CHESS GAME

Bahaeddin Ahmad Ibrahim Hammad
ATYPON

Outlines

Introduction

In this project, I implemented a Chess game and added most of its features for it.

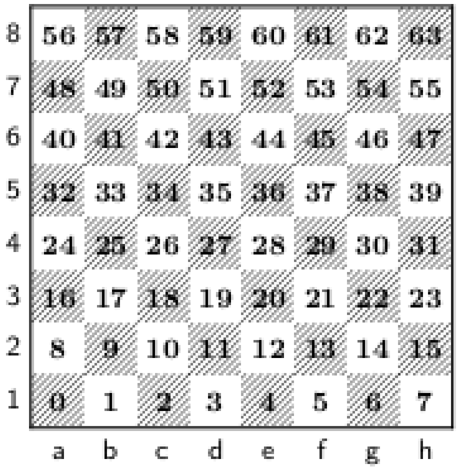
we will start by explaining what we had done in the project, at first I used some classes to Customize the methods in the project as we used object-oriented design, design patterns, and clean code principles.

I divided the work into 3 packages:

- Pieces
- Frame
- Game

And we will follow this order in the documentation.

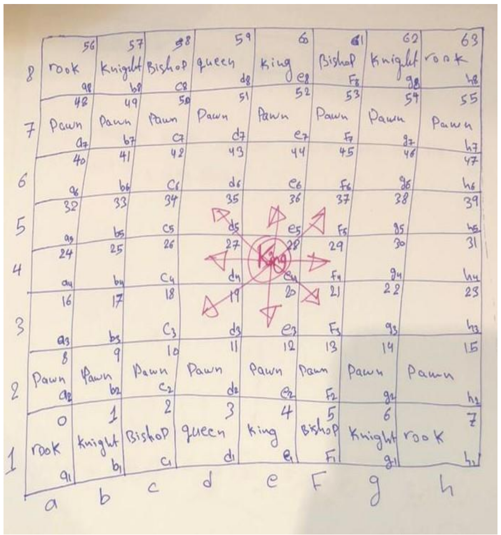
- I used a one-dimensional array to arrange the spots, the order is shown in the following figure:



8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	8	9	10	11	12	13	14	15
1	0	1	2	3	4	5	6	7
	a	b	c	d	e	f	g	h

*An illustration: spot number zero is (a1) for the user, and that will be explained further in the next parts.

- We defined a primary location for each piece and determined its movement by mathematical processes



An example shown in this photo,

The king has many movements each

One will have a different value for

Difference between the spot(index)

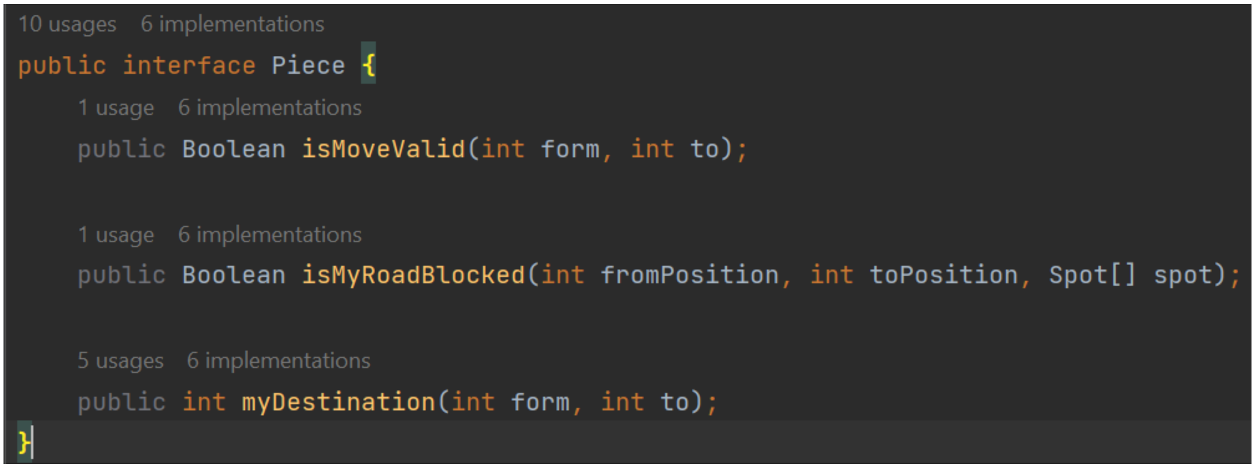
28 -> 29: 1

28 ->27: -1

28-> 37: 9

Pieces

I created an interface called Piece that contains the methods that will be used in pieces classes (king, queen, etc.…)



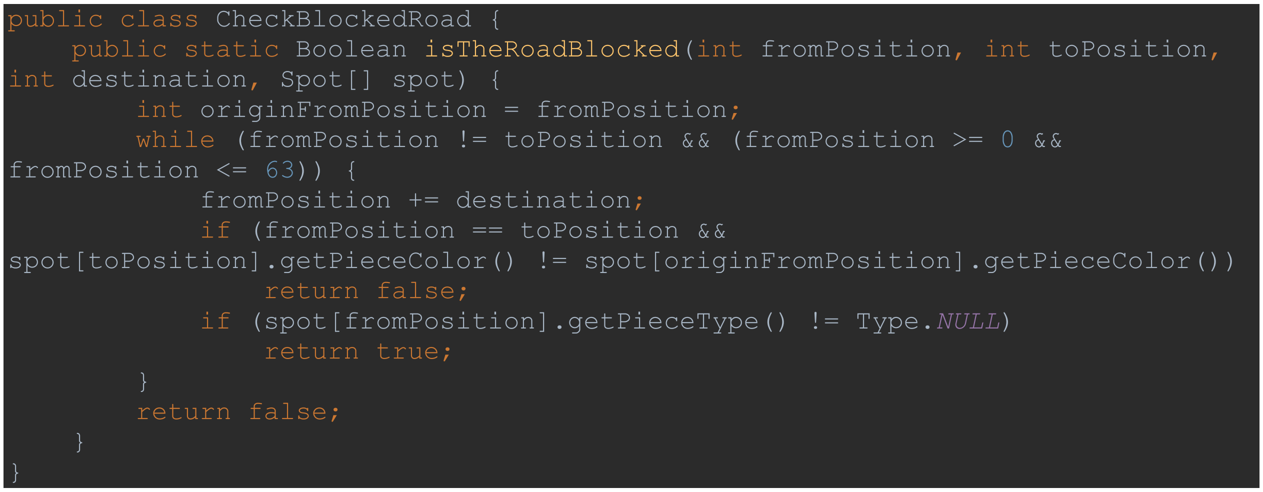
```
10 usages   6 implementations
public interface Piece {
    1 usage   6 implementations
    public Boolean isMoveValid(int form, int to);

    1 usage   6 implementations
    public Boolean isMyRoadBlocked(int fromPosition, int toPosition, Spot[] spot);

    5 usages   6 implementations
    public int myDestination(int form, int to);
}
```

Three main methods are shown in the picture

- isMoveValid : serves to check whether the move is correct for this piece or not
- myDestination : correlates between start point and end point spot numbers to know the correct path to move through (it chooses from two options or more)
- isMyRoadBlocked : serves to check whether there is any piece impaired the intended destination or not depending on the piece

CheckBlockedRoad

I created a class called CheckRoadBlocked that contains one static method used to determine whether the road is blocked by an enemy or friend or its not blocked and because all the pieces have the same method to check whether the road is blocked or not we created a static method to use it in all pieces, here is an overview of the class and the method

