

Abstract

The project 'Chess Game Assignment' implements a classic version of console-based Chess which follow the basic rules of chess, and all the chess pieces only move according to valid moves for that piece. Our implementation of Chess is for two players. It is played on an 8x8 checked board, with squares have two colors white and black.

Design

My code has simple design that take in consideration OOP principles such as inheritance, encapsulation and abstraction.

- We have abstract class 'Piece', which is the super class of all chess pieces types which are six subclasses: 'Pawn', 'Bishop', 'Knight', 'Rook', 'Queen' and 'King'.

- 'Board' class, which response of create the board of game and put the pieces on it. Board object is an 8x8 array of 'Piece' objects.

- We have 'Point' class which is response of presentation of any piece location, that each point has row and column to indicate piece location.

- 'PointFactory' is a utility class that create points. Point created when an move done, so we need to deal with start point and end point instead of two rows and two columns.

- 'Mover' class which does the general implementation on any move in the game

- 'ChessGame' class is where the game played and finished.

- 'Main' class is where the main method executed.

Design patterns

-Singleton design pattern, The board is a singleton because there is only one board in a game of chess, and that done by make the constructor private and instantiate only one object inside the Board class itself called '*boardInstance*' which is final to prevent it from being modified and static to make it associated with the class itself and we can access this instance from outside the class by use '*getInstance()*' method.

Implementation

-Color enum:

Defines two possible values 'White' and 'Black', these values can be used to set and get the color of pieces. these values can be used to set and get the color of piece.

-Board Class:

-Has one attribute '*theBoard*', and it's getter *getTheBoard()*

-I instantiate only one instance called '*boardInstance*', and constructor is private so no more instances will be made.

-*getInstance()*: get the '*boardInstance*'.

-*getPiece(int row, int col)*: method take coordinates of any piece/ square on the board and return Piece.

-*move(Point from, Point to)*: method that doing the move on the board with no checking or testing, so we call this method after testing the correctness of the move.

-*createTheBoard()*: Create the 8x8 array of pieces and fill it with pieces ordered on the board

-ChessGame Class:

-start(): start the game by taking each player move and check the king state after each move.

-makeTheBoard(): Call the *boardInstance* and print the board.

-makePlayerMove(Color player): take inputs from player, call 'checkInputs' and call the 'move' function.

checkInputs(String[] inputs, Color player): Check that the start point is not null so the moving piece != null. Keep asking for inputs if player moving piece that not same to his color or he entered not valid inputs. This method return the valid inputs from the user.

-notValidInputs(String[] inputs): if user entered any thing not correct about what the order of inputs should be: moving startPointName endPointName.

-isValidPointName(String input): Make sure that the single point name is correct in range of the board coordinates.

-move(String start, String end): Determine the type of piece and call the mover to move it.

-Mover Class:

Determine This class is response of any move happened in the game so even that each class of pieces has method called 'isValidMove' and the board it self has method called 'move', but the calling of these two methods and connecting them together is really happened is the mover, and any possible extending of moves such as adding castling move, will be in this class.

-Piece Class:

```
6 inheritors
public abstract class Piece {
    8 usages
    protected Color color;

    8 usages 6 implementations
    public abstract boolean isValidMove(Point startPoint, Point endPoint);

    12 usages
    public Color getColor() { return color; }

    protected Piece(Color color) {
        this.color = color;
    }

    protected Piece() {
    }
}
```

-It is the super class of all pieces. Subclasses are inheritance the Color, `isValidMove(Point startPoint, Point endPoint)` and `getColor()` from there parent.

-All pieces types (subclasses) are implements `isValidMove(Point startPoint, Point endPoint)` method to check if the move they asked to do is valid for there type or not. It is return boolean.

-When sub-class's object made (such as an pawn) it determine the color of itself on it's own constructor (Pawn class constructor) which is pass it to the superclass constructor by using 'super' keyword to set the color of that piece on superclass. For example:

```
board[1][i] = new Pawn(black);
```

```
protected Piece(Color color) {
    this.color = color;
}
```

```
2 usages
public Pawn(Color color) {
    super(color);
}
```

-An pawn is made here and it Is-a piece that has black color.

