importjava.util.Scanner;  
  
public classMain {  
  
public staticvoidmain(String[] args) {  
Scannerreader = newScanner(System.*in*);  
ChessBoard board = newChessBoard();  
board.printBoard();  
  
PiececapturedPiece = null;

1-)ScannerInitialization:

**Scannerreader = newScanner(System.in);**creates a newScannerobjecttoreadinputfrom the console.

2-)ChessBoard Initialization:

**ChessBoard board = new ChessBoard();**creates a newinstance of the ChessBoard class, initializing the chessboardandsettingup the gamestate.

3-)Board Printing:

**board.printBoard();**prints the initialstate of the chessboardusing the printBoard() method of the ChessBoard class.

4-) Capturedpiece:

**PiececapturedPiece = null**; declares a variablenamedcapturedPiece of typePieceandinitializes it with a value of null.

Thisvariable is usedtostore the capturedpieceduring the game. When a piececapturesanotherpieceduring a move, the capturedpiece can be assignedtothisvariable. Byinitializing it withnull, weindicatethatnopiece has beencaptured yet at the beginning of the game.

Later in the code, when a move is madeand a piececapturesanotherpiece, you can update the value of capturedPiecewith the capturedpiece. Thisallows you tokeeptrack of the capturedpiecesthroughout the game.

// Game loopuntil the game is ended  
while(!board.isGameEnded()) {  
System.*out*.println("It is " + (board.isWhitePlaying() ? "White" :"Black") + "'s turn");  
Piecepiece;  
  
 do {  
System.*out*.print("Enter the location of the piece:");  
Stringfrom = reader.next();  
piece = board.getPieceAt(from);  
if(piece == null) {  
System.*out*.println("There is nopiece at thislocation, pleasetryagain.");  
} else if(piece.getColor() != (board.isWhitePlaying() ? ChessBoard.*WHITE*:ChessBoard.*BLACK*)) {  
System.*out*.println("It is not " + (board.isWhitePlaying() ? "Black" :"White") + "'s turn, pleasetryagain.");  
piece = null;  
}  
 } while(piece == null);  
  
Stringto;  
 do {  
System.*out*.print("Enter the newlocation of the piece:");  
to = reader.next();  
if(!piece.canMove(to)) {  
System.*out*.println("Invalidmove. Pleasetryagain.");  
}  
 } while(!piece.canMove(to));  
// Capture the piece at the targetlocation, ifany  
PiecetargetPiece = board.getPieceAt(to);  
if(targetPiece != null) {  
capturedPiece = targetPiece;  
System.*out*.println("Capturedpiece: " + capturedPiece);  
}  
// Move the pieceto the newlocation  
piece.move(to);  
board.printBoard();  
  
  
}

5-)GameLoop:

The program enters a whileloopwith the condition**!board.isGameEnded(),**whichmeans the loopwillcontinueuntil the game has ended.

Inside the loop, it prompts the playerfortheirmoveandperforms the necessarychecksandactions.

Player'sTurn:

Itprints a messageindicatingwhoseturn it is, either "White" or "Black," based on the **isWhitePlaying()method** of the **ChessBoard object**.

6-Piece Selection:

The do-whileloop is usedtoget the location of the piece the playerwantstomove (from).

-Inside the **do-whileloop**, it prompts the usertoenter the location of the pieceandretrieves the correspondingPieceobjectfrom the chessboardusing the **getPieceAt() method**.

-Ifthere is nopiece at the specifiedlocation (piece == null), it displays an errormessageandasks the playertoenter the locationagain.

-If the piece at the specifiedlocationbelongsto the oppositecolor (i.e., it's not the player'sturn), it displays an errormessageandasks the playertoenter the locationagain.

-Once a validpiece is selected, the do-whileloop is usedagaintoget the newlocation (to) where the playerwantstomove the piece.

7-Move Selection:

-Inside the second**do-whileloop**, it prompts the usertoenter the newlocation of the pieceandchecksif the move is validusing the **canMove() method** of the Pieceobject.

-If the move is invalid, it displays an errormessageandasks the playertoenter the newlocationagain.

-After a validmove is made, the codechecksifthere is a piece at the targetlocation**(to)**using the **getPieceAt()method.**

-If a pieceexists, it captures the piecebyassigning it to the **capturedPiecevariable**andprints a messageindicating the capturedpiece.

8-)MoveExecutionand Board Update:

-The selectedpiece is thenmovedto the newlocationusing the **move() method** of the Pieceobject..

-Finally, the updatedchessboard is printedusing the **printBoard() methodfrom the ChessBoard class.**

**-**Thisprocessrepeatsuntil the gameends, as determinedby the **isGameEnded() method** of the ChessBoard object.

System.*out*.println("Game Over!");  
System.*out*.println((board.isWhitePlaying() ? "Black" :"White" )+ " wins!");  
if(capturedPiece != null) {  
System.*out*.println("Capturedpiece: " + capturedPiece);  
}  
  
reader.close(); // Close the scanneroutside the loop  
}  
}

9-Game Over:

After the whileloopends, it meansthat the game has ended. Itprints "Game Over!" toindicate the end of the game.

Itthenprintswhichplayerwins the gamebased on the **isWhitePlaying() method** of the **ChessBoard object**. IfisWhitePlaying() returnstrue, it meansthatblackwins, andif it returnsfalse, it meansthatwhitewins.

Next, it checksifthere is a capturedpiece, indicatedby the **capturedPiecevariable**being not null. If a piecewascapturedduring the game, it prints a messageindicating the capturedpieceusingSystem.out.println("Capturedpiece: " + capturedPiece);.