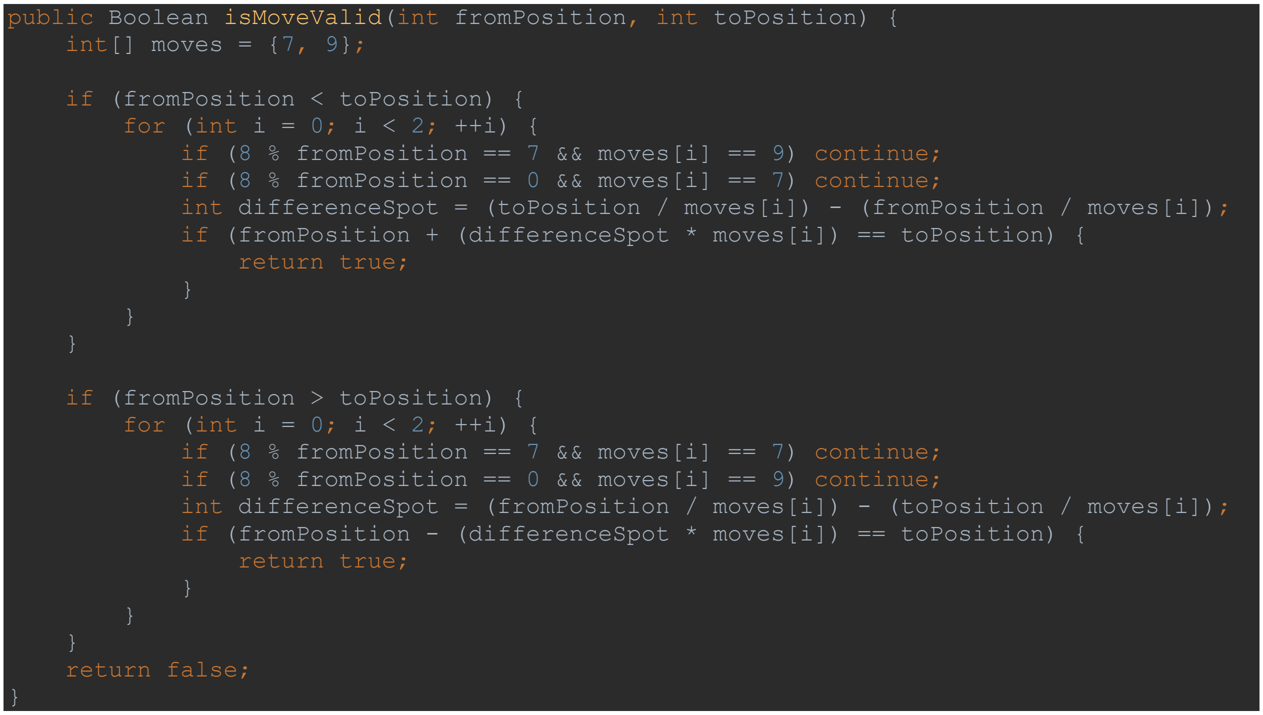


```
public class CheckBlockedRoad {
    public static Boolean isTheRoadBlocked(int fromPosition, int toPosition,
int destination, Spot[] spot) {
        int originFromPosition = fromPosition;
        while (fromPosition != toPosition && (fromPosition >= 0 &&
fromPosition <= 63)) {
            fromPosition += destination;
            if (fromPosition == toPosition &&
spot[toPosition].getPieceColor() != spot[originFromPosition].getPieceColor())
                return false;
            if (spot[fromPosition].getPieceType() != Type.NULL)
                return true;
        }
        return false;
    }
}
```

now I’ll show the used methods in each piece class

Bishop:

bishop has only two possible movements which create a destination that has a difference between spot indexes equal to 7 or 9 to the upper diagonal direction and -7 or -9 to the lower diagonal direction



```java
public Boolean isMoveValid(int fromPosition, int toPosition) {
    int[] moves = {7, 9};

    if (fromPosition < toPosition) {
        for (int i = 0; i < 2; ++i) {
            if (8 % fromPosition == 7 && moves[i] == 9) continue;
            if (8 % fromPosition == 0 && moves[i] == 7) continue;
            int differenceSpot = (toPosition / moves[i]) - (fromPosition / moves[i]);
            if (fromPosition + (differenceSpot * moves[i]) == toPosition) {
                return true;
            }
        }
    }

    if (fromPosition > toPosition) {
        for (int i = 0; i < 2; ++i) {
            if (8 % fromPosition == 7 && moves[i] == 7) continue;
            if (8 % fromPosition == 0 && moves[i] == 9) continue;
            int differenceSpot = (fromPosition / moves[i]) - (toPosition / moves[i]);
            if (fromPosition - (differenceSpot * moves[i]) == toPosition) {
                return true;
            }
        }
    }
    return false;
}
```

isMoveValid() will accept the correct movement:

- forward right diagonal
- forward left diagonal

```
public int myDestination(int fromPosition, int toPosition) {
    int[] moves = {7, 9};

    if (fromPosition < toPosition) {
        for (int i = 0; i < 2; ++i) {
            int differenceSpot = (toPosition / moves[i]) - (fromPosition / moves[i]);
            if (fromPosition + (differenceSpot * moves[i]) == toPosition) {
                return moves[i];
            }
        }
    }

    if (fromPosition > toPosition) {
        for (int i = 0; i < 2; ++i) {
            int differenceSpot = (fromPosition / moves[i]) - (toPosition / moves[i]);
            if (fromPosition - (differenceSpot * moves[i]) == toPosition) {
                return (-1 * moves[i]);
            }
        }
    }
    return 0;
}
```

myDistination() will determine the pattern of movement whether it’s 7 or 9 or -9 or

and if method return zero that mean user did a wrong destination

```
public Boolean isMyRoadBlocked(int fromPosition, int toPosition, Spot[] spot)
{
    int destination = myDestination(fromPosition, toPosition);
    return CheckBlockedRoad.isTheRoadBlocked(fromPosition, toPosition,
destination, spot);
}
```

isMyRoadBlocked() method will check if the road of the bishop is blocked or not depending on the destination that is taken by the bishop

Rook:

Rook moves in straight lines either vertically or horizontally

```
public Boolean isMoveValid(int fromPosition, int toPosition) {
    return (((fromPosition % 8) == (toPosition % 8))
            || ((fromPosition / 8) == (toPosition / 8)));
}
```

isMoveValid()

it will serve its function throw mod or division

if fromPosition % 8 equal toPosition %8 that’s means it moves vertically

if fromPosition / 8 equal toPosition / 8 that means it moves horizontally

```
public int myDestination(int fromPosition, int toPosition) {
    if ((fromPosition % 8) == (toPosition % 8)) {
        if (fromPosition > toPosition)
            return -8;
        return 8;
    }
    if ((fromPosition / 8) == (toPosition / 8)) {
        if (fromPosition > toPosition)
            return -1;
        return 1;
    }
    return 0;
}
```

myDestination()

will determine the direction of movement based on the difference between the spot index:

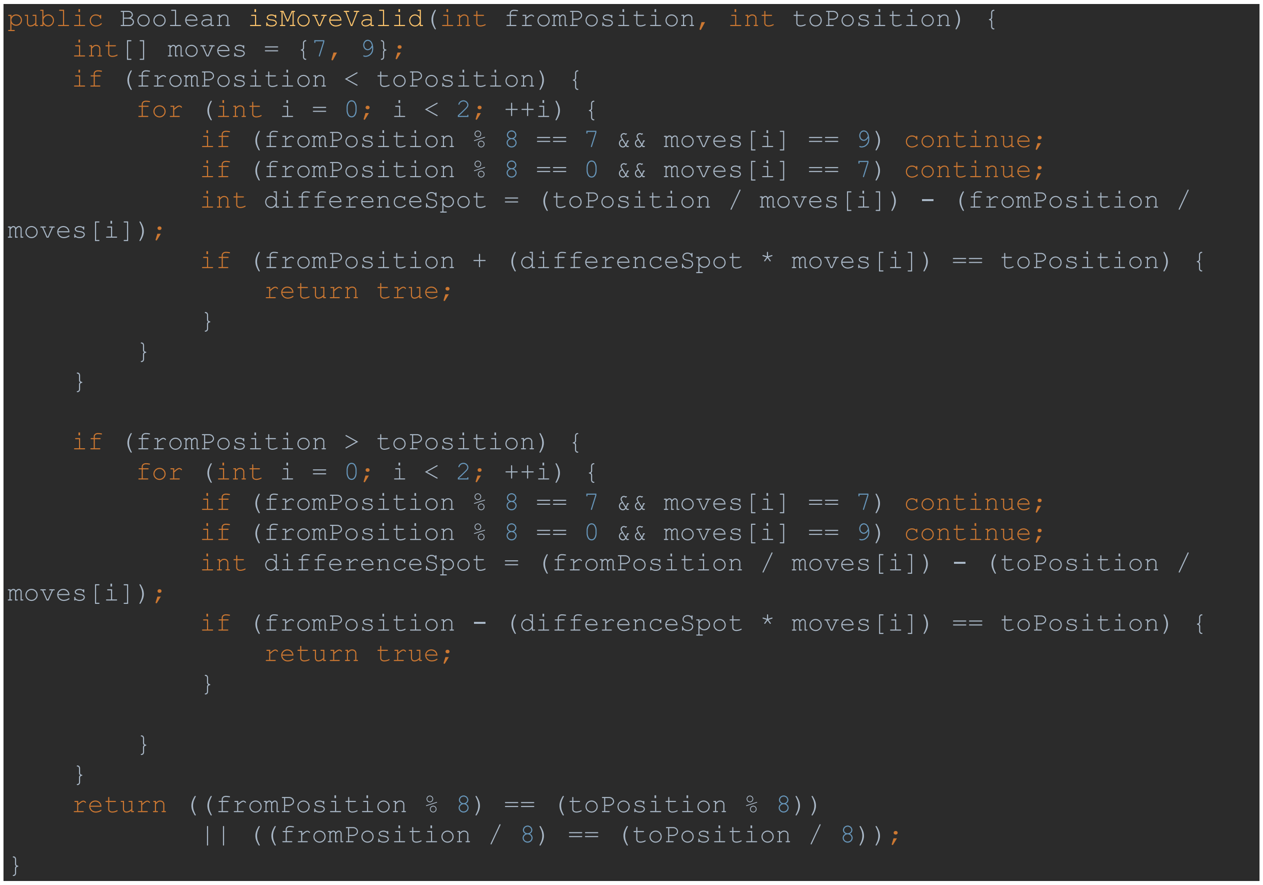
positive values mean it moves either upward or to the right while negative values mean it moves either downward or to the left

```
public Boolean isMyRoadBlocked(int fromPosition, int toPosition, Spot[] spot)
{
    int destination = myDestination(fromPosition, toPosition);
    return CheckBlockedRoad.isTheRoadBlocked(fromPosition, toPosition,
destination, spot);
}
```

