PROFESSIONAL TRAINING REPORT
at

Sathyabama Institute of Science and Technology (Deemed to be University)

Submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering Degree in Computer Science and Engineering

By

RAGHAVEE E M
REG. NO. 39110822

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF COMPUTING

SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY
JEPPIAAR NAGAR, RAJIV GANDHI SALAI,
CHENNAI – 600119, TAMILNADU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of **RAGHAVEE E M (Reg. No: 39110822)** who carried out the project entitled "**Online Chess Game**" under my supervision from January 2022 to April 2022.

Internal Guide

Ms.R.Aishwarya

Head of the Department

Submitted for Viva voce Examination held on ____________________

Internal Examiner **External Examiner**

DECLARATION

I, **Raghavee E M** hereby declare that the project report entitled Online chess game done by me under the guidance of **Ms.R.Aishwarya** is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering Degree in Computer Science and Engineering.

DATE:

PLACE: **SIGNATURE OF THE CANDIDATE**

[illegible]

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to **Board of Management** of **SATHYABAMA** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. T. Sasikala M.E., Ph.D**, **Dean**, School of Computing, **Dr. S. Vigneshwari, M.E., Ph.D. and Dr. L. Lakshmanan, M.E., Ph.D., Heads of the Department** of **Computer Science and Engineering** for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Ms.R.Aishwarya** for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the

Department of Computer Science and Engineering who were helpful in many ways for the completion of the project.

TRAINING CERTIFICATE

TABLE OF CONTENTS

ABSTRACT

This report showcases an online portal for chess game providing human to human capability as well as human to computer capability for both playing and learning respectively. For the human to human capability, the humans need not to be present in the same geographical location physically. They can play the game of chess from varied places. However, for the learning purpose the interaction is completely based with the computer and it is both offline and online. In this portal presentation, the users have to register themselves and are provided with a unique identification number to login in again and to learn or play against computer or other registered user. The moves as well as the previous history of the games played and the tournaments participated are kept in the repository database and will be shown to the registered users.

LIST OF FIGURE

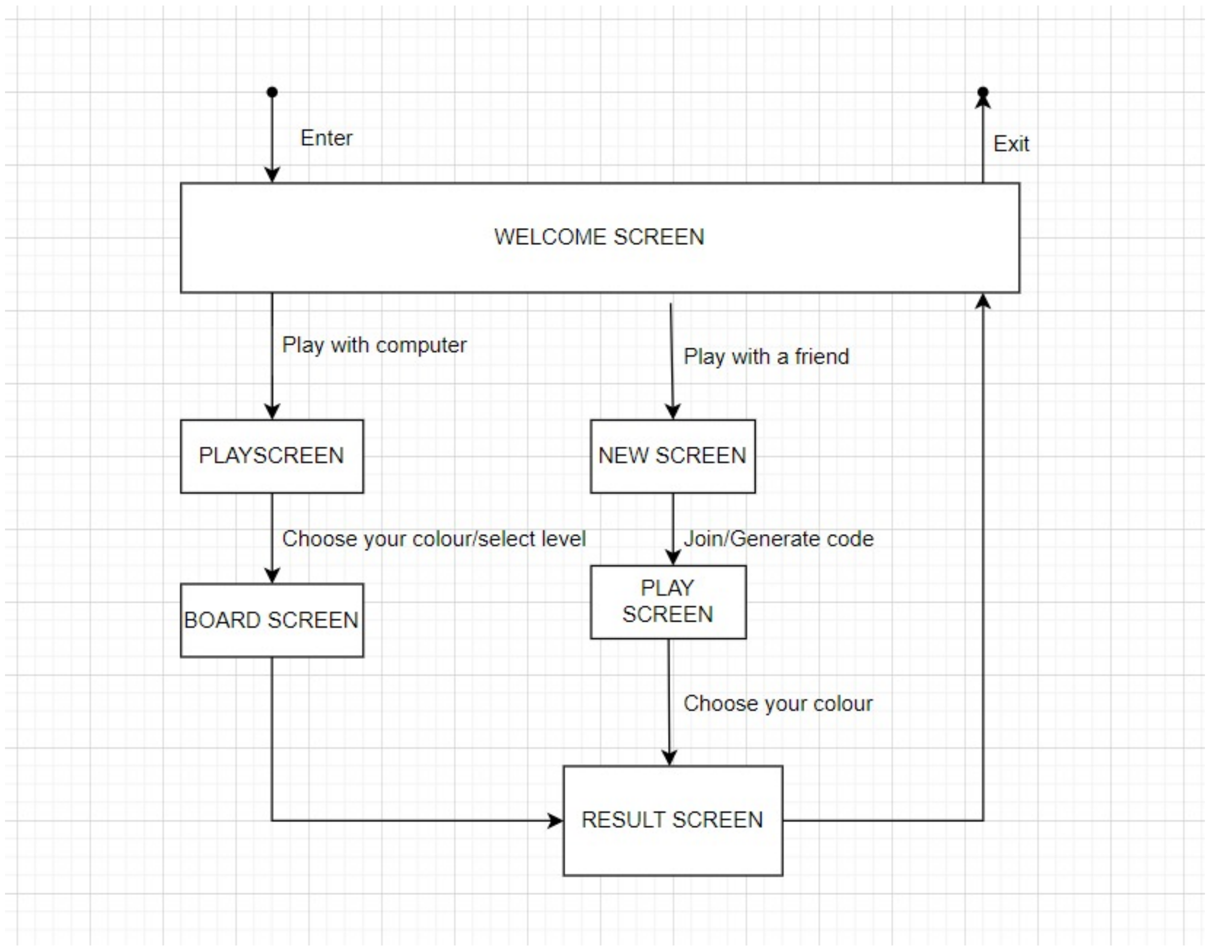


Fig 1-flow diagram for online chess game

CHAPTER 1:

INTRODUCTION

Chess is a game that has been developed to play offline i.e., it can be played against a computer or a human present at the same physical location. However, when the game is played online the opponent can be at any geographical location and may be totally unknown to the player. For the online game, the player has to register himself and a unique identification number will provided on the successful registration along with a password for further usages. The administrator handles the registration procedures and also checks on the duplicity of the users. The main portal comprises a normal lookalike chessboard providing two sided panel. The game can be played and is saved upon completion for future references. The players has to wait till the completion of the move of the opponent. The player can also redo a move with permission of the opponent and also forfeit a game after or before the commencement of the match. There is also a facility of communication between the players with the help of text, so that it can benefit both the players. After completion or stoppage of each game a log is saved about the game, so that for each game can be resumed after a certain interval of time. There is also provision about learning about the game of chess through various tutorials. The registered users can also play with the computer to improve their skills as well as learn new tricks. The users can also visit the history page for referring different games that has been played earlier to detect their mistakes in certain tournament

CHAPTER 2:

AIM:

To design an online chess game for two players using JAVA.

SCOPE:

1)First goal is to allow two users or players to play the game interactively
from
remote locations.
2)The second goal will be that the program should be working and allow the
users to play the game.
Benefits of the chess game:
1)Improves problem solving technique.
2)Improves logic and reasoning skills.
3)Increases patience and persistence.
4)Improves decision making skills.

CHAPTER 3:

METHODS USED FOR DEVELOPING ONLINE CHESS GAME:

1 - Basic Logic ♟

The logic for the terminal based game and the UI is essentially the same:

1. Instantiate a pieces class. This is the board of the game, and contains all of the pieces.
2. Select a piece and select a coordinate. If the destination coordinate is a valid coordinate for your given piece, and its the piece's colour turn to move, then make a move.
   - this is represented by changing pieces.
   - the piece's key changes from its origin to its destination coordinate
   - if a piece from the opposite colour was occupying the coordiante, that piece is eliminated
3. Check if its check mate. If so, the game ends. Otherwise, it is the turn of the other colour.

2 - Enums ♞

Enums were an integral part of this project, as I used them to represent important constants for the game. The enums I created
were ID, COLOUR and BOARD.

ID

Used as an identifier for a piece. The types of pieces are:

- KING
- QUEEN
- ROOK
- BISHOP
- KNIGHT
- PAWN

This enum contained 2 toString() methods. One (toString()) is used to print the piece's letter for describing moves. For example, if a bishop moves to e6, such move would be described as Be6. The other one (toFullString()) is mainly used for testing and "human" printing purposes. It returns the full name of the String. For example, ID.KING.toFullString() would return "King".

COLOUR

Used as a colour identifier for a piece. These are:

- B (a black piece)
- W (a white piece)

This enum also contains 2 toString() methods, albeit these are barely used (mainly only for tests). The most important method is the not(COLOUR) method. It is used to return the opposite colour to the argument it takes.
Hence, COLOUR.not(COLOUR.B) would return COLOUR.W. This is extremely useful, for example when handling the turn of play, or calculating when a move

leads to check.

BOARD

Used to contain the dimensions of the board. These are determined by 4 constants:

- FIRST_FILE('a')
- LAST_FILE('h')
- FIRST_RANK(1)
- LAST_RANK(8)

A file is used to represent a column, and is represented by a character from a to h. A rank represents a row, and is represented by an integer from 1 to 8. BOARD contains methods to access the values associated with these constants.

3 - Key Classes ♝

There are 4 key classes that sustain this project: Coordinate, Piece, Pieces and Move. The first 3 are used to create objects to represent the chess board and its pieces. They all contain getters, setters, alongside functionality to create deep copies of its instances. This is paramount, as will be explained later. The methods toString(), equals(Object o) and hashCode() have all been overridden. The last class, Move simply contains methods that are essential for the correct functioining of the project.

Coordinate

Uses a char (file) and an int (rank) to determine a square within a board, according to Chess nomenclature. Includes functionality to ensure that the arguments provided represent a valid coordinate within the board.

Piece

A class identifying the pieces of the game. A piece is initialised with an ID (type of piece), a COLOUR (black or white) and its initial Coordinate within the board. It acts as a super class to the more specific pieces: King, Queen, Rook, Bishop, Knight, and Pawn. The most important method in Piece have to do with the creation, updating and validation of the moves that a piece can move. We define raw moves as those moves that a piece can make independently of whether the King is in check of not. Potential moves are the actual moves that a piece can make, taking checks into accounts. Piece contains abstract methods that are then individually defined within the children classes. For example, getRawMoves(Pieces pieces) is used to obtain the raw moves that are available to an individual piece. Since each piece moves differently, the details of getRawMoves(Pieces pieces) are defined individually.
Perhaps the most important of all its methods is removeOwnCheck(Pieces pieces). This method is used to take in the raw moves available to a piece, and then filter out all of the moves that are impossible; namely those that would either:

- lead to check
- not stop a check (i.e if a piece moves away from the King, leading to a

check by the opposition)

In order to do this, we must create a deep copy of the board. From the raw moves of the piece, we make the piece execute the move within the copied board. We then check if that has lead to situation of check by the opposition. If it has, said move is deleted. Otherwise, it is maintained. This is a crucial process, as it allows the pieces to determine all of their moves, so checking whether the move provided by the user is legal becomes trivial. Moreover, for the UI, it allows us to display all the moves avaialbale to the given piece.

Pieces

Contains a HashMap with Coordinate-Piece key-value pairs. It contains all the methods used to handling the positioning of the pieces throughout the game. For example, we can use it to find the King of a certain colour, determine which pieces lie on the same file or whether it is the end of the game (via check mate or a draw/stalemate). Pieces also contains the method that executes the moves provided by the user. It is a particularly long method, which must check for all moves that constitute special cases, such as a King castling or a pawn queening/capturing in diagonal/en passant.

Move

This class contains all the classes pertinent to the movement of the pieces. It contains functionality to, given a board (Pieces) and a piece determine which range of movement it has. We can determine available moves in vertical, diagonal and horizontal direction, alongside the moves available to a Knight. It is these methods that are used within a Piece to determine the raw moves available to them. It must be noted that there are pieces, such as the King or the Pawn that have a special range of moves available to them. The handling of these moves is made directly within their classes.

4 - The UI ♜

To create the UI, I used the Swing package.

Printing the Interface

The interface is mainly made through the superposition of JPanels. To create the chess board, I created a 2D array of JButtons, each of alternating colours. I used a nested for loop to print the whole chess board. I also used this to assign coordinates to each JButton square, which allowed me to display a picture representing a piece on top of the square. Then, with each move, I executed the for loop again, but with pieces updated to reflect the current game. I also created an area to the right of the board that contained a section to see the moves played, alongside a button for saving a game, and an area displaying the outcome of a game (a draw, stalemate or win).

Handling movements

To make a move, the user needs to select a piece, and then select a destination square. I created a flag that would allow me to check whether the user has clicked twice, as this would represent a move. When the user clicks a square

(JButton), I looped through the array of JButtons until I found the JButton that had been clicked. I then turned this information into a Coordinate, which then allowed me to find the Piece occupying the square. This then made it so the squares corresponding to the potential moves of the piece got illuminated. It also set the flag to true. Once there was a second click the program checked to see if the selected square corresponded to one of the potential moves of the piece. If so, the move was executed and the board was updated, resetting the flag. Otherwise, the potential moves of the selected piece would be shown.

5 - Saving a Game ♛
The FileIO class is used to handle game saving. In order to save a game, we create a txt file containing the moves, as per pgn (portable game notation) format (albeit without additional information, such as the date, location, players involved, etc ...). FileIO contains a method that handles the creation of Strings representing sets of moves. The user can then introduce the name of the file to be saved.

CHAPTER 4:

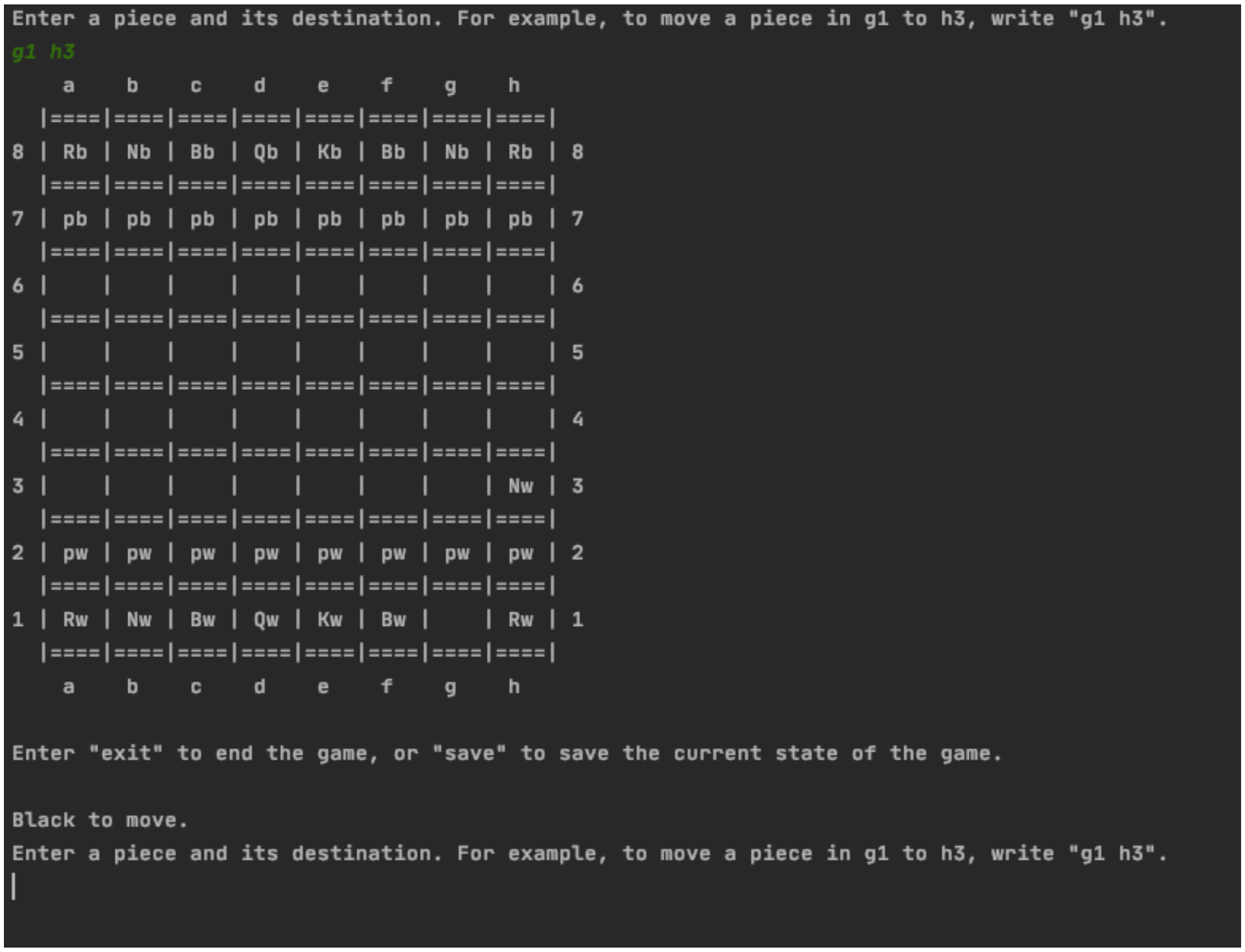
RESULTS AND DISCUSSION:

I believe I have merely constructed the beginning of the project. To further improve it, I would like to (in order to feasability and ease):

- add functionality to parse pgn/txt files and load their games
- add functionality so that a user can drag a piece to move it (currently need 2 clicks)
- add a player vs player functionality (switching the position of the board to face the player)
- add a timer
- add an opening move handbook
- create an AI of varying difficulty (player vs computer)
- create an ML algorithm

CHAPTER 5:

SUMMARY:



```
Enter a piece and its destination. For example, to move a piece in g1 to h3, write "g1 h3".
g1 h3
    a    b    c    d    e    f    g    h
  |====|====|====|====|====|====|====|====|
8 | Rb | Nb | Bb | Qb | Kb | Bb | Nb | Rb | 8
  |====|====|====|====|====|====|====|====|
7 | pb | pb | pb | pb | pb | pb | pb | pb | 7
  |====|====|====|====|====|====|====|====|
6 |    |    |    |    |    |    |    |    | 6
  |====|====|====|====|====|====|====|====|
5 |    |    |    |    |    |    |    |    | 5
  |====|====|====|====|====|====|====|====|
4 |    |    |    |    |    |    |    |    | 4
  |====|====|====|====|====|====|====|====|
3 |    |    |    |    |    |    |    | Nw | 3
  |====|====|====|====|====|====|====|====|
2 | pw | pw | pw | pw | pw | pw | pw | pw | 2
  |====|====|====|====|====|====|====|====|
1 | Rw | Nw | Bw | Qw | Kw | Bw |    | Rw | 1
  |====|====|====|====|====|====|====|====|
    a    b    c    d    e    f    g    h

Enter "exit" to end the game, or "save" to save the current state of the game.

Black to move.
Enter a piece and its destination. For example, to move a piece in g1 to h3, write "g1 h3".
|
```

CONCLUSION:

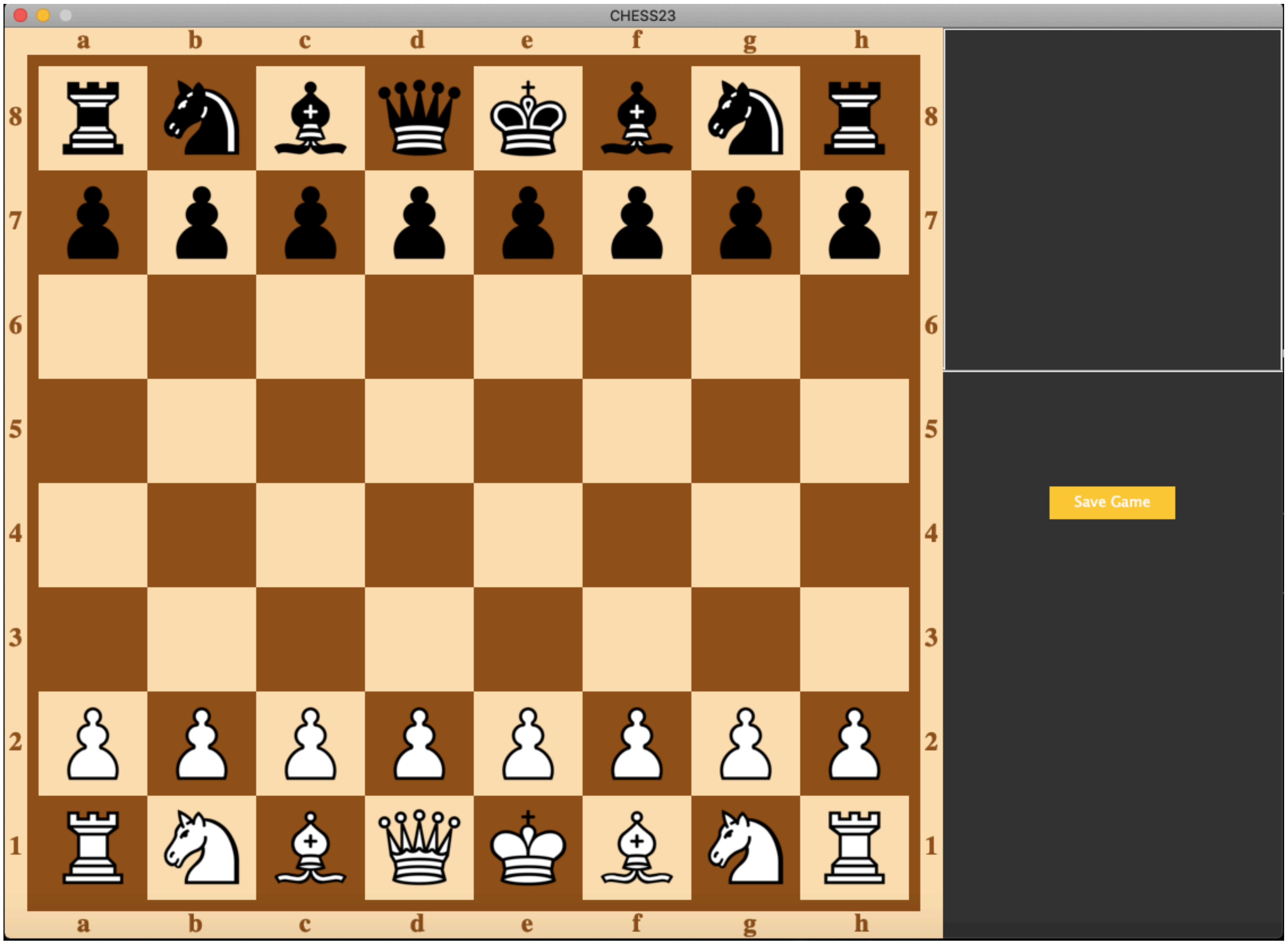
The main purpose of this work was to design a model for chess game which runs on the web. By, using the above portal system one can learn the various aspects of the game of chess. This will help in understanding the complete design and implementation of the functionalities in the online chess portal. This will help to reach all the chess enthusiast all over the globe to learn as well as play against each other. Thus, help to create an online world wide championship. This will also be economically viable, as players doesn't have to travel to attend a tournament or play a match. The log in the database will help new registered players to learn the tricks of the game, as by watching the old games from repositories and thus develop his/her standard of the game

REFERENCES:

[1] Adelson-Velskiy, G. M., Arlazarov, V. L., and Donskoy, M. V. (1975). Some methods of controlling the tree search in chess programs. Artificial Intelligence, 6(4), 361–371. http://doi.org/10.1016/0004-3702(75)90021-1
[2] Atherton, M., Zhuang, J., Bart, W. M., Hu, X., and He, S. (2003). A functional MRI study of high-level cognition. I. The game of chess. Cognitive Brain Research, 16(1), 26–31. http://doi.org/10.1016/S0926-6410(02)00207-0
[3] Bratko, I., Kopec, D., and Michie, D. (1978). Pattern-based representation of chess end-game knowledge. The Computer Journal, 21(2), 149–153. Retrieved from http://comjnl.oxfordjournals.org/content/21/2/149.full.pdf
[4] Chase, W. G., and Simon, H. A. (1973). Perception in chess. Cognitive Psychology, 4(1), 55–81. http://doi.org/10.1016/0010-0285(73)90004-2
[5] Ginsburg, M. (1974). Chessgames Services LLC Web design and database developmen. Retrieved March 15, 2016, from http://www.chessgames.com/player/mark_ginsburg.html
[6] Hajari, P., Iyer, R., and Patil, A. (2014). Implementation of Artificial Intelligence for Best Moves using Chessboard, 5(3), 3142–3144.
[7] Kiesel, A., Kunde, W., Pohl, C., Berner, M. P., and Hoffmann, J. (2009). Playing chess unconsciously. Journal of Experimental Psychology: Learning, Memory, and Cognition, 35(1), 292–298. http://doi.org/10.1037/a0014499
[8] Levinson, R., Hsu, F., Schaeffer, J., Marsland, T. A., and Willkins, D. E. (1991). Panel: The Role of Chess in Artificial Intelligence Research. In International Joint Conference on Artificial Intelligence (Vol. 84, pp. 547–552). Retrieved from http://www.ijcai.org/Proceedings/91-1/Papers/084.pdf
[9] Newell, A., Shaw, J. C., and Simon, H. A. (1958). Chess-Playing Programs and the Problem of Complexity. IBM Journal of Research and Development. http://doi.org/10.1147/rd.24.0320

APPENDIX:

A) SCREEN SHOT



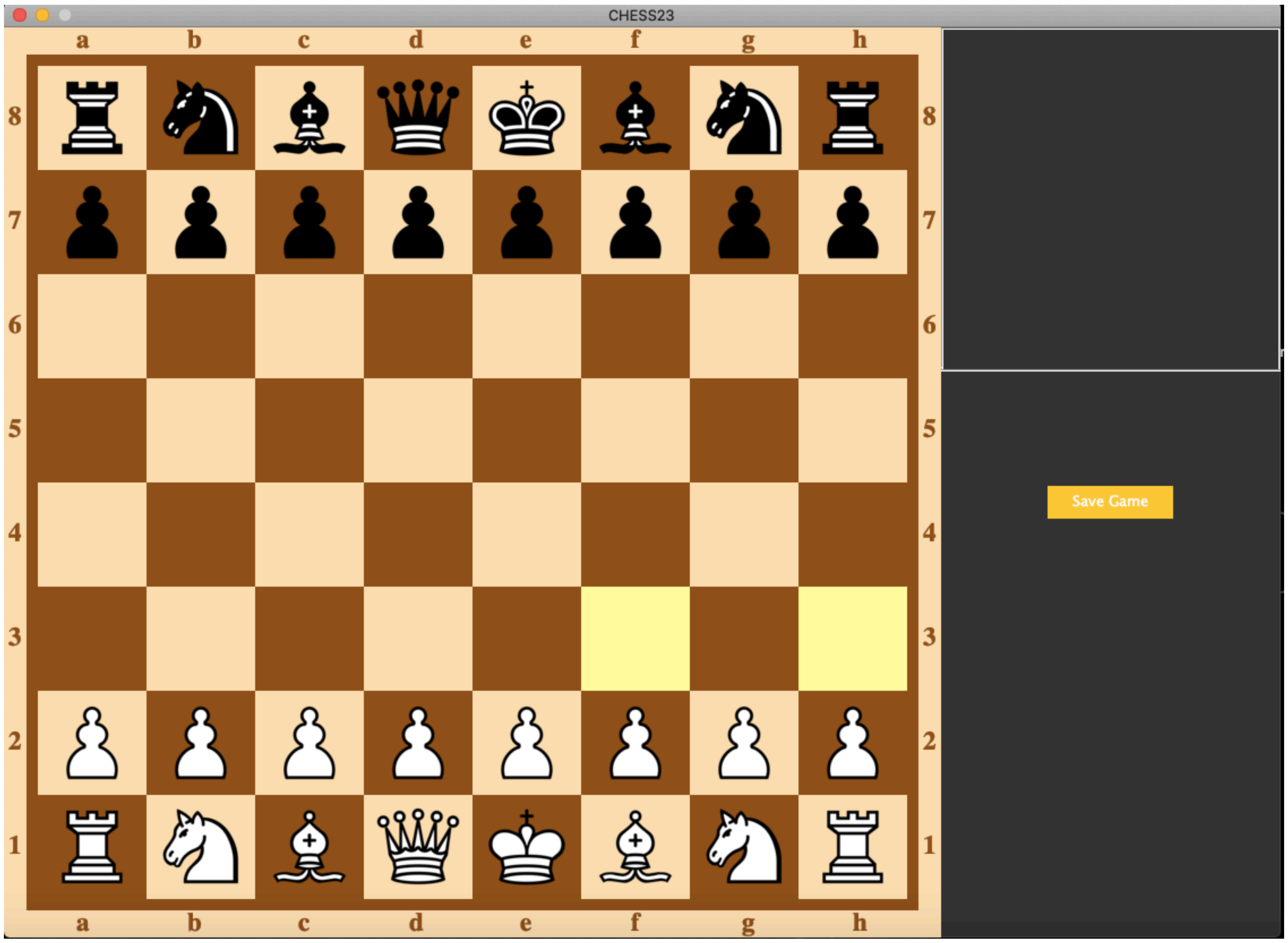
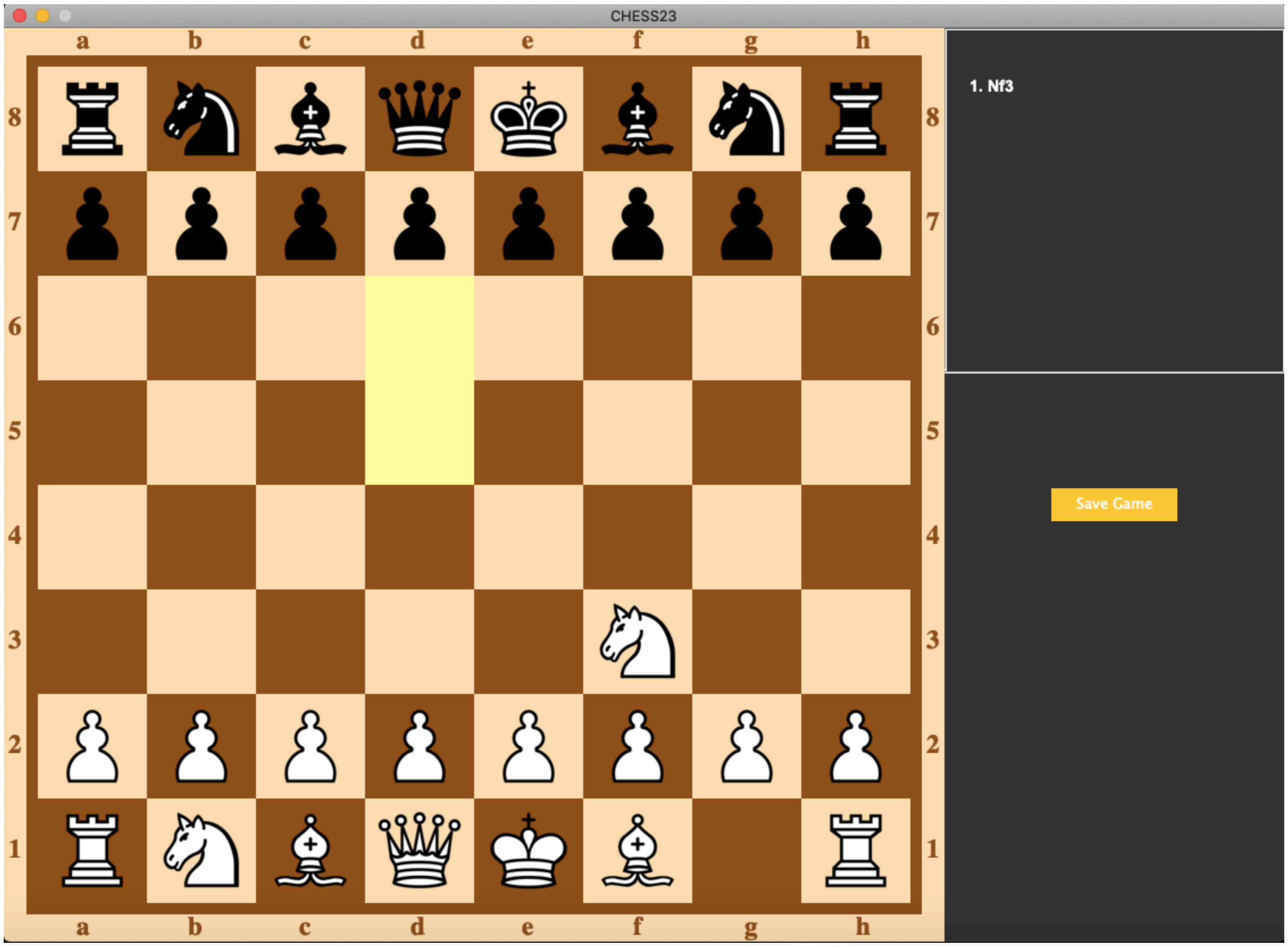