10-)ScannerClosure:

Finally, it closes the **Scanner**object**reader**using**reader.close()**

The main methodsetsup the gameloop, promptsplayersfortheirmoves, performsmovevalidation, andupdates the chessboard'sstateaccordingly. Itrepeatsthisprocessuntil the game is ended, providing an interactivechessgameexperience.

while(!board.isGameEnded()) {  
System.*out*.println("It is " + (board.isWhitePlaying() ? "White" :"Black") + "'s turn");  
Piecepiece= null;

In the providedcodesnippet, the whileloopiteratesuntil the **isGameEnded()**method of the **ChessBoard**objectreturns**false**. Thismeans the loopwillcontinue as long as the game has not ended.

Inside the loop, the currentplayer'sturn is determinedusing the **isWhitePlaying()**method of the **ChessBoard**object. If it returns**true**, it means it is White'sturn, andif it returns**false**, it means it is Black'sturn. Thisinformation is usedtoprint the appropriatemessageindicatingwhoseturn it is.

The variable**piece** is initially set to**null**. Itwill be usedtostore the **Piece**objectthat the playerwantstomove. The program willprompt the playertoenter the location of the piece they wanttomove.

A **do-while**loop is usedtorepeatedly ask the playerfor the piece'slocationuntil a validpiece is selected. Itchecksif the enteredlocationcorrespondsto a validpiece on the chessboardbyusing the **getPieceAt()**method of the **ChessBoard**object. If the selectedlocationdoes not have a piece (**piece == null**), the program willdisplay an errormessageandprompt the playertoenter a validlocation.

Additionally, it checksif the selectedpiecebelongsto the currentplayer. If the color of the piece (obtainedusing**getColor()**method) does not match the color of the currentplayer (determinedby**isWhitePlaying()**), it meansit's not the player'sturn. Inthatcase, an errormessage is displayed, and the **piece**variable is set to**null**againtorepeat the loopandpromptfor a validpiece.

Once a validpiece is selected, the program proceedsto ask the playerfor the newlocationwhere they wanttomove the piece. Itusesanother**do-while**looptorepeatedly ask for a newlocationuntil a validmove is entered. The validity of the move is checkedusing the **canMove()**method of the selected**Piece**object.

If the move is valid, the **move()**method of the **Piece**object is calledtoperform the actualmove on the chessboard. After the move is made, the updatedchessboard is printedusing the **printBoard()**method.

The loopcontinuesuntil the gameends (**isGameEnded()**returns**true**), at whichpoint the loop is exited, and the program displays the "Game Over" messageanddeclares the winnerbased on the currentplayer.

Overall, thiscode segment manages the flow of the chessgame, alternatingbetweenplayerturns, validatingmoves, andupdating the gamestateuntil it reaches a conclusion.

Formun Üstü

Formun Altı

The **ChessBoard class**represents the chessboardandcontains the logicforinitializing the board, managingplayerturns, checkingforgameendconditions, accessingpiecesandsquares, andprinting the board.

**public ChessBoard():**This is the **constructorfor the ChessBoard class**. Itinitializes the chessboard, sets the initialplayerturntowhite, andcalls the **initialize() method**to set up the board withpieces in theirstartingpositions.

**privatevoidinitialize():**This**method**initializes the chessboardwithemptysquaresandsetsup the blackandwhitepieces in theirstartingpositions.

**public booleanisWhitePlaying():**Thismethodreturns a booleanvalueindicatingwhether it is currently the whiteplayer'sturn

**public voidnextPlayer():**Thismethodswitches the turnto the nextplayer. Ittoggles the value of the whitePlayingvariable.

**public booleanisGameEnded():** Thismethodchecksif the game has ended. Itcounts the number of whiteandblackpieces on the board andreturnstrueiftherearenopieces of onecolorremaining.

**publicPiecegetPieceAt(Stringlocation**): Thismethodtakes a stringrepresentation of a location on the board (e.g., "a1", "e5") andreturns the pieceobjectlocated at thatposition.

**public SquaregetSquareAt(Stringlocation**): Thismethodtakes a stringrepresentation of a location on the board andreturns the squareobjectlocated at thatposition.

**public Square[] getSquaresBetween(Squarelocation, SquaretargetLocation):**Thismethodtakes two squareobjectsrepresenting the startingandtargetlocationsandreturns an array of squaresrepresenting the squaresbetweenthem (excluding the startingandtargetsquares).

**public voidprintBoard():**Thismethodprints the currentstate of the chessboard, including the pieces on eachsquare, in a visualrepresentation.

publicclassChessBoard {  
public static final int*WHITE* = 0;  
 public static final int*BLACK* = 1;  
privateSquare[][] board; // the 64 squares of the board  
privatebooleanwhitePlaying; // indicatesifit'scurrentlywhiteplayer'sturn  
protectedbooleanisGameEnded; // indicatesif the game has ended  
  
  
public ChessBoard() {  
board = newSquare[8][8];  
whitePlaying= true;  
isGameEnded= false;  
initialize();  
}

In the ChessBoard constructor, **the initialize() method** is calledto set up the chessboardwithpieces in theirstartingpositions. Let's break downwhathappens in the initialize() method:

**board = newSquare[8][8]**;: Thislinecreates a 2D array of **Squareobjects**withdimensions 8x8, representing the 64 squares of the chessboard.

**whitePlaying = true;:**Thislinesets the initialvalue of the **whitePlayingboolean**variabletotrue, indicatingthat it is currently the whiteplayer'sturn.

**isGameEnded = false**;: Thislinesets the initialvalue of **the isGameEndedbooleanvariable**tofalse, indicatingthat the game has not ended yet.

