Assignment 3

In Assignment 1, you created a chess board and placed a knight piece on the chess board. Further more, you allowed a knight piece to move around as long as the moves were legal:

- Setup a Chess Board (8x8) (with a grid and draw it textually perhaps)
- Introduce a Knight
    - placement on chessboard
    - legal / illegal moves

In Assignment 2, you added more pieces and functionality to our Chess Game, namely:

1. Introduce 1 Queen, 2 Bishops, 2 Rook, 1 King and 8 pawns for only 1 player
    a. placement on chessboard
    b. legal / illegal moves
2. For all pieces
    a. Identify shadows (a possible move is blocked due to another piece in the path)

In Assignment 3, now let's get a game going!

Overview

1. Set the owner of the existing piece's(pieces you created in Assignment 2) to "Player1"
2. Add pieces for Player 2.

    These pieces should be initialized on the other side of the board for Player2, such that you should have a chess board with (as shown in Figure 1):

Piece	Player 1			Player 2	
	Mark	**Location**		**Mark**	**Location**
King	K	(0,4)		k	(7,4)
Queen	Q	(0,3)		q	(7,3)
Bishop	B	(0,2) & (0,5)		b	(7,2) & (7,5)
Knight	N	(0,1) & (0,6)		n	(7,1) & (7,6)
Rook	R	(0,0) & (0,7)		r	(7,0) & (7,7)
Pawn	P	(1,0) => (1,7)		p	(6,0) => (6,7)

Table 1 – Mark Table of Chess Pieces

3. Capture pieces
4. Identify threatened locations by each piece
5. King can not move to a threatened location
6. Play game until checkmate
7. Sequence / Interface

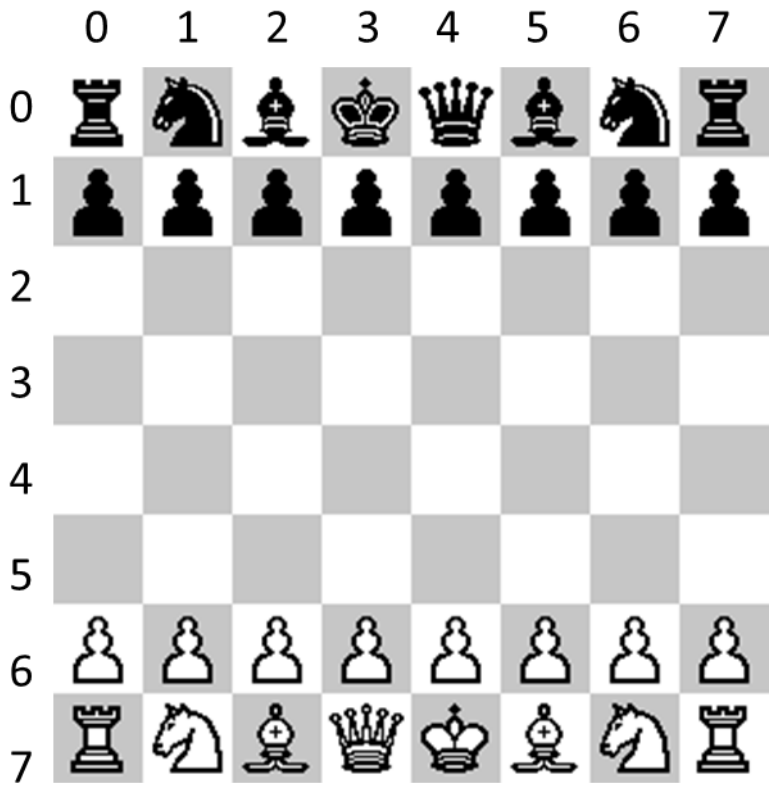


Figure 1 – Chess Game Layout

1. Chess Game Details

1.1. Set the owner of the existing pieces to Player 1

This is primarily what you have already done for in Assignment 2. If you have not done so, please go back and do Assignment 2. Please make sure of the following:

a) There should a chessboard with 16 pieces initialized from Assignment 2
b) These pieces should be initialized at row 0 and row 1 – Please refer to Table 1 for exact location on the Chess Board.
c) Set the owner of these pieces to Player1 so Player 1' pieces are on row 0 and 1.
d) Also initialize the mark accordingly to that of Player1

1.2. Add pieces for Player 2

Just like you have done earlier in 1.1 above, introduce another set of Chess Pieces (16 pieces) for Player 2 with the corresponding mark and at the locations described in Table 1.

1.3. Move and Capture Pieces

For each move, the players take turns. None of the chess pieces may move to a square occupied by another chess piece of the same owner(or player – owner = player). However, a piece may move onto a square occupied by an opponent's piece. When this occurs, the opponent's piece is 'captured' and is permanently removed from the chessboard.

A capture happens when a piece moves* onto a square occupied by an opponent's piece.

move* - When we say move* here, we say that the move of the piece was constrained by the selected piece's rules.

This is true for all the pieces except for "pawn". Pawn captures diagonally.

Pawn is special such that it may move 1 square (or 2 depending on initial move) forward, but it can only capture an opponent's piece if that opponent's piece is occupying a immediate "forward" diagonal.

Pawn Moves:

NEW: I had not discussed this in Assignment 2, but each piece may move in either a "forward" or a "backward" direction as long as the move is satisfying the legality condition of that piece. For example a rook at (5,2) in Figure 2 can occupy any of the locations in the row 5 or column 2 i.e. a legal move can be (5,2) to 4,2) or (5,2) to (6,2).

Moving forwards and backwards is allowed for all the pieces EXCEPT pawn.