Fig 2: Clicking on the Knight revels it has 2 potential moves (f3 & h3):



1. Nf3

Fig 3: The Knight has moved, as shown in the game log to the right; the black pawn has 2 potential moves (d6 & d5)

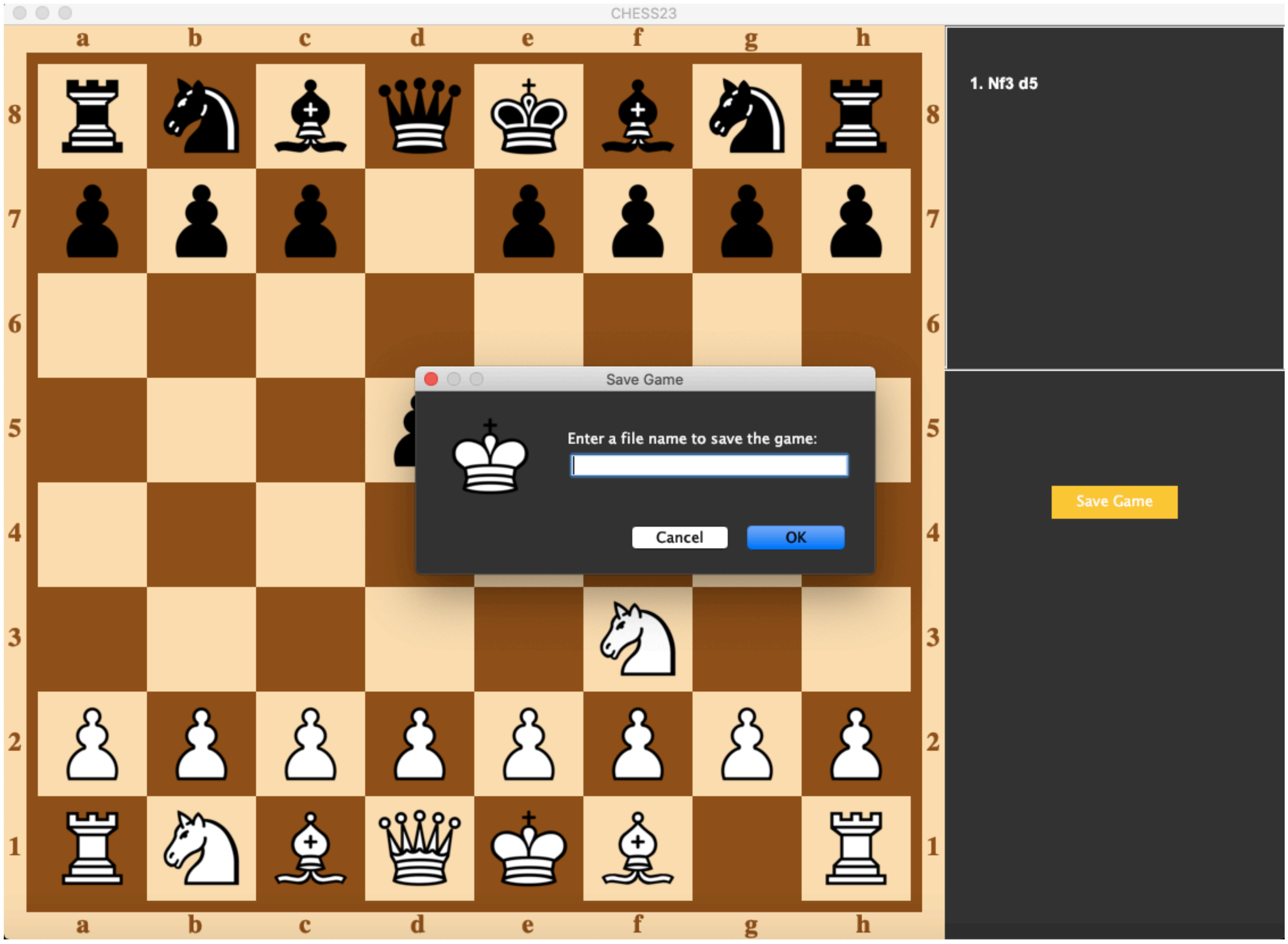
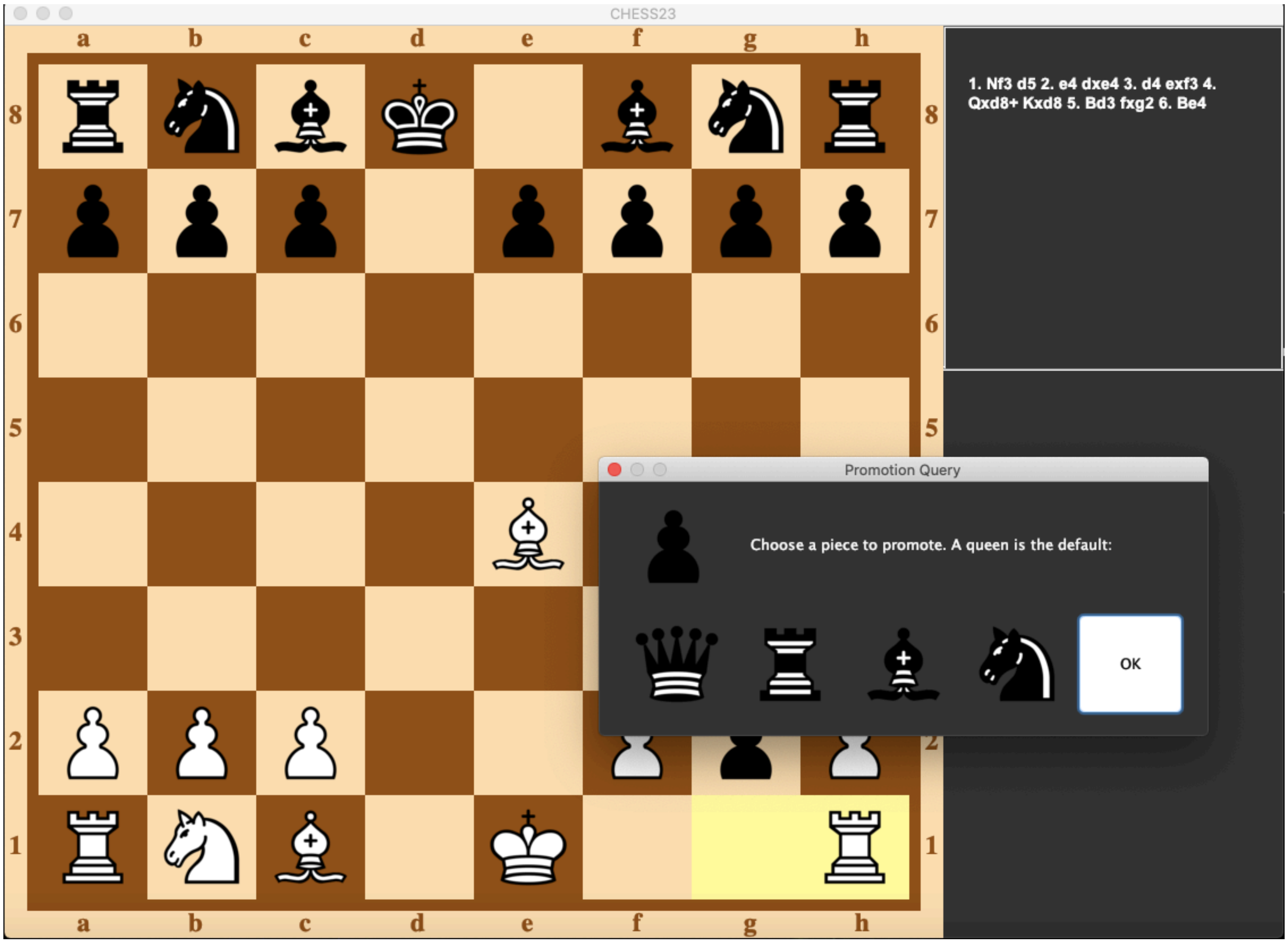


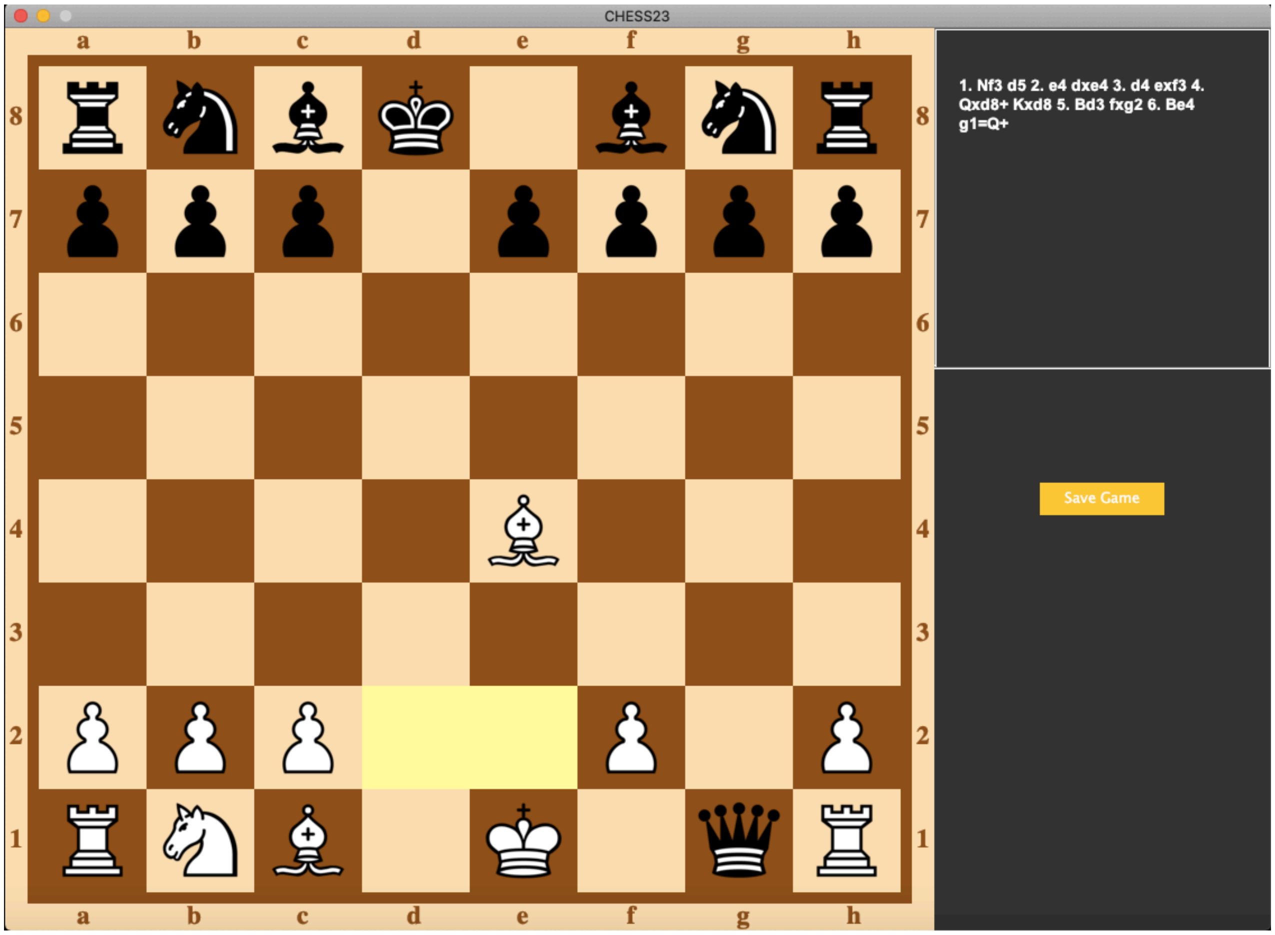
Fig 4: We can choose to save the game under any name we choose (as long as it doesn't already exist!):



1. Nf3 d5 2. e4 dxe4 3. d4 exf3 4. Qxd8+ Kxd8 5. Bd3 fxg2 6. Be4

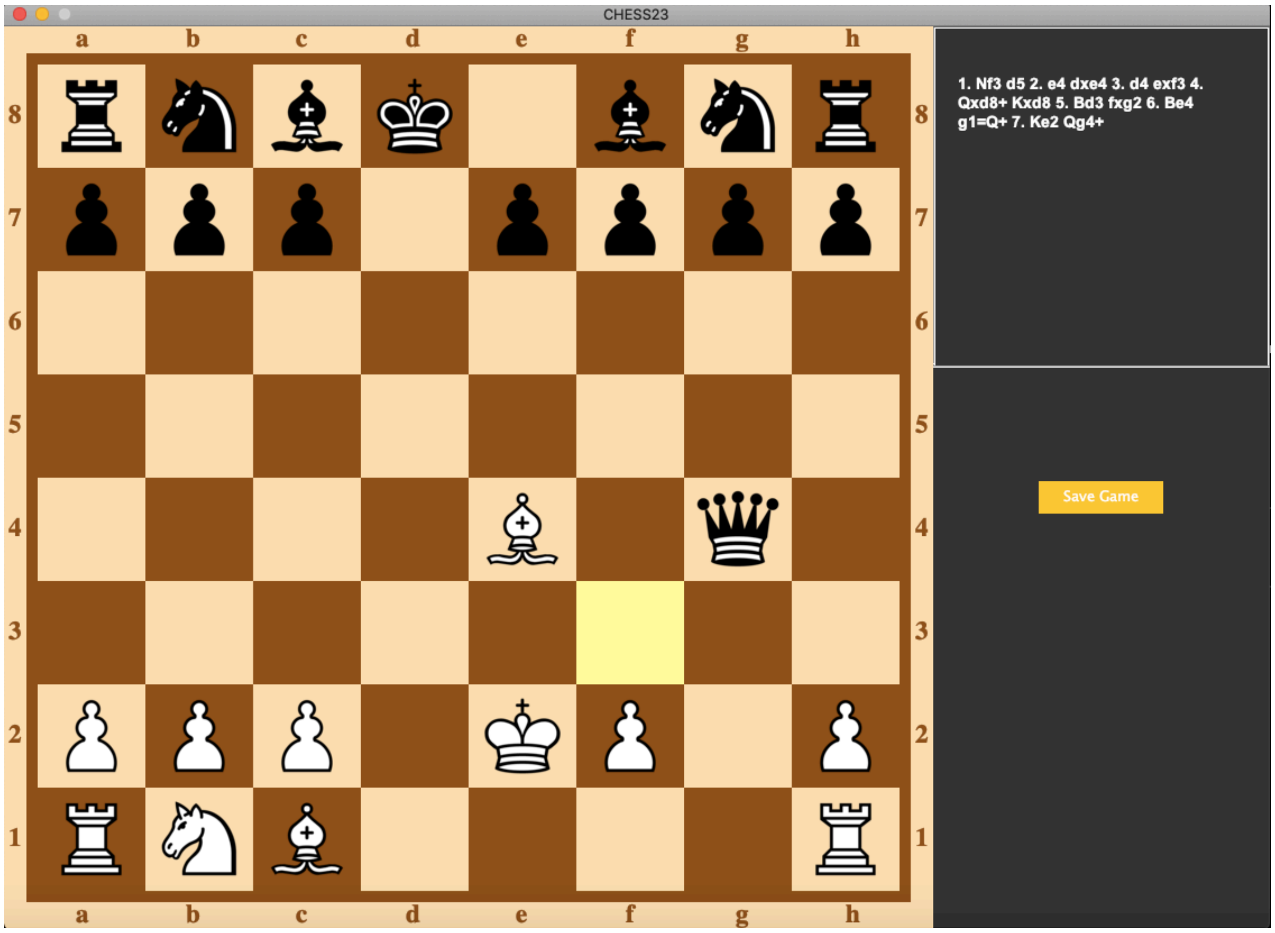
Choose a piece to promote. A queen is the default:

Fig 5: The black Pawn can now be promoted:



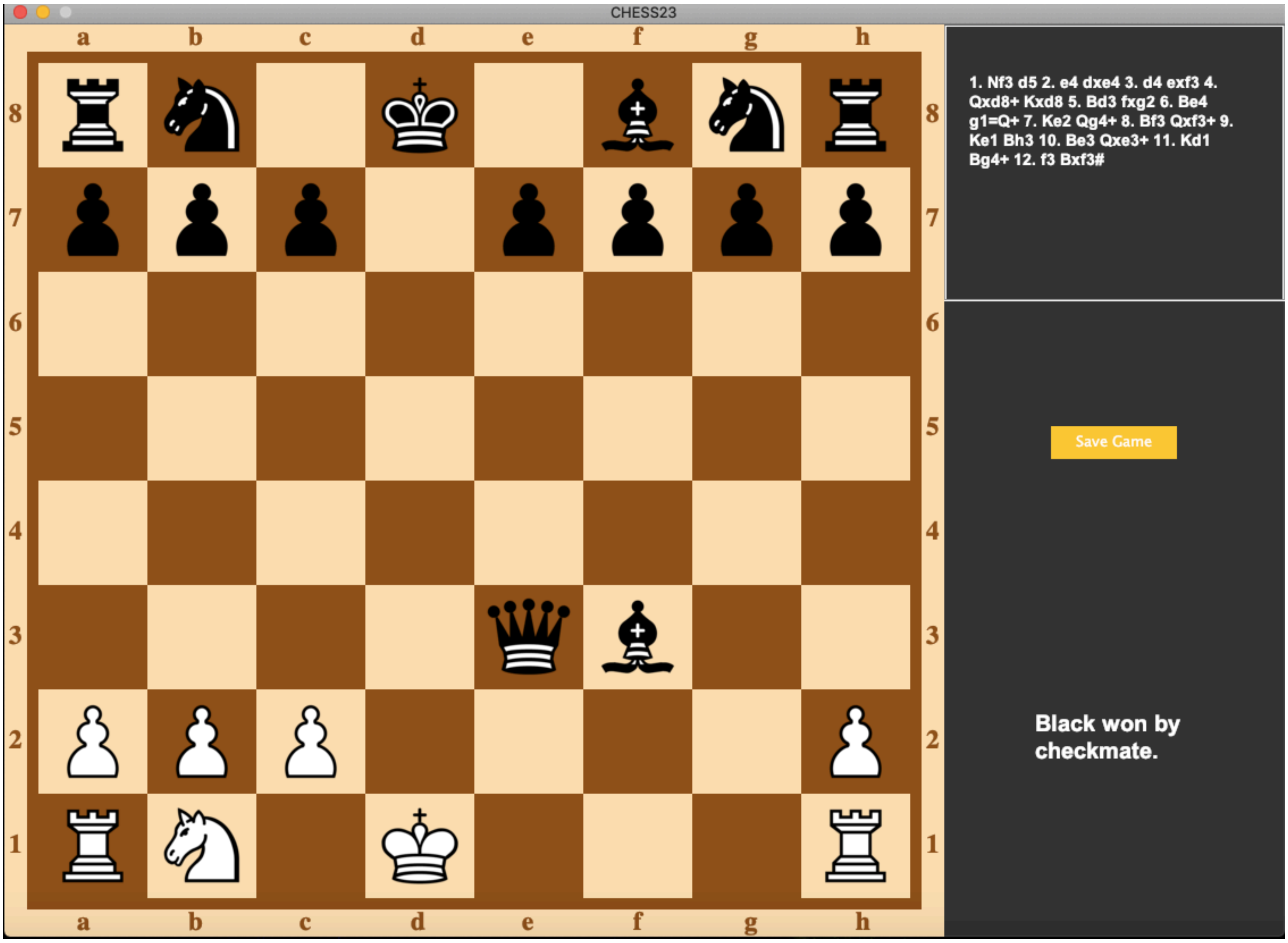
1. Nf3 d5 2. e4 dxe4 3. d4 exf3 4. Qxd8+ Kxd8 5. Bd3 fxg2 6. Be4 g1=Q+

Fig 6: The white King is in check, so its movements are limited:



1. Nf3 d5 2. e4 dxe4 3. d4 exf3 4. Qxd8+ Kxd8 5. Bd3 fxg2 6. Be4 g1=Q+ 7. Ke2 Qg4+

Fig 7: If we want to save the King, the white Bishop only has 1 square available



1. Nf3 d5 2. e4 dxe4 3. d4 exf3 4. Qxd8+ Kxd8 5. Bd3 fxg2 6. Be4 g1=Q+ 7. Ke2 Qg4+ 8. Bf3 Qxf3+ 9. Ke1 Bh3 10. Be3 Qxe3+ 11. Kd1 Bg4+ 12. f3 Bxf3#

Black won by checkmate.

Fig 8: A situation of checkmate

B)SOURCE CODE

BOARD.java :

```
/**
 * The BOARD enum class contains information abou tthe "limiting" constants
of a chess board
 * These are the first file ('a'), the last file ('h), the first rank (1) and the last rank
(8)
 * Also contains methods to get these values
 */

public enum BOARD {

    FIRST_FILE('a'),
    LAST_FILE('h'),
    FIRST_RANK(1),
    LAST_RANK(8);

    private char fileVal;
    private int rankVal;

    BOARD(char file) {
        fileVal = file;
    }

    BOARD(int rank) {
        rankVal = rank;
    }

    public char getFileVal() {
        return fileVal;
    }

    public int getRankVal() {
        return rankVal;
    }
}
```

BISHOP.java

```
import javax.swing.*;
import java.util.ArrayList;

/*
1) Class Constructors
2) Overridden Methods
*/

public class Bishop extends Piece{

   private ImageIcon icon;

   //________________________________________________Class
Constructors________________________________________________

   public Bishop(COLOUR colour, Coordinate OGcoord) {
      super(ID.BISHOP, colour, OGcoord);
      if (getColour() == COLOUR.B)
         icon = new ImageIcon("BBishop.png");
      else if (getColour() == COLOUR.W)
         icon = new ImageIcon("WBishop.png");
   }

   public Bishop(Bishop original) {
      super(original);
   }

   //________________________________________________Overridden
Methods________________________________________________

   @Override
   public Bishop makeCopy() {
      return new Bishop(this);
   }

   /**
    * Produces an ArrayList containing all the raw moves available to a Bishop
within a given board
    * @param pieces the board being played in
    * @return an ArrayList containing all the coordinates produced from the
Move class (all the diagonals)
```

```
     */
    @Override
    public ArrayList<Coordinate> getRawMoves(Pieces pieces) {
        ArrayList<Coordinate> frontRDig = Move.frontRDigFree(pieces,
this,dimension);
        ArrayList<Coordinate> backRDig = Move.backRDigFree(pieces, this,
dimension);
        ArrayList<Coordinate> backLDig = Move.backLDigFree(pieces,
this,dimension);
        ArrayList<Coordinate> frontLDig = Move.frontLDigFree(pieces, this,
dimension);

        frontRDig.addAll(backRDig);
        backLDig.addAll(frontLDig);
        frontRDig.addAll(backLDig);

        return frontRDig;
    }

    @Override
    public ImageIcon getImageIcon() {
        return icon;
    }




}





import java.util.ArrayList;
import java.util.HashMap;

public class Boards {

    public static HashMap<Coordinate, Piece> getChessBoard() {

        HashMap <Coordinate, Piece> pieces = new HashMap<>();

        int pawnBRank = 7;
        int bRank = 8;
        int pawnWRank = 2;
        int wRank = 1;
        //_______________________________________________Black
Pawns_______________________________________________

        Pawn pawnBa = new Pawn(COLOUR.B, new Coordinate('a', pawnBRank));
        Pawn pawnBb = new Pawn(COLOUR.B, new Coordinate('b', pawnBRank));
        Pawn pawnBc = new Pawn(COLOUR.B, new Coordinate('c', pawnBRank));
        Pawn pawnBd = new Pawn(COLOUR.B, new Coordinate('d', pawnBRank));
        Pawn pawnBe = new Pawn(COLOUR.B, new Coordinate('e', pawnBRank));
        Pawn pawnBf = new Pawn(COLOUR.B, new Coordinate('f', pawnBRank));
        Pawn pawnBg = new Pawn(COLOUR.B, new Coordinate('g', pawnBRank));
        Pawn pawnBh = new Pawn(COLOUR.B, new Coordinate('h', pawnBRank));

        //_______________________________________________Black
Rooks_______________________________________________

        Rook rookBa = new Rook(COLOUR.B, new Coordinate('a', bRank));
        Rook rookBh = new Rook(COLOUR.B, new Coordinate('h', bRank));

        //_______________________________________________Black
Knights_______________________________________________

        Knight knightBb = new Knight(COLOUR.B, new Coordinate('b', bRank));
        Knight knightBg = new Knight(COLOUR.B, new Coordinate('g', bRank));

        //_______________________________________________Black
Bishops_______________________________________________

        Bishop bishopBc = new Bishop(COLOUR.B, new Coordinate('c', bRank));
        Bishop bishopBf = new Bishop(COLOUR.B, new Coordinate('f', bRank));

        //_______________________________________________Black
Queen_______________________________________________

        Queen queenB = new Queen(COLOUR.B, new Coordinate('d', bRank));

        //_______________________________________________Black
King_______________________________________________

        King kingB = new King(COLOUR.B, new Coordinate('e', bRank));

        //_______________________________________________White
Pawns_______________________________________________

        Pawn pawnWa = new Pawn(COLOUR.W, new Coordinate('a', pawnWRank));
```

```
	Pawn pawnWb = new Pawn(COLOUR.W, new Coordinate('b',
pawnWRank));
	Pawn pawnWc = new Pawn(COLOUR.W, new Coordinate('c', pawnWRank));
	Pawn pawnWd = new Pawn(COLOUR.W, new Coordinate('d',
pawnWRank));
	Pawn pawnWe = new Pawn(COLOUR.W, new Coordinate('e', pawnWRank));
	Pawn pawnWf = new Pawn(COLOUR.W, new Coordinate('f', pawnWRank));
	Pawn pawnWg = new Pawn(COLOUR.W, new Coordinate('g',
pawnWRank));
	Pawn pawnWh = new Pawn(COLOUR.W, new Coordinate('h',
pawnWRank));

	//_______________________________________________White
Rooks_______________________________________________

	Rook rookWa = new Rook(COLOUR.W, new Coordinate('a', wRank));
	Rook rookWh = new Rook(COLOUR.W, new Coordinate('h', wRank));

	//_______________________________________________White
Knights_______________________________________________

	Knight knightWb = new Knight(COLOUR.W, new Coordinate('b', wRank));
	Knight knightWg = new Knight(COLOUR.W, new Coordinate('g', wRank));

	//_______________________________________________White
Bishops_______________________________________________

	Bishop bishopWc = new Bishop(COLOUR.W, new Coordinate('c', wRank));
	Bishop bishopWf = new Bishop(COLOUR.W, new Coordinate('f', wRank));

	//_______________________________________________White
Queen_______________________________________________

	Queen queenW = new Queen(COLOUR.W, new Coordinate('d', wRank));

	//_______________________________________________White
King_______________________________________________

	King kingW = new King(COLOUR.W, new Coordinate('e', wRank));

	//_______________________________________________Place Black in
HashMap_______________________________________________

	pieces.put(pawnBa.getCoords(), pawnBa);
	pieces.put(pawnBb.getCoords(), pawnBb);
```

```
        pieces.put(pawnBc.getCoords(), pawnBc);
        pieces.put(pawnBd.getCoords(), pawnBd);
        pieces.put(pawnBe.getCoords(), pawnBe);
        pieces.put(pawnBf.getCoords(), pawnBf);
        pieces.put(pawnBg.getCoords(), pawnBg);
        pieces.put(pawnBh.getCoords(), pawnBh);

        pieces.put(rookBa.getCoords(), rookBa);
        pieces.put(rookBh.getCoords(), rookBh);

        pieces.put(knightBb.getCoords(), knightBb);
        pieces.put(knightBg.getCoords(), knightBg);

        pieces.put(bishopBc.getCoords(), bishopBc);
        pieces.put(bishopBf.getCoords(), bishopBf);

        pieces.put(queenB.getCoords(), queenB);

        pieces.put(kingB.getCoords(), kingB);

        //_____________________________________________Place White in
HashMap_____________________________________________

        pieces.put(pawnWa.getCoords(), pawnWa);
        pieces.put(pawnWb.getCoords(), pawnWb);
        pieces.put(pawnWc.getCoords(), pawnWc);
        pieces.put(pawnWd.getCoords(), pawnWd);
        pieces.put(pawnWe.getCoords(), pawnWe);
        pieces.put(pawnWf.getCoords(), pawnWf);
        pieces.put(pawnWg.getCoords(), pawnWg);
        pieces.put(pawnWh.getCoords(), pawnWh);

        pieces.put(rookWa.getCoords(), rookWa);
        pieces.put(rookWh.getCoords(), rookWh);

        pieces.put(knightWb.getCoords(), knightWb);
        pieces.put(knightWg.getCoords(), knightWg);

        pieces.put(bishopWc.getCoords(), bishopWc);
        pieces.put(bishopWf.getCoords(), bishopWf);

        pieces.put(queenW.getCoords(), queenW);

        pieces.put(kingW.getCoords(), kingW);
        return pieces;
    }

    public static HashMap<Coordinate, Piece> getTestBoard() {

        HashMap <Coordinate, Piece> pieces = new HashMap<>();

        int pawnBRank = 7;
        int bRank = 8;
        int pawnWRank = 2;
        int wRank = 1;

        //_______________________________________________________Black
Pawns_______________________________________________________

        Pawn pawnB = new Pawn(COLOUR.B, new Coordinate('g', pawnBRank));

        //_______________________________________________________Black
Rooks_______________________________________________________

        Rook rookB = new Rook(COLOUR.B, new Coordinate('a', bRank));

        //_______________________________________________________Black
Knights_______________________________________________________

        Knight knightB = new Knight(COLOUR.B, new Coordinate('h', 7));

        //_______________________________________________________Black
Bishops_______________________________________________________


        //_______________________________________________________Black
Queen_______________________________________________________

        Queen queenB = new Queen(COLOUR.B, new Coordinate('d', bRank));

        //_______________________________________________________Black
King_______________________________________________________

        //_______________________________________________________White
Pawns_______________________________________________________

        Pawn pawnWe = new Pawn(COLOUR.W, new Coordinate('e', pawnWRank));
        Pawn pawnWh = new Pawn(COLOUR.W, new Coordinate('h',
pawnWRank));
```

```
        //___________________________________________________White
Rooks___________________________________________________

        Rook rookWa = new Rook(COLOUR.W, new Coordinate('a', wRank));
        Rook rookWh = new Rook(COLOUR.W, new Coordinate('h', wRank));

        //___________________________________________________White
Knights___________________________________________________

        //___________________________________________________White
Bishops___________________________________________________

        Bishop bishopW = new Bishop(COLOUR.W, new Coordinate('f', 6));

        //___________________________________________________White
Queen___________________________________________________

        Queen queenW = new Queen(COLOUR.W, new Coordinate('g', 5));

        //___________________________________________________White
King___________________________________________________

        King kingW = new King(COLOUR.W, new Coordinate('e', wRank));

        //___________________________________________________Place Black in
HashMap___________________________________________________


        pieces.put(pawnB.getCoords(), pawnB);

        pieces.put(rookB.getCoords(), rookB);

        pieces.put(knightB.getCoords(), knightB);

        pieces.put(queenB.getCoords(), queenB);

        //___________________________________________________Place White in
HashMap___________________________________________________

        pieces.put(pawnWe.getCoords(), pawnWe);
        pieces.put(pawnWh.getCoords(), pawnWh);

        pieces.put(rookWa.getCoords(), rookWa);
        pieces.put(rookWh.getCoords(), rookWh);
        pieces.put(bishopW.getCoords(), bishopW);

        pieces.put(queenW.getCoords(), queenW);

        pieces.put(kingW.getCoords(), kingW);

        return pieces;
    }

    public static HashMap<Coordinate, Piece> getCheckIngBoard() {
        HashMap <Coordinate, Piece> pieces = new HashMap<>();

        int pawnBRank = 7;
        int bRank = 8;
        int pawnWRank = 2;
        int wRank = 1;

        //_______________________________________________Black
Pawns_______________________________________________

        Pawn pawnB = new Pawn(COLOUR.B, new Coordinate('g', pawnBRank));

        //_______________________________________________Black
Rooks_______________________________________________

        Rook rookBh = new Rook(COLOUR.B, new Coordinate('h', bRank));
        Rook rookB = new Rook(COLOUR.B, new Coordinate('a', bRank));

        //_______________________________________________Black
Knights_______________________________________________

        Knight knightB = new Knight(COLOUR.B, new Coordinate('h', 7));

        //_______________________________________________Black
Bishops_______________________________________________


        //_______________________________________________Black
Queen_______________________________________________

        Queen queenB = new Queen(COLOUR.B, new Coordinate('d', bRank));

        //_______________________________________________Black
King_______________________________________________
```

```
        King kingB = new King(COLOUR.B, new Coordinate('e', bRank));

        //_______________________________________________White
Pawns_______________________________________________

        Pawn pawnWe = new Pawn(COLOUR.W, new Coordinate('e', pawnWRank));
        Pawn pawnWh = new Pawn(COLOUR.W, new Coordinate('h',
pawnWRank));

        //_______________________________________________White
Rooks_______________________________________________

        Rook rookWa = new Rook(COLOUR.W, new Coordinate('a', wRank));
        Rook rookWh = new Rook(COLOUR.W, new Coordinate('h', wRank));

        //_______________________________________________White
Knights_______________________________________________

        //_______________________________________________White
Bishops_______________________________________________

        Bishop bishopW = new Bishop(COLOUR.W, new Coordinate('f', 6));

        //_______________________________________________White
Queen_______________________________________________

        Queen queenW = new Queen(COLOUR.W, new Coordinate('g', 5));

        //_______________________________________________White
King_______________________________________________

        King kingW = new King(COLOUR.W, new Coordinate('e', wRank));

        //_______________________________________________Place Black in
HashMap_______________________________________________


        pieces.put(pawnB.getCoords(), pawnB);

        pieces.put(rookB.getCoords(), rookB);
        pieces.put(rookBh.getCoords(),rookBh);

        pieces.put(knightB.getCoords(), knightB);
        pieces.put(queenB.getCoords(), queenB);

        pieces.put(kingB.getCoords(),kingB);

        //__________________________________________________Place White in
HashMap__________________________________________________

        pieces.put(pawnWe.getCoords(), pawnWe);
        pieces.put(pawnWh.getCoords(), pawnWh);

        pieces.put(rookWa.getCoords(), rookWa);
        pieces.put(rookWh.getCoords(), rookWh);

        pieces.put(bishopW.getCoords(), bishopW);

        pieces.put(queenW.getCoords(), queenW);

        pieces.put(kingW.getCoords(), kingW);

        return pieces;
    }

    public static HashMap<Coordinate, Piece> getPromotingBoard() {
        HashMap <Coordinate, Piece> pieces = new HashMap<>();

        int pawnBRank = 7;
        int bRank = 8;
        int pawnWRank = 2;
        int wRank = 1;

        //__________________________________________________Black
Pawns__________________________________________________

        Pawn pawnW = new Pawn(COLOUR.W, new Coordinate('g', pawnBRank));
        Pawn pawnB = new Pawn(COLOUR.B, new Coordinate('b',pawnWRank));

        //__________________________________________________Black
Rooks__________________________________________________

        Rook rookBh = new Rook(COLOUR.B, new Coordinate('h', bRank));
        Rook rookB = new Rook(COLOUR.B, new Coordinate('a', bRank));

        //__________________________________________________Black
Knights__________________________________________________
```

```
        Knight knightB = new Knight(COLOUR.B, new Coordinate('h', 7));

        //__________________________________________________Black
Bishops__________________________________________________


        //__________________________________________________Black
Queen__________________________________________________

        Queen queenB = new Queen(COLOUR.B, new Coordinate('d', bRank));

        //__________________________________________________Black
King__________________________________________________

        King kingB = new King(COLOUR.B, new Coordinate('e', bRank));

        //__________________________________________________White
Pawns__________________________________________________

        Pawn pawnWe = new Pawn(COLOUR.W, new Coordinate('e', pawnWRank));
        Pawn pawnWh = new Pawn(COLOUR.W, new Coordinate('h',
pawnWRank));

        //__________________________________________________White
Rooks__________________________________________________

        Rook rookWa = new Rook(COLOUR.W, new Coordinate('a', wRank));
        Rook rookWh = new Rook(COLOUR.W, new Coordinate('h', wRank));

        //__________________________________________________White
Knights__________________________________________________

        //__________________________________________________White
Bishops__________________________________________________

        Bishop bishopW = new Bishop(COLOUR.W, new Coordinate('f', 6));

        //__________________________________________________White
Queen__________________________________________________

        Queen queenW = new Queen(COLOUR.W, new Coordinate('g', 5));

        //__________________________________________________White
King__________________________________________________
```

```
        King kingW = new King(COLOUR.W, new Coordinate('e', wRank));

        //______________________________________________________Place Black in
    HashMap______________________________________________________


        pieces.put(pawnW.getCoords(), pawnW);
        pieces.put(pawnB.getCoords(), pawnB);

        pieces.put(rookB.getCoords(), rookB);
        pieces.put(rookBh.getCoords(),rookBh);

        pieces.put(knightB.getCoords(), knightB);

        pieces.put(queenB.getCoords(), queenB);

        pieces.put(kingB.getCoords(),kingB);

        //______________________________________________________Place White in
    HashMap______________________________________________________

        pieces.put(pawnWe.getCoords(), pawnWe);
        pieces.put(pawnWh.getCoords(), pawnWh);

        pieces.put(rookWa.getCoords(), rookWa);
        pieces.put(rookWh.getCoords(), rookWh);

        pieces.put(bishopW.getCoords(), bishopW);

        pieces.put(queenW.getCoords(), queenW);

        pieces.put(kingW.getCoords(), kingW);

        return pieces;
    }


    //////////////////// - HELPER METHODS FOR displayBoard - ////////////////////

    public static String fancySeparator() {

        String unit = "|====";

        String str = "  " +
                unit.repeat(8) +
                "|";
        return str;
    }

    public static String spacer (int n,String type) {
        if (type.equals("L"))
            return n + " ";
        else
            return " " + n;
    }

    public static String fancyColumnIndex() {
        StringBuilder str = new StringBuilder();
        str.append("   ");
        for (char file = 'a'; file <= 'h'; file++) {
            str.append(" ").append(file).append("   ");
        }

        return str.toString();
    }


    ////////////////////////////////////////////////////////////////////////////////

    public static String displayBoard(Pieces pieces) {

        int dimRank = BOARD.FIRST_RANK.getRankVal();
        char dimFile = BOARD.FIRST_FILE.getFileVal();
        char lastFile = BOARD.LAST_FILE.getFileVal();

        StringBuilder str = new StringBuilder();

        str.append(fancyColumnIndex()).append("\n");
        str.append(fancySeparator()).append("\n");
        for (int rank = 8; rank >= dimRank; rank--) {
            str.append(spacer(rank,"L")).append("|");
            for (char file = dimFile; file <= lastFile; file++) {
                Coordinate coord = new Coordinate(file,rank);
                str.append((pieces.getPieces().get(coord) != null)
                        ? ("  " + pieces.getPieces().get(coord).toBoardString() + " |")
                        : "    |");
            }
            str.append(spacer(rank,"R")).append("\n");
            str.append(fancySeparator()).append("\n");
        }
```

```
        str.append(fancyColumnIndex()).append("\n");

        return str.toString();

    }


}
```