The subsequentlines set up the initialpositions of the pieces on the chessboard. Each**board[row][col]**square is assigned a specificpieceusing the **setPiece() method**. The blackpiecesareplaced on the first two rows (rows 0 and 1), and the whitepiecesareplaced on the last two rows (rows 6 and 7). The specificpiecetype (rook, knight, bishop, queen, king, orpawn) is createdusing

the correspondingclassconstructors (**newRook(), new Knight(), etc.**) andpassed the color (blackorwhite) and the square it occupies (**board[row][col]**).

Bycalling**the initialize() method** in the constructor, the chessboard is set upwith the initialconfiguration of pieces, readyfor the gameto start.

1. **initialize() method**:

privatevoidinitialize() { // method  
 // initialize the board withemptysquares  
for(introw = 0; row<8; row++) {  
for(intcol = 0; col<8; col++) {  
board[row][col] = newSquare(row, col, this);  
}  
 }  
// set upblackpieces  
board[0][0].setPiece(newRook(*BLACK*, board[0][0]));  
board[0][1].setPiece(newKnight(*BLACK*, board[0][1]));  
board[0][2].setPiece(newBishop(*BLACK*, board[0][2]));  
board[0][3].setPiece(newQueen(*BLACK*, board[0][3]));  
board[0][4].setPiece(newKing(*BLACK*, board[0][4]));  
board[0][5].setPiece(newBishop(*BLACK*, board[0][5]));  
board[0][6].setPiece(newKnight(*BLACK*, board[0][6]));  
board[0][7].setPiece(newRook(*BLACK*, board[0][7]));  
for(intcol = 0; col<8; col++) {  
board[1][col].setPiece(newPawn(*BLACK*, board[1][col]));  
}  
// set upwhitepieces  
board[7][0].setPiece(newRook(*WHITE*, board[7][0]));  
board[7][1].setPiece(newKnight(*WHITE*, board[7][1]));  
board[7][2].setPiece(newBishop(*WHITE*, board[7][2]));  
board[7][3].setPiece(newQueen(*WHITE*, board[7][3]));  
board[7][4].setPiece(newKing(*WHITE*, board[7][4]));  
board[7][5].setPiece(newBishop(*WHITE*, board[7][5]));  
board[7][6].setPiece(newKnight(*WHITE*, board[7][6]));  
board[7][7].setPiece(newRook(*WHITE*, board[7][7]));  
for(intcol = 0; col<8; col++) {  
board[6][col].setPiece(newPawn(*WHITE*, board[6][col]));  
}  
}

The method**initialize()**initializes the chessboardbycreating a 2D array of squaresandplacing the pieces in theirstartingpositions.

* No parametersarepassedto the method.
* The methoddoes not returnanything (**void**returntype).
* The functionality is implementedbyiteratingover the rowsandcolumns of the chessboardarrayandcreating a new**Square**object at eachposition. The blackandwhitepiecesarethen set on the correspondingsquaresusing the **setPiece()**method of the **Square**class, with the appropriatepiecetypeandcolorpassed as arguments.

The **initialize() method** is responsibleforsettingup the chessboardwith the initialpositions of the pieces.

-the first nested loopinitializes the **board array**with**Squareobjects**. Ititeratesovereachrowandcolumn (from 0 to 7) andassigns**a newSquareobjectto board[row][col].** The **Squareconstructor**takes the row, column, and a reference to the **ChessBoard object**itself (this).

-Afterinitializing the emptysquares, the methodproceedsto set up the blackpieces. Each**board[row][col]**squarecorrespondingto a specificposition on the board is assigned a specificpieceusing the **setPiece() method**. The piecetypesandpositionsare as follows:

Rooks: board[0][0] and board[0][7]

Knights: board[0][1] and board[0][6]

Bishops: board[0][2] and board[0][5]

Queen: board[0][3]

King: board[0][4]

Pawns: board[1][0] to board[1][7]

The pieceobjectsarecreatedusingtheircorrespondingclassconstructors (**newRook(), new Knight(),**etc.) andpassed the color (black) and the square they occupy (**board[row][col]).**

**-**Aftersettingup the blackpieces, the methodproceedsto set up the whitepieces. Each**board[row][col]**squarecorrespondingto a specificposition on the board is againassigned a specificpieceusing the **setPiece()**method. The piecetypesandpositionsare as follows:

Rooks: **board[7][0]**and**board[7][7]**

Knights: **board[7][1]**and**board[7][6]**

Bishops: **board[7][2]**and**board[7][5]**

Queen: **board[7][3]**

King: **board[7][4]**

Pawns: **board[6][0]**to**board[6][7]**

Similarto the blackpieces, the whitepieceobjectsarecreatedusingtheircorrespondingclassconstructors (**newRook()**, **new Knight()**, etc.) andpassed the color (white) and the square they occupy (**board[row][col]**).

By the end of the **initialize()**method, the **board**array is populatedwith**Square**objects, and the initialpositions of the blackandwhitepiecesare set on the board. Thisestablishes the startingconfigurationfor a game of chess.

**A board can have 64 squares. How did you define the relationbetween Board andSquareobjects? Explain**.

In the givenimplementation, the relationbetween the **ChessBoard**and**Square**objectsis establishedthrough the **board**array, which is a 2-dimensional array of **Square**objects.

The **board**arrayrepresents the chessboarditself, andeach element of the arraycorrespondsto a specificsquare on the board. Forexample, **board[0][0]**represents the top-leftsquare (a8 in standardalgebraicnotation), and**board[7][7]**represents the bottom-rightsquare (h1 in standardalgebraicnotation).

Byinitializing the **board**arraywith**Square**objects in the **ChessBoard**constructor, eachsquare on the board is associatedwith a specific**Square**object. The **Square**objectscontaininformationabouttheirownposition (rowandcolumn) on the board and a reference to the **ChessBoard**object.

