Muğla Sıtkı Koçman University

Faculty of Engineering

Computer Engineering

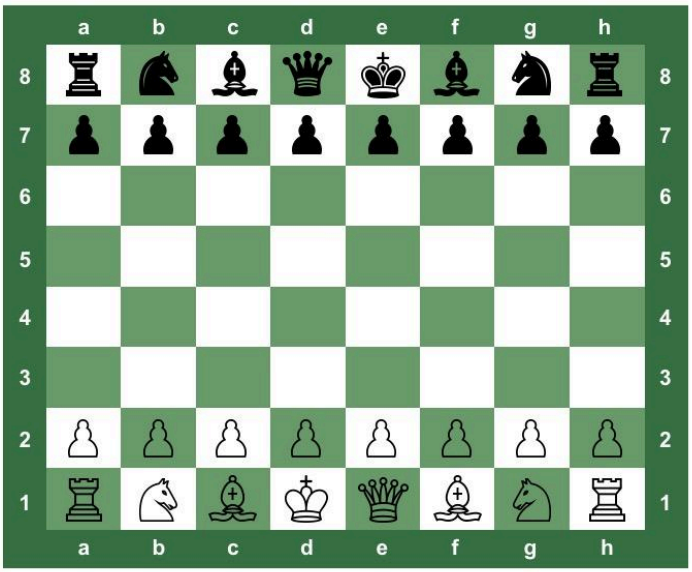
Midterm pdf documentation

CENG 1004 Spring 2023- OOP

Chess Game

Name	Seham Hakim Ahmed Othman
Section	2
Department :	Computer Engineering
Submitted date	22-05-2023

1- Pieces in Chess game :



Pieces :

1- Pawn  *P, p*

2- Rook :  *R , r*

3- Knight :  *N , n*

4- Bishop:  *B ,b*

5- King :  *K , k*

6- Queen:  *Q , q*

A) Printing Board

1- Describe how you manage to print the board representation.

2- How did you construct loops?

3-What are the challenges? How did you manage to solve them?

To print the chess game board :

public static Square[ ][ ] Board2D = new Square[8][8];

I needed to make a 2D array of squares(Board2D) that is initialized as an 8x8 array of Squares which will be 64 squares.

Its public : means reachable everywhere and static because we are using it several times in the classes, so it won't occupy several spaces in memory.

After that inside the default constructor of the chessBoard class I made a nested loop in order to iterate over the rows, and the inner loop iterates over the columns of the chessboard, then in order to put each piece at its place ,I created another 2 for loops to set pawns(black and white) and write down the other pieces locations on the board at their starting positions ,to make sure that every single piece is in its specific square.

Then I made an int method called getLtterIndex(); which its parameter is a char.

I thought about instead of using many if-else statements I made it to return a switch statement which has several cases : if the input matches the case char it will give us the index of it!

This method is used to get the letter index means it converts the letter part of the position into the corresponding column index in the array and this allows for easy retrieval of squares and pieces using their positions. if the letter was A or a means its at the index 0 , then if the letter was B or b its at the index 1 , then if none of above throw error exceptions as shown :

```
private int getLetterIndex(char c){
   // instead of many if else statements I made it to return a switch
statement which has several cases to get the pieces from specific squares .
   return switch (c){
       case 'A','a' -> 0; //means if the letter is 'A' or 'a' return index 0
       case 'B', 'b' -> 1; // if the letter is 'B' or 'b', return index 1
       case 'C', 'c' -> 2; // if the letter is 'C' or 'c', return index 2
       case 'D', 'd' -> 3; // if the letter is 'D' or 'd', return index 3
       case 'E', 'e' -> 4; // if the letter is 'E' or 'e', return index 4
       case 'F', 'f' -> 5; // if the letter is 'F' or 'f', return index 5
       case 'G', 'g' -> 6; // if the letter is 'G' or 'g', return index 6
       case 'H', 'h' -> 7; // if the letter is 'H' or 'h', return index 7
       default -> throw new IllegalStateException("Unexpected value: " + c);
// If the letter is none of the above so at that case throw an exception
   };
}
```

The reason that I put all of these statements inside the constructor is because when we run the code from the main class, constructors are working first , before the methods inside the class which have been called.

Then I created a to String method to print the board
First it will print the letters (A B C D E F G H) on the board, then I made a nested loop, it starts looping through columns and rows.
I made an integer var called “num” that is set to be row numbers which are on the edge or sides of the chess board which is 8 then inside the loop it will decrease by 1 at each time. then it will iterate over the rows and columns and append the pieces to their locations (specific squares), and if there is no piece at that location it will make it empty(or null), then adding spaces and (|) to make the board .

The challenges that I was facing at first was:
1- The creation of the board, putting all of the numbers and the letters all around the board .
2-How to connect the square, piece, and the board all together at the ChessBoard class.
I managed to solve them by creating 2D arrays of squares at the chessBoard class and then I set up all pieces in a nested for loop.
I made a private method called getLetterIndex(); which made the creation of the board easier.

B) Defining Board and Square Classes, A board can have 64 squares. How did you define the relation between Board and Square objects?

1- At the board
 class the chess board is composed of multiple Square objects.
Inside the chessBoard class I defined the relation between them as follows :
public static Square[ ][ ] Board2D = new Square[8][8];
Which says that a board is composed of 8 x 8 squares = 64 squares.
Also I have initialized a 2 D array and I use nested loops in order to put the square object in the board.

2- In the Square class
 the relation between them defined as :
Since the relation between the Square class and ChessBoard class is an obligation , I made up a private instance ” board” of type chessBoard, and inside the constructor it asked for a third parameter so I added the board of type ChessBoard as a third parameter looking as follows :

```
public Square(int row , int col , ChessBoard board) { //parametric
constructor for Square class
   //It takes the row, column, and the ChessBoard object that square
belongs to as a parameter.
   // we are using this to avoid shadowing since the name of the instance
is the same with the parameters of the constructor!
   this.row = row;
   this.col = col;
   this.board = board;
   //What it does is It initializes the instance variables (row , col ,
board ) with provided values.

}
```

C) Implementing methods of Board and Square Classes.