isMyRoadBlocked() method will check if the road of the rook is blocked or not depending on the destination that is taken by the rook

Queen:

Its movement is a combination between the bishop and rook



```
public Boolean isMoveValid(int fromPosition, int toPosition) {
    int[] moves = {7, 9};
    if (fromPosition < toPosition) {
        for (int i = 0; i < 2; ++i) {
            if (fromPosition % 8 == 7 && moves[i] == 9) continue;
            if (fromPosition % 8 == 0 && moves[i] == 7) continue;
            int differenceSpot = (toPosition / moves[i]) - (fromPosition /
moves[i]);
            if (fromPosition + (differenceSpot * moves[i]) == toPosition) {
                return true;
            }
        }
    }

    if (fromPosition > toPosition) {
        for (int i = 0; i < 2; ++i) {
            if (fromPosition % 8 == 7 && moves[i] == 7) continue;
            if (fromPosition % 8 == 0 && moves[i] == 9) continue;
            int differenceSpot = (fromPosition / moves[i]) - (toPosition /
moves[i]);
            if (fromPosition - (differenceSpot * moves[i]) == toPosition) {
                return true;
            }

        }
    }
    return ((fromPosition % 8) == (toPosition % 8))
            || ((fromPosition / 8) == (toPosition / 8));
}
```

isMoveValid()

it serves as what is mentioned in bishop and rook

```
public int myDestination(int fromPosition, int toPosition) {
    int[] moves = {7, 9};
    if (fromPosition < toPosition) {
        for (int i = 0; i < 2; ++i) {
            if (fromPosition % 8 == 7 && moves[i] == 9) continue;
            if (fromPosition % 8 == 0 && moves[i] == 7) continue;
            int differenceSpot = (toPosition / moves[i]) - (fromPosition /
moves[i]);
            if (fromPosition + (differenceSpot * moves[i]) == toPosition) {
                return moves[i];
            }
        }
    }

    if (fromPosition > toPosition) {
        for (int i = 0; i < 2; ++i) {
            if (fromPosition % 8 == 7 && moves[i] == 7) continue;
            if (fromPosition % 8 == 0 && moves[i] == 9) continue;
            int differenceSpot = (fromPosition / moves[i]) - (toPosition /
moves[i]);
            if (fromPosition - (differenceSpot * moves[i]) == toPosition) {
                return (-1 * moves[i]);
            }

        }
    }
    if ((fromPosition % 8) == (toPosition % 8)) {
        if (fromPosition > toPosition)
            return -8;
        return 8;
    }
    if ((fromPosition / 8) == (toPosition / 8)) {
        if (fromPosition > toPosition)
            return -1;
        return 1;
    }
    return 0;
}
```

myDestination()

it serves as what is mentioned in bishop and rook

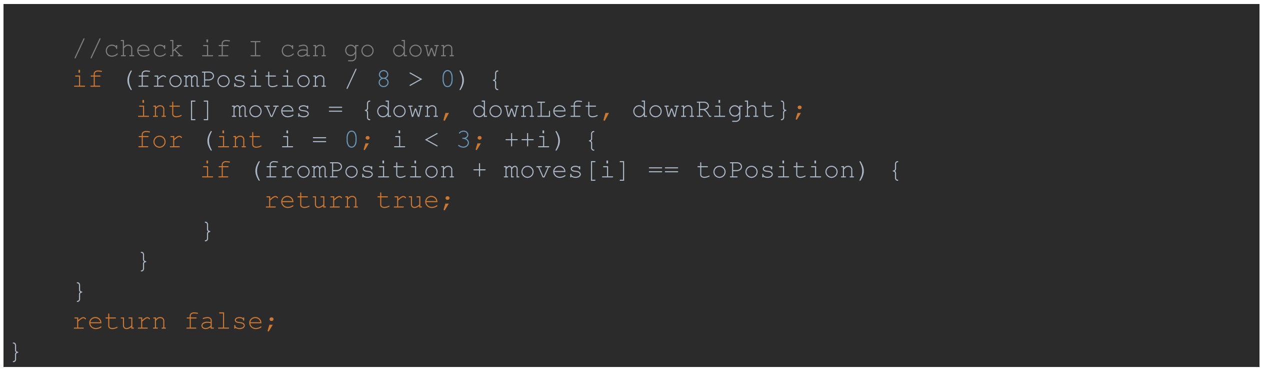
```
public Boolean isMyRoadBlocked(int fromPosition, int toPosition, Spot[] spot)
{
    int destination = myDestination(fromPosition, toPosition);
    return CheckBlockedRoad.isTheRoadBlocked(fromPosition, toPosition,
destination, spot);
}
```

isMyRoadBlocked() method will check if the road of the rook is blocked or not depending on the destination that is taken by the rook

King:

king moves one spot in any direction

```
public Boolean isMoveValid(int fromPosition, int toPosition) {
    int up = 8, upRight = 9, right = 1, downRight = -7,
            down = -8, downLeft = -9, left = -1, upLeft = 7;

    //check if I can go right
    if (fromPosition % 8 < 7) {
        int[] moves = {right, upRight, downRight};
        for (int i = 0; i < 3; ++i) {
            if (fromPosition + moves[i] == toPosition) {
                return true;
            }
        }
    }

    //check if I can go left
    if (fromPosition % 8 > 0) {
        int[] moves = {left, upLeft, downLeft};
        for (int i = 0; i < 3; ++i) {
            if (fromPosition + moves[i] == toPosition) {
                return true;
            }
        }
    }

    //check if I can go up
    if (fromPosition / 8 < 7) {
        int[] moves = {up, upRight, upLeft};
        for (int i = 0; i < 3; ++i) {
            if (fromPosition + moves[i] == toPosition) {
                return true;
            }
        }
    }
```



```
    //check if I can go down
    if (fromPosition / 8 > 0) {
        int[] moves = {down, downLeft, downRight};
        for (int i = 0; i < 3; ++i) {
            if (fromPosition + moves[i] == toPosition) {
                return true;
            }
        }
    }
    return false;
}
```