Thisarrangementallowsforeasyaccessto a specificsquare on the board byusingitsrowandcolumnindices in the **board**array. Forexample, toaccess the **Square**object at position (row = 2, column = 3), you woulduse**board[2][3]**.

Overall, the **board**arrayserves as a containerfor the **Square**objectsandprovides a convenientwaytorepresent the relationbetween the **ChessBoard**and**Square**objects in a grid-likestructurethatmimics the layout of a chessboard.

2-) isWhitePlaying() method:

public booleanisWhitePlaying() {  
returnwhitePlaying;  
}

The **isWhitePlaying()**method is a **gettermethod (****accessormethod)**thatreturns the value of the **whitePlaying**variable. It is usedtodeterminewhether it is currently the whiteplayer'sturn in the game.

**Description**: Thismethoddetermineswhether it is currently the whiteplayer'sturn in the game.

**Parameters**: None

Returns: A booleanvalueindicatingwhether it is the whiteplayer'sturn. **true**if it is the whiteplayer'sturn, **false**otherwise.

**Functionality:** The methodsimplyreturns the value of the **whitePlaying**variable, which is a booleanvariablethatkeepstrack of whetherthe whiteplayer is currentlyplaying. Itprovides a convenientwaytocheck the currentplayer'sturn in the gamelogic. If the value is **true**, it means it is the whiteplayer'sturn, andif the value is **false**, it means it is the blackplayer'sturn.

Usage is:

if(board.isWhitePlaying()) {  
// It's the whiteplayer'sturn  
} else {  
// It's the blackplayer'sturn  
}

3-) **nextPlayer() method**:

public voidnextPlayer() {  
whitePlaying= !whitePlaying;  
}

the method**nextPlayer()**switches the turnto the nextplayerbytoggling the **whitePlaying**booleanvariable.

* No parametersarepassedto the method.
* The methoddoes not returnanything (**void**returntype).
* The functionality is implementedbynegating the value of the **whitePlaying**variableusing the logical NOT operator**!**. Thiseffectivelyswitches the turnfrom the currentplayerto the nextplayer.

4**-) isGameEnded() method:**

public booleanisGameEnded() {  
//Endgamecontrol : you can end the gamewhennopiecesexistfromonecolor  
intnumWhitePieces = 0;  
intnumBlackPieces = 0;  
for(introw = 0; row<8; row++) {  
for(intcol = 0; col<8; col++) {  
Piecepiece = board[row][col].getPiece();  
if(piece != null) {  
if(piece.getColor() == *WHITE*) {  
numWhitePieces++;  
} else {  
numBlackPieces++;  
}  
 }  
 }  
 }  
returnnumWhitePieces == 0 || numBlackPieces == 0;  
}

The **isGameEnded()**methodchecksif the game has endedbydeterminingiftherearenopiecesremainingforonecolor on the chessboard.

* Parameters: No parameters.
* Return: Itreturns a booleanvalueindicatingwhether the game has ended (**true**if the game has ended, **false**otherwise).
* Implementation: The methoditeratesthrougheachsquare on the chessboardandchecksifthere is a piecepresent. If a pieceexists, it increments the count of piecesfor the respectivecolor (whiteorblack). Finally, it checksif the count of whitepiecesorblackpieces is zero, indicatingthatonecolor has nopiecesleftand the game has ended.

5-) **getSquareAt(Stringlocation**)method:

publicSquaregetSquareAt(Stringlocation) {  
intcol = location.charAt(0) - 'a';  
introw = 8 - (location.charAt(1) - '0');  
returnboard[row][col];  
}

The **getSquareAt(Stringlocation)**methodtakes a stringrepresenting a location on the chessboard (e.g., "a1", "e5") andreturns the corresponding**Square**object on the chessboard.

Parameters:

* **location**: A stringrepresenting the location on the chessboard (e.g., "a1", "e5").

Return:

* The **Square**objectlocated at the specifiedlocation on the chessboard.

Functionality:

1. The methodfirstextracts the columnindexbysubtracting the character 'a' from the firstcharacter of the **location**stringusing the expression**location.charAt(0) - 'a'**. Thisconverts the charactertoitscorrespondingnumericalvalue.
2. The methodextracts the rowindexbysubtracting the character '0' from the secondcharacter of the **location**string, representing the rownumber, using the expression**location.charAt(1) - '0'**. Thisconverts the charactertoitscorrespondingnumericalvalue.
3. The rowindex is furtheradjustedbysubtracting it from 8 tomatch the arrayindexingused in the **board**array.
4. The methodreturns the **Square**objectlocated at the calculatedrowandcolumnindices on the chessboard.

DIFFERENT WAY :

publicSquaregetSquareAt(Stringlocation) {  
intcol = Character.*toLowerCase*(location.charAt(0)) - 'a';  
introw = Integer.*parseInt*(location.substring(1)) - 1;  
returnboard[row][col];  
}

Inthisalternativeimplementation, weconvert the firstcharacter of the **location**stringtolowercaseusing**Character.toLowerCase()**tohandlebothlowercaseanduppercaseletters. Wesubtract**'a'**from the lowercasecharactertodetermine the columnindex.

For the rowindex, weextract the numericpart of the **location**stringusing**substring(1)**andconvert it to an integerusing**Integer.parseInt()**. Wesubtract 1 from the parsedvaluetomatch the arrayindexing.