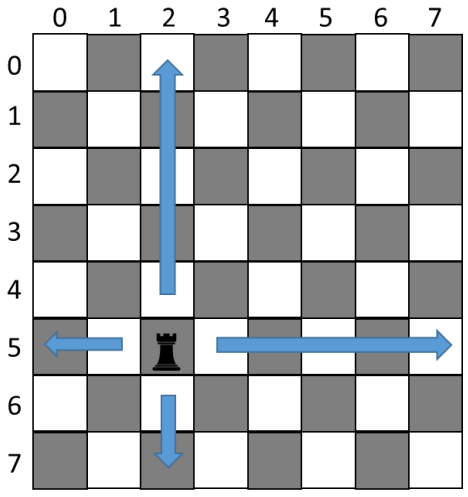
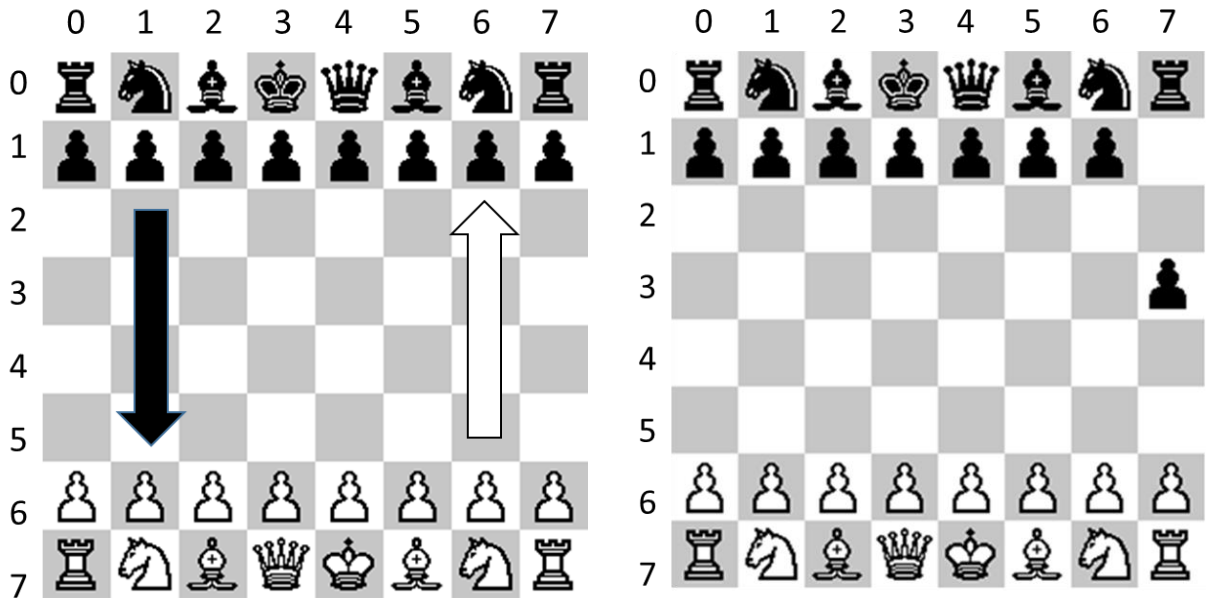


Figure 2 – Rook's Possible Moves

Pawn can only move in a "forward" direction. For our game, that means all the pawns for player 1 can move from row 1 to row 2 to row 3 etc as shown by the dark arrow in Figure 3a. Pawns of player 2 can move from row 6 to row 5 to row 4 etc as shown by the white arrow in Figure 2a

A pawn may not move backwards. For example in Figure 3b, pawn at (3,7) may move to (4,7) but is not allowed to move to (2,7)

Any piece directly in front of a pawn, friend or foe, blocks its advance.



(a) (b)

Figure 3 – Pawn's Legal Moves

Pawn Capture:

A pawn can only capture its opponent's piece if the opponent's piece is occupying the pawn's immediate diagonal "forward" motion.

Unlike other pieces, the pawn does not capture in the same direction as it otherwise moves. A pawn captures diagonally, one square forward and to the left or right. In Figure 4, the white pawn may capture either the black rook or the black knight.

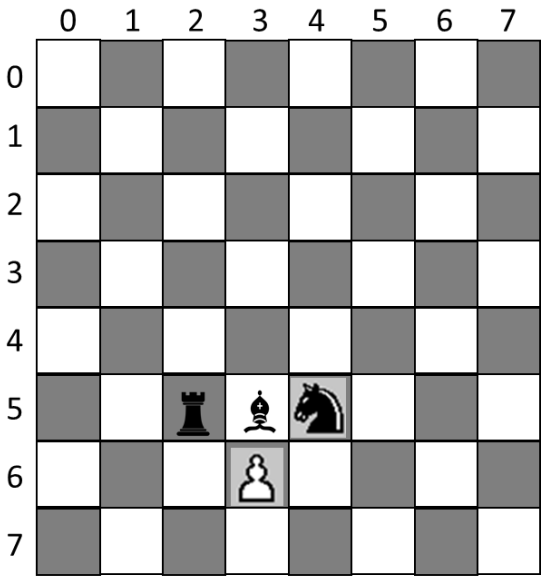


Figure 4 – Pawn's Capture Example

1.4. Identify Threatened Locations by Each Piece

You need to keep a track of all the possible locations threatened by each piece. A location is considered to be threatened for a given piece if that piece can possibly capture the opponent's piece at that given location. (please read 1.3 for the rules of capture).

Do not forget: a location may not be threatened if it is in the "shadow"s – a shadow may be casted by a friendly piece or an enemy piece. Also, shadows are not applicable for knights, as they can capture opponent's pieces if that opponent's piece is at a location to which a knight move to legally. For the description of shadows, please read Section 1.3 of Assignment 2.

1.5. King can not move to a threatened location

A king is the most important piece on the chess board. If king is captured, then the game is over and the capturing side wins.