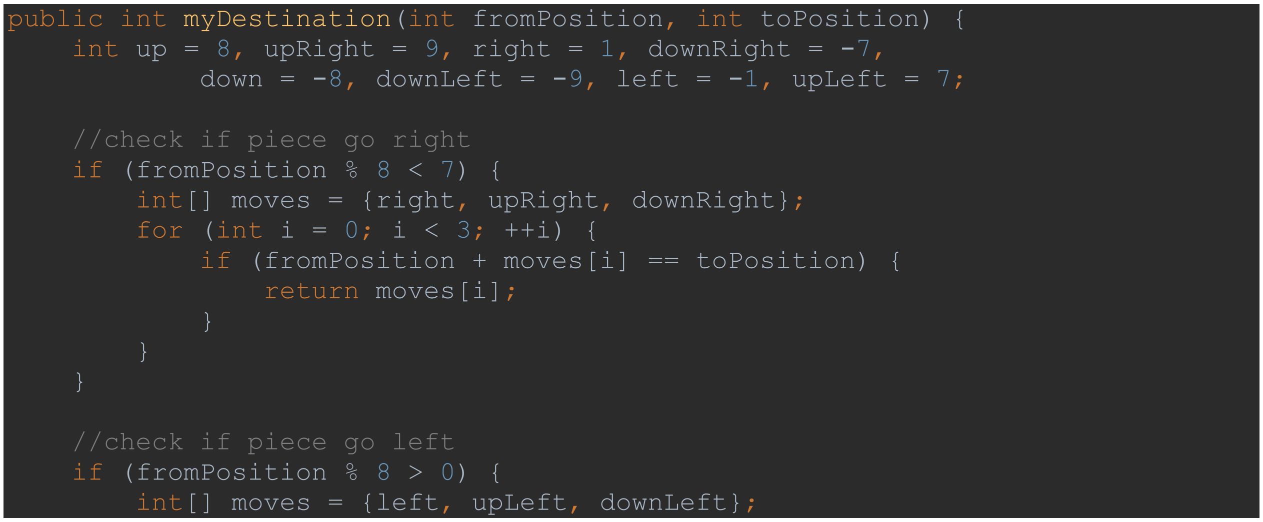
king has 4 possible movements:

- right (1, 9, -7)

  notice that, in certain cases, we can’t move right because we may exceed the boundary of the board and these cases happen if the king stands at the most right column, in other words, if the (position mod 8 equals 7) it will not move right movements

- left (-1, 7, -9)
  notice that, in certain cases, we can’t move left because we may exceed the boundary of the board and these cases happen if the king stands at the most left column, in other words, if the (position mod 8 equals 0) it will not move left movements

- up(8)
  notice that, in certain cases, we can’t move up because we may exceed the boundary of the board and these cases happen if the king stands at the most up row, in other words, when the (position divided by 8 gives 7) it will not move up movements
- down(-8)
  notice that, in certain cases, we can’t move down because we may exceed the boundary of the board and these cases happen if the king stands at the most bottom row, in other words, when the (position divided by 8 gives 0) it will not move down movements



```
public int myDestination(int fromPosition, int toPosition) {
    int up = 8, upRight = 9, right = 1, downRight = -7,
            down = -8, downLeft = -9, left = -1, upLeft = 7;

    //check if piece go right
    if (fromPosition % 8 < 7) {
        int[] moves = {right, upRight, downRight};
        for (int i = 0; i < 3; ++i) {
            if (fromPosition + moves[i] == toPosition) {
                return moves[i];
            }
        }
    }

    //check if piece go left
    if (fromPosition % 8 > 0) {
        int[] moves = {left, upLeft, downLeft};
        for (int i = 0; i < 3; ++i) {
            if (fromPosition + moves[i] == toPosition) {
                return moves[i];
            }
        }
    }

    //check if piece go up
    if (fromPosition / 8 < 7) {
        int[] moves = {up, upRight, upLeft};
        for (int i = 0; i < 3; ++i) {
            if (fromPosition + moves[i] == toPosition) {
                return moves[i];
            }
        }
    }

    //check if piece go down
    if (fromPosition / 8 > 0) {
        int[] moves = {down, downLeft, downRight};
        for (int i = 0; i < 3; ++i) {
            if (fromPosition + moves[i] == toPosition) {
                return moves[i];
            }
        }
    }
    return 0;
}
```

myDestination() method will determine the direction of movement based on the difference between the spot index,Depending on how the piece move according to what I am divided

```
public Boolean isMyRoadBlocked(int fromPosition, int toPosition, Spot[] spot)
{
    int destination = myDestination(fromPosition, toPosition);
    return CheckBlockedRoad.isTheRoadBlocked(fromPosition, toPosition,
destination, spot);
}
```

isMyRoadBlocked() method will check if the road of the king is blocked or not depending on the destination that is taken by the king

Knight:

The knight moves in an L-shape in all directions

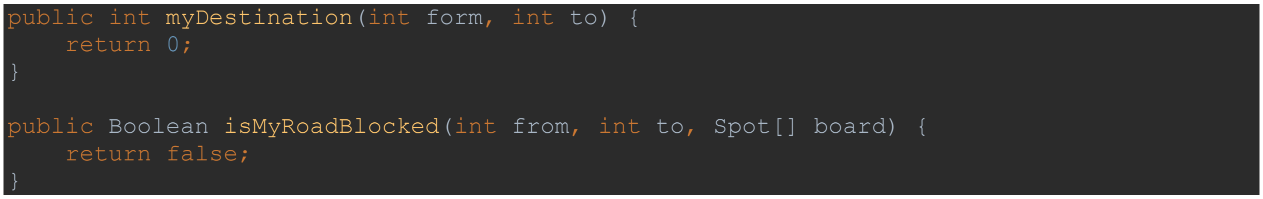
```
public Boolean isMoveValid(int fromPosition, int toPosition) {
    int[] up = {17, 15};
    int[] down = {-17, -15};
    int[] right = {10, -6};
    int[] left = {6, -10};

    //check if I can do up
    if (fromPosition / 8 <= 5) {
        for (int i = 0; i < 2; ++i) {
            if (fromPosition + up[i] == toPosition)
                return true;
        }
    }

    //check if I can do down
    if (fromPosition / 8 >= 2) {
        for (int i = 0; i < 2; ++i) {
            if (fromPosition + down[i] == toPosition)
                return true;
        }
    }

    //check if I can do right
    if (fromPosition % 8 <= 5) {
        for (int i = 0; i < 2; ++i) {
            if (fromPosition + right[i] == toPosition)
                return true;
        }
    }

    //check if I can do left
    if (fromPosition % 8 >= 2) {
        for (int i = 0; i < 2; ++i) {
            if (fromPosition + left[i] == toPosition)
                return true;
        }
    }
    return false;
}
```

knight has 4 possible movements:

- right (10, -6)
  notice that, in certain cases, we can’t move right because we may exceed the boundary of the board and these cases happen if the knight stands at the most right column and its adjacent to the left, in other words, if the (position mod 8 equals [6,7]) it will not move right movements
- left (6, -10)
  notice that, in certain cases, we can’t move left because we may exceed the boundary of the board and these cases happen if the knight stands at the most left column and its adjacent to the right, in other words, if the (position mod 8 equals [0,1]) it will not move left movements
- up (17, 15)
  notice that, in certain cases, we can’t move up because we may exceed the boundary of the board and these cases happen if the knight stands at the most up row and its adjacent to the down, in other words, when the (position

divided by 8 gives[7,6]) it will not move up movements

- down(-17, -15)
- notice that, in certain cases, we can’t move down because we may exceed the boundary of the board and these cases happen if the knight stands at the most down row and its adjacent to the up, in other words, when the (position divided by 8 gives[0,1]) it will not move down movements