The alternativeimplementationachieves the sameresult but handlesbothlowercaseanduppercaselettersforcolumnindexingandparses the numericpart of the **location**stringtodetermine the rowindex

6-) **getPieceAt(Stringlocation**) method:

public PiecegetPieceAt(Stringlocation) {  
//Square board = getSquareAt(location)  
returngetSquareAt(location).getPiece();  
}

The **getPieceAt(Stringlocation)**methodretrieves the piecelocated at the specifiedposition on the chessboard.

* Parameter: **location** is a stringrepresenting the position on the chessboard (e.g., "a1", "e5").
* Return: Itreturns the **Piece**objectlocated at the specifiedposition, or**null**ifthere is nopiece at thatposition.
* Implementation: The methoduses the **getSquareAt(location)**methodtoretrieve the **Square**objectcorrespondingto the specifiedposition. Itthencalls the **getPiece()**method on the **Square**objecttoretrieve the piecelocated at thatpositionandreturns it.

7-) **getSquaresBetween(Squarelocation, SquaretargetLocation)**method:

public Square[] getSquaresBetween(Squarelocation, SquaretargetLocation) {  
introwDiff = targetLocation.getRow() - location.getRow();  
intcolDiff = targetLocation.getCol() - location.getCol();  
  
// Calculate the direction of the movement  
introwDir = Integer.*compare*(rowDiff, 0);  
intcolDir = Integer.*compare*(colDiff, 0);  
  
intsteps = Math.*max*(Math.*abs*(rowDiff), Math.*abs*(colDiff)) ;  
Square[] between = newSquare[steps-1];  
  
for(inti = 0; i < steps-1; i++) {  
introw = location.getRow() + rowDir \* (i+1);  
intcol = location.getCol() + colDir \* (i+1) ;  
between[i] = board[row][col];  
}  
  
returnbetween;  
}

The **getSquaresBetween(Squarelocation, SquaretargetLocation)**methodcalculates the squaresbetween two givensquares on a chessboardandreturns an array of thosesquares.

Parameters:

* **location**: The startingsquare.
* **targetLocation**: The targetsquare.

Return:

* Itreturns an array of **Square**objectsrepresenting the squaresbetween the **location**and**targetLocation**.(excluding the endpoints)

Functionality:

1. The methodcalculates the rowdifferencebetween**targetLocation**and**location**bysubtracting the row of **location**from the row of **targetLocation**, andsimilarlyfor the columndifference.
2. Itdetermines the direction of movementbyusing (bycalculating the sign of the rowandcolumndifferences.)**Integer.compare()** on the rowandcolumndifferences, assigning 1 ifpositive, -1 ifnegative, or 0 if the difference is zero.
3. Itcalculates the number of stepsrequiredtomovefrom**location**to**targetLocation**bytaking the maximumabsolutevaluebetween the rowdifferenceandcolumndifference.
4. A newarray**between** is createdwith a size of **steps-1**tostore the squaresbetween**location**and**targetLocation**.
5. Using a loop, it iterates**steps - 1**timesandcalculates the rowandcolumnindices of the squaresbetween**location**and**targetLocation**byadding the respectivedirectionaloffsetsmultipliedby the loopindexto the rowandcolumn of **location**.
6. Itassigns the correspondingsquarefrom the **board**arrayto the **between**array.
7. Finally, it returns the **between**arraycontaining the squaresbetween**location**and**targetLocation**.

8-) **printBoard()**method:

public voidprintBoard() {  
System.*out*.println(" A B C D E F G H");  
System.*out*.println(" -------------------------------");  
for(introw = 0; row<8; ++row) {  
System.*out*.print(8 - row + " ");  
for(intcol = 0; col<8; ++col) {  
System.*out*.print("| " + board[row][col].toString() + " ");  
  
}  
System.*out*.print("| " + (8 - row));  
System.*out*.println();  
System.*out*.println(" -------------------------------");  
}  
System.*out*.println(" A B C D E F G H");  
}

Thismethodprints the currentstate of the chessboard, displaying the positions of the pieces.

* Therearenoparameters.
* Itdoesn'treturnanything (**void**).
* The methodprints the columnheadersand a horizontal linetorepresent the board'slayout.
* Ittheniteratesthrougheachrowandcolumn of the board, printing the stringrepresentation of eachsquare'spieceenclosed in "|".
* Afterprinting the row of pieces, it prints the rownumber on the side.
* Another horizontal line is printedtoseparateeachrow.
* Finally, it prints the columnheadersagain at the bottom of the board.
* The functionality is implementedbyaccessingeachsquare on the board andcalling the **toString()**methodtoobtain the stringrepresentation of the piece on thatsquare. The stringsareconcatenatedandformattedtocreate the visualrepresentation of the board.

Formun Üstü

**Square Class:**

publicclassSquare {  
private final introw;  
private final intcol;  
privatePiecepiece;  
privateChessBoard board;

1. **private final introw**: Thismembervariablerepresents the rownumber of the square on the chessboard. It is marked as final, indicatingthatonceinitialized, itsvaluecannot be changed.
2. **private final intcol**: Thismembervariablerepresents the columnnumber of the square on the chessboard. Like the rowvariable, it is alsomarked as final.
3. **privatePiecepiece**: Thismembervariable**holds the Pieceobject**that is currentlyoccupying the square. It can be nullifthere is nopiece on the square.
4. **private ChessBoard board**: Thismembervariableholds a reference to the ChessBoardobjecttowhich the squarebelongs. Itallows the squaretoaccessandinteractwith the chessboard'sstateandothersquares.

Thesemembervariablesaredeclared as private, whichmeans they can only be accessedormodifiedwithin the Squareclassitself. Byencapsulatingthesevariables, the classensuresthattheirvaluesarecontrolledandmaintainedproperly.

Overall, the Squareclassrepresents an individualsquare on the chessboard, storingitsposition (rowandcolumn), the pieceoccupying it, and a reference to the chessboard it belongsto.