For this reason, king can not move to a location which maybe a "threatened" location. A threatened location is where a king can be capture by an opponent. For each move of the king, you are to examine all the possible moves of the opponent's pieces such that the "next" move of the opponent could not result in the capture of the king. (See section 1.4 for more information on Threatened locations)

Do not allow the player to move the king to a "threatened" location –

a. Let the player know that their move will result in the king being threatened. Also let them know from which piece at what location.

b. Allow the player to try again.

c. If the King is the last piece on the board, and all the possible moves are "threatened", then it is checkmate and the game is over and the player, who is threatening, wins!

Please note that when you check for this, make sure you check the king is really the last piece on the board. You can easily run into a situation, where all of king's moves may be being threatened, but not the king's own location. If there is another piece on the board, then it is not game over, as the player can move its remaining pieces.

For example in Figure 5a, the white king is trapped. He is the last piece on the board. The white king's possible moves are

(7,4) => (7,3)	(7,4) => (6,3)	(7,4) => (6,4)	(7,4) => (6,5)	(7,4) => (7,5)

However, each of these possible moves are "threatened" as seen from the 2 rooks and 1 bishop (green, blue and red arrows). The rational behind this is that in each move, you must move 1 piece. For white player, as king is the only piece left, you must move the white king. However, as soon as you move the white king, the white king will be captured in the opponent's next move. This is also known as checkmate.

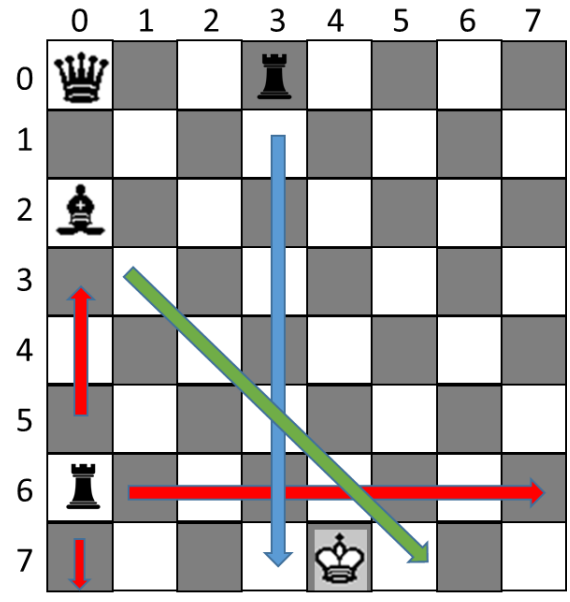


Figure 5a – Checkmate

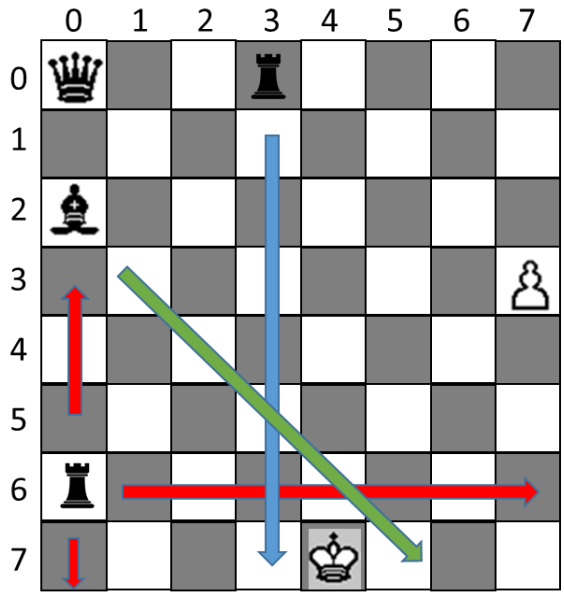


Figure 5b – Not a Checkmate

However, if you have a situation such as in Figure 5b, where all of the white king's possible moves are threatened, but the white player still has 1 pawn remaining, then it is not a check mate. The rational behind this is that when it is the "white" player's turn, it still has to move a piece – it can move its pawn – It does not have to move its king.

1.6. Play the Game until Checkmate or Quit

Both players are to play the game until either a checkmate happens or any one quits. When this happens, the winning player is announced as "Player 1 wins" or "Player 2 wins" and you are prompted to either start a new game or quit the game.

1.7. Sequence (Interface)

When the game starts up, the user is shown the chessboard initialized with all 32 pieces at their default locations.

Each player must be given at least 3 options on every turn:

i) Move
ii) Quit
iii) Restart

Quit – quits the game. Player who quits looses the game and a message is displayed to convey this. The game ends

Restart – restarts the game. All the pieces go back to their original locations and everything goes back to as if the game just started.

Move – Player is prompted: "from" and "to". The "from" states the location of the chess piece to be moved. "To" is the destination. Remember player should enter in the following format: row, column where row is 0 to 7 and column is 0 to 7

Also, note to make sure the player should NOT be able to move the opponent's piece.

Once a move is executed successfully, board with the new move is displayed and the other player is prompted for these options.

The game continues until a check mate occurs or quit or restart is selected.

2. Classes

To help you develop the chess game, find below description of some of the classes you'll need

2.1. Class ChessBoard (nothing has changed in this class from Assignment 2)

Your ChessBoard class should allow you to store different pieces at different locations (that is what a real life chessboard does after all)

Variables:

You decide how to represent your chess board (perhaps a 2 D array, etc). Strictly speaking, you're free to choose whatever data structure you like, provided the methods pieceAt, addPiece, and removePiece all work like they're supposed to. Let's just say, for example, that you wanted to use a two-dimensional array to represent the spaces on the chess board (let us call it the "spaces" array).

Methods:

Public accessors and mutators methods for your private variables.

Additionally, it should have at least the following methods and a constructor.

Constructor: *public ChessBoard()*

The constructor will initialize your spaces array to an empty 8x8 array.

