Coding Progression Report

Author:   
Alex Mason

Moodle Name:   
ZEAC061

Student Number:  
100873598

Email Address:   
[ZEAC061@live.rhul.ac.uk](mailto:ZEAC061@live.rhul.ac.uk)

# Abstract

In this report, I will be focusing on talking about the development and coding progression of my Chess game now that most of the base code is completed. It will include code examples and discuss various data structures and methods used to program the game. There will also be a section covering any limitations or difficulties I faced, as well as factors I didn’t originally anticipate for, towards the end of the report.

# Introduction

My main motivation for writing about the coding progression in my project is because I feel it is a good way to document my code and any difficulties I have faced so far in report form. Not only this, but the documentations made in this report will aid my interim and final reports as the progression of my code can be expanded on as more code is written in term 2.

Another motivation for writing this report is that it would be interesting for readers to see how a Chess game (with Artificial Intelligence being added at a later stage) is developed from start to finish, and the methodologies used to do so.

Coding my Chess game thus far has been very enjoyable, but also challenging in places. Lots of conditions and validation goes into the game and thinking about the way in which I will program these conditions was sometimes a challenge, but at other times (depending on the condition I am programming) not as complex as first thought.

Something else which I have realised, especially since programming the game, is that it isn’t just about writing the conditions for a move or setting the position of a piece, but it is also the accuracy in doing so, so that the game’s rules are followed precisely and do not cause conflicts with the rest of the conditions for other methods inside the many other classes required for the game.

# Data Structures and Their Methods

In my Chess game, my main data structures are two TreeMaps. These are called “piecePos” and “tileOccupation”. These TreeMaps are very significant in the game because they store the absolute basics for the game, such as the positions for each piece, the positions on a chess board, and whether a tile is occupied or empty. They are both stored in a class named “TreeMaps”, where there are also five methods related to these data structures.

I also have smaller data structures in the form of ArrayLists. I have an ArrayList named ‘capturedPieces’ which contains a list of the captured pieces and gets updated as a piece is captured by an opponent piece. This is also inside the “TreeMaps” class. Although it is not actually a TreeMap itself, it assists the TreeMaps in storing pieces which no longer need to be mapped to a tile.

## Data Structures

PiecePos TreeMap

This TreeMap takes a tile co-ordinate (e.g. 1A) and maps it to a chess piece. This means that the chess piece is currently in that tile. When a chess piece is captured, its position becomes null so that it no longer exists (no longer mapped to a tile co-ordinate) on the chess board.

TileOccupation TreeMap

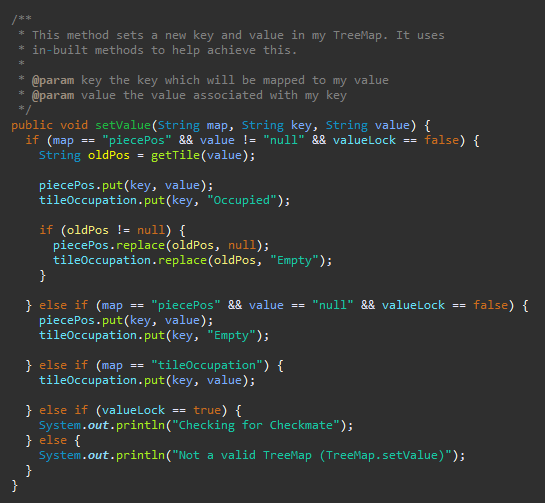
This TreeMap takes a tile co-ordinate (e.g. 1A) and maps it to a value which is the tile’s/square’s occupation status. A tile can either be “Occupied” or “Empty”. This correlates to if a piece is inside the tile or not. For example, if a piece is inside a given tile, the status will be “Occupied”. If isn’t a piece in the given tile, its status is “Empty”.

## Data Structure Methods

‘Set Value’ Method

My ‘setValue’ method takes three parameters; a String map, a String key, and a String value. The String map will be a map name (either “piecePos” or “tileOccupation”), the String key will be a map key (a tile co-ordinate), and the String value will be either a tile occupation status or a piece position (so “Occupied” or “1A”, for example).

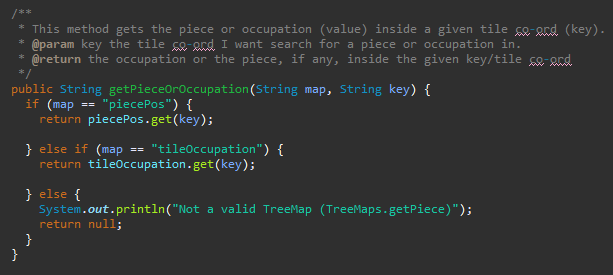
I designed the Set Value method so that it can be used my both TreeMaps to avoid code duplication, as most of the code would have been very similar in places and the same in other places if I were to code the method twice for the two TreeMaps. I have conditions to check if the values being entered is acceptable/appropriate for that map.