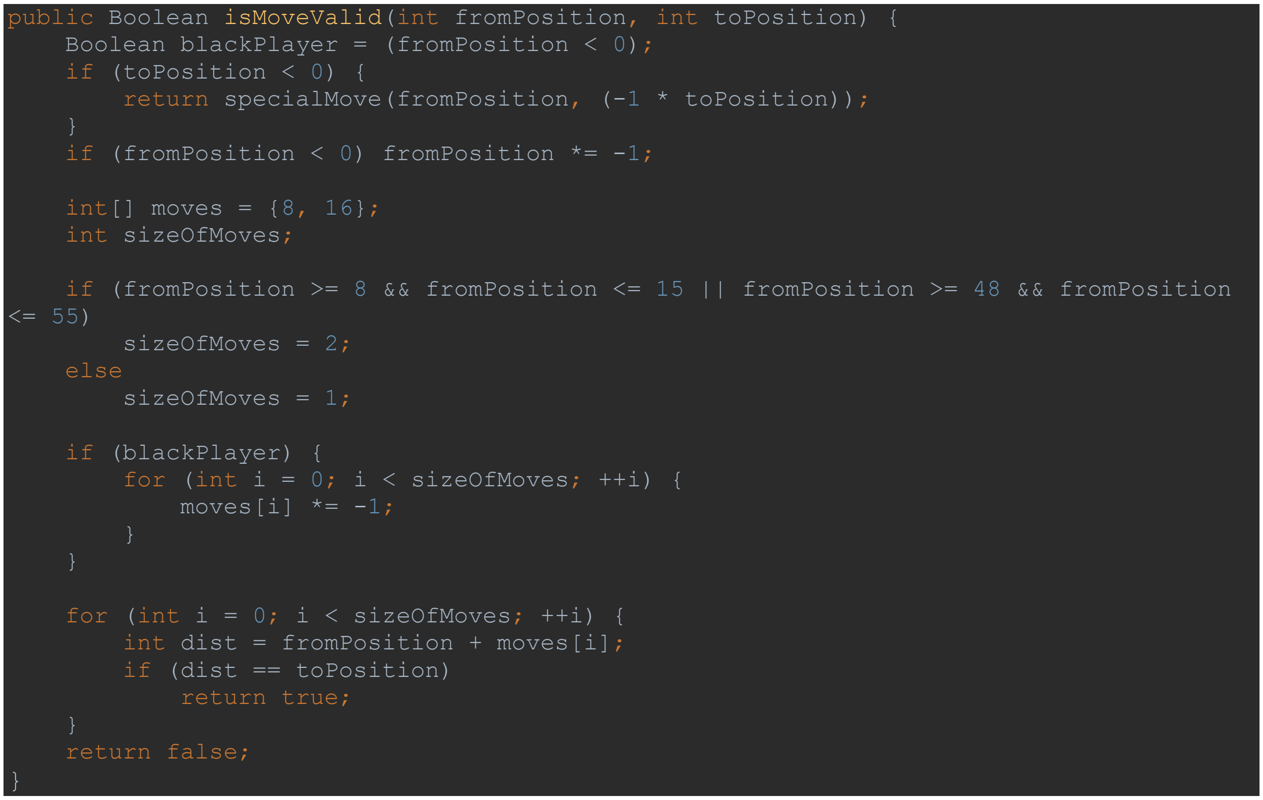


```
public int myDestination(int form, int to) {
    return 0;
}

public Boolean isMyRoadBlocked(int from, int to, Spot[] board) {
    return false;
}
```

The knight can jump over any piece in his path, so isMyRoadBlocked() method will always return false, and we don't need to know his gait pattern through myDestination() method it will return zero because we don’t need the destination to track the knight path

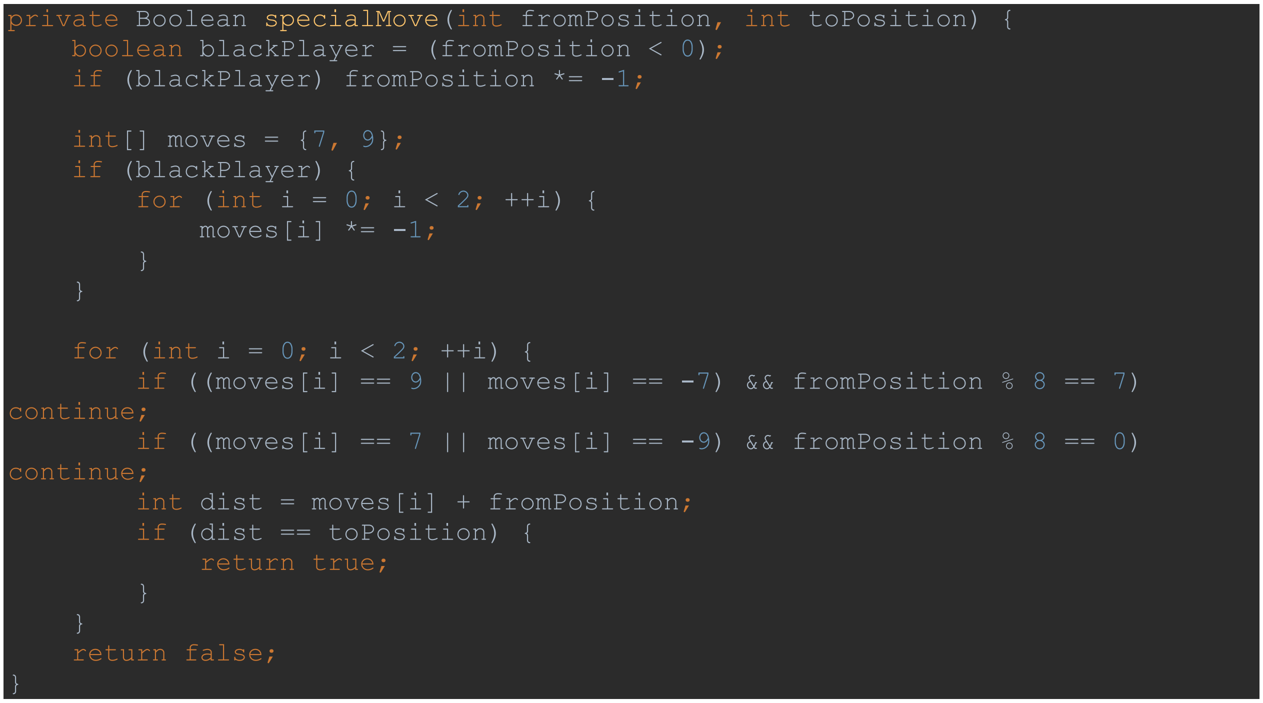
Pawn:

The pawn can take one or two steps the first time, and the rest of the time only one step and if an enemy piece meets him diagonally, he can walk diagonally to kill it



```
public Boolean isMoveValid(int fromPosition, int toPosition) {
    Boolean blackPlayer = (fromPosition < 0);
    if (toPosition < 0) {
        return specialMove(fromPosition, (-1 * toPosition));
    }
    if (fromPosition < 0) fromPosition *= -1;

    int[] moves = {8, 16};
    int sizeOfMoves;

    if (fromPosition >= 8 && fromPosition <= 15 || fromPosition >= 48 && fromPosition
<= 55)
        sizeOfMoves = 2;
    else
        sizeOfMoves = 1;

    if (blackPlayer) {
        for (int i = 0; i < sizeOfMoves; ++i) {
            moves[i] *= -1;
        }
    }

    for (int i = 0; i < sizeOfMoves; ++i) {
        int dist = fromPosition + moves[i];
        if (dist == toPosition)
            return true;
    }
    return false;
}
```

isMoveValid() method will accept the correct movement which is 8 or 16 for the first move and 8 for another movement, if pawn do a special move isValidMove() will lead me to specialMove()



```
private Boolean specialMove(int fromPosition, int toPosition) {
    boolean blackPlayer = (fromPosition < 0);
    if (blackPlayer) fromPosition *= -1;

    int[] moves = {7, 9};
    if (blackPlayer) {
        for (int i = 0; i < 2; ++i) {
            moves[i] *= -1;
        }
    }

    for (int i = 0; i < 2; ++i) {
        if ((moves[i] == 9 || moves[i] == -7) && fromPosition % 8 == 7)
continue;
        if ((moves[i] == 7 || moves[i] == -9) && fromPosition % 8 == 0)
continue;
        int dist = moves[i] + fromPosition;
        if (dist == toPosition) {
            return true;
        }
    }
    return false;
}
```

the special move for the pawn is One step forward for the right diagonal(9) , for the left diagonal(7) and -9 ,-7 for the black player

```
public int myDestination(int fromPosition, int toPosition) {
    Boolean blackPlayer = (fromPosition < 0);
    if (toPosition < 0) {
        return specialDestination(fromPosition, (-1 * toPosition));
    }
    if (fromPosition < 0) fromPosition *= -1;


    int[] moves = {8, 16};
    int sizeOfMoves;

    if (fromPosition >= 8 && fromPosition <= 15 || fromPosition >= 48 && fromPosition
<= 55)
        sizeOfMoves = 2;
    else
        sizeOfMoves = 1;

    if (blackPlayer) {
        for (int i = 0; i < sizeOfMoves; ++i) {
            moves[i] *= -1;
        }
    }
    for (int i = 0; i < sizeOfMoves; ++i) {
        int dist = fromPosition + moves[i];
        if (dist == toPosition)
            return moves[i];
    }

    return 0;
}
```

myDestination() will determine the pattern of movement whether it’s one step(8) or two step(16) forward and -8 ,-16 for the black player, and if it special move myDestination() method will lead me to