Method #1: public ChessPiece getPieceAt(ChessLocation location)

This method should return the piece at the specified location

Method #2: *public void placePieceAt (ChessPiece piece, ChessLocation location)*

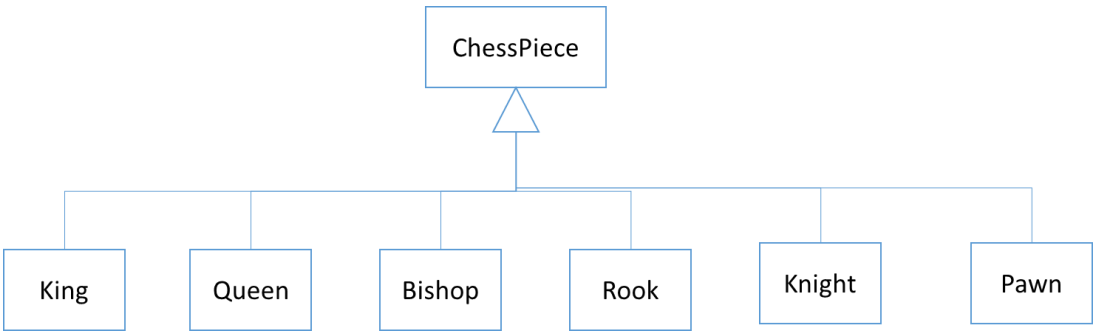
placePieceAt should place the given piece on your ChessBoard (i.e. in your "spaces" array) at the given location. **If the user attempts to add a piece to a location where one already exists,** *placePieceAt* **should overwrite the old piece with the new one.** Do not forget to update the piece's own location in its own class. ~~Also, do not forget to call the piece's updateThreateningLocations method to update the locations which this piece can now threaten.~~

Method #3: *public void removePiece(ChessLocation location)*

The method *removePiece* should remove whatever piece is at the specified row and column from your "spaces" data structure, setting the value null at that row and column to be null. This method can be used to remove a piece on the board from the original location after a move. Do not forget to update the piece's own location to null.

In addition to these three required methods, you're free to add any other methods to the ChessBoard class in order to make your job easier. For instance, it might not be a bad idea to have toString method which draws out your ChessBoard

2.2. Class Abstract ChessPiece *implements ChessPieceInterface*



Chess Piece will be an abstract superclass for the classes: King, Queen, Bishop, Rook, Knight and Pawn and should implement the ChessPieceInterface

Every piece needs to know where it is and which player it belongs to. Make sure you have at least the following:

Variables:

private ChessGame game
- To determine which game does this piece belong to

private String player
- Chess is a 2 person game identified by colours black and white.

  Now you have a 2nd person playing:
  - identify the owner of each piece (i.e. player = "player1" or "player2").

private ChessLocation location
- Current location of the chess piece

private ArrayList<Location> threateningLocations
- Array for all the locations which this chess piece is threatening

protected char id
- This identifies the piece on your board when you are displaying your chess board. The following should be your table:

Piece	**Mark**			**Piece**	**Mark**	
	Player 1	**Player 2**			**Player 1**	**Player 2**
King	K	k		Knight	N	n
Queen	Q	q		Rook	R	r
Bishop	B	B		Pawn	P	p

Now that you have a 2nd person playing, make sure you keep the appropriate ids of each player.

Methods

Public accessors and mutators methods for all your variables.

ChessPiece(String owner, ChessLocation initialLocation, ChessGame game)

- Initialize the owner, and the game to the given variables
- Initialize the location of this piece to null, meaning set the local variable "location" to null
- Initialize the arraylist for threatening location
- Place yourself on the chessboard with the initialLocation (hint: you need to get the chessboard reference from the game object passed)

public boolean moveTo(ChessLocationClass newLocation)

- (For each piece's move, the subclass will check the legality of the move with respect to the piece type. Hence, this superclass does NOT do any checks on the legality of the move)
- Conditions: A chess piece can move to a "newLocation" if and only if:
    - If there is no piece at the newLocation OR
    - If there is a piece at the newLocation, then that piece should belong to the opponent – in this case a capture happens (see section 1.3 for details).

- for making the move:
    - get ChessBoard from the object game,
    - remove any pieces at the newLocation (call removePiece of ChessBoard)
    - place this piece at the newLocation
    - return true

- If the move is legal, make sure the pieces can not be moved to indexes outside of the bounds of the chessboard
- If there are any issues with either the legality of the move etc, return false

public abstract void updateThreateningLocation(ChessLocation newLocation)

- There should be no implementation for this in the super class. Each of the sub classes will override this method to update the threateningLocations array based on what piece it is.

// nothing changed here for Assignment 3
protected boolean checkLineOfSight(ChessLocation start, ChessLocation end)

- this is a complex function
- it should return if there is a line of sight from the position start to the position end. This means there are no obstructions from start to end i.e. there are no pieces between start to end.
- Between start to end can means:
    - you should check horizontally,
    - you should check vertically and
    - you should diagonally (perfect diagonal) with a slope of either 1 or -1

2.3. Interface ChessPieceInterface

public boolean moveTo(ChessLocation location);

ChessPiece class should implement this interface. Basically, this forces a moveTo functionality in the ChessPiece class (Note: you have already defined this functionality in assignment 2).

2.4. Classes Rooks, Bishop, Knight, Queen, Pawn

In the constructor of each of these classes, you are setting the "mark" variable accordingly (piece either belongs to Player1 or Player2 – see Table 1.)

Add the following for each subclass:

protected void updateThreateningLocation(ChessLocation newLocation)

- Each of these classes will override this method from the superclass to update the threateningLocations array (threateningLocations array is in the superclass ChessPiece) based on what piece it is. See section 1.4 for more details.

For each subclass, the signature of the moveTo method has changed now to return a Boolean instead of a void. Now the signature is:
public boolean moveTo(ChessLocationClass newLocation)

- Returns true if moveTo is successful else false

2.5. Class King

Everything from Assignment 2 applies. In addition, King should **not** be allowed to move to a location which is being "threatened" by one of the opponent's pieces.