An example of such conditions includes the checking of the name of the map passed from the String map (which is one of the parameters). For example, if the map’s name is “tileOccupation”, I utilise the tileOccupation map and input a status to a corresponding tile. If the map’s name is “piecePos”, I set a piece’s position to a corresponding tile co-ordinate.  
The method also utilises a “valueLock” variable. This is a Boolean variable which is used to “lock”/prevent the setting of values when checking for a king in checkmate.

‘Get Piece or Occupation’ Method

My ‘getPieceOrOccupation’ method takes two parameters; a String map, and a String key. It simply returns a value (the piece or tile occupation status) from a given key (tile co-ordinate). It uses conditions to check the map name, so that we get a value from the key given in the relevant TreeMap.

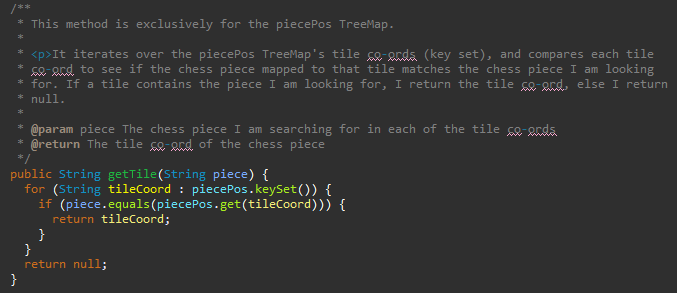
If the name of the TreeMap isn’t “piecePos” or “tileOccupation”, the method will use its else statement which prints “Not a valid TreeMap” and returns null as the value.



‘Get Tile’ Method

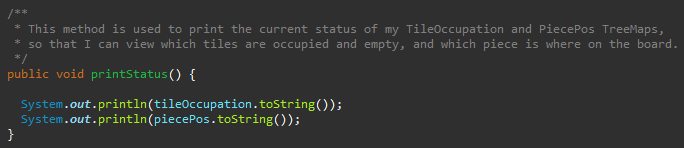
My ‘getTile’ method takes a single parameter, String piece, which is the name of a chess piece on the board. The purpose of this method is to iterate over the key set in the “piecePos” TreeMap and search for the tile containing the piece passed to the method. For example, if I used the getTile method just after starting the game and passed “WhitePawn1” as String piece, the method would return “2A”.

If the piece is not mapped to a tile, the method will return null. This could occur when a piece is captured or due to an error initialising the piece.

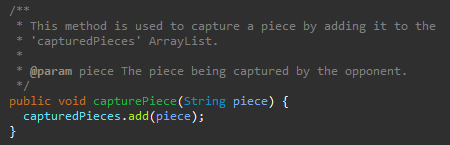


‘Print Status’ Method

My ‘printStatus’ method takes no parameters. It simply prints the contents of both my TreeMaps to the console. It is extremely useful for checking/validating that a Chess piece has moved from one tile to another and checking the current game state. Not only this, but it is a way of checking my keys and values updating without the need for a GUI which will come at a later stage in the project.



‘Capture Piece’ Method

My ‘capturePiece’ method takes a single parameter, String piece, which is the piece being captured by the opponent. It simply adds the chess piece to the ArrayList ‘capturedPieces’, so that the piece is no longer mapped to a tile on the chess board and is now recognised as captured.

# Validation

In each Piece type class, there is validation in place to verify that a piece type is being moved legally and in accordance to the rules of Chess. To code this validation, I used my research collated in my History and Rules of Chess report to ensure my validation was programmed in line with the rules of the game.

## Pawns

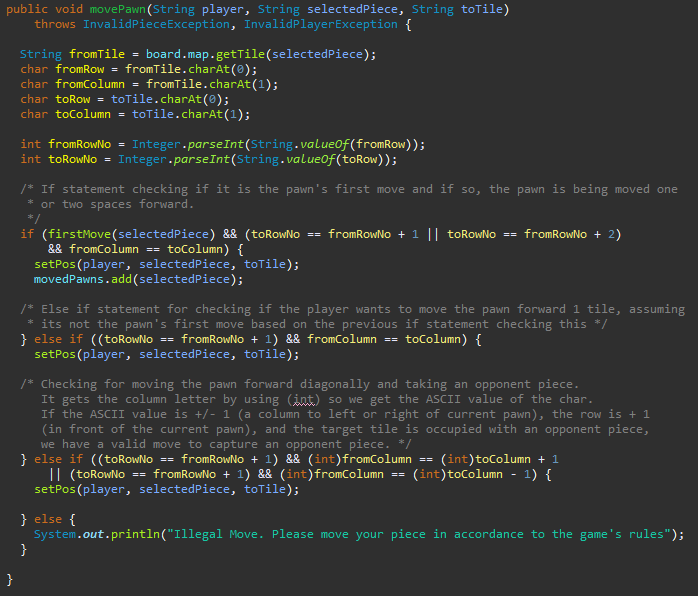
According to the rules for a Pawn; “*Pawns can move forward one square, if that square is unoccupied. If it has not yet moved, the pawn has the option of moving two squares forward provided both squares in front of the pawn are unoccupied. A pawn cannot move backward. Pawns are the only pieces that capture differently from how they move. They can capture an enemy piece on either of the two spaces adjacent to the space in front of them (i.e., the two squares diagonally in front of them) but cannot move to these spaces if they are vacant.”* [[1]](#_References)

Therefore, in my code for the movePawn method in my Pawn class, my first condition (surrounded in a red square below) uses a Boolean method firstMove to check if the particular pawn passed to the method has moved before or not. This firstMove method scans through an ArrayList which contains a list of pawns that get added to the list once they have moved. If the pawn isn’t in the list, the pawn hasn’t moved yet and can take advantage of moving two squares forward.