-Subclasses of 'Piece':

1-Pawn:

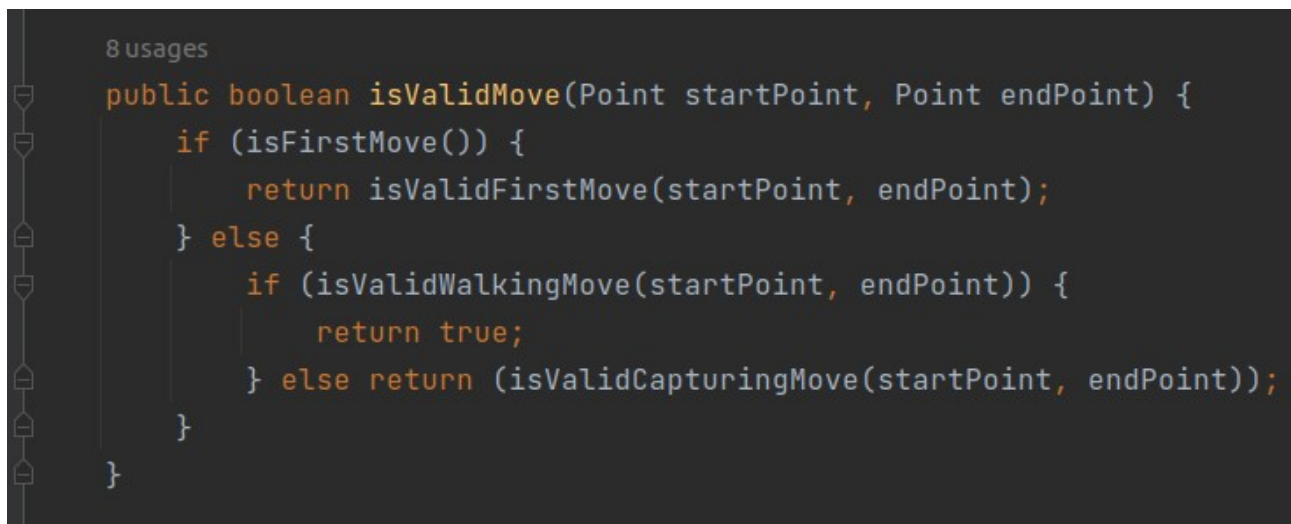
- **isValidMove(Point startPoint, Point endPoint)**: it is the backbone method of all subclasses with different implementation.

In Pawn it is check if the pawn piece is moving for the first time, that the pawn can move two steps forward if it was his first move, after that it can move only one step forward.

So we have two states of pawn piece: moving the first move OR it is not the first move.

If it was the first move then check if it is valid move by:

isValidFirstMove(startPoint, endPoint). Else if it is not the first move then it must be normal walking move or capturing move, so we check both situations: **isValidWalkingMove(startPoint, endPoint)**, **isValidCapturingMove(startPoint, endPoint)**.



```
8 usages
public boolean isValidMove(Point startPoint, Point endPoint) {
    if (isFirstMove()) {
        return isValidFirstMove(startPoint, endPoint);
    } else {
        if (isValidWalkingMove(startPoint, endPoint)) {
            return true;
        } else return (isValidCapturingMove(startPoint, endPoint));
    }
}
```

- **isFirstMove()**: each pawn piece has integer attribute 'moves' that increased after first valid move done, and this method check if moves still equal zero or not.

- **isValidFirstMove(Point start, Point end)**: check if the first move done vertically forward and if it not more than two steps. It increase 'moves' by one if it returning true.

- **isValidWalkingMove(Point start, Point end)**: check if the end point is empty so the pawn can move for if, and if it is empty we checked if it is only one step forward.

- **isValidCapturingMove(Point start, Point end)**: check if end point has an piece (not null) and if the move is valid move diagonally, after check these two, we check if the piece in there has the opposite color of the moving piece.

```

1 usage
private boolean isValidCapturingMove(Point start, Point end) {
    Color movingPieceColor = start.getPiece().getColor();
    if ((end.getPiece() != null && isValidDiagonalMove(start, end))) {
        Color capturedPieceColor = end.getPiece().getColor();

        return (movingPieceColor != capturedPieceColor);
    } else return false;
}

```

-**isValidDiagonalMove(Point start, Point end)**: check if the pawn move is only one step in diagonal line (forward and backward).