No variables (except the ones it inherits from the superclass automatically)

Methods:

Constructor:
public King(String player, ChessGame game, ChessLocation initial_ location)

- call the constructor of the super class with appropriate arguments
- check for the player and set the mark accordingly (player can either be "Player1" or "Player2"

public boolean moveTo(ChessLocation destination)

- check legality of the move
    - check if the move is legal accordingly for a King
    - during this check, also check if the destination location will put the king in danger: invoke the method locationInDanger
        - if okay, call the moveTo of the super class and return true,

        - otherwise output an error message stating which piece threatens the king and return false

public void updateThreateningLocation(ChessLocation newLocation)

- this method will be overridden from the superclass to update the threateningLocations array based on what piece it is. See section 1.4 for more details.

public ChessPiece locationInDanger(ChessLocation destinationLocation)

- update the threateningLocations for all the opponent's pieces
    - for this you have to retrieve chessBoard object from your instance "game" variable and go through all the opponent's pieces and update the threateningLocations
- returns the ChessPiece which may threaten the king if the king is at the "destinationLocation" location else returns nulls

public boolean anyMovesLeft()

- returns true if the King has any moves left which are not being threatened (hint: make use of locationInDanger function)
- returns false if the king can not move without being threatened

public ChessPiece check()

- returns the ChessPiece which is threatening the king at the "current" position
    - hint: call locationInDanger with the current location as the parameter
- If the king is not being threatened, return null

2.6. Class ChessLocation

No change from Assignment 2

2.7. Class ChessGame

If you are not familiar with the game of Chess, see Section 1 and the Appendix A for a game description*

In this game, you will alternate turns between Player 1 and Player 2.

<u>Variables</u>

ChessBoard board

- Chess board to play the game on

String player1

- Models player 1

String player2

- Models player 2

Methods:

- Accessors and mutators methods for the private variables

Constructor: *public ChessGame(String player1, String player2)*

- Initialize the players
- Initialize the chessboard object
- Initialize all **32 pieces (16 for player 1 and 16 for player 2)** with their appropriate locations on their board and add it to your board (hint: initialization of each piece can be done in one statement by calling each of the chess piece's constructor and using the "**this"** keyword)

2.8. Class PlayGame

Take a look at the description of the interface given in 1.7.

Variables:

- none

Methods:

public static void main(String[] args)

- display an initial menu on the screen describe what the program is about
- make sure you talk about the row and columns are to be used for indexes and give example

- initialize a new chessGame with player "Player1" and player "Player2"
- retrieve and display the chessBoard from this initialized game
- Take turns between player1 and player2
- Ask each player if they want to "move" or "quit" or "restart"
    - If they want to quit, then the program gracefully ends with a goodbye message
    - If they want to restart the game, then the game (not the program) should gracefully end and a new game starts
    - If they want to "move"
        - Check for a checkmate
            - No other piece is left except the KING AND
            - anyMoveLeft (method from king class) returns false AND
            - check (method from king class) returns a ChessPiece (and not a null)
            - If all of the above is satisfied, then display a message stating "checkmate" with the information of the ChessPiece threatening the king
        - Check for a check (is the king in danger?)
            - check (method from king class) returns a ChessPiece (and not a null)
            - If the above is satisfied, then display a message stating "check" with the information of the ChessPiece threatening the king
        - Ask user for the source
            - states the location of the chess piece to be moved
        - Ask user for the destination
            - states the location of the chess piece to move to

        - Check if there is a piece at the source (there should be one – if not, ask to retry)
            - If there is, then then move the piece to the new location.
                - If return type is true, then it is the opponent's turn.
                - If return type is false, then the player is asked for another move.
    - Keep on doing this either player quits or restarts or checkmate occurs
        - each time a piece is moved, display the board with the new locations

Move – Player is prompted: "from" and "to". The "from" states the location of the chess piece to be moved. "To" is the destination. Remember player should enter in the following format: row, column where row is 0 to 7 and column is 0 to 7

Also, note to make sure the player should NOT be able to move the opponent's piece.

Once a move is executed successfully, board with the new move done is displayed and the other player is prompted for these options.

The game continues until a check mate occurs or quit or restart is selected.

** The above main method is quite large and can easily be broken down into multiple private methods. I leave it up to you to design your implementation to break the main method into such private methods **

Deliverables:

1. From within BlueJ, create a JAR file containing your code (Project menu, click "Create JAR File").

   When the "Create Jar File" dialog is displayed, make sure the "Include Source" and "Include BlueJ Project" checkboxes are both checked! If you forget to include the source code, you will get a zero, so make sure you double-check after submission as well. Multiple submissions are allowed up until the deadline. Only the latest one will be graded

   Submit the resulting Jar file on cuLearn.

2. Draw a UML Class Diagram reflecting your design. This should be done using a graphical application such as powerpoint etc.

Marking Scheme:

0	- not submitted - jar file submitted with out the source code - project file submitted only -
25	- submitted but doesn't compile or very incomplete
50	- compiles but crashes easily (i.e., terminates by throwing an exception) and/or doesn't follow the assignment requirements. For example, illegal moves are being allowed. - Allows 2 player game with alternating turns - (King, Queen, Bishop, Knight, Rook and Pawn) pieces only work as outlined - Instructions are not included and/or are not clear on how to run the program - Missing UML Diagram
75	- compiles but crashes easily (i.e., terminates by throwing an exception) and/or doesn't follow the assignment requirements. For example, illegal moves are being allowed. - Capture is working - Only either check or checkmate is working (not both) - Missing Javadoc - Missing minor things (such as out of bounds check etc) - incorrect/incomplete UML Class Diagram - zipped file submitted instead of the jar file
100	- Games works beautifully - Players take turns - Capture is working - Check and checkmake is working - java doc exists for each - Complete UML Class Diagram

Appendix A – How the Game of Chess is Played

The Game

The basic idea behind chess is pretty simple. It's a two-player game played on an 8 by 8 grid (Figure 1), with each player controlling his or her own army of colored pieces (traditionally colored white and black). The game always begins with the pieces arranged as shown in Figure 1, and starts with White moving one of his/her pieces. The players then alternate turns, with each player moving one of their own pieces on their turn (the exact rules governing how each piece can move will be discussed later). If a player moves one of his/her pieces onto a square occupied by his/her opponent's piece, then the enemy piece is said to be "captured" and is removed from the board (Figure 7).