1-) SquareConstructor

publicSquare(introw, intcol, ChessBoard board) {  
this.row= row;  
this.col= col;  
this.piece= null;  
this.board= board;  
}

thisconstructor is responsibleforinitializing a **Square**objectwith the providedrowandcolumnvalues, setting the pieceto**null**, andassociating it with the given**ChessBoard**.

* **introw**: Itrepresents the *rownumber of the square on the chessboard*. Thisparameter is usedtoinitialize the **row**membervariable of the **Square**object.
* **intcol**: Itrepresents the *columnnumber of the square on the chessboard*. Thisparameter is usedtoinitialize the **col**membervariable of the **Square**object.
* **ChessBoard board**: Itrepresents the *chessboardtowhich the squarebelongs*. Thisparameter is usedtoinitialize the **board**membervariable of the **Square**object.
* **this.row = row;**:Itassigns the value of the **row**parameterto the **row**membervariable of the current**Square**object.
* **this.col = col;**:Itassigns the value of the **col**parameterto the **col**membervariable of the current**Square**object.
* **this.piece = null;**: Itsets the **piece**membervariableto**null**, indicatingthatthere is nochesspieceplaced on thissquareinitially.
* **this.board = board;**: Itassigns the **ChessBoard**objectpassed as a parameterto the **board**membervariable of the current**Square**object, establishing the associationbetween the squareand the chessboard
* The constructordoes not returnanything.
* The methodassigns the rowandcolumnvaluespassed as parametersto the correspondingmembervariables (**this.row**and**this.col**). Itsets the **piece**variableto**null**, indicatingthatthere is nopiece on the square. Finally, it assigns the provided**ChessBoard**objectto the **board**variable, establishing the associationbetween the squareand the chessboard.

2-) **getRow()**method:

The **getRow()**method is a gettermethod in the **Square**class.

* Parameters: Thismethoddoes not takeanyparameters.
* Return: Itreturns the rownumber of the square.
* Implementation: The implementation is straightforward. Itsimplyreturns the value of the **row**membervariable, whichrepresents the rownumber of the square. Bycallingthismethod on a **Square**object, you can retrieve the rownumberassociatedwiththatsquare.

3-) **getCol()**method:

public intgetCol() {  
returncol;  
}

The **getCol()**method is a gettermethod in the **Square**class.

* Parameters: Thismethoddoes not takeanyparameters.
* Return: Itreturns the columnnumber of the square.
* Implementation: The implementation is straightforward. Itsimplyreturns the value of the **col**membervariable, whichrepresents the columnnumber of the square. Bycallingthismethod on a **Square**object, you can retrieve the columnnumberassociatedwiththatsquare.

4-) **getPiece()**method:

public PiecegetPiece() {  
returnpiece;  
}

The **getPiece()**method in the **Square**classreturns the **Piece**objectassociatedwith the square.

* Therearenoparametersforthismethod.
* Itreturns a **Piece**objectthatrepresents the piece on the square. Ifthere is nopiece on the square, it returns**null**.
* The implementation is straightforward. Itsimplyreturns the **piece**instancevariable, whichholds the reference to the pieceobjectassociatedwith the square.

it can be useful in scenarioswhere you needtoretrieve the piece on a particularsquareforvariousoperations, such as checkingforvalidmoves, capturingpieces, orevaluating the gamestate.

5-) **setPiece()**method:

publicvoidsetPiece(Piecepiece) {  
this.piece= piece;  
}

The **setPiece()**method in the **Square**classsets the **Piece**object on the square.

* The **piece**parameter is the **Piece**objectto be set on the square.
* The methoddoesn'treturnanything (**void**returntype).
* The implementation is simple. Itassigns the **piece**parameterto the **piece**instancevariable of the **Square**object, effectivelysetting the piece on the square.

6-) **clear()**method:

publicvoidclear() {  
this.piece= null;  
}

The **clear()**method in the **Square**classclears the piecefrom the square, effectivelyremovinganypiecethatwaspreviously set on it.

* The methoddoesn'ttakeanyparameters.
* Itdoesn'treturnanything (**void**returntype).
* The implementation is straightforward. Itsets the **piece**instancevariable of the **Square**objectto**null**, indicatingthatthere is nopiece on the square.

7-) **isEmpty()**method:

public booleanisEmpty() {  
returnpiece== null;  
}

The **isEmpty()**method in the **Square**classchecksif the square is empty, meaningthere is nopiece on it.

* The methoddoesn'ttakeanyparameters.
* Itreturns a booleanvalue (**true**if the square is empty, **false**otherwise).
* The implementation is straightforward. Itchecksif the **piece**instancevariable of the **Square**object is **null**. If it is **null**, it means the square is empty, so the methodreturns**true**. Otherwise, it returns**false**.

8**-) isAtLastRow(intcolor)**method:

publicbooleanisAtLastRow(intcolor) {  
if(color == ChessBoard.*WHITE*) {  
returnrow== 0;  
} else { // Black color  
returnrow== 7;  
// Implementcheckforlastrowbased on color  
}  
}

The **isAtLastRow(intcolor)**method in the **Square**classchecksif the square is located at the lastrow of the board for the givencolor.

* The **color**parameterspecifies the color of the player (**ChessBoard.WHITE**or**ChessBoard.BLACK**).
* Itreturns a booleanvalue (**true**if the square is at the lastrowfor the givencolor, **false**otherwise).
* The implementationchecksif the **row**instancevariable of the **Square**objectmatches the lastrowindexbased on the givencolor. If the color is **ChessBoard.WHITE**, it checksif the **row** is 0. If the color is **ChessBoard.BLACK**, it checksif the **row** is 7. The methodreturns**true**if the condition is satisfied, indicatingthat the square is at the lastrowfor the givencolor. Otherwise, it returns**false**.
* isAtLastRowmethod in the Squareclasschecksif the row of the squarematches the lastrowbased on the color of the piece. Forwhitepieces, the lastrow is row 0, andforblackpieces, the lastrow is row 7.