2-Bishop:

-**isValidMove(Point startPoint, Point endPoint)**: it check first if the the bishop move in valid diagonal move and if the path of moving is clear, if these two are true then we have two possible types of move. It might be walking move or capturing move. So we check if the end point was empty then it is walking move and if not then it capturing move so we checked the color of the piece on end point, it must be the opposite color of the moving piece to be valid capturing move.

```

8 usages
public boolean isValidMove(Point startPoint, Point endPoint) {
    Color movingPieceColor = startPoint.getPiece().getColor();
    Piece endPointPiece = endPoint.getPiece();
    if (isDiagonalMove(startPoint, endPoint) && isClearPath(startPoint, endPoint)) {
        if (endPointPiece == null) {
            return true;
        } else {
            Color capturedPieceColor = endPointPiece.getColor();
            return (movingPieceColor != capturedPieceColor);
        }
    } else return false;
}

```

-**isDiagonalMove(Point start, Point end)**: check if the bishop moving diagonally.

-**isClearPath(Point start, Point end)**: to fill a list of pieces that represent the path between the start and end point (all pieces must be null to be clear path) we have four different calculation based on direction of move: upward right move, upward left move, downward right move and downward left move.

-**isUpwardRightMove(Point start, Point end), isUpwardLeftMove(Point start, Point end), isDownwardRightMove(Point start, Point end), isDownwardLeftMove(Point start, Point end)**: These methods determine the direction of move.

-**getUpwardRightPath(Point start, Point end)**: When moving upward to the right this method does the calculation to return a list of pieces that are between start and end point.

-**getUpwardLeftPath(Point start, Point end)**: When moving upward to the left this method does the calculation to return a list of pieces that are between start and end point.

-**getDownwardRightPath(Point start, Point end)**: When moving downward to the right this method does the calculation to return a list of pieces that are between start and end point.

-**getDownwardLeftPath(Point start, Point end)**: When moving downward to the left this method does the calculation to return a list of pieces that are between start and end point.

```
1 usage
private boolean isClearPath(Point start, Point end) {
    List<Piece> inBetweenPath;
    if (isUpwardRightMove(start, end)) {
        inBetweenPath = getUpwardRightPath(start, end);
    } else if (isDownwardLeftMove(start, end)) {
        inBetweenPath = getDownwardLeftPath(start, end);
    } else if (isUpwardLeftMove(start, end)) {
        inBetweenPath = getUpwardLeftPath(start, end);
    } else if (isDownwardRightMove(start, end)) {
        inBetweenPath = getDownwardRightPath(start, end);
    } else
        inBetweenPath = null;
    int x = 0;
    for (Piece piece : inBetweenPath) {
        if (piece == null) {
            x++;
        }
    }
    return x == inBetweenPath.size();
}
```

3-Rook:

-**isValidMove(Point startPoint, Point endPoint)**: it check first if the the rook move in valid straight move and if the path of moving is clear, if these two are true then we have two possible types of move. It might be walking move or capturing move. So we check if the end point was empty then it is walking move and if not then it capturing move so we checked the color of the piece on

end point, it must be the opposite color of the moving piece to be valid capturing move.

8 usages

```
public boolean isValidMove(Point startPoint, Point endPoint) {
    Color movingPieceColor = startPoint.getPiece().getColor();
    Piece endPointPiece = endPoint.getPiece();

    if (isStraightMove(startPoint, endPoint) && isClearPath(startPoint, endPoint)) {
        if (endPointPiece == null) {
            return true;
        } else {
            Color capturedPieceColor = endPointPiece.getColor();
            return (movingPieceColor != capturedPieceColor);
        }
    } else return false;
}
```

-isStraightMove(Point start, Point end): check if the rook moving straight regardless of direction.

-isClearPath(Point start, Point end): to fill a list of pieces that represent the path between the start and end point (all pieces must be null to be clear path) we have two different calculation based on the four direction of move. First case when the row or the column coordinate is increasing and the second case when it is decreasing, for example when moving to the right column coordinate is increase and when moving downward the row coordinate is decrease. So that I make method of increasing case: getPathOfRightOrDownMove(int start, int end) and other method of decreasing coordinates cases: getPathOfLeftOrUpMove(int start, int end). Hafter get the path we check if it is clear using two methods: isHorizontalPathClear(List<Integer> path, int row), isVerticalPathClear(List<Integer> path, int column).

-isRightward(start, end), isDownward(start, end), isLeftward(start, end), isUpward(start, end): These methods determine the direction of move.