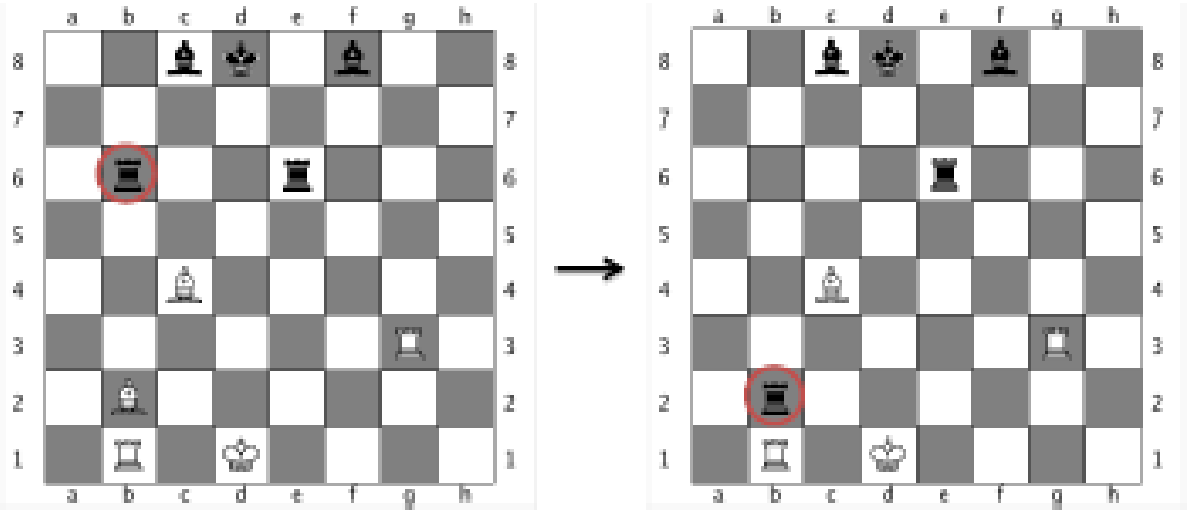


Figure 7. Example of capturing. The black Rook (originally at b6) moves to the same square as the white Bishop (located at b2), so the Bishop is "captured" and removed from the board.

The game ends when one player captures the other player's King. To this end, there are a couple of special scenarios known as *check* and *checkmate*. A player is said to be in *check* if their King is in danger of being captured by an opposing piece in one move. That is, a player is in *check* if his/her opponent could capture his/her King by making a single move. Once a player is in *check*, they are required to make a move to get themselves out of *check* (because if he/she did not, he/she would lose). Similarly, a player is not allowed to make a move that would put him/herself into *check* (again, doing so would immediately cause that player to lose). Sometimes, however, there are situations in which a player is in *check* and every legal move to available would also result in him/her being in check. This is known as a *checkmate* and results in the conclusion of the game, with the checkmated player losing.

Interestingly enough, however, the majority of high-quality chess games does not end with *checkmates*, but instead ends with draws, often the result of a condition called a *stalemate*. A stalemate occurs when a player is currently not in *check*, but any legal move left available would result in him/her moving into *check*. There are other ways to end a chess game by a draw, but we won't go into them here, and you don't need to worry about them for your game (they're fairly

infrequent, but hey, they make for great extensions. If you want to read about them check out http://en.wikipedia.org/wiki/Draw_(chess)).

The Pieces

The Knight

This is what a knight looks like on a chessboard:

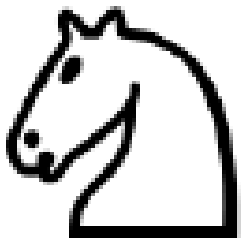


Figure 8. Image of a knight.

So what make a knight really interesting, and really easy, is how it moves. In particular the knight moves in L-shapes, moving first two squares in one direction and then one square in a perpendicular direction:

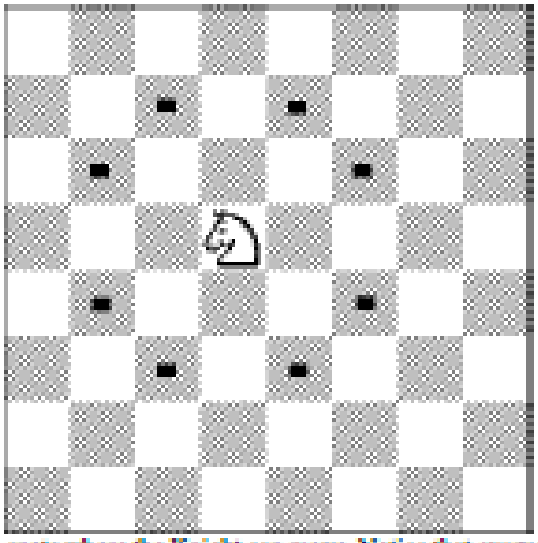


Figure 9. Diagram showing all of the spots where the Knight can move. Notice that every viable spot is two spaces away in one direction and then one space away in a perpendicular direction. Courtesy of www.thechesszone.com

The Knight is the only piece in chess that can jump over other pieces. What this means is, if you want to check to see if a given move with a Knight is valid, all you need to do is look at the state of the square the Knight is trying to move to.

The Queen

The Queen is considered the most powerful piece in chess. This is because of its extensive ability to move and capture pieces.