Describe each method in these classes except mutator and accessor methods. In each method description you should provide:
1- What does the method do ?
2- One sentence description of each parameter. What does it return? How did you implement the functionality?

- Square Class -

1- **public Square(int row , int col , ChessBoard board) { }**

- This is a default constructor for the Square class.
- It takes the row, column, and the ChessBoard object that square belongs to as a parameter.
- What it does is It initializes the instance variables (row , col , board) with provided values.

```
public Square(int row , int col , ChessBoard board) { //parametric
constructor for Square class
   // we are using this to avoid shadowing since the name of the
  instance is the same with the parameters of the constructor!
   this.row = row;
   this.col = col;
   this.board = board;
}
```

2- **public ChessBoard getBoard() { }**
This method returns the ChessBoard object that the square belongs to.
no parameter
returns the board object.

```
public ChessBoard getBoard() { //a public method which is
reachable everywhere that will get or returns the ChessBoard
object
   return board;
}
```

3- **public int getRowDistance(Square location) { }**

- This int method calculates the row distance between the current square and another specific square (which is the location).
- it just have one parameter which is the Square location
- It subtracts the row of location from the row of the current square and returns the result of it.

```
public int getRowDistance(Square location) {
   //this method calculates the Row distance between 2 square objects.
   return this.row - location.row;  //this line calculates the row distance
between the : current Square (which we use the keyword this to represent
it), and another specific location Square.
}
```

4- **public int getColDistance(Square location) { }**

- This int method calculates the column distance between the current square and another specific square (which is the location).
- it just have one parameter which is the Square location
- This method calculates the column distance between the current location and target location.

```
public int getColDistance(Square location) { //the method calculates
the column distance between 2 square objects .
   return this.col - location.col; //this line calculates the column
distance between the : current Square (which we use the key word this
to represent it), and another specific location Square.
}
```

5- public boolean isEmpty() {}

- this boolean method to check whether a square is empty or does have a piece.
- it doesn't have any parameters, it just have a returning value type which is true or false : if piece is = null (empty) return true else false.

```
public boolean isEmpty() { //To check if a square is empty (does not
contain any Piece)
   return piece == null;
}
```

6- public void clear() {}

- This void method doesn't return anything; what it does is clear or delete the previous location of the piece when it moves to a new location.
- piece = null (means empty)

```
public void clear() { //to clear the place of any piece after it
moves
   piece = null; //we will make the piece object null means empty (
removing the piece from the square)
   //we can use the keyword null to objects to give them a value
which is empty
}
```

7- public boolean isAtSameColumn(Square targetLocation) { }:

- This method returns a boolean value type, which will check if the square and the target location are in the same column.
- it has a one parameter which is the square targetLocation(new location)
- it return a boolean expression which will check if both locations on the same col

```
public boolean isAtSameColumn(Square targetLocation) { //This method
returns a boolean value, which check if this square and the target location
are in the same column
   return col == targetLocation.getCol();
}
```

8- public boolean isNeibourColumn(Square targetLocation) { }

- This method returns a boolean value type, which will check if the square and the target location are neighbors in the term of columns.
- it has a one parameter which is the square targetLocation(new location)
- it returns a boolean expression which will check if both locations are column neighbors or not.

```
public boolean isNeighborColumn(Square targetLocation) { //This method
returns a boolean value, checks if this square and the target location are
neighbor in terms of columns
   return col == targetLocation.getCol() || col + 1 ==
targetLocation.getCol() || col - 1 == targetLocation.getCol();  //we are
making if else statements to check this statement
   //if the current column is equal to the column of target location,  or
check the next column (column +1) is equal to the column of target location
or the (column -1) going back , is equal to the target location
   //if one of those cases is true it will stop checking the others. it
will return true, and the oppiste is false!
}
```

9- public boolean isAtLastRow(){ }

- This method returns a boolean value type, which will check if the piece reaches the last row (the pawn)
- it has a one parameter which is the (PieceColor color) : to hold the color of the piece
- it has nested if else statements : which will check if the piece is not empty so check the color if it's white and reach the last row which is 7. then check black color and reaches row =0

```
public boolean isAtLastRow(PieceColor color) {
   //we can use this method in : turning the pawn into Queen, and that happenese when the pawn reaches the last
     row

   if (this.piece != null) { //if the piece is not empty
       //checking if the piece on the square is at the last row for the given colur
       if (color == PieceColor.WHITE && this.piece.location.row == 7) { //check White if It's at the last row
           return true;
       }
       //check Black if It's at the last row
       return color == PieceColor.BLACK && this.piece.location.row == 0;
   }
   return false; //other than that return false
}
```

10- **public boolean isDiagonal(Square targetLocation){ }**

- This method returns a boolean value type, which will check if the square and the target location are on diagonal.
- it has a one parameter which is the (Square targetLocation): to hold the color of the piece
- It will return the absolute value of the difference of rows (current and target) and check if it's equal to the absolute value of difference of col (current and target).

```
public boolean isDiagonal(Square targetLocation) { //to check Diagonal ( if
this square and the target location are on a diagonal line)
   return (Math.abs(this.row - targetLocation.row) == Math.abs(this.col -
targetLocation.col));

   // return Math.abs(this.getRowDistance(square)) ==
Math.abs(this.getColDistance(square));

}
```

11- public void putNewQueen(PieceColor color) {}

- It creates a new queen.
- it has one parameter PieceColor color: The color of the piece that has reached the last row.
- It doesn't return anything (void).
- It takes the piece that has reached the last row and it creates a new Queen object and it places the queen in the same coordinate and color!

```
public void putNewQueen(PieceColor color) { //when a pawn reaches the last
row it will turn into a Queen
   //in fact it can be turned into any piece you lost (Queen , Bishop ,
Knight).
   this.piece = new Queen(color, this); //make this piece a new Queen piece
on the square for the given color whether it was black of white!
}
```