The first condition also checks the Pawn is moving straight forward (staying in the same column) and is moving either one or two spaces forward (as the pawn doesn’t have to move two spaces forward in its first go).

If the first condition isn’t met, the program goes to the second condition/else if statement (surrounded in a green square below). This condition checks to see if the Pawn simply wants to move forward a tile, by seeing that the pawn is moving forward/staying in the same column, but there is a difference of 1 from its origin tile to its destination tile. It can be safely assumed that if the program goes into this condition that the pawn has already moved before, otherwise it would have executed the code in the first condition.

In the third condition (surrounded in a blue square below), the program checks to see if the Pawn is moving diagonally one space either North West or North East. This would indicate an offensive move by a Pawn to capture an opponent player’s piece. If none of the above conditions are met, we must have an illegal move because the moves valid with a Pawn piece type are coded and have already been checked against.

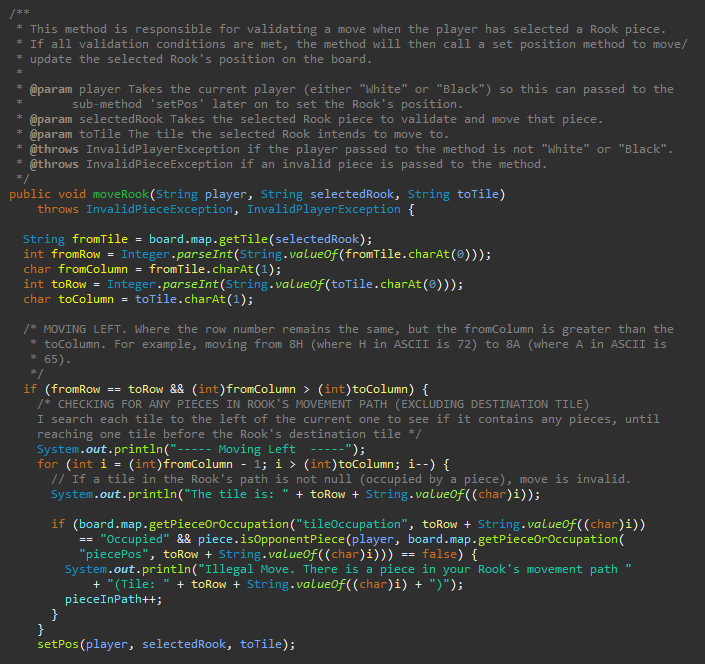


## Rooks

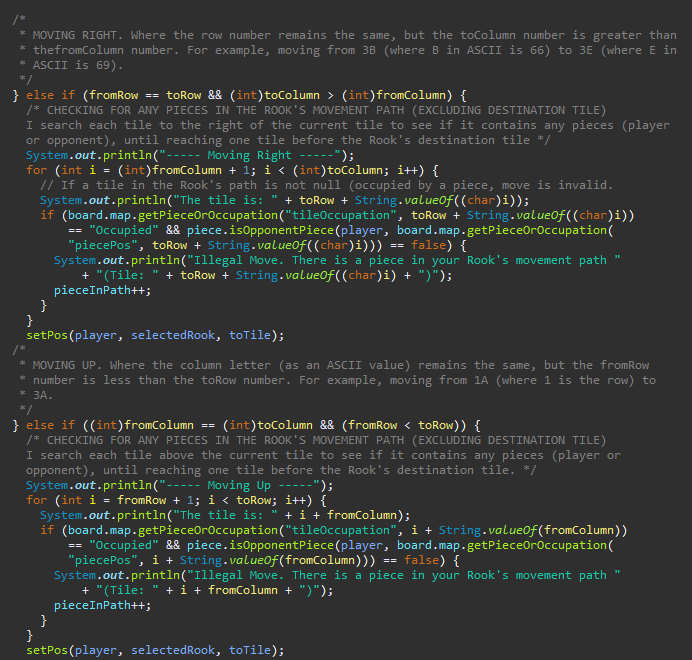
In terms of a legal move for a Rook, a Rook can move any number of **vacant** squares vertically or horizontally. [[1]](#_References)

To reflect these rules in validation code, I create various conditional statements each checking for the direction the Rook can move in which is horizontally, so left and right, as well as vertically, which is up and down.