specialDestination() method

```
private int specialDestination(int fromPosition, int toPosition) {
    Boolean blackPlayer = (fromPosition < 0);
    if (fromPosition < 0) fromPosition *= -1;
    if (toPosition < 0) toPosition *= -1;

    int[] moves = {7, 9};
    if (blackPlayer) {
        for (int i = 0; i < 2; ++i) {
            moves[i] *= -1;
        }
    }

    for (int i = 0; i < 2; ++i) {
        if ((moves[i] == 9 || moves[i] == -7) && fromPosition % 8 == 7) continue;
        if ((moves[i] == 7 || moves[i] == -9) && fromPosition % 8 == 0) continue;
        int dist = moves[i] + fromPosition;
        if (dist == toPosition) {
            return moves[i];
        }
    }
    return 0;
}
```

specialDestination() method will determine the pattern of movement whether it’s right diagonal(9) or left diagonal(7) and -9,-7 for the black palyer

```
public Boolean isMyRoadBlocked(int fromPosition, int toPosition, Spot[] spot)
{
    int destination = myDestination(fromPosition, toPosition);
    if (fromPosition < 0)
        fromPosition = fromPosition * -1;
    if (toPosition < 0)
        toPosition = toPosition * -1;
    return CheckBlockedRoad.isTheRoadBlocked(fromPosition, toPosition,
destination, spot);
}
```

isMyRoadBlocked() method will check if the road of the pawn is blocked or not depending on the destination that is taken by the pawn

Frame package

I created this package with two classes called Board and Spot.

Spot

```
public class Spot {
    private Type pieceType;
    private Colors pieceColor;
    public Spot(Type pieceType, Colors
pieceColor) {
        this.pieceType = pieceType;
        this.pieceColor = pieceColor;
    }
    public Type getPieceType() {
        return pieceType;
    }
    public Colors getPieceColor() {
        return pieceColor;
    }
}
```

This class contains the contents of the spot in the chessboard

Board

```
private Spot[] board = new Spot[64];
private PieceFactory pieceFactory;
private static Board SingletonBoard = new Board();

private Board() {
    initializeSpots();
    pieceFactory = new PieceFactory();
}
```

in this class defined three attributes:

- board array type of Spot

I'll explain pieceFactory and singletonBoard in the design patterns part

```
private void initializeSpots() {
    board[0] = new Spot(Type.ROOK, Colors.WHITE);
    board[1] = new Spot(Type.KNIGHT, Colors.WHITE);
    board[2] = new Spot(Type.BISHOP, Colors.WHITE);
    board[3] = new Spot(Type.QUEEN, Colors.WHITE);
    board[4] = new Spot(Type.KING, Colors.WHITE);
    board[5] = new Spot(Type.BISHOP, Colors.WHITE);
    board[6] = new Spot(Type.KNIGHT, Colors.WHITE);
    board[7] = new Spot(Type.ROOK, Colors.WHITE);
    for (int i = 8; i <= 15; ++i) {
        board[i] = new Spot(Type.PAWN, Colors.WHITE);
    }

    board[56] = new Spot(Type.ROOK, Colors.BLACK);
    board[57] = new Spot(Type.KNIGHT, Colors.BLACK);
    board[58] = new Spot(Type.BISHOP, Colors.BLACK);
    board[59] = new Spot(Type.QUEEN, Colors.BLACK);
    board[60] = new Spot(Type.KING, Colors.BLACK);
    board[61] = new Spot(Type.BISHOP, Colors.BLACK);
    board[62] = new Spot(Type.KNIGHT, Colors.BLACK);
    board[63] = new Spot(Type.ROOK, Colors.BLACK);
    for (int i = 48; i <= 55; ++i) {
        board[i] = new Spot(Type.PAWN, Colors.BLACK);
    }

    for (int i = 16; i <= 47; ++i) {
        board[i] = new Spot(Type.NULL, Colors.NULL);
    }
}
```

initializeSpots() method will initialize spots with the initial location for every piece in the board array.

```
public void printBoard()
```

I created a printBoard() method to print board in console

```
private String getPieceShape(Type type) {
    if (type == Type.KING)
        return "KI";
    if (type == Type.BISHOP)
        return "B";
    if (type == Type.KNIGHT)
        return "KN";
    if (type == Type.PAWN)
        return "P";
    if (type == Type.QUEEN)
        return "Q";
    if (type == Type.ROOK)
        return "R";
    return "*";
}
```

getPieceShape() method will help me with printBoard() method by returning the shape for every piece will printed in the console.

```
public Boolean rightPieceColor(int fromPosition, Player player)
{
    return (board[fromPosition].getPieceColor() ==
player.getPlayerColor());
}
```

rightPieceColor() method will return true if the user started with his piece

```
public Type occupiedToPosition(int fromPosition, int toPosition)
{
    if (board[toPosition].getPieceColor() !=
board[fromPosition].getPieceColor())
        return board[toPosition].getPieceType();
    return Type.NULL;
}
```

occupiedToPosition() method will tell me if the user moved the piece to a spot containing a piece form his team

```
public Boolean pieceValidMove(int fromPosition, int toPosition)
{
    Piece piece =
pieceFactory.createPiece(board[fromPosition].getPieceType());
    if (piece == null) return false;
    int fromValidate = 1, toValidate = 1;
    if (board[fromPosition].getPieceType() == Type.PAWN &&
board[fromPosition].getPieceColor() == Colors.BLACK)
        fromValidate = -1;
    if ((board[fromPosition].getPieceType() == Type.PAWN) &&
(occupiedToPosition(fromPosition, toPosition) != Type.NULL))
        toValidate = -1;

    return piece.isMoveValid(fromValidate * fromPosition,
toValidate * toPosition);
}
```

pieceValidMove() will create a piece type (I will explain it in more details in the design pattern part)

then will go to isMoveValid() and see if the piece did a valid move or not

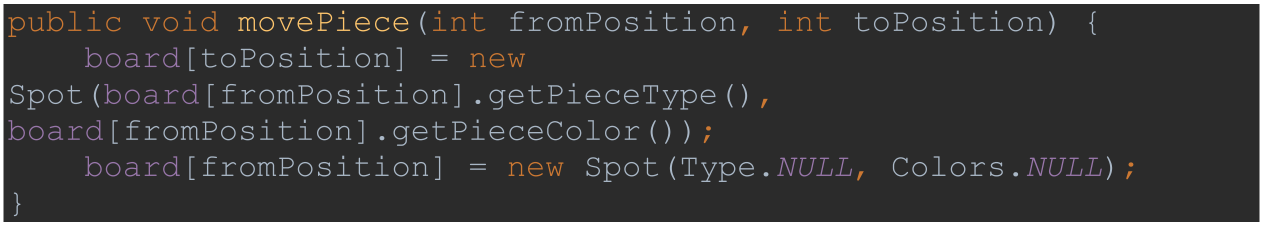
```
public Boolean pieceBlockedMove(int fromPosition, int
toPosition) {
    Piece piece =
pieceFactory.createPiece(board[fromPosition].getPieceType());
    if (piece == null) return false;
    int fromValidate = 1, toValidate = 1;
    if (board[fromPosition].getPieceColor() == Colors.BLACK &&
board[fromPosition].getPieceType() == Type.PAWN)
        fromValidate = -1;
    if ((board[fromPosition].getPieceType() == Type.PAWN) &&
(occupiedToPosition(fromPosition, toPosition) != Type.NULL))
        toValidate = -1;

    return piece.isMyRoadBlocked((fromPosition * fromValidate), (toPosition *
toValidate), board);
}
```

pieceBlockedMove()will create a piece type (I will explain it in more details in the design pattern part)

then will go to isMyRoadBlocked() and see if any piece blocked my road or not

```
public Boolean checkMate(Player player) {
    int kingPosition = -1;
    int[] moves = {-1, 7, 8, 9, 1, -7, -8, -9};
    for (int i = 0; i <= 63; ++i) {
        if (board[i].getPieceType() == Type.KING && board[i].getPieceColor() ==
player.getPlayerColor()) {
            kingPosition = i;
            break;
        }
    }

    for (int move : moves) {
        int k = kingPosition;
        while (k >= 0 && k <= 63) {
            if ((move == 7 || move == -9 || move == -1) && k % 8 == 0) break;
            if ((move == 9 || move == -7 || move == 1) && k % 8 == 7) break;
            if (move == 8 && k / 8 == 7) break;
            if (move == -8 && k / 8 == 0) break;
            k += move;
            if (k >= 0 && k <= 63) {
                if (board[k].getPieceColor() != Colors.NULL &&
board[k].getPieceColor() != player.getPlayerColor()) {
                    if (pieceValidMove(k, kingPosition))
                        return true;
                } else if (board[k].getPieceColor() != Colors.NULL)
                    break;
            }
        }
    }
    return false;
}
```