11- public int getDifRowCol(Square location) { }

- This method calculates the difference between the row and column of the current square and another square.
- it has one parameter which is the Square location.
- its returning an int value of the different between the row and col

```
public int getDifRowCol(Square location) { //this method calculates the
difference between the row and column of a current square(this) and another
square (used in Queen class)
   return location.getRow() - row;
}
```

12 - public String toString() { }

- **I made it override since the string representation is different at each piece**
- **it returns the if else condition if the piece is empty(null) return “ “ space else write the string representation of the chosen piece.**

```
@Override
// the string method returns a string representation of the square if It's
either empty or the piece's string representation
public String toString() {
   return this.piece == null ? " " : this.piece.toString();
}
```

ChessBoard Class :

1- public ChessBoard() { }

- default constructor which will inizilalize the chess board
- It places all the pieces in their own place
- There is no parameter.
- It doesn't return anything (void).
- I created 2 different for loops , one for each Pawn color , and added all the pieces in the correct squares by using the 2- dimensional array that I have created while making the ChessBoard.

```
public ChessBoard() { //default constructor
   //initializes the chess board (Board 2D) by creating a 2D array list of squres!
   for (int i = 0; i < 8; i++) {
       for (int j = 0; j < 8; j++) {
           Board2D[i][j] = new Square(i, j, this);
       }
   }
   /*
   setting up the initial arrangement of pieces on the Board 2D.
  1- The first 8 rows are white pieces and the next 8 rows are black pieces.
  2-it creates a pawn on row 6 and sets its color to black.
  3-It then creates a rook on row 7 and sets its color to black as well.
  4-next step it creates a knight on row 7 and sets its color to black as well.
  5-Finally, it creates two bishops on row 7 with colors set to white and places
them in their respective positions (one at each end).
    */
   for (int i = 0; i < 8; i++) { //White pieces
       Board2D[6][i].setPiece(new Pawn(PieceColor.BLACK, Board2D[6][i]));
   }
   Board2D[7][0].setPiece(new Rook(PieceColor.BLACK, Board2D[7][0]));
   Board2D[7][1].setPiece(new Knight(PieceColor.BLACK, Board2D[7][1]));
   Board2D[7][2].setPiece(new Bishop(PieceColor.BLACK, Board2D[7][2]));
   Board2D[7][3].setPiece(new Queen(PieceColor.BLACK, Board2D[7][3]));
   Board2D[7][4].setPiece(new King(PieceColor.BLACK, Board2D[7][4]));
   Board2D[7][5].setPiece(new Bishop(PieceColor.BLACK, Board2D[7][5]));
   Board2D[7][6].setPiece(new Knight(PieceColor.BLACK, Board2D[7][6]));
   Board2D[7][7].setPiece(new Rook(PieceColor.BLACK, Board2D[7][7]));

   for (int i = 0; i < 8; i++) { //Black pieces
       Board2D[1][i].setPiece(new Pawn(PieceColor.WHITE, Board2D[1][i]));
   }
   Board2D[0][0].setPiece(new Rook(PieceColor.WHITE, Board2D[0][0]));
   Board2D[0][1].setPiece(new Knight(PieceColor.WHITE, Board2D[0][1]));
   Board2D[0][2].setPiece(new Bishop(PieceColor.WHITE, Board2D[0][2]));
   Board2D[0][3].setPiece(new Queen(PieceColor.WHITE, Board2D[0][3]));
   Board2D[0][4].setPiece(new King(PieceColor.WHITE, Board2D[0][4]));
   Board2D[0][5].setPiece(new Bishop(PieceColor.WHITE, Board2D[0][5]));
   Board2D[0][6].setPiece(new Knight(PieceColor.WHITE, Board2D[0][6]));
   Board2D[0][7].setPiece(new Rook(PieceColor.WHITE, Board2D[0][7]));

}
```

2- private int getLetterIndex(char c) { }

- private method to get the letter index.

- it just has a one parameter which is the char
- Instead of many if-else statements I made it return a switch statement which has several cases. This method is used to convert the letter part of the position into the corresponding column index in the array and this allows for easy retrieval of squares and pieces using their positions. if the letter was A or a means its at the index 0 , then if the letter was B or b its at the index 1. then the default means if none of the above throws exceptions.
- returning the c which is the char

```
private int getLetterIndex(char c){
   // instead of many if else statements I made it to return a switch
statement which has several cases .
   //This method is used to convert the letter part of the position into the
corresponding column index in the array and this allows for easy retrieval of
squares and pieces using their positions
   return switch (c){
       case 'A','a' -> 0; //means if the letter is 'A' or 'a' return index 0
       case 'B', 'b' -> 1; // if the letter is 'B' or 'b', return index 1
       case 'C', 'c' -> 2; // if the letter is 'C' or 'c', return index 2
       case 'D', 'd' -> 3; // if the letter is 'D' or 'd', return index 3
       case 'E', 'e' -> 4; // if the letter is 'E' or 'e', return index 4
       case 'F', 'f' -> 5; // if the letter is 'F' or 'f', return index 5
       case 'G', 'g' -> 6; // if the letter is 'G' or 'g', return index 6
       case 'H', 'h' -> 7; // if the letter is 'H' or 'h', return index 7
       default -> throw new IllegalStateException("Unexpected value: " + c);
// If the letter is none of the above so at that case throw an exception
   };
}
```

3- public boolean isGameEnded() { }

- public boolean method to check if the game ended or not
- there is no parameters
- it works if there are no pieces on the board (whites and blacks) which means the board has to be empty.

```
public boolean isGameEnded() { //to check if the game has ended : when
there is no pieces on the board
   int numOfWhites = 16;
   int numOfBlacks = 16;
   return false;
}
```