My first conditional statement (surrounded in a red square below) checks if the Rook is moving left by seeing if the fromColumn as an ASCII value is greater than the toColumn as an ASCII value. This works because if the Rook is moving from 3H to 3A, A has a lower ASCII value than H so this is how I can distinguish the Rook is moving left. I also check that the row number is remaining the same, therefore if this condition is met, the Rook is indeed moving left only.

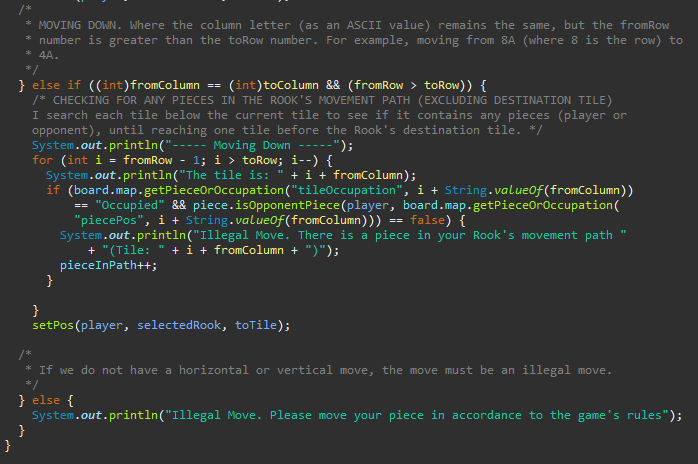


My second condition checks for moving right (surrounded in a green square below). It does this by checking that the toColumn is greater than the fromColumn. This applies the theory above in regard to ASCII values, because the Rook will be moving right if the toColumn letter is greater than the fromColumn letter. The row number must also remain the same.



The third condition checks for moving up (surrounded in a blue square above). This time I check the column letters remain the same, and that the toRow number is greater than the fromRow number. This means that if the Rook is moving from 1A to 3A, the Rook is recognised as moving up as the column letter is the same and the row number for the tile it is moving to is greater than its origin tile.

Finally, my fourth condition checks for moving down (surrounded in a yellow square below). This works similarly to the third condition, except that the fromRow number remains greater than the toRow number, with column letters remaining the same. If none of these four conditions are met, the program will go into its ‘else’ statement which indicates to the user that the move selected is invalid for the Rook piece type.



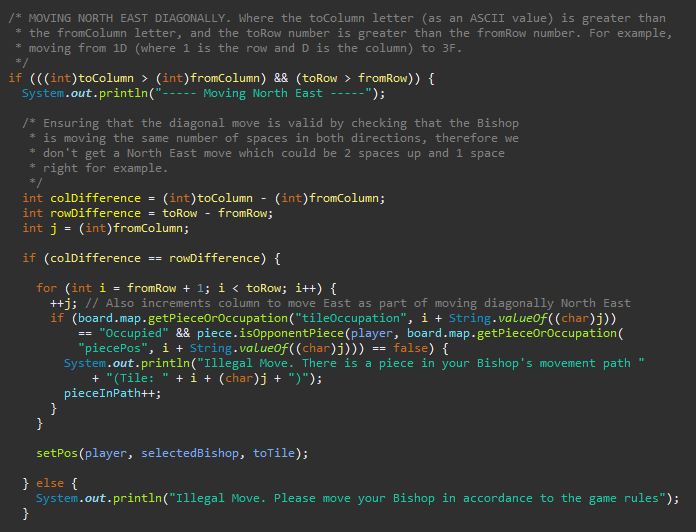
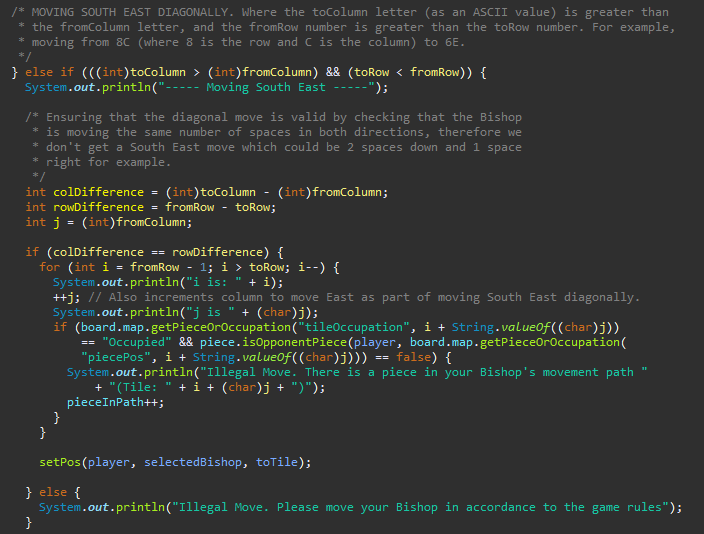
In all the conditional statements that are checking for the Rook’s direction, there is code to count if there is a piece in Rook’s path (all four surrounded in purple squares above). It does this by iterating through the tiles in the path of the Rook’s given direction. If there is a piece of EITHER of the two players’ inside the Rook’s movement path (this excludes the destination tile as this is treated separately), then the pieceInPath variable is incremented. This is used in the set position method later, as if the pieceInPath variable equals 0, the Rook has a clear movement path to move.