checkMate() method will check if the king in dangerous,by examining all roads leading to the king



```
public void movePiece(int fromPosition, int toPosition) {
    board[toPosition] = new
Spot(board[fromPosition].getPieceType(),
board[fromPosition].getPieceColor());
    board[fromPosition] = new Spot(Type.NULL, Colors.NULL);
}
```

movePiece() method will move any piece The user selects it if it’s valid

Game package

Player

```
public class Player {
    private final String playerName;
    private final Colors playerColor;
     public Player(String playerName, Colors
playerColor) {
         this.playerName = playerName;
         this.playerColor = playerColor;
     }
     public String getPlayerName() {
         return playerName;
     }
     public Colors getPlayerColor() {
         return playerColor;
     }
}
```

this method contains the information for the player

game

```
private Player whitePlayer;
private Player blackPlayer;
private HashMap<String, Integer> spotName= new HashMap<String,
Integer>();;
private HashMap<Type, Integer> aliveTypes= new HashMap<Type,
Integer>();;
```

I will store in the whitePlayer and BlackPlayer objects player information

spotName map: his key will contain the name of the spot and his value will contain the number of this spot in the array

aliveTypes: his key will contain the type of piece and his value will contain the number of this piece still alive

```
private void initializeSpotName() {
    int numberOfSpot = 0;
    for (int i = 1; i <= 8; ++i) {
        for (int ch = 97; ch <= 104; ++ch) {
            String spotAlphabetical = "" + (char) ch + i;
            spotName.put(spotAlphabetical, numberOfSpot);
            numberOfSpot++;
        }
    }
}
```

initializeSpotName() will initialize the spot name and save it in the key of spotName map and its value will be the number of the spot in the array

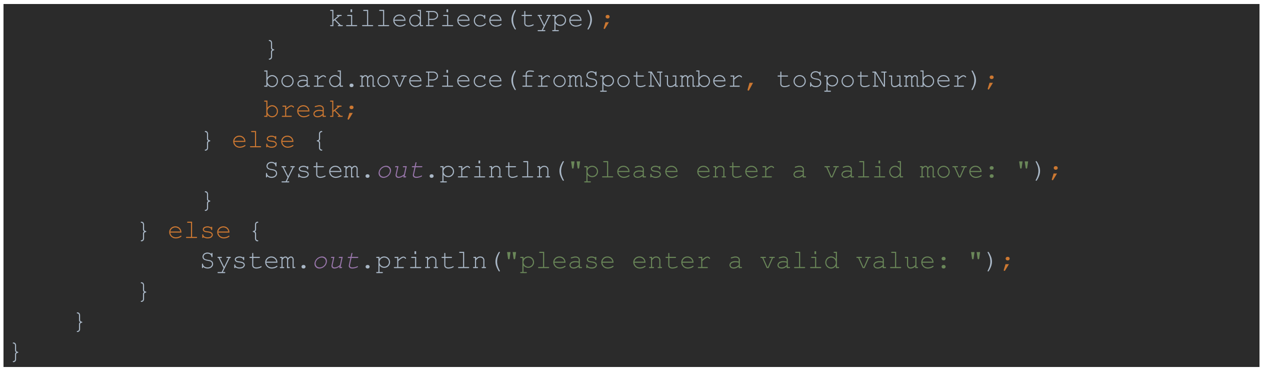
```
private void playerInformation() {
    System.out.println("white player name: ");
    whitePlayer = new Player(sc.next(), Colors.WHITE);
    System.out.println("black player name: ");
    blackPlayer = new Player(sc.next(), Colors.BLACK);
}
```

playerInformation() will give whitePlayer and blackPlayer his Information.

```
private void initializeAlive() {
    Type[] type = {Type.KING, Type.BISHOP, Type.KNIGHT,
Type.QUEEN, Type.PAWN, Type.ROOK};
    for (int i = 0; i < 6; ++i) {
        if (type[i] == Type.KING || type[i] == Type.QUEEN)
            aliveTypes.put(type[i], 2);
        if (type[i] == Type.PAWN)
            aliveTypes.put(type[i], 16);
        else
            aliveTypes.put(type[i], 4);
    }
}
```

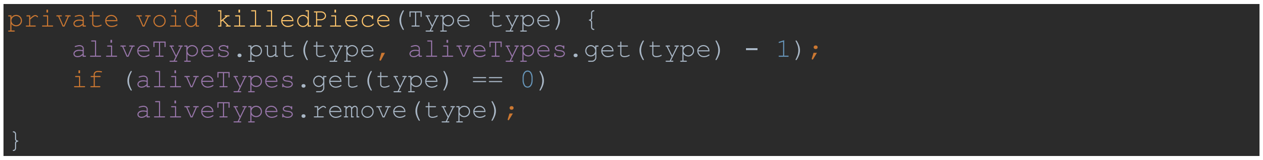
initializeAlive() method will give the key of Alive map type of piece and give the value of Alive map initial number of piece still alive

```
private void checkInputs(Player player, Board board) {
    while (true) {
        String fromPosition = sc.next();
        String toPosition = sc.next();
        Boolean validFromPosition = fromPosition.matches("[a-h][1-8]");
        Boolean validToPosition = toPosition.matches("[a-h][1-8]");

        if (validFromPosition && validToPosition) {
            int fromSpotNumber = spotName.get(fromPosition),
                    toSpotNumber = spotName.get(toPosition);

            Boolean rightColor = board.rightPieceColor(fromSpotNumber,
player);
            Boolean validMove = (board.pieceValidMove(fromSpotNumber,
toSpotNumber));
            Boolean notBlockedMove = !(board.pieceBlockedMove(fromSpotNumber,
toSpotNumber));
            if (rightColor && validMove && notBlockedMove) {
                if (board.occupiedToPosition(fromSpotNumber, toSpotNumber) !=
Type.NULL) {
                    Type type = board.occupiedToPosition(fromSpotNumber,
toSpotNumber);
```



```
                    killedPiece(type);
                }
                board.movePiece(fromSpotNumber, toSpotNumber);
                break;
            } else {
                System.out.println("please enter a valid move: ");
            }
        } else {
            System.out.println("please enter a valid value: ");
        }
    }
}
```

checkInputs() method will take the moves from the players and check whether is a valid move or not, if it’s a valid move it will execute the suitable action for this move and if it’s not it will ask the player to enter a valid move



```
private void killedPiece(Type type) {
    aliveTypes.put(type, aliveTypes.get(type) - 1);
    if (aliveTypes.get(type) == 0)
        aliveTypes.remove(type);
}
```

killedPiece() method it will decrease the number of alive pieces if the piece is down by the opposite player

```
private void playerTurn(Board board) {
    int wholeNumberOfTurns = 50;
    boolean flag = false;
    while (wholeNumberOfTurns >= 0) {
        if (board.checkMate(blackPlayer))
        {

            System.out.println("white player wins");
            flag = true;
            break;
        }
        System.out.println("white player turn: ");
        if (board.checkMate(whitePlayer))
            System.out.println("king in dangerous !");
        checkInputs(whitePlayer, board);
        if (aliveTypes.size() == 2) {
            break;
        }
        board.printBoard();

        if (board.checkMate(whitePlayer)) {
            System.out.println("black player wins");
            flag = true;
            break;
        }
        System.out.println("black player turn: ");
        if (board.checkMate(blackPlayer))
            System.out.println("king in dangerous !");
        checkInputs(blackPlayer, board);
        if (aliveTypes.size() == 2) {
            break;
        }

        board.printBoard();
        wholeNumberOfTurns--;
    }

    if (!flag)
        System.out.println("draw");
}
```

playerTurn() method will give the players turns to play starting from the white player and if one of them wins it will print the winner and if they both draw or the turns over it will also print draw

```
public void startGame() {
    Board board = Board.getBoard();
    board.printBoard();
    playerInformation();
    playerTurn(board);
}
```

startGame() method will create the board and read the player information and start the game by giving the turns to the player

oop part

my project had been done with classes also I used an interface to classify the Piece classes, as we used