-getPathOfRightOrDownMove(int start, int end): it take the coordinate of increasing row or column and return a list of pieces that are located between.

-getPathOfLeftOrUpMove(int start, int end): it take the coordinate of decreasing row or column and return a list of pieces that are located between.

-isHorizontalPathClear(List<Integer> path, int row), isVerticalPathClear(List<Integer> path, int column).


```

1 usage
private boolean isClearPath(Point start, Point end) {
    List<Integer> inBetweenPath;
    if (isRightward(start, end)) {
        inBetweenPath = getPathOfRightOrDownMove(start.getCol(), end.getCol());
        return isHorizontalPathClear(inBetweenPath, start.getRow());
    } else if (isDownward(start, end)) {
        inBetweenPath = getPathOfRightOrDownMove(start.getRow(), end.getRow());
        return isVerticalPathClear(inBetweenPath, start.getCol());
    } else if (isLeftward(start, end)) {
        inBetweenPath = getPathOfLeftOrUpMove(start.getCol(), end.getCol());
        return isHorizontalPathClear(inBetweenPath, start.getRow());
    } else if (isUpward(start, end)) {
        inBetweenPath = getPathOfLeftOrUpMove(start.getRow(), end.getRow());
        return isVerticalPathClear(inBetweenPath, start.getCol());
    } else {
        return false;
    }
}
}

```

4-Knight:

-**isValidMove(Point startPoint, Point endPoint)**: it check first if the the knight move in valid L move. if true then we have two possible types of move. It might be walking move or capturing move. So we check if the end point was empty then it is walking move and if not then it capturing move so we checked the color of the piece on end point, it must be the opposite color of the moving piece to be valid capturing move.

8 usages

```
public boolean isValidMove(Point startPoint, Point endPoint) {
    Color movingPieceColor = startPoint.getPiece().getColor();
    Piece endPiece = endPoint.getPiece();
    if (isLMove(startPoint, endPoint)) {
        if (endPiece == null) {
            return true;
        } else {
            Color capturedPieceColor = endPiece.getColor();
            return (movingPieceColor != capturedPieceColor);
        }
    } else return false;
}
```

-isLMove(Point start, Point end): return if it vertical L move or horizontal L move.

-private boolean isVerticalL(Point start, Point end): Check if it is vertical L move.

-private boolean isHorizontalL(Point start, Point end): Check if it is horizontal L move.

5-King

-isValidMove(Point startPoint, Point endPoint): it check first if the the King move only one step. if true then we have two possible types of move. It might be walking move or capturing move. So we check if the end point was empty then it is walking move and if not then it capturing move so we checked the color of the piece on end point, it must be the opposite color of the moving piece to be valid capturing move.

8 usages

```
public boolean isValidMove(Point startPoint, Point endPoint) {  
    Color movingPieceColor = startPoint.getPiece().getColor();  
    Piece endPointPiece = endPoint.getPiece();  
    if (isOneStepMove(startPoint, endPoint)) {  
        if (endPointPiece == null) {  
            return true;  
        } else {  
            Color capturedPieceColor = endPointPiece.getColor();  
            return (movingPieceColor != capturedPieceColor);  
        }  
    } else return false;  
}
```

isOneStepMove(Point start, Point end): possible one step are four cases:

- isOneStepVertically(Point start, Point end).
- isOneStepHorizontally(Point start, Point end).
- isOneStepUpwardDiagonally(Point start, Point end).
- isOneStepDownwardDiagonally(Point start, Point end).

1 usage

```
private boolean isOneStepMove(Point start, Point end) {  
  
    if (isOneStepVertically(start, end)) {  
        return true;  
    } else if (isOneStepHorizontally(start, end)) {  
        return true;  
    } else if (isOneStepUpwardDiagonally(start, end)) {  
        return true;  
    } else return isOneStepDownwardDiagonally(start, end);  
}
```

6-Queen

-isValidMove(Point startPoint, Point endPoint): Queen can move as a rook piece and as a bishop piece so I passes the start point and end point to the 'isVaildMove' method of both rook and bishop so if any of them return true then it is a valid move for queen too.

8 usages

```
public boolean isValidMove(Point startPoint, Point endPoint) {  
    Rook rook = new Rook( color: null);  
    Bishop bishop = new Bishop( color: null);  
    return rook.isValidMove(startPoint, endPoint) ||  
           bishop.isValidMove(startPoint, endPoint);  
}
```