## Bishops

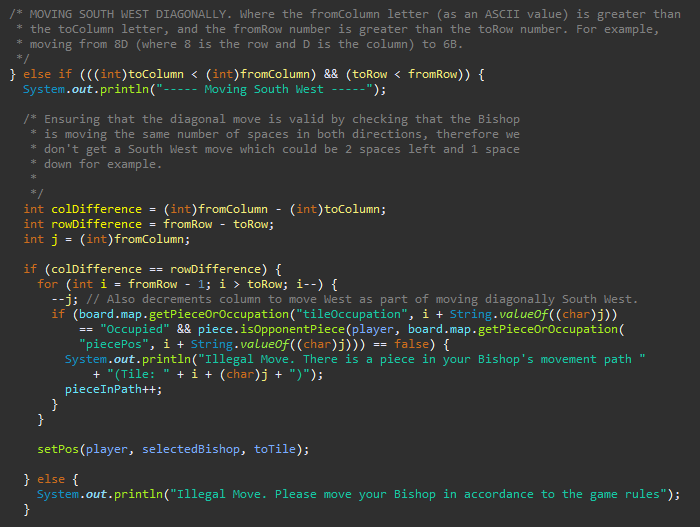
From my research of the rules of each Chess piece, a Bishop can move any number of squares diagonally, as long as the squares are vacant. [[1]](#_References)

To translate this rule into code, I have written conditions to check for the four different diagonal movements a Bishop can make. These are; North East, South East, South West, and North West diagonally.

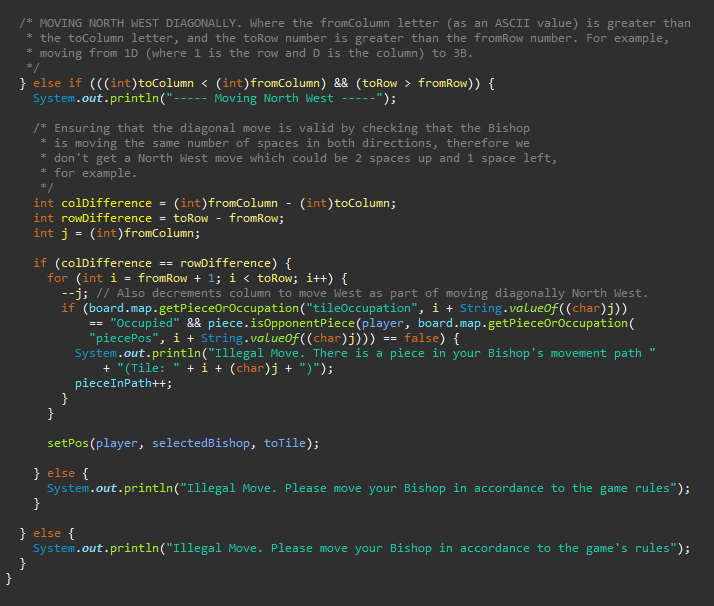
My first condition is to check for moving North East diagonally (surrounded in red square below). I achieve this by checking the toColumn as an ASCII value is greater than the fromColumn, as part of moving East, and the toRow number is greater than the fromRow number, as part of moving North. For example, moving from 1D (where 1 is the row and D is the column) to the tile 3F.

My second condition is checking for moving South East diagonally (surrounded in green square above). I check that the toColumn letter (as an ASCII value) is greater than the fromColumn letter, as part of moving East. I also check that the toRow number is greater than the fromRow number, as part of moving South. For example, moving from 8C (where 8 is the row number and C is the column letter) to 6E.

My third condition checks for moving South West diagonally (surrounded in a blue square below). To achieve this, I ensure the fromColumn letter as an ASCII value is greater than the toColumn letter, as part of moving West. I also ensure that the fromRow number is greater than the toRow number, to move South.



My fourth and final condition is responsible for moving North West diagonally (surrounded in a yellow square below). As part of moving West, I check that the fromColumn letter as an ASCII value is greater than the toColumn letter. Then, I check that the toRow number is greater than the fromRow number as part of moving up/North. For example, when moving from 1D where 1 is the row and D is the column letter to 3B.



In each of my four conditions checking for a diagonal move, I also identified that I needed to check that the Bishop is moving in a perfect diagonal line. For example, for each square north we take a square east, as opposed to 2 squares north and 1 east. In each condition (surrounded in purple squares above), I check that the difference between the toColumn and fromColumn is the same as the difference between the toRow and fromRow. That is, if the column letter has changed by 4, for example, the row number should have changed by 4 as well for the move to be valid.

Furthermore, in each of the four conditions the final condition I need to check for is ensuring that there are no pieces in the Bishop’s movement path (surrounded in white squares above). This is regardless of whether the piece in the Bishop’s movement path belongs to the player or the opponent. This is so that we can prevent the Bishop from jumping over that piece as this is not a property this piece type has, unlike a Knight. If there are any pieces in the Bishop’s movement path then the pieceInPath counter is incremented, which will be used in the set position method later on to verify the Bishop has a clear movement path.