COLOURS.java

```
/**
 * The COLOUR enum class is used to assign a certain colour to a given piece.
 * The 2 possibilities are B (black) or W (white)
 * There are 2 toString methods
 */

public enum COLOUR {
    B {
        @Override
        public String toString() {
            return "Black";
        }

        @Override
        public String toSmallString() {
            return "b";
        }
    },
    W {
        @Override
        public String toString() {
            return "White";
        }

        @Override
        public String toSmallString() {
            return "w";
        }
    };
    public abstract String toSmallString();

    /**
     * @param colour the colour to be considered
     * @return the opposite colour as the one passed as an argumnet
     */

    public static COLOUR not(COLOUR colour) {
        if (colour == COLOUR.B)
            return COLOUR.W;
        else
            return COLOUR.B;

    }

}
```

CHESSIO.java

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Objects;
import java.util.Scanner;

/*
1) Save String Methods
2) Game Saving Method
*/

public class ChessIO {

    private static final String errorSave = "$$$";

    //________________________________________________Save String
Methods________________________________________________

    /**
```

```
     * Produces a String corresponding to a move
     * @param pieces the board being played in
     * @param coordinate the destination coordinate of the moves
     * @param piece the piece making the move
     * @return a String representing a move, according to the PGN naming
conventions
     */

    public static String moveString (Pieces pieces, Coordinate coordinate, Piece
piece) {

        boolean isCastle = false;

        StringBuilder str = new StringBuilder();
        Pieces previousBoard = new Pieces(pieces.getPreviousPieces());
        Coordinate previousCoordinate = previousBoard.findPiece(piece);
        Piece previousPiece = previousBoard.getPiece(previousCoordinate);

        if (piece.getName() != ID.KING) {
            str.append(piece.getName().toString());
        }
        else {
            King king = (King) piece;
            King previousKing = (King) previousPiece;

            if (coordinate.equals(king.getCastleCoordKingQ()) &&
previousKing.canCastleQueen(previousBoard)) {
                str.append("O-O-O");
                isCastle = true;
            }
            else if (coordinate.equals(king.getCastleCoordKingK()) &&
previousKing.canCastleKing(previousBoard)) {
                str.append("O-O");
                isCastle = true;
            }
            else
                str.append(piece.getName().toString());
        }

        str.append(removeAmbiguous(previousBoard, coordinate,
previousPiece));

        if (pieces.getIsCapture()) {
            if (piece.getName() == ID.PAWN) {
                assert piece instanceof Pawn;
            Pawn pawn = (Pawn) piece;
            str.append(pawn.getPreviousCoordinate().getFile()).append("x");
         }
         else
            str.append("x");
      }

      if (!isCastle) {
         str.append(coordinate.toString());
      }

      if (piece.getName() == ID.PAWN) {
         Pawn pawn = (Pawn) piece;
         if (pawn.canPromoteBlack(coordinate) ||
pawn.canPromoteWhite(coordinate))

str.append("=").append(pawn.getPromotedPiece().getName().toString());
      }

      if (pieces.isMate(COLOUR.not(piece.getColour())))
         str.append("#");
      else if (pieces.isCheck(COLOUR.not(piece.getColour())))
         str.append("+");

      return str.toString();
   }

   /**
    * Produces a String to remove the ambiguity in printing a move,
    * should 2 pieces of the same colour and same type be capable of moving to
the same coordinate
    *  @param pieces the board being played in
    *  @param coordinate the destination coordinate of the moves
    *  @param piece the piece making the move
    * @return the file of the piece, if there is another piece of the same colour
and type in the same rank
    * the rank of the piece, if there is another piece of the same colour and type
in the same file
    * otherwise, returns the empty String
    */

   public static String removeAmbiguous (Pieces pieces, Coordinate
coordinate, Piece piece) {
      if (pieces.pieceToSameCoordinate(coordinate, piece)) {
         if (pieces.pieceInSameRank(piece))
```

```
            return piece.getFile() + "";
         else if (pieces.pieceInSameFile(piece))
            return piece.getRank() + "";
         else
            return "";
      }
      else
         return "";
   }

   /**
    * Uses a scanner to request a file to save a game
    * @param test_in the scanner being used to tae the input
    * @return a Path, containing the name of the save file
    */

   public static Path fileQuery(Scanner test_in) {

      System.out.print("Enter file path: ");
      String filePath = test_in.nextLine();
      return Paths.get(filePath);
   }

   /**
    * Checks that a String constitutes a valid file path, and returns a valid path if
so.
    * @param filePath the path being checked
    * @return either a String with a valid ".txt" extension,
    * or errorSave ("$$$") should filePath be an invalid String for a file path
    */

   public static String toTxt (String filePath) {

      filePath = filePath.replaceAll("\\s","");

      if (filePath.length() == 0)
         return errorSave;

      int periodCheck = filePath.lastIndexOf(".");

      if (periodCheck == -1)
         return filePath + ".txt";
      else if (filePath.substring(periodCheck).length() == 4)
         return filePath;
      else
```

```
        return errorSave;
    }

    /**
     * @param potentialFile the path being checked
     * @return true if potentialFile is an incorrect file path
     */

    public static boolean isErrorSave (String potentialFile) {
       return errorSave.equals(potentialFile);
    }

    //_______________________________________________Game Saving
Methods_______________________________________________

    /**
     * Saves a game to the provided file path
     * @param game a String containing the information of the game
     * @param saveFile the path for saving the game
     * @return
     */

    public static boolean saveGame(String game, Path saveFile) {

       Objects.requireNonNull(game,"You can't have a null game!");
       Objects.requireNonNull(saveFile,"You can't save a game to a null path!");

       String path = String.valueOf(saveFile);
       File gameFile = new File(path); // we create an instance of a file with the
given path
       if (!gameFile.exists()) { // ensures that we don't overwrite an existing file
          try {
             FileWriter writer = new FileWriter(path);
             writer.write(game);
             writer.close();
             return true;
          } catch (IOException e) {
             e.printStackTrace();
          }
       }
       return false;
    }
}
```

COORDINATE.java

```
import java.util.Objects;

/*
1) Class Constructors
2) Getters
3) Validating Method
4) Overridden Methods
*/

/**
 * The Coordinate class is used to determine the position of the pieces within
the board.
 * A chess coordinate is determined by its file (column), represented as a
lowercase character from a to h;
 * and its rank (row), represented as a number from 1 to 8.
 */

public class Coordinate {

   public char file; //column or letter
   public int rank; //row or number
   public static Coordinate emptyCoordinate = new Coordinate((char) 0,0); // a
filler coordinate that isn't in the board

   //_______________________________________________Class
Constructors_______________________________________________

   public Coordinate (char file, int rank) {

      this.file = Character.toLowerCase(file);
      this.rank = rank;
   }

   public Coordinate (Coordinate original) {
      file = original.file;
      rank = original.rank;
   }
```

```
    public Coordinate (String coordinate) {

        if (coordinate.length() == 2 && Character.isLetter(coordinate.charAt(0)) &&
Character.isDigit(coordinate.charAt(1))) {
            file = Character.toLowerCase(coordinate.charAt(0));
            rank = Character.getNumericValue(coordinate.charAt(1));
        }
        else {
            System.out.println("Invalid coordinate format. Empty coordinate
provided.");
            file = 0;
            rank = 0;
        }
    }

    public Coordinate() {
        file = 0;
        rank = 0;
    }

    //
________________________________________________Getters_____________
___________________________________

    public char getFile() {
        return file;
    }

    public int getRank() {
        return rank;
    }

    //________________________________________________Validating
Method________________________________________________

    /**
     * inBoard checks that an instance of a Coordinate is within the chess board
     * @param coord is the coordinate that is checked for validity within the
board
     * @return true if the coordinate is in the board
     */

    public static boolean inBoard (Coordinate coord) {
        char coordFile = coord.getFile();
        int coordRank = coord.getRank();
        return (coordFile >= BOARD.FIRST_FILE.getFileVal()
                && coordFile <= BOARD.LAST_FILE.getFileVal()
                && coordRank >= BOARD.FIRST_RANK.getRankVal()
                && coordRank <= BOARD.LAST_RANK.getRankVal());
    }

    //______________________________________________Overridden
Methods______________________________________________

    @Override
    public String toString() {
        return file + "" + rank;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Coordinate that = (Coordinate) o;
        return file == that.file &&
                rank == that.rank;
    }

    @Override
    public int hashCode() {
        return Objects.hash(file, rank);
    }


}
```

GUI BOARD.java

```
import javax.swing.*;
import javax.swing.border.Border;
import javax.swing.border.EmptyBorder;
import javax.swing.border.MatteBorder;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.nio.file.Paths;
import java.util.HashSet;


// https://www.youtube.com/watch?v=SNYFjgz4bU4
// png files from https://marcelk.net/chess/pieces/