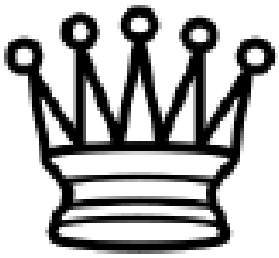


Figure 10. Image of a Queen. Like the King, it's depicted by a crown, but it's very different than a King. Don't get them confused.

A queen is allowed to move and capture enemy pieces along any straight line (horizontal, vertical, or diagonal) from the square that she currently occupies:

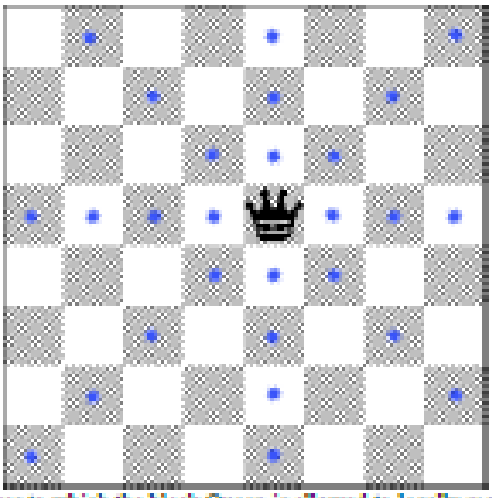


Figure 11. Diagram showing all squares to which the black Queen is allowed to legally move. Notice that every viable spot is on a straight line emanating out from the Queen's current location. Courtesy of www.thechesszone.com

The Bishop and Rook

Figure 12. Images of a Bishop (left) and Rook (right). The Rook is often called a Castle, but the official name is a Rook so that's what we're going to use.

Since we already talked about the Queen in a fair amount of depth, I'm going to go much faster through the Bishop and Rook. They can effectively be thought of as less-powerful variants on the Queen. In particular, the Bishop is a variant that can only move along diagonals, whereas the Rook is a variant that can only move along horizontal or vertical lines.

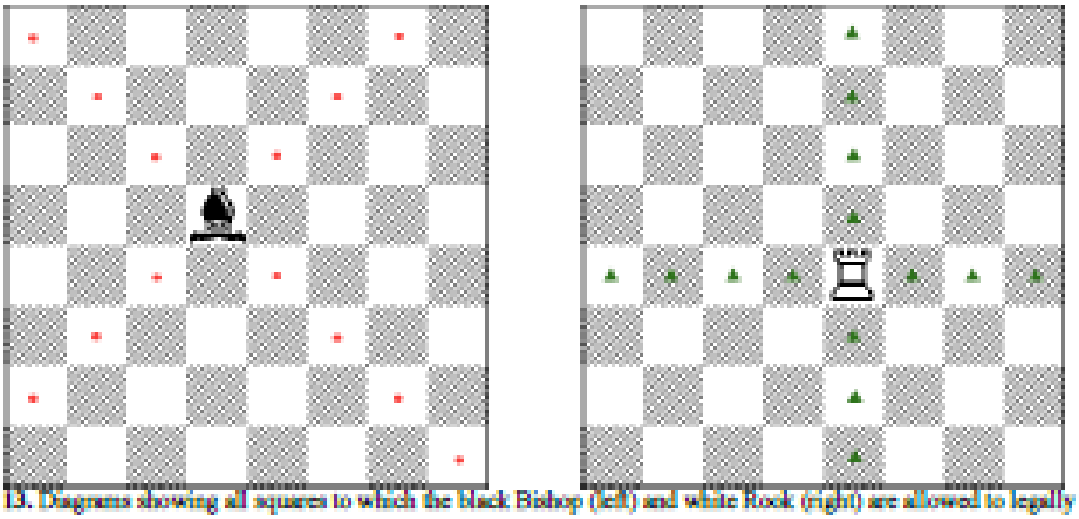


Figure 13. Diagrams showing all squares to which the black Bishop (left) and white Rook (right) are allowed to legally move. Notice that every viable spot is on a straight line emanating out from the Bishop or Rook's current location, but the straight lines allowed are different for the Bishop and Rook. Courtesy of www.thechesszone.com

The King

Figure 14. Image of a King.

Like the Knight, the King is also quite a simple piece. This is because its range of motion is quite limited. In particular, the King can only move one space, but it can move in any direction. To answer your question (if you haven't played Chess before), yes it is a little weird that the most important piece in the game is also one of the weakest as far as its ability to attack and move. That's just how it goes.

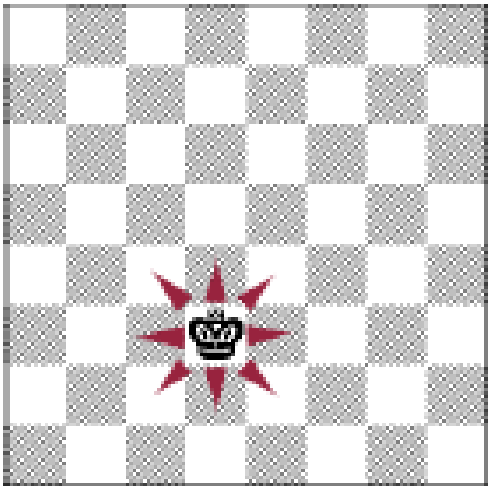


Figure 15. Diagram showing all the spaces where the black King can legally move. Courtesy of www.thechesszone.com

The Pawn

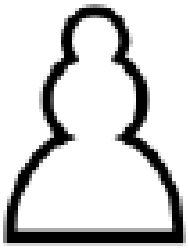


Figure 16 Image of a pawn. The picture of it is boring, but the way it moves is quite interesting.

Interestingly enough, the pawn, which is definitely the weakest piece in the game of Chess, is also the most complicated to get right. We'll go slowly to make sure everything's clear. The first interesting thing to say about pawns is that they only can move "forward." In this case forward is defined as towards the opponent's side of the board. In our orientation, this means that White pawns must always move up the screen, and Black pawns must always move down.