If none of the four conditions above are met, the program will go into the ‘else’ statement, which indicates to the user that the move they selected is invalid and to move their Bishop in accordance to the game’s rules.

## Knights

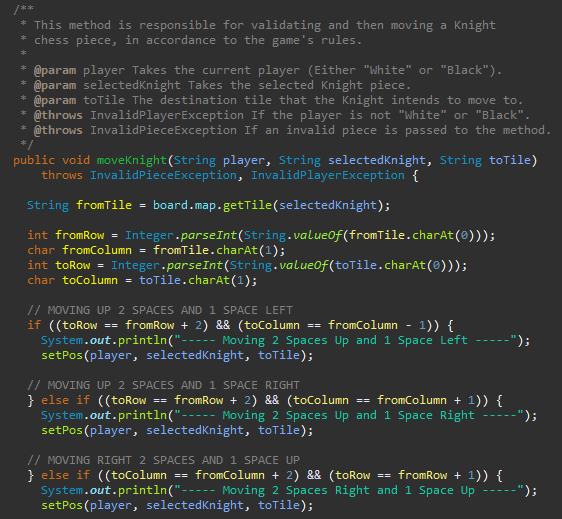
As part of my research of the rules of each Chess piece, a Knight can move in an L shape in any direction. It can move two squares forwards, backwards, left or right, then one square to the left or right. Alternatively, it can move one square forwards, backwards, left or right, then two squares to the left or right. Knights can jump over pieces in its intended path, so the squares in its path do not have to be vacant*.* [[1]](#_References)

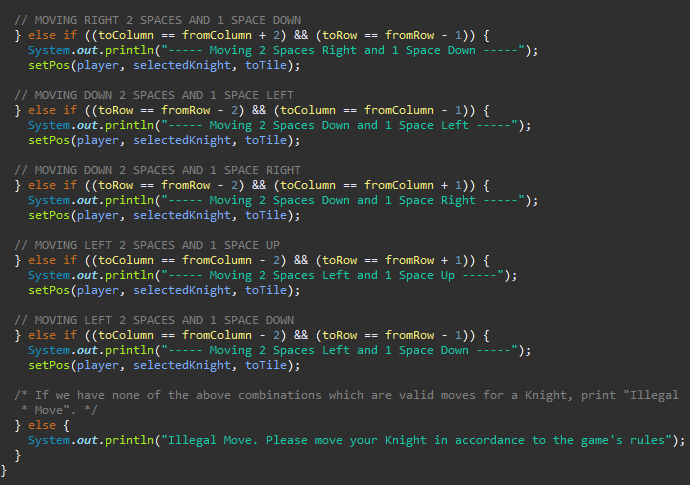
To show my understanding of this in the code I have written, I programmed the eight different possible moves that a Knight can make as a result of the combinations from the rules above.

The first condition checks for moving up 2 spaces and moving left 1 space. It does this by checking the destination row number is 2 higher than the origin row number, and that the destination column letter is one below the origin column letter to move left.

The second condition will check for moving up 2 spaces and moving right 1 space. Same as above, except it sees if the destination column letter is one above the origin column letter to move right.

The rest of the conditions follow similar principles, the only differences are the values being added or subtracted to determine the tile combination the Knight is moving in. The Knight class does not need to check for piece in its movement path because the Knight is the only piece type that can “jump” over other pieces.



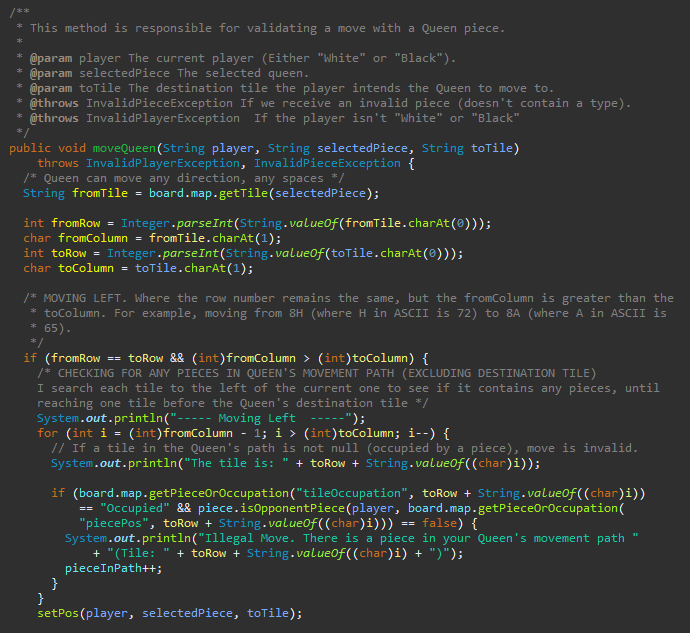


If none of the above conditions or combinations are met, the program goes into the else statement to tell the user they are trying to make an illegal move and that they should move their Knight piece in accordance to the game’s rules.