/*
1) Class Constructor
2) Board Labelling Methods
3) Board Formatting Methods
4) Information Panel Method
5) Piece Movement Methods
6) Game Saving Class
7) Piece Movement Methods
8) Overridden Methods
*/

/**
 * The GUIBoard class contains all the methods and classes used to display the
game of chess
 */

public class GUIBoard extends JFrame {

    // the pieces used for the game
    private final Pieces pieces;
    private final int dimension = BOARD.LAST_RANK.getRankVal();
    private final int firstRank = BOARD.FIRST_RANK.getRankVal();
    private final char firstFile = BOARD.FIRST_FILE.getFileVal();
    private final char lastFile = BOARD.LAST_FILE.getFileVal();
    private final char charFile = (char) (firstFile - 1);
    // the turn of the game being played
    private static COLOUR turn = COLOUR.W;

    // text pane containing the moves of a game
    private final JTextPane movePane = new JTextPane();
    // text pane containing the message at the end of the game (mate or draw)
    private final JTextPane matePane = new JTextPane();

    // the board on which the game is played
    private final JButton[][] board = new JButton[dimension][dimension];
    // the button used to save a game
    private final JButton saveButton = new JButton("Save Game");
    // colours for the board
    private final Color brown = new Color(150, 75, 0); //brown #964B00
    private final Color pastel = new Color(255, 222, 173); //navajorwhite #FFDEAD
    private final Color intermediate = new Color(255, 255, 153);
    public static final Color infoColour = new Color(51,51,51);

    // size of a square in board
    private static final int tileSize = 88;

    // icon for squares without pieces
    private final BufferedImage invisible = new BufferedImage(80, 80,
BufferedImage.TYPE_INT_ARGB);
    private final ImageIcon invisibleIcon = new ImageIcon(invisible);

    // used to determine number of clicks
    private int counter = 0;

    // number of turns used for game saving
    private int numberOfTurns = 0;

    // build up the game save file
    private static final StringBuilder str = new StringBuilder();

    // piece selected to move
    private Piece movingPiece;

    // ActionHandler used in GUIBoard construcotr
    ButtonHandle gameClick = new ButtonHandle();

    //________________________________________________Class
Constructor________________________________________________

    /**
     * The GUIBoard construcotr creates the chess board
     * It assigns buttons to the boards 2D array, and adds their ActionListeners
(gameClick)
     * Adds all the additional JPanels (i.e to show files, rows, save panels, etc...)
     * Sets up the JFrame
     * @param p the pieces HashMap used for the game
     */

    public GUIBoard(Pieces p) {
        setTitle("CHESS23");
        setBackground(Color.black);
        Container contents = getContentPane();
        contents.setLayout(new BorderLayout());

        pieces = p;

        JPanel boardPanel = new JPanel(new GridLayout(dimension, dimension));
        for (int rank = dimension; rank >= firstRank; rank--) {
            for (int file = 1; file <= dimension; file++) {
                char fileChar = (char) (charFile + file);
                Coordinate tileCoord = new Coordinate(fileChar, rank);
                board[rank - 1][file - 1] = setButton(tileCoord, p);
                board[rank - 1][file - 1].addActionListener(gameClick);
                boardPanel.add(board[rank - 1][file - 1]);
            }
        }

        JPanel fullBoardPanel = new JPanel(new BorderLayout());
        fullBoardPanel.add(createFileLabelsTop(),BorderLayout.NORTH);
        fullBoardPanel.add(createRankLabelsLeft(),BorderLayout.WEST);
        fullBoardPanel.add(boardPanel, BorderLayout.CENTER);
        fullBoardPanel.add(createRankLabelsRight(),BorderLayout.EAST);
        fullBoardPanel.add(createFileLabelsBottom(), BorderLayout.SOUTH);
        contents.add(fullBoardPanel, BorderLayout.WEST);
        contents.add(createInfoPanel(), BorderLayout.EAST);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setResizable(false);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    //_______________________________________________Board Labelling
Methods_______________________________________________

    /**
     * Shows the ranks (1-8) of the board
     * @return a JPanel with JLabels containing the ranks of the chess board
     */

    public JPanel createRanks() {
        JPanel ranks = new JPanel(new GridLayout(dimension,0));
        ranks.setBackground(pastel);

        int rankPad = 4;
        for (int rank = dimension; rank >= firstRank; rank--) {
            JLabel rankLabel = new JLabel(String.valueOf(rank));
            rankLabel.setFont(new Font("TimesRoman", Font.BOLD, 23));
            rankLabel.setForeground(brown);
            rankLabel.setBorder(new EmptyBorder(0,rankPad,0,rankPad));
            ranks.add(rankLabel);
        }

        return ranks;
    }

    /**
     * Shows the files (a - h) of the board
     * @return a JPanel with JLabels containing the files of the chess board
     */

    public JPanel createFiles() {
        JPanel files = new JPanel(new GridLayout(0,dimension));
        files.setBackground(pastel);

        int filePad = 42;

        for (char file = firstFile; file <= lastFile; file++) {
            JLabel fileLabel = new JLabel(String.valueOf(file));

            fileLabel.setFont(new Font("TimesRoman", Font.BOLD, 23));
            fileLabel.setForeground(brown);
            fileLabel.setBorder(new EmptyBorder(0,filePad,0,filePad));
            files.add(fileLabel);
        }
        return files;
    }

    /**
     * Shows a brown border for the board
     * @return a JPanel that acts as a border for the baord
     */

    public JPanel createBorder() {
        JPanel border = new JPanel();
        border.setBackground(brown);

        return border;
    }
```

```
    /**
     * @return a JPanel, acting as a corner for the board
     */

    public Border createCorner() {
        int borderPad = 20;
        return new MatteBorder(0,borderPad,0,borderPad,pastel);
    }

    /**
     * Displays the ranks to the left of the board
     * @return a JPanel displaying the ranks and the border displayed to the left
of the board
     */

    public JPanel createRankLabelsLeft() {
        JPanel rankLabels = new JPanel(new BorderLayout());

        rankLabels.add(createRanks(), BorderLayout.WEST);
        rankLabels.add(createBorder(), BorderLayout.EAST);

        return rankLabels;
    }

    /**
     * Displays the files to the bottom of the board
     * @return a JPanel displaying the files and the border displayed to the
bottom of the board
     */

    public JPanel createFileLabelsBottom() {
        JPanel fileLabels = new JPanel(new BorderLayout());

        fileLabels.add(createFiles(), BorderLayout.SOUTH);
        fileLabels.add(createBorder(), BorderLayout.NORTH);

        fileLabels.setBorder(createCorner());

        return fileLabels;
    }

    /**
     * Displays the ranks to the right of the board
     * @return a JPanel displaying the ranks and the border displayed to the right
of the board
     */

    public JPanel createRankLabelsRight() {
        JPanel rankLabels = new JPanel(new BorderLayout());

        rankLabels.add(createRanks(), BorderLayout.EAST);
        rankLabels.add(createBorder(), BorderLayout.WEST);

        return rankLabels;
    }

    /**
     * Displays the files to the top of the board
     * @return a JPanel displaying the files and the border displayed to the top of
the board
     */

    public JPanel createFileLabelsTop() {
        JPanel fileLabels = new JPanel(new BorderLayout());

        fileLabels.add(createFiles(), BorderLayout.NORTH);
        fileLabels.add(createBorder(), BorderLayout.SOUTH);

        fileLabels.setBorder(createCorner());

        return fileLabels;
    }

    //________________________________________________Board Formatting
Methods________________________________________________

    /**
     * Sets the background of a button, based on whether its a white or black
square
     * @param coordinate the coordinate of the square being considered
     * @param b the button at the given square
     */

    private void backgroundSetter (Coordinate coordinate, JButton b){
        int signature = coordinate.getFile() - charFile + coordinate.getRank();
        if (signature % 2 == 0) {
            b.setBackground(brown);
        } else {
```

```
        b.setBackground(pastel);
    }
}

/**
 * Formats a JButton to be used in the board 2D array
 * @param coordinate the coordinate of the square being considered
 * @param pieces the pieces being used for the game
 * @return a JButton, correctly formatted with the piece it represents
 */

private JButton setButton (Coordinate coordinate, Pieces pieces){
    JButton b = new JButton();

    Piece piece;

    backgroundSetter(coordinate, b);

    if (pieces.getPieces().get(coordinate) == null) {
        piece = null;
    } else {
        piece = pieces.getPiece(coordinate);
    }

    if (piece != null)
        b.setIcon(piece.getImageIcon());
    else
        b.setIcon(invisibleIcon);

    formatButton(b);

    return b;
}

/**
 * Correctly formats the properties of the JButton used in the boards array
 * @param b the JButton being formatted
 */

private void formatButton (JButton b) {
    b.setSize(tileSize, tileSize);
    b.setOpaque(true);
    b.setContentAreaFilled(true);
    b.setBorderPainted(false);
    b.setVisible(true);
```

```
   }

   /**
    * Correctly formats the properties of an invisible JButton used in the boards
array
    * @param b the JButton being formatted
    */

   public static void formatInvisibleButton (JButton b) {
      b.setSize(tileSize, tileSize);
      b.setOpaque(false);
      b.setContentAreaFilled(false);
      b.setBorderPainted(false);
      b.setVisible(true);
   }

   /**
    * Removes the ActionListener of all the JButtons of the board
    * This is done once a game is finished
    */

   private void disableBoardButtons() {
      for (int row = 0; row < dimension; row++) {
         for (int file = 0; file < dimension; file++) {
            board[row][file].removeActionListener(gameClick);
         }
      }
   }

   //_______________________________________________Information Panel
Method________________________________________________

   /**
    * The "Info Panel" is a JPanel containing information of the game (the moves
being played)
    * and whether its check mate or a draw.
    * It contains the saveButton, used to save a game at any given time
    * @return a JPanel containing information on the game
    */

   private JPanel createInfoPanel() {

      JPanel movePanel = new JPanel(new GridBagLayout());

      GridBagConstraints gbc = new GridBagConstraints();
```

```
movePanel.setBackground(infoColour);
movePanel.setVisible(true);

movePanel.setPreferredSize(new Dimension(300,800));

movePane.setEditable(false);
movePane.setForeground(Color.white);
movePane.setBackground(infoColour);
movePane.setFont(new Font("Arial", Font.BOLD, 14));
movePane.setBorder(new EmptyBorder(40,20,40,20));
JScrollPane scrollMoves = new JScrollPane(movePane);

gbc.gridx = 0;
gbc.weightx = 1;
gbc.weighty = 0.4;
gbc.gridy = 1;
gbc.fill = GridBagConstraints.BOTH;

movePanel.add(scrollMoves,gbc); // show the moves being played

saveButton.setBackground(Color.orange);
saveButton.setForeground(Color.white);
saveButton.setOpaque(true);
saveButton.setContentAreaFilled(true);
saveButton.setBorderPainted(false);
SaveHandle saver = new SaveHandle();
saveButton.addActionListener(saver);

gbc.gridy = GridBagConstraints.RELATIVE;
gbc.fill = GridBagConstraints.NONE;

movePanel.add(saveButton,gbc); // add saving button

matePane.setEditable(false);
matePane.setForeground(Color.white);
matePane.setBackground(infoColour);
matePane.setFont(new Font("Arial", Font.BOLD, 20));
matePane.setBorder(new EmptyBorder(0,80,40,80));

gbc.fill = GridBagConstraints.HORIZONTAL;

movePanel.add(matePane,gbc); // add information on check mate / draw

return movePanel;
```

```
    }

    //_______________________________________________Piece Movement
Methods_______________________________________________

    /**
     * Changes the turn of the game
     */

    private void setTurn() {
        turn = COLOUR.not(turn);
    }

    /**
     * Clears the counter. Used after a second click.
     * This is used for the logic of the game, as we can know when destination
coordinate has been set,
     * or whether we have yet to choose and origin coordinate.
     */

    private void clearCounter() {
        counter = 0;
    }

    /**
     * This is used to illuminate the potential moves of the piece that is being
clicked on
     * @param potentials the potential moves of the piece being clicked on
     */
    private void processClick(HashSet<Coordinate> potentials) {

        for (int rank = 1; rank <= dimension; rank++) {
            for (char file = firstFile; file <= lastFile; file++) {
                int processedRank = rank - firstRank;
                int processedFile = file - firstFile;
                Coordinate potentialCoord = new Coordinate(file, rank);
                if (potentials.contains(potentialCoord))
                    board[processedRank]
[processedFile].setBackground(intermediate);
                else
                    backgroundSetter(potentialCoord, board[processedRank]
[processedFile]);
            }
        }
    }
    /**
     * This class is used to handle the game logic for the GUI.
     */

    private class ButtonHandle implements ActionListener {

        /**
         * Displays the potential moves, increases counter and sets the
movingPiece to the piece selected
         * @param piece the piece selected
         */
        private void selectPiece(Piece piece) {
            processClick(piece.getPotentialMoves());
            counter++;
            movingPiece = piece;
        }

        /**
         * Used to turn 2D array "coordinates" into chess board coordinates
         * @param row the row of the 2D array
         * @param column the column of the 2D array
         * @return the Coordinate corresponding to an element of the board array
with the given row and column
         */

        private Coordinate toCoordinate(int row, int column) {
            int rank = row + firstRank;
            char file = (char) (column + firstFile);

            return new Coordinate(file, rank);
        }

        /**
         * If the counter is at 0, and a button is clicked, we increase the counter
and execute game click.
         * We also save the piece that has been clicked on, and assign it to
movingPiece
         * If the counter is not at 0, we check the coordinate that has been selected.
         * If its a valid move for the movingPiece, then it executes makeMove on
the pieces.
         * It also appends the move to str, changes the turn, reprints the board with
the new pieces and clears the counter.
         * If it is an invalid move, then we consider the new clicked piece as the
moving piece, and again, execute gameClick
```

```
        * If its mate, stalemate or draw, all the board buttons are disabled and the
game ends.
        */

       @Override
       public void actionPerformed(ActionEvent actionEvent) {

           Coordinate coordinate = null;

           JButton source = (JButton) actionEvent.getSource();

           for (int i = 0; i < dimension; i++) {
               for (int j = 0; j < dimension; j++) {
                   if (board[i][j].equals(source)) {
                       coordinate = toCoordinate(i, j);
                   }
               }
           }

           Piece originPiece = pieces.getPieces().get(coordinate);

           if (coordinate != null && Coordinate.inBoard(coordinate)) {
               if (counter == 0) {
                   if (originPiece != null && originPiece.getColour() == turn) {
                       selectPiece(originPiece);
                   }
               } else {
                   Piece piece = movingPiece;

                   if (piece.isValidMove(coordinate, turn)) {
                       pieces.makeMove(coordinate, piece);
                       if (turn == COLOUR.W) {
                           numberOfTurns++;
                           str.append(numberOfTurns).append(".
").append(ChessIO.moveString(pieces, coordinate, piece)).append(" ");
                       } else
                           str.append(ChessIO.moveString(pieces, coordinate,
piece)).append(" ");
                       movePane.setText(str.toString());
                       for (int rank = 1; rank <= dimension; rank++) {
                           for (char file = firstFile; file <= lastFile; file++) {
                               int processedRank = rank - firstRank;
                               int processedFile = file - firstFile;
                               Coordinate potentialCoord = new Coordinate(file, rank);
```

```
                        backgroundSetter(potentialCoord, board[processedRank]
[processedFile]);
                        board[processedRank]
[processedFile].setIcon(invisibleIcon);
                        if (pieces.getPieces().get(potentialCoord) != null) {
                            Piece updatePiece = pieces.getPiece(potentialCoord);
                            board[processedRank]
[processedFile].setIcon(updatePiece.getImageIcon());
                        }
                    }
                }

                clearCounter();
                setTurn();

                if (pieces.isMate(turn)) {
                    matePane.setText(COLOUR.not(turn).toString() + " won by
checkmate.");
                    disableBoardButtons();
                } else if (pieces.isStalemate(COLOUR.not(turn))) {
                    matePane.setText("Draw by stalemate.");
                    disableBoardButtons();
                } else if (pieces.isDraw()) {
                    matePane.setText("It's a draw.");
                    disableBoardButtons();
                }
            } else {
                if (originPiece != null && originPiece.getColour() == turn) {
                    selectPiece(originPiece);
                }
            }
        }
      }
    }
  }

  //________________________________________________Game Saving
Class________________________________________________