Now, ordinarily, pawns only move along a vertical line (always towards the opponent's side of the board) and they only move one space at a time. Except for two scenarios: a pawn's first move and capturing. For instance, each pawn is given the option of moving forward either one or two spaces on its first move (assuming that these paths lie unblocked by any pieces):

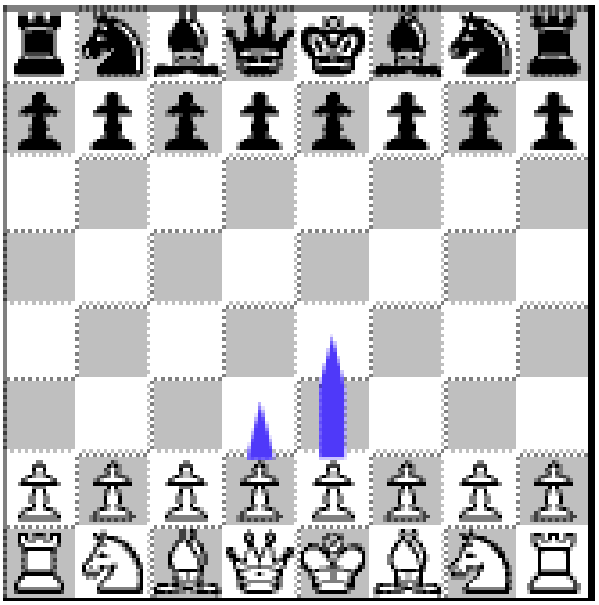


Figure 17. Diagram showing the unique movement of a pawn on its first move. Since the white Pawns shown have not been moved, they have the option of moving forward either one or two spaces (as indicated by the blue arrows) on their first move. This only applies if this move is not blocked by any piece, friend or enemy. Courtesy of www.thechesszone.com

Additionally, Pawns have another interesting wrinkle in that they capture differently than they move. In particular, while Pawns are only allowed to move along a vertical line forward, they capture on a diagonal line forward:

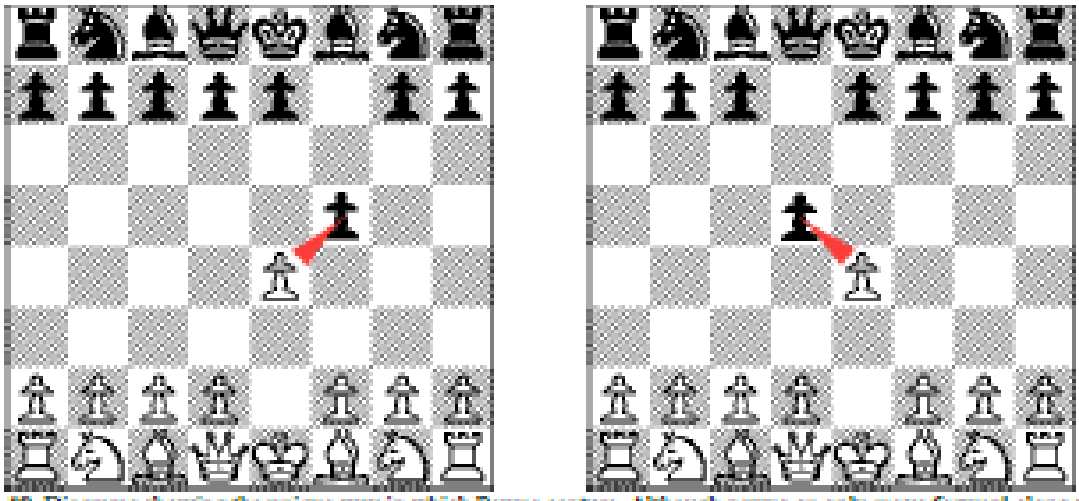


Figure 18. Diagrams showing the unique way in which Pawns capture. Although pawns an only move forward along vertical lines, they are only allowed to capture forward along diagonal lines.

This also means, however, that Pawns cannot capture directly forward:

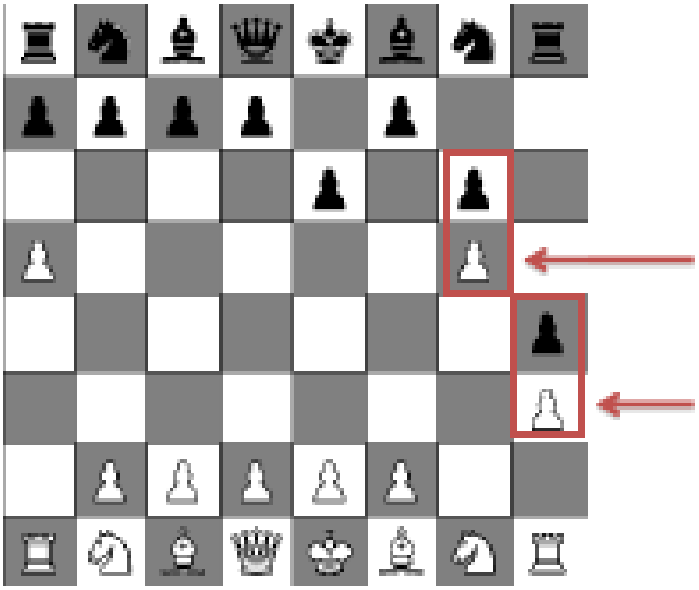


Figure 19. Diagram showing pawns who cannot capture each other (red boxes). Since pawns can only capture on diagonals, neither the black nor the white pawn in either of the red boxes can move in the current configuration.

Finally (and this you don't have to worry about), if a Pawn makes it all the way to the other side of the board, the player in command of the Pawn gets to replace that Pawn with their choice of a Knight, Bishop, Rook, or Queen (in this version of Chess is taken care of automatically for you – try it, it's kind of cool).