## Queen

The rules for a Queen piece type state that a Queen can; move in any number of squares in all directions (horizontally, vertically, or diagonally), as long as the squares are vacant. [[1]](#_References)

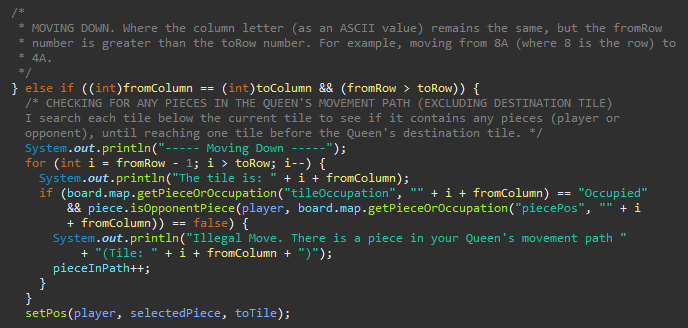
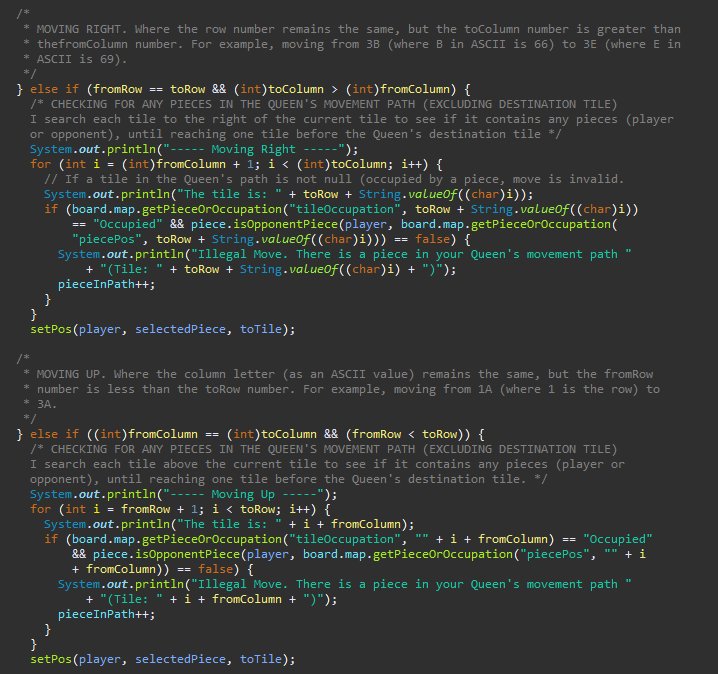
This means there are eight different directions the Queen can possibly move in. The first of these directional moves is moving left. In this condition, it checks that the row number is staying the same so that the Queen is not moving up or down, and that the fromColumn letter is greater than the toColumn letter.



The second direction to check for is moving right. To check this, I look to see if the row number is the same (so that the Queen is going up or down at all), and that the toColumn letter is greater than the fromColumn letter. For example, if moving from 3B to 3E, where B in ASCII is 66 and E in ASCII is 69, the column letter for the destination tile is greater than the origin tile column letter.

The third direction to check for is moving up. This time I look for the column remaining the same (not moving horizontally), and that the toRow is greater than the fromRow. For example, moving from 1A to 3A, where 1 and 3 are row numbers.

The fourth direction is moving down. Like with moving up, I check that the column remains the same. However, this time I check that the fromRow number is greater than the toRow number. For example, moving down from 3A to 1A, where 3 and 1 are row numbers.



The next four directional moves are for moving diagonally. The code for this is the same as the code used for checking diagonal moves in the Bishop class. Refer to the ‘Bishop’ heading in this report to see how this code is written and for an explanation on how the code works.

As Queens can cover multiple tiles in any direction, a for loop is needed to check for any pieces in the tiles between the Queen’s origin and destination tile. If a piece is in the Queen’s movement path then the pieceInPath counter gets incremented, if the pieceInPath counter is not 0 when the set position method is called then the move is invalid as the Queen’s path is obstructed.

## King

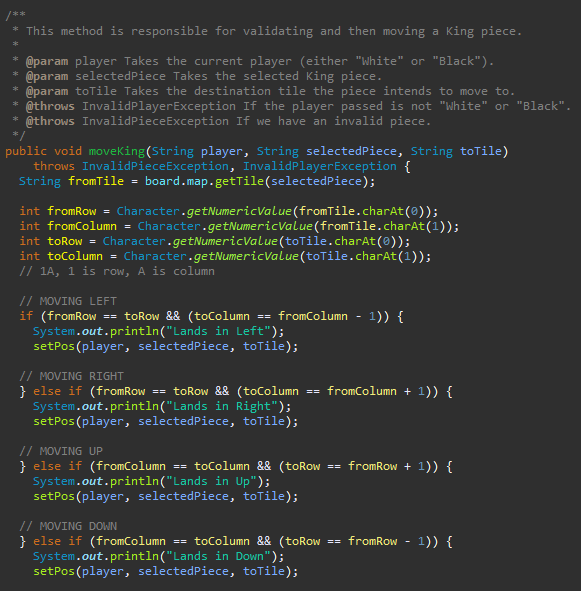
The rules for a King piece state that a King can move **one** square at a time in all directions (horizontally, vertically, or diagonally). [[1]](#_References)

As the King can move in any direction, there are again eight possible directions. The first of these conditions check that between the origin tile and destination tile that the row number remains the same, but that the column letter in the destination tile is one letter lower than the column letter in the origin tile.