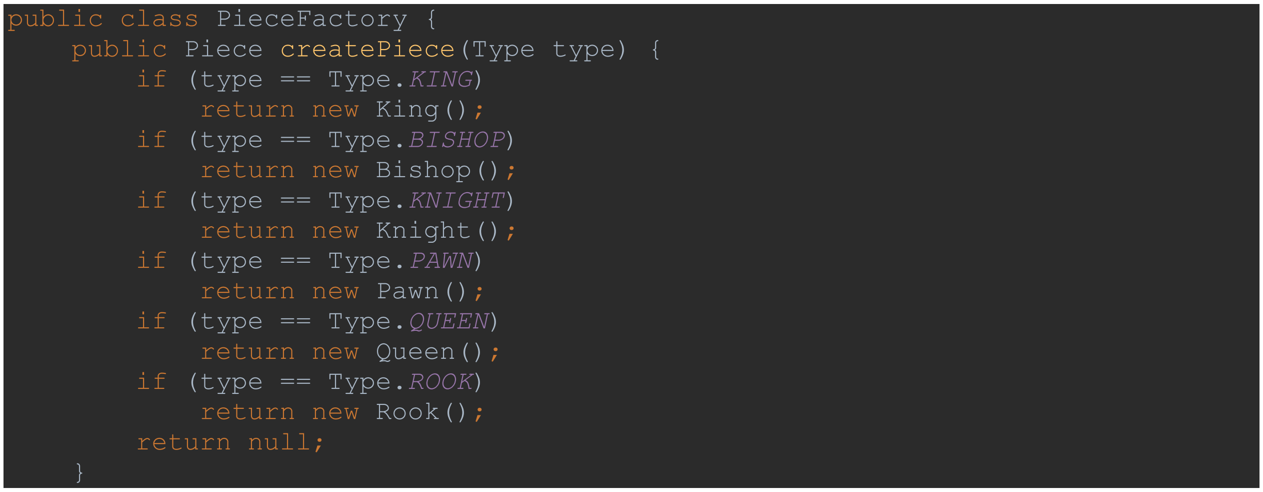
- abstraction
- encapsulation
- polymorphism

as shown in the project and the code we done

design pattern part

I used in my project two design patterns principles

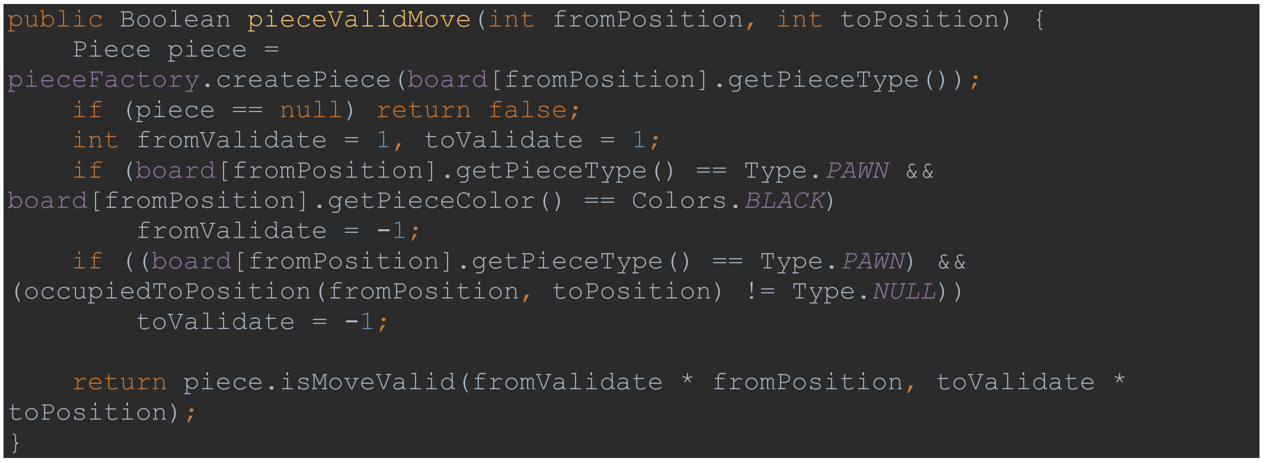
- factory



```
public class PieceFactory {
    public Piece createPiece(Type type) {
        if (type == Type.KING)
            return new King();
        if (type == Type.BISHOP)
            return new Bishop();
        if (type == Type.KNIGHT)
            return new Knight();
        if (type == Type.PAWN)
            return new Pawn();
        if (type == Type.QUEEN)
            return new Queen();
        if (type == Type.ROOK)
            return new Rook();
        return null;
    }
```

in general, I used the piece factory class to instantiate a suitable piece for the suitable situation

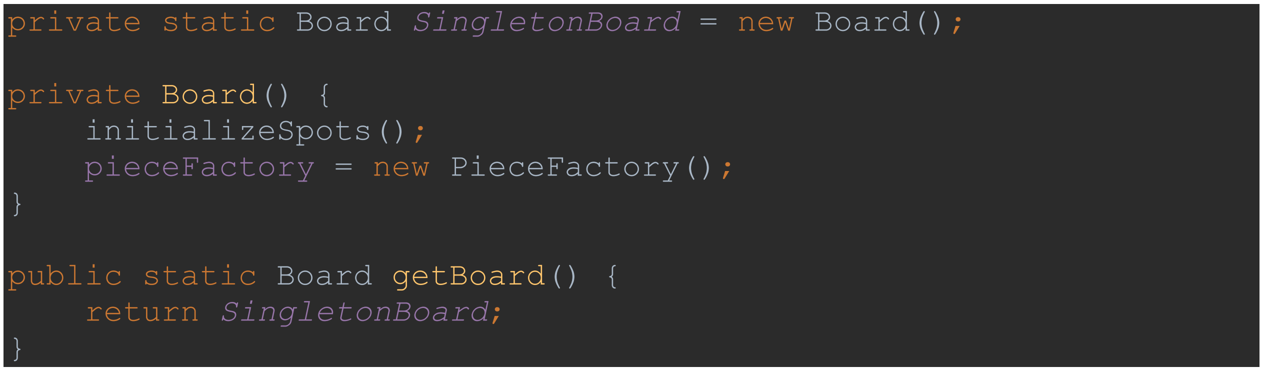
as example



```
public Boolean pieceValidMove(int fromPosition, int toPosition) {
    Piece piece =
pieceFactory.createPiece(board[fromPosition].getPieceType());
    if (piece == null) return false;
    int fromValidate = 1, toValidate = 1;
    if (board[fromPosition].getPieceType() == Type.PAWN &&
board[fromPosition].getPieceColor() == Colors.BLACK)
        fromValidate = -1;
    if ((board[fromPosition].getPieceType() == Type.PAWN) &&
(occupiedToPosition(fromPosition, toPosition) != Type.NULL))
        toValidate = -1;

    return piece.isMoveValid(fromValidate * fromPosition, toValidate *
toPosition);
}
```

I used the piece factory with the help of the piece type that is stored in the spot contents to create the suitable piece and call the suitable isMoveValid() function

Sineglton



```
private static Board SingletonBoard = new Board();

private Board() {
    initializeSpots();
    pieceFactory = new PieceFactory();
}

public static Board getBoard() {
    return SingletonBoard;
}
```

I used the singleton design pattern for the board because I need one board for each game, and I did that by instantiating board objects as private in the class and I created getBoard() method to deal with this object

Clean code part

One of the best methods to make the code clear for teamwork is to make it in clean code

I used the most clean-code principles in my project

Like,

- meaning full names for objects, methods, classes, and other elements
- write code easy to read and understandable
- Avoid duplication in the methods
- fewer comments and very short when needed
- I don't use long methods
- I don't use long parameters

As shown in general we used more principles in the code, as we know this code is for humans not just for machines so it must be readable and understanble for the rest of the team

SOLID principles

There are five types of solid principles:

- Single Responsibility Principle (SRP):

I divided my code for more than one class to reach the target of SRP that used every class for one goal

- Open-Closed Principle (OCP):

the code we had done is completed and able for an extension if we want to add another feature in the future

- Liskov Substitution Principle (LSP):

  We used LSP in the code in the interface that we named Piece to use it in pieces classes (king, bishop, etc.…)

- Interface Segregation Principle (ISP):

  I created a Piece interface to use in pieces classes (king, bishop, etc.…) and all pieces classes need the attributes in this interface

Despite it being the first time, I deal with a chess game and its details, but it a was a good chance to learn it

The end...