9**-) isAtSameColumn(Square s )**method:

publicbooleanisAtSameColumn(Square s) {  
returnthis.col== s.getCol();  
}

The **isAtSameColumn(Square s)**method in the **Square**classchecksif the currentsquareand the specifiedsquare**s**arelocated in the samecolumn.

* The **s**parameter is the **Square**objecttocomparewith.
* Itreturns a booleanvalue (**true**if the squaresare in the samecolumn, **false**otherwise).
* The implementationcompares the **col**instancevariable of the current**Square**object (**this**) with the column of the specified**Square**object**s**obtainedbycallingits**getCol()**method. If the columnsmatch, the methodreturns**true**, indicatingthat the squaresare in the samecolumn. Otherwise, it returns**false**.

10-) **isNeighborColumn(Square s)**method:

publicbooleanisNeighborColumn(Square s) {  
returnMath.*abs*(col- s.getCol()) == 1;  
}

The **isNeighborColumn(Square s)**method in the **Square**classchecksif the currentsquareand the specifiedsquare**s**arelocated in neighboringcolumns.

* The **s**parameter is the **Square**objecttocomparewith.
* Itreturns a booleanvalue (**true**if the squaresare in neighboringcolumns, **false**otherwise).
* The implementationcalculates the absolutedifferencebetween the **col**instancevariable of the current**Square**object (**this**) and the column of the specified**Square**object**s**obtainedbycallingits**getCol()**method. If the absolutedifference is equalto 1, it means the squaresare in neighboringcolumns, and the methodreturns**true**. Otherwise, it returns**false**.

11-)**getRowDistance(Square s)**method:

public intgetRowDistance(Square s) {  
returns.getRow() - row;  
}

The **getRowDistance(Square s)**method in the **Square**classcalculates the rowdistancebetween the currentsquareand the specifiedsquare**s**.

* The **s**parameter is the **Square**objecttocalculate the rowdistanceto.
* Itreturns an integervaluerepresenting the difference in rowsbetween the currentsquareand**s**.
* The implementationsubtracts the **row**instancevariable of the current**Square**objectfrom the row of the specified**Square**object**s**obtainedbycallingits**getRow()**method. The resultrepresents the rowdistancebetween the two squares.

12-) **getColDistance(Square s)**method:

publicintgetColDistance(Square s) {  
returnMath.*abs*(col- s.getCol());  
}

The **getColDistance(Square s)**methodcalculates the columndistancebetween the current**Square**objectand the specified**Square**object**s**.

* Parameter: **s** is the **Square**objecttocalculate the columndistanceto.
* Return: Itreturns an integerrepresenting the absolutedifference in columnsbetween the currentsquareand**s**.
* Implementation: The methodsubtracts the column of the currentsquare (**col**) from the column of **s** (**s.getCol()**) andtakes the absolutevalueusing**Math.abs()**toensure a positiveresult.

13-) **getBoard()**method:

public ChessBoard getBoard() {  
returnboard;  
}

The **getBoard()**methodreturns the **ChessBoard**objectassociatedwith the current**Square**.

* Parameter: The methoddoes not takeanyparameters.
* Return: Itreturns the **ChessBoard**objectassociatedwith the current**Square**.
* Implementation: The **board**instancevariable is simplyreturned.

14-) **toString()**method:

@Override  
public StringtoString() {  
if(piece== null) {  
return" ";  
} else {  
returnpiece.toString();  
}  
}

The **toString()**methodoverrides the defaultimplementation of **toString()**from the **Object**classtoprovide a stringrepresentation of the **Square**object.

* Parameter: The methoddoes not takeanyparameters.
* Return: Itreturns a stringrepresentation of the **Square**object. If the **piece** is **null**, it returns a spacecharacter. Otherwise, it returns the stringrepresentation of the **piece**.
* Implementation: The methodchecksif the **piece** is **null**. If it is, it returns a spacecharacter. Otherwise, it calls the **toString()**method of the **piece**objecttogetitsstringrepresentation.

15-) **putNewQueen(intcolor)**method:

public voidputNewQueen(intcolor) {  
// Create a new Queen pieceandassign it to the square  
Queen queen = newQueen(color, this);  
setPiece(queen);  
}

The **putNewQueen**methodcreates a new Queen piece of the specifiedcolorandassigns it to the currentsquare.

* Parameter: **color**represents the color of the Queen pieceto be created. It can be either**ChessBoard.WHITE**or**ChessBoard.BLACK**.
* Return: The methoddoes not returnanything (**void**).
* Implementation: Inside the method, a new**Queen**object is createdwith the specifiedcolorand the currentsquare (**this**) as itslocation. The **setPiece**method is thencalledtoassign the created**Queen**objectto the currentsquare.

Formun Üstü

if (piece.getColor() == WHITE) {

numWhitePieces++;

} else {

numBlackPieces++;

}

Inthiscode, piecerefersto an instance of the Piececlass, andgetColor() is a methodthatreturns the color of the piece. The conditionpiece.getColor() == WHITE compares the color of the pieceto the constant WHITE (presumablydefinedelsewhere in the code). If the color of the piece is white, thennumWhitePieces is incrementedby 1. Otherwise, if the color of the piece is not white (implyingit'sblack), thennumBlackPieces is incrementedby 1.