The second condition is for moving right. It again checks the row numbers are staying the same between the origin and destination tile, but that the column letter in the destination tile is one letter higher than the column letter in the origin tile, thus moving right.

The third condition looks for if the King is moving up. It can tell this by the two column letters in both the origin and destination tile being the same, but the destination tile’s row number being one above the origin tile’s row number, thus indicating a move upwards.

The fourth condition is searching for a downwards move. If the column letters in both the origin and destination tile remain the same, but the destination tile’s row number is one row lower than the origin tile’s row number, then the King is moving down.



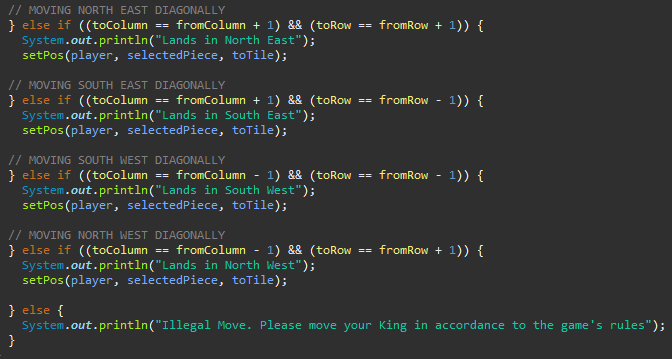
The fifth condition is checking for a diagonal move in the North East direction. To check this, I see if the column letter is increasing by one (to move East), and that the row number is increasing by one also (to move North).

The sixth condition looks for a diagonal move in the South East direction. The same as before, I check to see if the column letter is increasing by one (to move East), however I now check that the row number is decreasing by one so that the piece is moving South as well.

The seventh condition is checking for a diagonal move in the South West direction. To check this, I see if the column letter is decreasing by one (to move West), and that the row number is decreasing by one so that the piece is moving South.

The eighth and final condition to check for is a diagonal move in the North West direction. If the column letter is decreasing by one (to move West), and the row number is increasing by one so that the piece is moving North, then we have a valid North West move.

If none of these conditions are met, the program will execute the ‘else’ statement to indicate to the player that the move they have selected is illegal and that they must move their King in accordance to the game’s rules.



# Setting Positions

In this section, I will refer to the setPos method that I have in each of my Piece type classes. This setPos method is always called after the bulk of the validation has been completed in the move piece methods.

This setPos method performs a final validation check to see what piece is in the destination tile that the player is trying to move their piece into. This is to eliminate the chance of a player trying to move a piece into a tile which is already occupied with another one of their pieces. It also means that if the player is moving their piece into a tile containing an opponent piece, then the player can capture this piece and move there.

Once this check is completed, the position of the piece the player is trying to move is actually set in the TreeMap data structures I have, thus setting the piece’s new position on the Chess board.

Below is an example of one of my setPos methods from the Queen class. The setPos methods are very similar between all my piece type classes, sometimes the setPos method in one class may be the same as the setPos method in another. The only main differences between one version of the method to another is the usage of the ‘pieceInpath’ variable. This variable is only used in setPos methods where that particular piece type is able to move more than one square at a time in a single move.

Pieces that only move one square at a time will have the same validation checking, just without the pieceInPath variable being checked for in the conditional statements.



In the example above, you can see that I have three different cases, the first is for checking that the tile the player intends to move to is empty and that there are no other pieces in the movement path of the piece being moved. If it is empty and there are no other pieces in the way, the position of the piece being moved is set in the TreeMaps.

The second case is if there are no pieces in the movement path of the piece being moved, and there is a piece in the destination tile which is the opponent player’s. This means the Queen, for example, can move into that destination tile and then a capture piece method is called to remove the opponent player’s piece from the Chess board. The Queen’s new position is then set on the board once the capture piece method has executed.

The third and final case is if there isn’t an opponent player’s piece in the destination tile, but it is still occupied. This means that the player’s own piece will be in the destination tile so the piece being moved cannot also go there. A message indicating this will be printed to the player.

After the execution of one of these three cases, the pieceInPath counter is reset to zero ready for new validation checks when that piece is selected again, and to avoid any unforeseen conflicts later on with the validation methods.

# References

[1] Chess Setup and Rules – Chess Coach Online  
<http://www.chesscoachonline.com/chess-articles/chess-rules>