4- public boolean isWhitePlaying() { }

- checks if its whites turn or not ?
- has no parameters

- returns a boolean expression whether the turn of the current playing piece color is equal to white piece so it means white is playing, colors are the same (White and white).

```
public boolean isWhitePlaying() { //this method checks if the turn variable
is equal to  PieceColor.WHITE. If the turn is PieceColor.WHITE, it means it
is currently white's turn to play, and the method returns true
   // otherwise, it returns false
   return turn == PieceColor.WHITE;
}
```

5- public void nextPlayer () { }

- This method alterns the turns. If White has played, the method will give the turn to the Black play
- There is no parameter.
- It doesn't return anything (void).
- If the value of whitePlaying is True, it will become false so it will be the Black player's turn.

```
public void nextPlayer() {  // Switch the current player ; because a teacm
can't play twice!
   whitePlaying = !isWhitePlaying();
}
```

6-public void switchTurn() {

- SwitchTurn method switches the turn between the two players.
- no parameters.
- sets the turn instance variable and check if turn is == white (it means it was white turn so now return black) else(means it was blacks turn) return white

```
public void switchTurn(){ // Switching the turn between the two players!
   turn = turn == PieceColor.WHITE ? PieceColor.BLACK : PieceColor.WHITE;
}
```

7- public square [ ] getSquaresBetween(Square location , Square targetLocation) { }

- Determines the squares between two given squares based on their positions.
- it has two parameters the current square location and the target location (that I want to move my piece into)
- it returns the array of squares between the two given locations!
- It considers different cases : same columns and neighbor col , or non neighbor columns , finally it goes through loops to iterate over the relevant rows or columns and store them in an array.

```
public Square[] getSquaresBetween(Square location, Square targetLocation) {
   /*
```

```
   This method determines the squares between location and targetLocation
based on their relative positions
    It considers three cases:
    1- when the squares are in the same column
    2-when they are in neighboring columns
     3-when they are not in the same column or neighboring columns, so It
does iterate over the rows or columns between the two squares and stores
the squares in the array
    */
   Square[] squares = new Square[8]; // array to store the squares between
location and targetLocation
   int num = 0; //num is used to keep track of the number of squares

   if (location.isAtSameColumn(targetLocation)) { //checking if the
location and the target location are at the same column
       int startRow = Math.min(location.getRow(), targetLocation.getRow())
+ 1;
       int endRow = Math.max(location.getRow(), targetLocation.getRow());

       for (int i = startRow; i < endRow; i++) { //it goes over the rows
between startRow and endRow
           squares[num] = Board2D[i][location.getCol()]; //store the square
in the array
           num++; //incrementing num
       }

       // this for loop will fill the remaining elements of the array with
the square strats from startRow - 1
       for (int x = num; x < 8; x++) {
           squares[x] = Board2D[startRow - 1][location.getCol()];
       }
   } else if (location.isNeighborColumn(targetLocation)) { //checking if
the location and the target location are neighbordes in the term of columns
       int startCol = Math.min(location.getCol(), targetLocation.getCol())
+ 1;
       int endCol = Math.max(location.getCol(), targetLocation.getCol());


       for (int i = startCol; i < endCol; i++) {
           squares[num] = Board2D[location.getRow()][i];
           num++;
       }

       for (int x = num; x < 8; x++) { // filling the remaining elements of
the array with the square from location.getRow()
           squares[x] = Board2D[location.getRow()][startCol - 1];
       }
   } else { // At this case the location and targetLocation are not in the
same column or neighboring columns
       int startRow = location.getRow();
       int startCol = location.getCol();
       int endRow = targetLocation.getRow();
       int endCol = targetLocation.getCol();
```

```
        int rowIncrement = (endRow > startRow) ? 1 : -1;
        int colIncrement = (endCol > startCol) ? 1 : -1;

        startRow += rowIncrement;
        startCol += colIncrement;

        while (startRow != endRow && startCol != endCol) { // Iterate over
the squares diagonally between location and targetLocation
            squares[num] = Board2D[startRow][startCol];
            num++;
            startRow += rowIncrement;
            startCol += colIncrement;
        }

        for (int x = num; x < 8; x++) { // Fill the remaining elements of
the array with the square from startRow and startCol
            squares[x] = Board2D[startRow][startCol];
        }
    }
    return squares; //return the array of squares
}
```

8- public Square getSquareAt(String from) { }

- this methods gets a square at a specific location on the board
- it has one parameter (from) which is the string of the position on the board like a1.
- it does return the square which located at a specified place on the board
- getLetter method helped here : it parses the input string to obtain the row and column, then retrieves the corresponding square from the Board 2D array.

```
public Square getSquareAt(String from) {
    /*
    The getSquareAt method use the getLetterIndex method to determine the
column index and retrieve the corresponding square from the Board2D array
based on the input position!
     */
    char letter = from.charAt(0);
    int num = Integer.parseInt(String.valueOf(from.charAt(1)));
    return Board2D[num-1][getLetterIndex(letter)];
}
```

9-public String toString() { }

- Generates a string implementation of the board
- returns a string representation of the board.
- It fills the borders of the ChessBoard with the numbers and letters.
- There is no parameter.
- Iterates over the Board2D array and constructs a string with the visual representation of the board and pieces

```
public String toString() {
   int num = 8; //inizilalizes the value of the num
   StringBuilder board = new StringBuilder(); //create a StringBuilder object named
board to build the string representation of the board
   board.append("      A   B   C   D   E   F   G   H " + "\n
---------------------------------"); //appending the tup row which is letters then
the -----
   for (int i = 7; i >= 0; i--) {
       /*
       Iterate over the rows of the Board2D array in reverse order (from 7 to 0)
using the variable i.
       For each row append the row number which is (num) followed by a vertical
line (|) to the board string
        */
       board.append("\n  ").append(num).append(" |");
       for (int j = 0; j < 8; j++) {
           Piece p = Board2D[i][j].getPiece();
           if (p == null) { //If p is null meaning there is no piece at that square
so append three spaces followed by a vertical line (" |") to the board string.
               board.append("   |");
               continue;
           }
           if (p.getColor() == PieceColor.WHITE) //checking colors
               board.append("
").append(Board2D[i][j].getPiece().toString()).append(" |");
           if (p.getColor() == PieceColor.BLACK)
               board.append("
").append(Board2D[i][j].getPiece().toString()).append(" |");
       }
       board.append(" ").append(num);
       board.append("\n    ---------------------------------"); //appending
horizontal line
       num -= 1; //we are decreasing the num variable by 1 to move to the next row

   }
   board.append("\n      A   B   C   D   E   F   G   H "); //appending the last row
which is letters
   return board.toString();
}
```

10 -public getPicecAt(String from) { }

- put the piece at the specific location on the board.