  /**
   * Handles the logic to save the game
   */

  public static class SaveHandle implements ActionListener {
    /**
     * This handles the saving of a game.
     * It produces a JOptionPane with an InputDialog.
     * We use ChessIO to validate the file path provided.
     * JOptionPane with message dialogs are then presented, based on
whether the save was successful or not
     */

    @Override
    public void actionPerformed(ActionEvent actionEvent) {

       ImageIcon icon = new ImageIcon("WKing.png");

       UIManager.put("OptionPane.background", infoColour);
       UIManager.put("Panel.background", infoColour);
       UIManager.put("OptionPane.messageForeground", Color.white);

       String fileSave = (String) JOptionPane.showInputDialog(null,
             "Enter a file name to save the game:",
             "Save Game",
             JOptionPane.INFORMATION_MESSAGE,
             icon,null,null);

       if (fileSave != null) {

          String filePath = ChessIO.toTxt(fileSave);

          if (ChessIO.isErrorSave(filePath)) {
             JOptionPane.showMessageDialog(null,
                   fileSave + " is an incorrect file name.",
                   "Failed Saving",
                   JOptionPane.ERROR_MESSAGE,
                   icon);
          } else {
             if (ChessIO.saveGame(str.toString(), Paths.get(filePath)))
                JOptionPane.showMessageDialog(null,
                      "Game saved succesfuly on path " + filePath,
                      "Save Succesful",
                       JOptionPane.INFORMATION_MESSAGE,
                       icon);
             else
                JOptionPane.showMessageDialog(null,
                      "There was an error saving the file on the path " + filePath +
". The name provided might be a duplicate.",
                      "Failed Saving",
                            JOptionPane.ERROR_MESSAGE,
                            icon);

                }
            }
        }
    }


    public static void main (String[]args){
        Pieces pieces = new Pieces();
        pieces.setGUIGame(true);
        new GUIBoard(pieces);
    }

}
```

ID.java

```
/**
 * The ID enum class is used to identify the "type" of a piece (King, Queen,
Rook, Bishop, Knight, and Pawn)
 * Each has 2 toString methods
 */

public enum ID {
    KING {
        @Override
        public String toString() {
            return "K";
        }
        public String toFullString() {
            return "King";
        }
    },
    QUEEN {
        @Override
        public String toString() {
            return "Q";
        }
        public String toFullString() {
            return "Queen";
        }
    },
    ROOK {
        @Override
        public String toString() {
            return "R";
        }
        public String toFullString() {
            return "Rook";
        }
    },
    BISHOP {
        @Override
        public String toString() {
            return "B";
        }
        public String toFullString() {
            return "Bishop";
        }
    },
    KNIGHT {
        @Override
        public String toString() {
            return "N";
        }
        public String toFullString() {
            return "Knight";
        }
    },
    PAWN {
        @Override
        public String toString() {
            return "";
        }
        public String toFullString() {
            return "Pawn";
        }
    };

    public abstract String toFullString();

}
```

KING.java

```
import javax.swing.*;
import java.util.*;

/*
1) Class Constructors
2) Getters & Setters
3) Castling Validation Methods
4) Overridden Methods
*/

public class King extends Piece{

   // the coordinate to which the king goes after castling King side
   private Coordinate castleCoordKingK;
   // the coordinate to whcih the king goes after castling Queen side
   private Coordinate castleCoordKingQ;
   // the coordinate through which the king goes when castling King side
   private Coordinate transitionCoordKingK;
   // the coordinate through which the king goes when castling Queen side
   private Coordinate transitionCoordKingQ;
   // the rook on the King side
   private Rook rookKing;
   // the rook on the Queen side
   private Rook rookQueen;
   private ImageIcon icon;

   //_______________________________________________Class
Constructors_______________________________________________

   public King(COLOUR colour, Coordinate OGcoord) {
      super(ID.KING, colour, OGcoord);
      if (getColour() == COLOUR.B)
         icon = new ImageIcon("BKing.png");
      else if (getColour() == COLOUR.W)
         icon = new ImageIcon("WKing.png");
   }

   public King(King original) {
      super(original);
   }
```

```
    //_______________________________________________Getters & 
Setters_______________________________________________

    public Coordinate getCastleCoordKingK() {
        return castleCoordKingK;
    }

    public Coordinate getCastleCoordKingQ() {
        return castleCoordKingQ;
    }

    public Coordinate getTransitionCoordKingK() {
        return transitionCoordKingK;
    }

    public Coordinate getTransitionCoordKingQ() {
        return transitionCoordKingQ;
    }

    public Rook getRookKing() {
        return rookKing;
    }

    public Rook getRookQueen() {
        return rookQueen;
    }

    //_______________________________________________Castling Validation 
Methods_______________________________________________

    /**
     * Determines whether a King can castle king side (short castling)
     * @param pieces the board being played in
     * @return true if and only if:
     * the king isn't in check
     * there is a rook on the king side (h1 for whites, h8 for blacks)
     * there are no pieces between the king and the rook
     * neither the rook nor the king have moved
     * If the king can castle, then the coordinate to which the king goes to castle 
is added to castleCoordKingk
     * It sets the corresponding coordinate to the king side rook
     * It also saves the coordinate through which the king goes when castling.
     * This is used in "removeOwnCheck" to ascertain that castling is possible
     * ("One may not castle out of, through, or into check.")
     */
```

```
   public boolean canCastleKing (Pieces pieces) {

      if (pieces.isCheck(getColour()))
         return false;

      HashMap<Coordinate, Piece> colouredPieces =
pieces.getColourPieces(getColour());

      for (Piece value : colouredPieces.values()) {
         if (value.getName() == ID.ROOK && value.getFile() ==
BOARD.LAST_FILE.getFileVal())
            rookKing = (Rook) value;
      }

      int distanceRookKing = 2;
      ArrayList<Coordinate> castleCoords;

      if (getColour() == COLOUR.B)
         castleCoords = Move.leftFree(pieces, this, dimension);
      else
         castleCoords = Move.rightFree(pieces, this, dimension);

      boolean isSpace = castleCoords.size() == distanceRookKing;

      boolean canCastle = rookKing != null &&
            !rookKing.getHasMoved() &&
            !getHasMoved() &&
            isSpace;

      if (canCastle) {
         castleCoordKingK = castleCoords.get(1);
         transitionCoordKingK = castleCoords.get(0);
         rookKing.setCastleCoordRook(castleCoords.get(0));
         return true;
      }
      return false;
   }

   /**
    * Determines whether a King can castle queen side (long castling)
    * @param pieces the board being played in
    * @return true if and only if:
    * the king isn't in check
    * there is a rook on the queen side (a1 for whites, a8 for blacks)
```

```
     * there are no pieces between the king and the rook
     * neither the rook nor the king have moved
     * If the king can castle, then the coordinate to which the king goes to castle
is added to castleCoordKingQ
     * It sets the corresponding coordinate to the queen side rook
     * It also saves the coordinate through which the king goes when castling.
     * This is used in "removeOwnCheck" to ascertain that castling is possible
     * ("One may not castle out of, through, or into check.")
     */

    public boolean canCastleQueen (Pieces pieces) {

        if (pieces.isCheck(getColour()))
            return false;

        HashMap<Coordinate,Piece> colouredPieces =
pieces.getColourPieces(getColour());

        for (Piece value : colouredPieces.values()) {
            if (value.getName() == ID.ROOK && value.getFile() ==
BOARD.FIRST_FILE.getFileVal())
                rookQueen = (Rook) value;
        }

        int distanceRookQueen = 3;
        ArrayList<Coordinate> castleCoords;

        if (getColour() == COLOUR.W) {
            castleCoords = Move.leftFree(pieces, this, dimension);
        }
        else {
            castleCoords = Move.rightFree(pieces, this, dimension);
        }

        boolean isSpace = castleCoords.size() == distanceRookQueen;


        boolean canCastle = rookQueen != null &&
                !rookQueen.getHasMoved() &&
                !getHasMoved() &&
                isSpace;

        if (canCastle) {
            castleCoordKingQ = castleCoords.get(1);
            transitionCoordKingQ = castleCoords.get(0);
            rookQueen.setCastleCoordRook(castleCoords.get(0));
            return true;
        }
        return false;
    }

    //___________________________________________Overridden
Methods___________________________________________

    @Override
    public King makeCopy() {
        return new King(this);
    }

    /**
     * Produces an ArrayList containing all the raw moves available to a King
within a given board
     * @param pieces the board being played in
     * @return an ArrayList containing all the coordinates produced from the
Move class
     * (all the diagonals, all verticals and all horizontals, using 1 as a limit).
     * Also adds coordinates if castling is possible.
     */

    @Override
    public ArrayList<Coordinate> getRawMoves(Pieces pieces) {
        //get and add all "raw" moves (reachable from position)
        ArrayList<Coordinate> front = Move.frontFree(pieces,this,single);
        ArrayList<Coordinate> right = Move.rightFree(pieces,this,single);
        ArrayList<Coordinate> back = Move.backFree(pieces,this,single);
        ArrayList<Coordinate> left = Move.leftFree(pieces,this,single);
        ArrayList<Coordinate> frontRDig = Move.frontRDigFree(pieces,
this,single);
        ArrayList<Coordinate> backRDig = Move.backRDigFree(pieces, this,
single);
        ArrayList<Coordinate> backLDig = Move.backLDigFree(pieces,
this,single);
        ArrayList<Coordinate> frontLDig = Move.frontLDigFree(pieces, this,
single);

        front.addAll(right);
        back.addAll(left);
        front.addAll(back);

        frontRDig.addAll(backRDig);
        backLDig.addAll(frontLDig);
        frontRDig.addAll(backLDig);

        front.addAll(frontRDig);

        // if can castle, add coordinates
        if (canCastleKing(pieces))
           front.add(castleCoordKingK);
        if (canCastleQueen(pieces))
           front.add(castleCoordKingQ);


        return front;
    }

    @Override
    public ImageIcon getImageIcon() {
        return icon;
    }


}
```

KNIGHT.java

```
import javax.swing.*;
import java.util.ArrayList;

/*
1) Class Constructors
2) Overridden Methods
*/

public class Knight extends Piece{

    private ImageIcon icon;

    //_______________________________________________Class
Constructors________________________________________________
```

```
   public Knight(COLOUR colour, Coordinate OGcoord) {
      super(ID.KNIGHT, colour, OGcoord);
      if (getColour() == COLOUR.B)
         icon = new ImageIcon("BKnight.png");
      else if (getColour() == COLOUR.W)
         icon = new ImageIcon("WKnight.png");
   }

   public Knight(Knight original) {
      super(original);
   }

   //________________________________________________Overridden
Methods________________________________________________

   @Override
   public Knight makeCopy() {
      return new Knight(this);
   }

   /**
    * Produces an ArrayList containing all the raw moves available to a Knight
within a given board
    * @param pieces the board being played in
    * @return an ArrayList containing all the coordinates produced from the
Move class (all the Knight moves)
    */

   @Override
   public ArrayList<Coordinate> getRawMoves(Pieces pieces) {
      ArrayList<Coordinate> front = Move.frontKnight(pieces,this);
      ArrayList<Coordinate> right = Move.backKnight(pieces,this);
      ArrayList<Coordinate> back = Move.rightKnight(pieces,this);
      ArrayList<Coordinate> left = Move.leftKnight(pieces,this);

      front.addAll(right);
      back.addAll(left);
      front.addAll(back);

      return front;
   }

   @Override
   public ImageIcon getImageIcon() {
      return icon;
    }

}
```

MAIN.java

```
import java.util.Scanner;

public class Main {


    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        System.out.println("Welcome to CHESS23. Enter \"GUI\" to play on a
Graphical User Interface. Alternatively, enter \"TBI\" to play on the terminal.");

        Pieces pieces = new Pieces();

        boolean correctType = false;

        while (!correctType) {

            String gameType = sc.next();

            if (gameType.equals("GUI")) {
                pieces.setGUIGame(true);
                new GUIBoard(pieces);
                correctType = true;
            }
            else if (gameType.equals("TBI")) {
                pieces.setGUIGame(false);
                TBIBoard.gameLoop(pieces);
                correctType = true;
            }
            else
                System.out.println("Incorrect game format. Try again.");
        }
    }

}
```

MOVE.java

```
import java.util.ArrayList;
import java.util.Objects;
import java.util.Scanner;

/*
1) Null Messages
2) Support Methods
3) Vertical Movement Methods
4) Horizontal Movement Methods
5) Diagonal Right Movement Methods
6) Diagonal Left Movement Methods
7) Vertical Knight Movement Methods
8) Horizontal Knight Movement Methods
9) Move Input Methods
*/

/**
 * The Move class contains all methods that are used to find all raw moves for a
given piece.
 * The move calculating methods all have a Pieces, Piece and int argument.
 * Pieces is where the game is being played
 * Piece is the piece that is being moved
 * the int (known as limit) is how many "steps" in a given direction we want to
lok for
 * We have generic methods for calculating vertical, horizontal and diagonal
moves in all directions.
 * We also have individual methods for calculating the moves of a Knight
 * We use a "factor" (either 1 or -1) to calculate raw coordinates, depending on
whether the piece being moved is white or black
 * (black moving forward is the same direction as white moving backward)
 * For diagonals, we also have a factorV (for vertical movement) and factorH (for
horizontal movement)
```

```
 * We determine all coordinates in a given direction, beginning from the
position of a piece.
 * We return an ArrayList containing al these valid coordinates in the direction
 * If the coordinate is occupied, we check whether it is occupied by a piece of
the same colour.
 * If it is of the same colour, we return the ArrayList of all the previous
coordinates
 * If it is of different colour, we include the coordinate in the ArrayList and
return it.
 * If the coordinate isn't occupied, we add it and continue checking in the given
direction
 */

public class Move {

   //________________________________________________Null
Messages________________________________________________

   private static final String nullPieces = "You can't play with a null board.";
   private static final String nullCoord = "Board coordinates must not be null.";

   //________________________________________________Support
Methods________________________________________________

   /**
     * Given a certain pieces HashMap and a given Coordinate, checks whether
the coordinate is occupied by a piece of agiven colour
     * @param pieces the board on which the game is being played
     * @param destination the coordinate being considered
     * @param colour the colour being considered
     * @return true if the given coordinate is occupied by a piece of the opposite
colour (that is provided as an argument)
     * @throws NullPointerException if the coordinate or the pieces HashMap are
null
     */

   public static boolean isNotTileColour(Pieces pieces, Coordinate destination,
COLOUR colour) {
       Objects.requireNonNull(pieces,nullPieces);
       Objects.requireNonNull(destination,nullCoord);

       return !(pieces.getPieces().get(destination).getColour() == (colour));
   }

   /**
```

```
     * Given a certain pieces HashMap and a given Coordinate, checks whether
the coordinate is occupied by a piece
     * @param pieces the board on which the game is being played
     * @param destination the coordinate being considered
     * @return true if the coordinate is occupied by a piece
     * @throws NullPointerException if the coordinate or the pieces HashMap are
null
     */

    public static boolean tileFull (Pieces pieces, Coordinate destination) {

        Objects.requireNonNull(pieces,nullPieces);
        Objects.requireNonNull(destination,nullCoord);

        return pieces.getPieces().get(destination) != null;
    }

    //________________________________________________Vertical Movement
Methods________________________________________________

    // checks vertical moves directly in front (that is, moves along the same file)

    public static ArrayList<Coordinate> frontFree(Pieces pieces, Piece piece, int
limit) {

        ArrayList<Coordinate> moves = new ArrayList<>();
        int factor;

        if (piece.getColour().equals(COLOUR.B))
            factor = -1;
        else
            factor = 1;

        for (int advance = 1; advance <= limit; advance++) {
            int change = factor*advance;
            Coordinate checkCoord = new Coordinate(piece.getFile(),
piece.getRank() + change);
            if (Coordinate.inBoard(checkCoord)) {
                boolean occupiedTile = tileFull(pieces, checkCoord);
                if (occupiedTile && isNotTileColour(pieces, checkCoord,
piece.getColour())) {
                    moves.add(checkCoord);
                    return moves;
                }
                else if (occupiedTile)
                return moves;
             else
                moves.add(checkCoord);
          }
       }

       return moves;
    }

    // checks vertical moves directly behind (that is, moves along the same file)

    public static ArrayList<Coordinate> backFree(Pieces pieces, Piece piece, int
limit) {