Thiscode is usedtokeeptrack of the number of whiteandblackpieces on the chessboard. Byincrementing the respectivecountersbased on the color of the piece, you can maintain a count of how manypieces of eachcolorarepresent on the board

publicbooleanisGameEnded() {

booleanwhitePiecesExist = false;

booleanblackPiecesExist = false;

for (introw = 0; row< 8; row++) {

for (intcol = 0; col< 8; col++) {

Piecepiece = board[row][col].getPiece();

if (piece != null) {

if (piece.getColor() == WHITE) {

whitePiecesExist = true;

} else {

blackPiecesExist = true;

}

}

}

}

return !whitePiecesExist || !blackPiecesExist;

}

Inthismodifiedimplementation, weintroduce two booleanvariableswhitePiecesExistandblackPiecesExisttokeeptrack of whetherthereareanyremainingpiecesforeachcolor. Weiteratethrough the board andcheckif a pieceexists at eachsquare. If a piece is found, weupdate the correspondingbooleanvariablebased on itscolor. Finally, wereturntrueifeitherwhitePiecesExistorblackPiecesExist is false, indicatingthatonecolor has noremainingpiecesand the game has ended.

--------------------------------------------------------------------------------------------------------------------------------------

**Explain how Main class benefits from polymorphism. Explain, which methods and classes can be defined abstract in Piece hierarchy. Is there a code reuse in your implementation?**

In the **Main** class, polymorphism is evident in several places:

Here's how the **Main** class benefits from polymorphism with examples:

1. **Dynamic Method Dispatch:** In the **Main** class, the **Piece** reference variables, such as **Piece piece** and **Piece targetPiece**, are used to interact with different pieces on the chessboard. Let's take an example with the **canMove()** method. When the **Main** class invokes **piece.canMove(to)**, the actual implementation of **canMove()** that gets executed is determined dynamically at runtime based on the type of the object referred to by the **piece** variable. This is known as dynamic method dispatch. Here's an example:
2. // Assuming the piece variable refers to a Bishop object  
   piece.canMove(to);
3. In this case, the **canMove()** method of the **Bishop** class will be executed because the **piece** variable is referring to an object of the **Bishop** class. Similarly, when the **Main** class invokes **targetPiece.canMove(to)**, the appropriate **canMove()** method implementation of the corresponding piece will be called.
4. **Uniform Access to Different Piece Types:** The **Main** class benefits from polymorphism by treating different pieces uniformly through the abstract **Piece** type. This allows the **Main** class to work with different types of pieces without knowing their specific implementations. Here's an example:
5. / Assuming the piece variable refers to a Knight object  
   System.out.println(piece.toString());

In this case, the **toString()** method implementation of the **Knight** class will be called. The **Main** class doesn't need to have separate code for each piece type; it can treat all pieces generically through the common **Piece** interface.

1. **Flexible Piece Movement:** By leveraging polymorphism, the **Main** class can handle piece movement without explicit type checks or conditional statements. The **Main** class relies on the **canMove()** method to determine the validity of a move for any given piece. Here's an example:
2. // Assuming the piece variable refers to a Pawn object  
   if (piece.canMove(to)) {  
    piece.move(to);  
    }

In this case, the **Main** class can simply invoke **piece.canMove(to)** without knowing the specific type of the piece. The appropriate implementation of **canMove()** in the **Pawn** class will be called, and if the move is valid, the **piece.move(to)** method will be executed.

By utilizing polymorphism, the **Main** class achieves flexibility, extensibility, and uniformity in interacting with different types of pieces, enabling clean and concise code without the need for excessive type checking or conditional logic.

The **capturedPiece** variable is of type **Piece**, allowing it to hold any captured piece object during the game. This allows for flexibility in handling captured pieces, regardless of their specific type.

The **Piece** class is an abstract class that serves as a base class for different types of chess pieces. It contains common properties and methods that are shared among all chess pieces, while leaving the implementation of specific behavior to its subclasses.

Explanation of the class components:

* **color**: An integer representing the color of the piece.
* **location**: A reference to the **Square** object on which the piece is currently located.
* **Piece(int color, Square location)**: The constructor method that initializes the **color** and **location** properties of the piece.
* **canMove(String to)**: An abstract method that checks if the piece can move to the specified location (**to**). The implementation of this method is left to the subclasses.
* **move(String to)**: An abstract method that performs the movement of the piece to the specified location (**to**). The implementation of this method is left to the subclasses.
* **getColor()**: A method that returns the color of the piece.
* **toString()**: An abstract method that returns a string representation of the piece. The implementation of this method is left to the subclasses.

Now let's discuss which methods and classes can be defined abstract in the **Piece** hierarchy:

1. **Abstract Methods:** In the **Piece** class hierarchy, you can define the **canMove()** and **move()** methods as abstract in the **Piece** class. By declaring these methods as abstract, you require every subclass of **Piece** to provide its own implementation. The specific movement rules and logic for each chess piece type are different, so making these methods abstract ensures that each subclass implements its unique behavior. Abstract methods can be declared in the **Piece** class as follows:
2. public abstract boolean canMove(String to);
3. public abstract void move(String to);

By making these methods abstract, you enforce that every subclass of **Piece** must provide its own implementation of these methods.

1. **Abstract Class:** You can also consider making the **Piece** class itself an abstract class. An abstract class cannot be instantiated, but it serves as a common base for its subclasses. Since a chess piece itself is an abstract concept, defining the **Piece** class as abstract makes sense. This would also allow you to define common fields and methods shared by all chess pieces in the **Piece** class. However, it depends on your specific design and whether you have any shared functionality among all chess pieces. If you don't have any shared functionality, you can keep the **Piece** class as a concrete class.