- it has one parameter : from which is a string representation on the position of piece on the board.
- getLetter method helped here : it parses the input string to obtain the row and column, then retrieves the corresponding square from the Board 2D array

```
public Piece getPieceAt(String from) { //getting a piece from a spicific
location
   char letter = from.charAt(0);
   int num = Integer.parseInt(String.valueOf(from.charAt(1)));
   System.out.println(getLetterIndex(letter) + " - " + num); //it will
print the column index and row index to know which place I have choosen!
   return Board2D[num-1][getLetterIndex(letter)].getPiece();
}
```

D) Defining Piece Hierarchy

1- Explain how the Main class benefits from polymorphism.

2- Explain which methods and classes can be defined abstractly in the Piece hierarchy.
3- Is there a code reuse in your implementation?

Ans :

1-
concept of polymorphism: is using the function or method in a different way, but the result would be the same, here they are all pieces, but they are different pieces!

This part of the Main class : Piece piece, allows to refer to objects of any subclass of the parent class Piece such as (Pawn , Knight , King , Queen , Bishop , Rook). Which means if I want my piece to be a pawn, it will be converted into a pawn, or If I want it to be Queen also it will be converted into a Queen piece.
That's the way that the main class is benefiting from the concept of polymorphism

2-

- The piece class is defined as an abstract class , and this allows us to make a abstract method inside this class (Mark : abstract methods can only be defined inside abstract classes or interfaces)

- we have a 3 abstract methods in the piece class which is :
  public abstract boolean canMove(String to);
  Since we defined the canMove method inside the Piece as abstract. is not the same for all pieces, this method will be inherited and overridden in each sub-class, which means it allows each specific subclass (any class of pieces) to define its only and unique valid and invalid move

  public abstract void move (String to);
  abstract means it has to be overridden in each of the pieces classes, I was going to not make it abstract until I saw a difference at the pawn class.
  This method occurs at the Pawn class but in a slightly different way in its movement which is : if the pawn reaches the last row, it will be replaced with a Queen!

  public abstract String toString();
  Since the String representation of each point will be different so I made it abstract at the parent class.

E) Implementing methods in Piece Hierarchy
Do the same as described in C for the classes in Piece hierarchy

All pieces implement the move(); and canMove(); and toString() method from the Piece class.
I was going to make the move method an inherited method but since the Pawn is moving differently(When it reaches the last row it becomes a Queen) so I decided to make it abstract since it will be different in one class.

```
public abstract void move(String to);
```

- makes the movement of the pieces possible to their new locations.
- it is the coordinate of the target movement.
- It takes the parameter as an input from the user in order to know the new coordinate. then it deletes the piece in the previous location and creates the same piece in the new location.
- Finally it sets the location of the piece in the new coordinate. (For the pawn the only difference for this method is that if the pawn is in the last row, the pawn will be replaced with a Queen.

```
public abstract boolean canMove(String to);
```

- To check whether a move is valid or invalid move.
- It takes the parameter as an input from the user in order to know the new coordinate of the new location.
- its abstract at each class since it will be overridden once again in all the subclasses (it's not like the move method which is just different in one class this method is different at all classes).

1- Piece Class :

```
public abstract class Piece  { //the parent class
```

```
   public PieceColor color; //implementing the color of piece from a class
calls PieceColor(enum class) (it could be WHITE =0  , Black =1)
   public Square location; // implementing the current location of the piece
on the chessboard!


   public Piece(PieceColor color, Square location) { //parametric constructor
to take the color  and  Square location as parameters
       // we are using this to avoid shadowing and set the color and location
       this.color = color;
       this.location = location; //location of the piece in the chessBoard at
a spisific square
       this.location.setPiece(this); // setting this piece on a specific
location on the chessboard
   }

   public PieceColor getColor() { //create a getter, so I can get the color
(the return value type of the getter is the same as the value! )
       return color; // returning the value type which is (the color of the
piece)
   }

   public abstract boolean canMove(String to); //Abstract method to check if
the move is a valid move or not!


   public abstract void move(String to); //abstract method to move the pieces
because it will change at each sub class
   protected boolean isEnemy(Piece p){ //this method could used in Bishop to
check if the given piece is an Enemy piece.
       return !(this.color == p.color); // return true if the colors are
different (they are enemy pieces)!
   }
   public abstract String toString(); //abstract method to string because it
will change at each sub class

}
```

