human player plays first, and turns alternate between the human player and the game AI.
- On each turn, the human player is prompted for the piece they want to move (by entering the row and column of the piece) and where they want to move (by entering the row and column of the destination). Alternatively, the human may choose to forfeit and quit the match.
- Output a line of text describing the move made and whether any pieces were captured, and display the new state of the board.
- The computer moves. Output a line of text describing the move made and whether any pieces were captured, and display the new state of the board.
- Play continues by alternating between human and computer.
- If one player wins, the match stops and the winner is announced.
- After the match ends, the human player is then prompted to see if they want another match. The board is reset and the human is prompted for the difficulty of the opponent they want.

A rough structure to get you started on creating the tools for accomplishing all of these tasks is described below. The three major parts of the project are the Game Display, the Game Logic, and the AI. The functioning of each of these three parts will be dictated by three interfaces.

Simplified Chess Rules

Your game will implement the following rules/moves for each type of piece. If you are familiar with the game of chess, any moves not listed below are not required in this assignment (e.g. castling and capturing en passant do not need to be implemented). Additionally, you are not required to verify whether a player is "in check" (i.e. the king is in a position where it could be captured on the next move) after each move or display a notification when such a situation occurs. In this assignment, a player could move his king into check (which is not normally allowed). You are not required to test for conditions of checkmate or stalemate. The game will end when one player captures their opponent's king, or the human player decides to exit the match.

There are six types of pieces: rook, knight, bishop, king, queen, and pawn. See "Game Display" below for the initial setup of the board. Each type of piece is allowed to move as follows.

Rook:

- Moves forward/backward or sideways (i.e. along a row or column), as many spaces as desired but can NOT jump over pieces.
- Can capture the first piece in its path, by landing on that square.
- Do not implement castling.

Knight:

- Moves in an "L" pattern, two squares in one direction and one square in a perpendicular direction.
- Can jump over pieces.
- Can capture a piece in the square it lands on.

Bishop:

- Moves diagonally, as many spaces as desired but can NOT jump over pieces.
- Can capture the first piece in its path, by landing on that square.

King:

- Moves one square at a time, in any direction.
- Can capture a piece in the square it lands on.
- Normally, a king can not move himself into check. For this assignment, you DO NOT need to test whether this is the case. A king could move himself into check, and the match could end on the next opponent move without warning.
- You DO NOT need to test whether a king is in check after a move, and do not need to display a warning to the player. Normally if a king is in check, the player must either move the king or move another piece to block the opponent. For this assignment, you DO NOT need to require that a player deal with the situation if their king is in check. A player could move any piece and leave their king in check.

Queen:

- Moves in a straight line in any direction (along a row, column, or diagonal), as many spaces as desired but can NOT jump over pieces.
- Can capture the first piece in its path, by landing in that square.

Pawn:

- Moves one square at a time in the forward direction. The square must be empty.
- Do not implement the possibility of moving two squares for the first move.
- Captures by moving ahead one square on a diagonal to a square occupied by an opponent's piece.
- Do not implement capturing en passant.
- DO implement promotion: If a pawn reaches the far side of the board, the player can choose the type of piece the pawn will convert to (rook, knight, bishop, or queen).

Game Display

There is an interface for the game display and for obtaining user input. This should be kept separate from the game logic and AI, so that you could reuse the latter two with a different display in the future.

In this assignment, you will use a simple text output to display the board, with rows and columns labelled with integers to allow the human player to easily specify which piece they would like to move.

The board should initially appear as follows, with the human player's pieces in rows 7 and 8. Use lowercase letters to represent the computer's pieces and uppercase letters to represent the human's pieces. R - Rook, N - Knight, B - Bishop, K - King, Q - Queen, P - Pawn

```
  1 2 3 4 5 6 7 8
------------------
1|r|n|b|k|q|b|n|r|
------------------
2|p|p|p|p|p|p|p|p|
------------------
3| | | | | | | | |
------------------
4| | | | | | | | |
------------------
5| | | | | | | | |
------------------
6| | | | | | | | |
------------------
7|P|P|P|P|P|P|P|P|
------------------
8|R|N|B|K|Q|B|N|R|
------------------
```

The suggested starting point for creating the game display is a GameDisplay interface that looks something like the one below. (As part of your design you will need to fill in some information on data types for parameters & return values, and you may choose to add additional methods to the interface.) You should create a class (e.g. TextGameDisplay) that implements this interface.

```
public interface GameDisplay {

      public int promptForOpponentDifficulty(int maxDifficulty);

      public … promptForMove();

      public void displayBoard(...pass the current game board...);

      public void summarizeMove(...pass the last move...);

      public void gameOver(int winner);

      public boolean promptPlayAgain();

}
```