2- Pawn Class :

```java
public class Pawn extends Piece { //the relation here is inheritance from the
Piece class (Parent class) . The pawn will have all functions that the piece
class has

   /*
     MARKS :
   1- Extends from Piece class
   2-first move for pawn can be two steps forward then it will be 1.
   3-eats a piece diagonally. (The very first diagonal)
   4-if the pawn reaches the last row it becomes a Queen!
    */


   boolean initialLocation = true; //the piece here hasn't been moved yet
   //if the pawn is at its initial location or has moved.

   public Pawn(PieceColor color, Square location) { //parameterized
constructor for Pawn
       super(color, location); //use the key word super to call the
parametric constructor from the Piece class (The parent class)
   }

   @Override
   public boolean canMove(String to) { //this method to check whether a pawn
have a valid move or not

       //initializes the validMove variable to false, it will be use to track
whether the move is valid or not for the pawn
       boolean validMove = false;
       Square targetLocation = location.getBoard().getSquareAt(to);
       /*
       using getRowDistance(location) so we can evaluate if the pawn is
moving forward
        or backward and how many rows it needs to move to reach the target
location.

       For example: white pawns can move forward most 2 squares from their
       initial position and by 1 square afterward. The getRowDistance method
helps
       determine if the pawn is attempting to move within the allowed
distance!
        */
       int rowDistance = targetLocation.getRowDistance(location); //Getting
the row difference between the current location and target location

       //checks if the target coordinate is at the same column
       if (this.location.isAtSameColumn(targetLocation)) {
           //for white check pawn is moving forward at most 2 Squares
           if (color == PieceColor.WHITE && rowDistance > 0 && rowDistance
<= 2) {
               if (rowDistance == 2) {
                   if (initialLocation) {
```

```
                        //pawn is moving twice , check two squares in front
are empty
                        Square[] between =
location.getBoard().getSquaresBetween(location, targetLocation);
                        validMove = targetLocation.isEmpty() &&
between[0].isEmpty();
                    }
                } else {
                    validMove = targetLocation.isEmpty();
                }
                return validMove;

            } else if (color == PieceColor.BLACK && rowDistance < 0 &&
rowDistance >= -2) { //for black also  check pawn is moving forward at most 2
Squares
                if (rowDistance == -2) {
                    if (initialLocation) {
                        //pawn is moving twice, check two squares in front are
empty
                        Square[] between =
location.getBoard().getSquaresBetween(location, targetLocation);
                        validMove = targetLocation.isEmpty() &&
between[0].isEmpty();
                    }
                } else {
                    validMove = targetLocation.isEmpty();
                }
            }
            // if the target column is not at the same column, it should be a
neighbour column
        } else if (this.location.isNeighborColumn(targetLocation)) {
            //pawn can only move to forward diagonals if there is an attack
            if (color == PieceColor.WHITE && rowDistance == 1) {
                validMove = !targetLocation.isEmpty() &&
targetLocation.getPiece().getColor() == PieceColor.BLACK;
            } else if (color == PieceColor.BLACK && rowDistance == -1) {
                validMove = !targetLocation.isEmpty() &&
targetLocation.getPiece().getColor() == PieceColor.WHITE;
            }

        }
        return validMove;
    }

    @Override
    public void move(String to) { //move method moves the pawn to a new
location on the chessboard
        Square targetLocation = location.getBoard().getSquareAt(to);
        //turning the pawn into Queen if its at the last row!
        if (targetLocation.isAtLastRow(color)) {
            targetLocation.putNewQueen(color); //we will turn the pawn into
Queen of the same color.
```

```
        } else { //if the pawn is not at the last row so (the condition fails)
else part will work
            // the code executes targetLocation.setPiece(this); this sets the
target location's piece as the current pawn itself
            targetLocation.setPiece(this);
        }
        location.clear(); //clear previous location of pawn, making it empty
        location = targetLocation; //update current location of pawn to the
target location
        location.getBoard().nextPlayer(); //proccesed to next player turn! for
example white will make the first move for sure then black will make a move ,
the same player can't play twice at the same round!
        //piece has been moved at least once
        initialLocation = false;

    }
    @Override //it is being overriden because its abstract at the parent class
because it changes at each class thats why I made it abstract method
    public String toString() { //to String Representation to have a string
representation of an object
        return color == PieceColor.WHITE ? "P" : "p"; //This is if else
statement (?) means if , (:) means else
        // if the color is white so write P plays , if black  p plays
    }
}
```

3- Bishop Piece :

```
public class Bishop extends Piece { //the relation here is inheritance from
the Piece class (Parent class) . The Bishop will have all functions that
the piece class has.

   // Bishop class represents a bishop chess piece

   /* MARKS :
   1- Extends from Piece class.
   2-How does it move? like letter X, so we have to check Diagonals!
   3- can't move when pawn is in front of it .

    */
   public Bishop(PieceColor color , Square location){ //parameterized
constructor for bishop that takes the color of the bishop and its initial
location.
       super(color, location); //use the key word super to call the
parametric constructor from the Piece class (The parent class)
   }

   //since it moves like ( X ) we have to check Diagonals !
        /*
       this method checks if the bishop can move to the specific location.
the method is Overrides the canMove method defined in the Piece class.
       returns true if the movee is valid, otherwise false .
       */
   @Override
   public boolean canMove(String to) {//method checks valid move or
invalid.
       boolean validMove;  //initializes the validMove variable to false,
it will be use to track whether the move is valid or not for the Bishop
       Square targetLocation = location.getBoard().getSquareAt(to);

       if (!location.isDiagonal(targetLocation)) { //this line of if
condition checks if the target location is not on a diagonal line
           validMove = false;
       } else {
           // get all the squares between the current location and the
target location
           Square[] between =
location.getBoard().getSquaresBetween(location, targetLocation);
           // Check if all the squares between are empty
           boolean allEmpty = true;
           if (between != null) {
               for (Square sq : between) {
                   if (!sq.isEmpty()) {
                       allEmpty = false;
                       break; //if any Square is not empty exit the loop
                   }
               }
           }
```

```
            //checking if the target location is empty or occupied by an
enemy piece(other team), we just have two teams : black and white
            validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != this.color) && allEmpty;
        }

        return validMove;
    }
    @Override
    public void move(String to) { //method to move the  Bishop piece
        Square targetLocation = location.getBoard().getSquareAt(to); //havig
a target location
        targetLocation.setPiece(this); //
        location.clear(); //clear previous location of pawn, making it empty
        location = targetLocation; //update current location of pawn to the
target location
        location.getBoard().nextPlayer(); //proccesed to next player turn!
for example white will make the first move for sure then black will make a
move , the same player can't play twice at the same round!
        //piece has been moved at least once
    }

    @Override //it is overriden because its abstract at the parent class
because it changes at each class thats why I made it abstract method
    public String toString() { //to String Representation to have a string
representation of an object
        return color == PieceColor.WHITE ? "B" : "b"; //This is if else
statement (?) means if , (:) means else
        // if the color is white so write B plays , if black  b plays
    }


}
```

4- enum Piece class :

I added this class because my project wasn't able to run and move pieces

```
public enum PieceColor {
   //it  does define two values: BLACK and WHITE, representing the colors
of chess pieces
   BLACK,
   WHITE
}
```

5- Queen

```
public class Queen extends Piece { //inheriting all functions from the
piece class (the parent class)
   //Strogest Piece!!
   //it moves like everything but not L (can't move like knight)
   //moves unlimited number of steps
   boolean initialLocation = true;
   public Queen (PieceColor color , Square location ) { //a constructor for
the queen class which has the piece color and its location
       super(color, location); //we are using the key word super to call
the constructor of the parent class
   }
   @Override
   public boolean canMove(String to) { //this method is used to check
weather the move is valid or not!
       //it moves like everything but not L (can't move like knight)
       //moves unlimited number of steps
       //so we will check of the pieces moves but not the knight!
       boolean validMove = false;
       Square targetLocation = location.getBoard().getSquareAt(to);
       int rowDistance = targetLocation.getRowDistance(location);
       int colDistance = targetLocation.getColDistance(location);
       int difRowCol = targetLocation.getDifRowCol(location);
       if (this.location.isAtSameColumn(targetLocation)) {
           if (rowDistance > 0 && rowDistance <= 8) {
               if(rowDistance == 1) {
                   validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
               }
               else if(initialLocation) {
                   Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                   validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
```

```
                }
                return validMove;
            } else if (rowDistance < 0 && rowDistance >= -8) {
                if(rowDistance == -1) {
                    validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
                }
                else if(initialLocation) {
                    Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                    validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
                }
            }
        } else if (this.location.isNeighborColumn(targetLocation) &&
location.getRow() == targetLocation.getRow()) {
            if (colDistance > 0 && colDistance <= 8) {
                if(colDistance == 1) {
                    validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
                }
                else if(initialLocation) {
                    Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                    validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
                }
                return validMove;
            } else if (colDistance < 0 && colDistance >= -8) {
                if(colDistance == -1) {
                    validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
                }
                else if(initialLocation) {
                    Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                    validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
                }
            }
        } else if(rowDistance == colDistance) {
            if(difRowCol == 1 || difRowCol == -1){
                validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
            }
            else if(initialLocation) {
```

```
                Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
            }
            return validMove;
        } else if (rowDistance == -colDistance) {
            if(difRowCol == 1 || difRowCol == -1){
                validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
            }
            else if(initialLocation) {
                Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
            }
        }
        return validMove;
    }

    @Override
    public void move(String to) { //method to move the  Queen piece
        Square targetLocation = location.getBoard().getSquareAt(to); //havig
a target location
        targetLocation.setPiece(this); //
        location.clear(); //clear previous location of Queen, making it
empty
        location = targetLocation; //update current location of Queen to the
target location
        location.getBoard().nextPlayer(); //proccesed to next player turn!
for example white will make the first move for sure then black will make a
move , the same player can't play twice at the same round!
    }

    @Override //it is overriden because its abstract at the parent class
because it changes at each class thats why I made it abstract method
    public String toString() {//to String Representation to have a string
representation of an object
        return color == PieceColor.WHITE ? "Q" : "q"; //This is if else
statement (?) means if , (:) means else
        // if the color is white so write Q plays , if black  q plays
    }
}
```

6- Rook

```
public class Rook extends Piece { //inheriting all functions from the piece
class (the parent class)
   //moves like + with unlimited number of steps
   //you can move like this or . . . . . . . . . . . .back and forth
   /*or also this way back and forth
   .
   .
   .
   .
   .
    */

   boolean initialLocation = true;
   public Rook (PieceColor color , Square location){ //parameterized
constructor.
       super(color, location); //use the key word super to call the
parametric constructor from the Piece class (The parent class)
   }
   @Override
   public boolean canMove(String to) { //checks if a move is valid or not!

       /* 1
              since it moves like + back and forth we will check first :
       1- the target location is in the same column as the current
location:
       2- If they are in the same column, it proceeds to check the row
distance between them: If the row distance is 1 and the target location is
either empty or occupied by an opponent's piece, the move is considered
valid.
       3- if the row distance is greater than 1, it further checks if the
initial location flag is set to true. If it is, it checks the squares
between the current location and the target location to ensure they are all
empty. If they are, the move is considered valid.

        */

       /* 2
       then if the target location is not in the same column as the current
location, it checks if they are neighboring columns and have the same row
check :

       1- If they are neighboring columns and have the same row, it
proceeds to check the column distance between them.
       2- If the column distance is 1 and the target location is either
empty or occupied by an opponent's piece, the move is considered valid.
       3- If the column distance is greater than 1, it follows the same
logic as described in step 1 for moves in the same column.

        */
```

```
        // 3 If none of the above conditions is true, the move is considered
invalid, and the validMove variable will still false

        boolean validMove = false;
        Square targetLocation = location.getBoard().getSquareAt(to);
        int rowDistance = targetLocation.getRowDistance(location);
        int colDistance = targetLocation.getColDistance(location);
        if (this.location.isAtSameColumn(targetLocation)) {
            if (rowDistance > 0 && rowDistance <= 8) {
                if(rowDistance == 1) {
                    validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
                }
                else if(initialLocation) {
                    Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                    validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
                }
                return validMove;
            } else if (rowDistance < 0 && rowDistance >= -8) {
                if(rowDistance == -1) {
                    validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
                }
                else if(initialLocation) {
                    Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                    validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
                }
            }
        } else if (this.location.isNeighborColumn(targetLocation) &&
location.getRow() == targetLocation.getRow()) {
            if (colDistance > 0 && colDistance <= 8) {
                if(colDistance == 1) {
                    validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
                }
                else if(initialLocation) {
                    Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                    validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
                }
                return validMove;
            } else if (colDistance < 0 && colDistance >= -8) {
```

```
                if(colDistance == -1) {
                    validMove = targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color;
                }
                else if(initialLocation) {
                    Square[] between =
location.getBoard().getSquaresBetween(location,targetLocation);
                    validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color) && (between[0].isEmpty() &&
between[1].isEmpty() && between[2].isEmpty() && between[3].isEmpty() &&
between[4].isEmpty() && between[5].isEmpty() && between[6].isEmpty());
                }
            }
        }
        return validMove;
    }

    @Override
    public void move(String to) { //method to move the  Rook piece
        Square targetLocation = location.getBoard().getSquareAt(to); //havig
a target location
        targetLocation.setPiece(this);
        location.clear(); //clear previous location of Rook, making it empty
        location = targetLocation; //update current location of Rook to the
target location
        location.getBoard().nextPlayer(); //proccesed to next player turn!
for example white will make the first move for sure then black will make a
move , the same player can't play twice at the same round!
    }
    @Override
    public String toString() {
        return color == PieceColor.WHITE ? "R" : "r"; //This is if else
statement (?) means if , (:) means else
        // if the color is white so write R plays , if black  r plays
    }


}
```