Your game logic class will call the methods of your game display class. The above methods would work as follows:

- promptForOpponentDifficulty will be called at the beginning of each match. The method will be given the maximum difficulty of the AI (a positive integer greater than or equal to two). The method will return a value between 1 and maxDifficulty, indicating the human player's choice of which game AI to play against.
- promptForMove will ask the human user to input their chosen move (e.g. by inputting row and column of piece to move, and row and column for the destination). It should return an object containing the information the game logic would need to make the move.
- displayBoard will be called after each move, and at the beginning of the game. This method will output a character in each square of the board, indicating the location of each piece. Use a space to represent an empty square. This method will assume that it is always passed a valid board (i.e. the number of pieces is valid, the initial configuration of the board is correct, etc.).
- summarizeMove will also be called after each move. This method will output text indicating the last move made (i.e. the type of piece that moved, where it started, and where it landed) and whether a piece was captured. Some output examples:
  ```
  Bishop moved from (1, 3) to (3, 5).  No capture made.
  Queen moved from (1, 5) to (5, 5).  Pawn captured.
  ```
- gameOver will be called when a match ends. This method accepts an integer that indicates whether the human or computer won, or whether the game was forfeitied. If either player wins, a notification to that effect should be printed. If the human players forfeits, the notification should reflect that.

Game Logic

You are responsible for implementing the back end that manages the game logic. Your game logic will call the methods of the GameDisplay interface. It must also satisfy a ChessController interface such as:

```
public interface ChessController {

      public boolean movePiece(...details of move to be made...);

      public void reset();

      public void playGame();

}
```

The methods work as follows:

- movePiece is passed a desired move for the human player. It should test whether the move is valid, and make the move (updating the status of the board) if it is allowed.
  Note that the AI will NOT call this method (see below for the method makeMove(), which asks the AI which move they want to make).
  The method should return false is the move is invalid. If the method returns false, the board

should be unaffected by the call (i.e., if a bad move is made, the board remains unchanged) and the user should be re-prompted for a move.

- reset will reset the game. It will be called when the player requests a new game.
- playGame will drive the game. It calls methods from the display to prompt the human user for input, summarize moves, and display the updated board. It also controls whose turn it is, tests whether the game is over, prompts the human on whether they want to play again, etc.

NOTE that your Game Logic class should have a constructor that accepts a single parameter of type GameDisplay (so that it can call the GameDisplay interface's methods, like gameOver, etc.).

Game AI

You are responsible for developing two or more game AIs. These AIs must satisfy the ChessPlayer interface:

```
public interface ChessPlayer {

   public ...returnType... makeMove(...provide last move made by human
                                    and current state of the board...);

}
```

The method makeMove should be provided with the last move made by the (human) player and the current configuration of the game board. It will return the next move made by the AI (e.g. so that you can pass that to the game display to summarize). It is left to you to determine the variables that need to be passed to keep track of a move.

The AI must make a valid play. Since the AI always plays second, there will always be a previous move provided to the AI. It is up to you to decide how to use the move (i.e. whether the AI will play in response to the human's move or use some other strategy to choose its move).

Note that the method only supplies the location of the *last* piece moved by the human player, not all previous plays. If the AI wants to save a series of previous moves it will have to do so on its own.

main

Your main class should contain only a very simple main method. main() should create an instance of your game display class, an instance of your game logic class (passing it the game display instance, as described above), and call the playGame() method on the game logic instance. That's it! playGame() then drives the game until play ends.

Additional Notes

- DATA STRUCTURES: In this assignment **you are permitted to use a 2D array to store the game board**. Any data structures constructed elsewhere in the game must be implemented by you.
- This is a large programming problem with many pieces to the puzzle – remember your incremental development strategy from COMP1020 and get small pieces working rather than trying to tackle the whole thing at once.
- Your program will be tested by the markers on **aviary.cs.umanitoba.ca**, and must run with a simple javac/java.

Hand-in

Submit all your source code for all classes. Each class should be declared as public in its own file.

You **MUST** submit all of your files in a zip file. Additionally, you **MUST** follow these rules:

• All of the files required for your project should be in a single directory.

• Include a README.TXT file that describes exactly how to compile and run your code from the command line. The markers will be using these instructions exactly and if your code does not compile directly, you will lose marks.

• Markers will run your program on **aviary.cs.umanitoba.ca**. Test your program there! You are not required to submit sample output and the markers must be able to run your program to evaluate it.

The easier it is to mark your assignment, the more marks you are likely to get. **Do yourself a favour**.