7 - King :

```
public class King extends Piece {
   //all the game we are trying to protect it !
   //it can move just one square forward , backwards, right , left to do
the checkmate! But checkmate isn't implemented in the game!!

   public King(PieceColor color , Square location){ //parameterized
constructor.
       super(color, location); //use the key word super to call the
parametric constructor from the Piece class (The parent class)
   }
   @Override
   public boolean canMove(String to) { //to check if the king has a valid
or invalid move!
       /*
       Checks the following conditions :
      1- if the target location at the sameColumn as the location , if so :
       It checks if the row distance is either 1 or -1, indicating a valid
move in the same column.
       then If the row distance meets the condition, it checks if the
target location is either empty or occupied by an opponent's piece. If it
is, the move is considered valid.

       2- If the target location is on a neighbor column with the location:
       - It checks if the column distance is 1 or -1.
       - If the column distance meets the condition then it checks if the
target location is either empty or occupied by another  enemy piece .
        If it is, the move is considered valid.
        */
       boolean validMove = false;
       Square targetLocation = location.getBoard().getSquareAt(to);
       int rowDistance = targetLocation.getRowDistance(location);
       int colDistance = targetLocation.getColDistance(location);
       if(this.location.isAtSameColumn(targetLocation)) {
           if(rowDistance == 1 || rowDistance == -1) {
               validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color);
           }
       } else if(this.location.isNeighborColumn(targetLocation)) {
           if(colDistance == 1 || colDistance == -1) {
               validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color);
           }
       }
       return validMove;
   }

   @Override
   public void move(String to) { //move method to move the piece
       Square targetLocation = location.getBoard().getSquareAt(to);
//getting the square target location
```

```
        targetLocation.setPiece(this); //sets the piece on the target
location
        location.clear(); //delets the piece which was on the prevoius
location
        location = targetLocation;
        location.getBoard().nextPlayer(); //next players turn!
    }
    @Override
    public String toString() { //To string representation to represent the
piece
        // it is overriden because its abstract at the parent class because
it changes at each class thats why I made it abstract method
        return color == PieceColor.WHITE ? "K" : "k"; //This is if else
statement (?) means if , (:) means else
        // if the color is white so write K plays , if black  k plays
    }
}
```

8- Knight :

```
public class Knight extends Piece {  //inheriting all functions from the
piece class (the parent class)

    //limited steps
    //knight moves like letter L
    //the only piece can move even if there is a pawn in front of it
    // one two three then left or right
    //we have 2 knights
    //It can jump and make letter L

    public Knight (PieceColor color , Square location){ //parameterized
constructor.
        super(color, location); //use the key word super to call the
parametric constructor from the Piece class (The parent class)
    }

    @Override
    public boolean canMove(String to) { //checks if a knight have a valid or
invalid move!
        /*
         It determines the row and column distance between the current
location and the target location.
         1-If the row distance is 1 or -1 and the column distance is 2 or
-2, or if the row distance is 2 or -2 and the column distance is 1 or -1,
the move is considered valid.
          2-It checks if the target location is either empty or occupied by
an enemy piece.
          The method returns true if the move is valid and false otherwise.
          The implementation of other methods and classes used in the code
is not provided.
```

```
        */
        boolean validMove = false;
        Square targetLocation = location.getBoard().getSquareAt(to);
        int rowDistance = targetLocation.getRowDistance(location);
        int colDistance = targetLocation.getColDistance(location);
        if((rowDistance == 1 || rowDistance == -1) && (colDistance == 2 ||
colDistance == -2)) {
            validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color);
        }else if((rowDistance == 2 || rowDistance == -2) && (colDistance ==
1 || colDistance == -1)) {
            validMove = (targetLocation.isEmpty() ||
targetLocation.getPiece().getColor() != color);
        }
        return validMove;
    }

    @Override
    public void move(String to) { //method to move the  Knight piece
        Square targetLocation = location.getBoard().getSquareAt(to); //havig
a target location
        targetLocation.setPiece(this);
        location.clear(); //clear previous location of Knight, making it
empty
        location = targetLocation; //update current location of Knight to
the target location
        location.getBoard().nextPlayer(); //proccesed to next player turn!
for example white will make the first move for sure then black will make a
move , the same player can't play twice at the same round!
    }
    @Override
    public String toString() { //To string representation to represent the
piece
        // it is overriden because its abstract at the parent class because
it changes at each class thats why I made it abstract method
        return color == PieceColor.WHITE ? "N" : "n"; //This is if else
statement (?) means if , (:) means else
        // if the color is white so write N plays , if black  n  plays
    }
}
```