       ArrayList<Coordinate> moves = new ArrayList<>();
       int factor;

       if (piece.getColour().equals(COLOUR.B))
          factor = 1;
       else
          factor = -1;

       for (int advance = 1; advance <= limit; advance++) {
          int change = factor*advance;
          Coordinate checkCoord = new
Coordinate(piece.getFile(),piece.getRank()+change);
          if (Coordinate.inBoard(checkCoord)) {
             boolean occupiedTile = tileFull(pieces, checkCoord);
             if (occupiedTile && isNotTileColour(pieces, checkCoord,
piece.getColour())) {
                moves.add(checkCoord);
                return moves;
             }
             else if (occupiedTile)
                return moves;
             else
                moves.add(checkCoord);
          }
       }
       return moves;
    }

    //________________________________________________Horizontal
Movement Methods________________________________________________
```

```
   // checks horizontal moves directly to the right (that is, moves along the
same rank)

   public static ArrayList<Coordinate> rightFree(Pieces pieces, Piece piece, int
limit) {

      ArrayList<Coordinate> moves = new ArrayList<>();
      int factor;

      if (piece.getColour().equals(COLOUR.B))
         factor = -1;
      else
         factor = 1;

      for (int advance = 1; advance <= limit; advance++) {
         int change = factor*advance;
         Coordinate checkCoord = new Coordinate((char) (piece.getFile()
+change),piece.getRank());
         if (Coordinate.inBoard(checkCoord)) {
            boolean occupiedTile = tileFull(pieces, checkCoord);
            if (occupiedTile && isNotTileColour(pieces, checkCoord,
piece.getColour())) {
               moves.add(checkCoord);
               return moves;
            }
            else if (occupiedTile)
               return moves;
            else
               moves.add(checkCoord);
         }
      }
      return moves;
   }

   // checks horizontal moves directly to the left (that is, moves along the same
rank)

   public static ArrayList<Coordinate> leftFree(Pieces pieces, Piece piece, int
limit) {

      ArrayList<Coordinate> moves = new ArrayList<>();
      int factor;

      if (piece.getColour().equals(COLOUR.B))
         factor = 1;
```

```
        else
           factor = -1;

        for (int advance = 1; advance <= limit; advance++) {
           int change = factor*advance;
           Coordinate checkCoord = new Coordinate((char) (piece.getFile()
+change),piece.getRank());
           if (Coordinate.inBoard(checkCoord)) {
              boolean occupiedTile = tileFull(pieces, checkCoord);
              if (occupiedTile && isNotTileColour(pieces, checkCoord,
piece.getColour())) {
                 moves.add(checkCoord);
                 return moves;
              }
              else if (occupiedTile)
                 return moves;
              else
                 moves.add(checkCoord);
           }
        }
        return moves;
     }

     //________________________________________________Diagonal Right
Movement Methods_______________________________________________

     // checks diagonal moves to the front and right

     public static ArrayList<Coordinate> frontRDigFree(Pieces pieces, Piece
piece, int limit) {

        ArrayList<Coordinate> moves = new ArrayList<>();
        int factorV;
        int factorH;

        if (piece.getColour().equals(COLOUR.B)) {
           factorV = -1;
           factorH = -1;
        }
        else {
           factorV = 1;
           factorH = 1;
        }

        for (int advance = 1; advance <= limit; advance++) {
            int changeV = factorV*advance;
            int changeH = factorH*advance;
            Coordinate checkCoord = new Coordinate((char) (piece.getFile()
+changeV),piece.getRank()+changeH);
            if (Coordinate.inBoard(checkCoord)) {
                boolean occupiedTile = tileFull(pieces, checkCoord);
                if (occupiedTile && isNotTileColour(pieces, checkCoord,
piece.getColour())) {
                    moves.add(checkCoord);
                    return moves;
                }
                else if (occupiedTile)
                    return moves;
                else if (!tileFull(pieces, checkCoord))
                    moves.add(checkCoord);
            }
        }
        return moves;
    }

    // checks diagonal moves to the back and right

    public static ArrayList<Coordinate> backRDigFree(Pieces pieces, Piece
piece, int limit) {

        ArrayList<Coordinate> moves = new ArrayList<>();
        int factorV;
        int factorH;

        if (piece.getColour().equals(COLOUR.B)) {
            factorV = -1;
            factorH = 1;
        }
        else {
            factorV = 1;
            factorH = -1;
        }

        for (int advance = 1; advance <= limit; advance++) {
            int changeV = factorV*advance;
            int changeH = factorH*advance;
            Coordinate checkCoord = new Coordinate((char) (piece.getFile()
+changeV),piece.getRank()+changeH);
            if (Coordinate.inBoard(checkCoord)) {
                boolean occupiedTile = tileFull(pieces, checkCoord);
            if (occupiedTile && isNotTileColour(pieces, checkCoord, 
piece.getColour())) {
               moves.add(checkCoord);
               return moves;
            }
            else if (occupiedTile)
               return moves;
            else
               moves.add(checkCoord);
         }
      }
      return moves;
   }

   //____________________________________________________Diagonal Left 
Movement Methods___________________________________________________

   // checks diagonal moves to the back and left

   public static ArrayList<Coordinate> backLDigFree(Pieces pieces, Piece 
piece, int limit) {

      ArrayList<Coordinate> moves = new ArrayList<>();
      int factorV;
      int factorH;

      if (piece.getColour().equals(COLOUR.B)) {
         factorV = 1;
         factorH = 1;
      }
      else {
         factorV = -1;
         factorH = -1;
      }

      for (int advance = 1; advance <= limit; advance++) {
         int changeV = factorV*advance;
         int changeH = factorH*advance;
         Coordinate checkCoord = new Coordinate((char) (piece.getFile()
+changeV),piece.getRank()+changeH);
         if (Coordinate.inBoard(checkCoord)) {
            boolean occupiedTile = tileFull(pieces, checkCoord);
            if (occupiedTile && isNotTileColour(pieces, checkCoord, 
piece.getColour())) {
               moves.add(checkCoord);
```

```
                return moves;
            }
            else if (occupiedTile)
                return moves;
            else
                moves.add(checkCoord);
        }
    }
    return moves;
}

// checks diagonal moves to the front and left

public static ArrayList<Coordinate> frontLDigFree(Pieces pieces, Piece
piece, int limit) {

    ArrayList<Coordinate> moves = new ArrayList<>();
    int factorV;
    int factorH;

    if (piece.getColour().equals(COLOUR.B)) {
        factorV = 1;
        factorH = -1;
    }
    else {
        factorV = -1;
        factorH = 1;
    }

    for (int advance = 1; advance <= limit; advance++) {
        int changeV = factorV*advance;
        int changeH = factorH*advance;
        Coordinate checkCoord = new Coordinate((char) (piece.getFile()
+changeV),piece.getRank()+changeH);
        if (Coordinate.inBoard(checkCoord)) {
            boolean occupiedTile = tileFull(pieces, checkCoord);
            if (occupiedTile && isNotTileColour(pieces, checkCoord,
piece.getColour())) {
                moves.add(checkCoord);
                return moves;
            }
            else if (occupiedTile)
                return moves;
            else
                moves.add(checkCoord);
        }
    }
    return moves;
}

//________________________________________________Vertical Knight
Movement Methods________________________________________________

// checks Knight moves directly in front

public static ArrayList<Coordinate> frontKnight(Pieces pieces, Piece piece) {

    ArrayList<Coordinate> moves = new ArrayList<>();
    int factor;
    int sideDistance = 1;

    if (piece.getColour().equals(COLOUR.B)) {
        factor = -1;
    }
    else {
        factor = 1;
    }

    int changeV = 2*factor;
    int newRank = piece.getRank() + changeV;

    //right and left taken with reference to white pieces
    Coordinate frontRight = new Coordinate((char) (piece.getFile() +
sideDistance),newRank);
    Coordinate frontLeft = new Coordinate((char) (piece.getFile() -
sideDistance),newRank);

    if (Coordinate.inBoard(frontLeft)) {
        boolean occupiedTile = tileFull(pieces, frontLeft);
        if (!occupiedTile || isNotTileColour(pieces, frontLeft, piece.getColour()))
            moves.add(frontLeft);
    }

    if (Coordinate.inBoard(frontRight)) {
        boolean occupiedTile = tileFull(pieces, frontRight);
        if (!occupiedTile || isNotTileColour(pieces, frontRight, piece.getColour()))
            moves.add(frontRight);
    }
    return moves;
}
```

```
   // checks Knight moves directly behind

   public static ArrayList<Coordinate> backKnight(Pieces pieces, Piece piece) {

      ArrayList<Coordinate> moves = new ArrayList<>();
      int factor;
      int sideDistance = 1;

      if (piece.getColour().equals(COLOUR.B)) {
         factor = 1;
      }
      else {
         factor = -1;
      }

      int changeV = 2*factor;
      int newRank = piece.getRank() + changeV;

      //right and left taken with reference to white pieces
      Coordinate backRight = new Coordinate((char) (piece.getFile() +
sideDistance),newRank);
      Coordinate backLeft = new Coordinate((char) (piece.getFile() -
sideDistance),newRank);

      if (Coordinate.inBoard(backLeft)) {
         boolean occupiedTile = tileFull(pieces, backLeft);
         if (!occupiedTile || isNotTileColour(pieces, backLeft, piece.getColour()))
            moves.add(backLeft);
      }

      if (Coordinate.inBoard(backRight)) {
         boolean occupiedTile = tileFull(pieces, backRight);
         if (!occupiedTile || isNotTileColour(pieces, backRight, piece.getColour()))
            moves.add(backRight);
      }
      return moves;
   }

   //________________________________________________Horizontal Knight
Movement Methods________________________________________________

   // checks Knight moves directly to the right

   public static ArrayList<Coordinate> rightKnight(Pieces pieces, Piece piece) {
      ArrayList<Coordinate> moves = new ArrayList<>();
      int factor;
      int sideDistance = 1;

      if (piece.getColour().equals(COLOUR.B)) {
         factor = -1;
      }
      else {
         factor = 1;
      }

      int changeH = 2*factor;
      char newFile = (char) (piece.getFile() + changeH);

      //right and left taken with reference to white pieces
      Coordinate rightTop = new Coordinate(newFile,piece.getRank() +
sideDistance);
      Coordinate rightBottom = new Coordinate(newFile, piece.getRank() -
sideDistance);

      if (Coordinate.inBoard(rightTop)) {
         boolean occupiedTile = tileFull(pieces, rightTop);
         if (!occupiedTile || isNotTileColour(pieces, rightTop, piece.getColour()))
            moves.add(rightTop);
      }

      if (Coordinate.inBoard(rightBottom)) {
         boolean occupiedTile = tileFull(pieces, rightBottom);
         if (!occupiedTile || isNotTileColour(pieces, rightBottom,
piece.getColour()))
            moves.add(rightBottom);
      }
      return moves;
   }

   // checks Knight moves directly to the left

   public static ArrayList<Coordinate> leftKnight(Pieces pieces, Piece piece) {

      ArrayList<Coordinate> moves = new ArrayList<>();
      int factor;
      int sideDistance = 1;

      if (piece.getColour().equals(COLOUR.B)) {
         factor = 1;
      }
      else {
         factor = -1;
      }

      int changeH = 2*factor;
      char newFile = (char) (piece.getFile() + changeH);

      //right and left taken with reference to white pieces
      Coordinate leftTop = new Coordinate(newFile,piece.getRank() +
sideDistance);
      Coordinate leftBottom = new Coordinate(newFile, piece.getRank() -
sideDistance);

      if (Coordinate.inBoard(leftBottom)) {
         boolean occupiedTile = tileFull(pieces, leftBottom);
         if (!occupiedTile || isNotTileColour(pieces, leftBottom,
piece.getColour()))
            moves.add(leftBottom);
      }

      if (Coordinate.inBoard(leftTop)) {
         boolean occupiedTile = tileFull(pieces, leftTop);
         if (!occupiedTile || isNotTileColour(pieces, leftTop, piece.getColour()))
            moves.add(leftTop);
      }
      return moves;
   }

   //_______________________________________________Move Input
Methods_______________________________________________

   /**
    * Uses a scanner input to provide origin and destination coordinates for a
move in a TBIBoard.
    * It checks that the inputted String is in the correct format,
    * and uses a while loop to continue asking for input until a valid format is
provided
    * @param sc the scanner being used to get the input
    * @return a String array, containing two coordinates
    */

   public static String[] moveQuery(Scanner sc) {
      String[] move = new String[2];
      boolean validFormat = false;

      while (!validFormat) {
         System.out.println("Enter a piece and its destination. For example, to
move a piece in g1 to h3, write \"g1 h3\".");
         String userInput = sc.nextLine();

         if (userInput.length() != 5 || !userInput.contains(" ")) {
            System.out.println("Move instructions provided in an incorrect
format. Try again!");
         }
         else {
            move = userInput.split(" ");
            validFormat = true;
         }
      }
      return move;
   }

}
```

PAWN.java

```
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.Objects;
import java.util.Scanner;

/*
1) Class Constructors
2) Getters & Setters
3) Special Pawn Move Methods
4) Pawn Promotion Methods
5) Overridden Methods
*/

public class Pawn extends Piece {

    // whether the pawn has moved 2 squares forward
    private boolean hasMovedTwo = false;
```

```
   // whether the pawn can capture en passant to the left
   private boolean enPassantLeft = false;
   // whether the pawn can capture en passant to the right
   private boolean enPassantRight = false;
   // the coordinate previously occupied by the pawn
   private Coordinate previousCoordinate = new Coordinate();
   // the piece to which the pawn will be promoted
   private Piece promotedPiece;
   private ImageIcon icon;

   //________________________________________________Class
Constructors________________________________________________

   public Pawn(COLOUR colour, Coordinate OGcoord) {
      super(ID.PAWN, colour, OGcoord);
      if (getColour() == COLOUR.B)
         icon = new ImageIcon("BPawn.png");
      else if (getColour() == COLOUR.W)
         icon = new ImageIcon("WPawn.png");
   }

   public Pawn(Pawn original) {
      super(original);
   }

   //________________________________________________Getters &
Setters________________________________________________

   public void setPreviousCoordinate(Coordinate previousCoordinate) {
      this.previousCoordinate = previousCoordinate;
   }

   public Coordinate getPreviousCoordinate() {
      return previousCoordinate;
   }

   public void setHasMovedTwo() {
      this.hasMovedTwo = true;
   }

   public boolean getHasMovedTwo() {
      return hasMovedTwo;
   }

   public boolean getEnPassantLeft() {
        return enPassantLeft;
    }

    public boolean getEnPassantRight() {
        return enPassantRight;
    }

    //________________________________________________Special Pawn Move
Methods________________________________________________

    /**
     * Determines whether the pawn can eat to the left.
     * @param pieces the board being played in
     * @return it calculates the coordinate directly to the left front diagonal of the
pawn.
     * Returns true if the coordinate is occupied by a piece of the opposite colour
to the pawn moving.
     */

    private boolean canEatLeftDig(Pieces pieces) {

        int factorV;
        int factorH;

        if (getColour().equals(COLOUR.B)) {
            factorV = -1;
            factorH = 1;
        } else {
            factorV = 1;
            factorH = -1;
        }

        char newFile = (char) (getFile() + factorH);
        int newRank = getRank() + factorV;
        Coordinate leftDig = new Coordinate(newFile, newRank);

        return Move.tileFull(pieces, leftDig) && Move.isNotTileColour(pieces,
leftDig, getColour());
    }

    /**
     * Determines whether the pawn can eat to the right.
     * @param pieces the board being played in
     * @return it calculates the coordinate directly to the right front diagonal of
the pawn.
```

```
     * Returns true if the coordinate is occupied by a piece of the opposite colour
to the pawn moving.
     */

    private boolean canEatRightDig(Pieces pieces) {

        int factorV;
        int factorH;

        if (getColour().equals(COLOUR.B)) {
            factorV = -1;
            factorH = -1;
        } else {
            factorV = 1;
            factorH = 1;
        }

        char newFile = (char) (getFile() + factorH);
        int newRank = getRank() + factorV;
        Coordinate rightDig = new Coordinate(newFile, newRank);

        return Move.tileFull(pieces, rightDig) && Move.isNotTileColour(pieces,
rightDig, getColour());
    }

    /**
     * Determines how many squares a pawn can move forward
     * @param pieces the board being played in
     * @return an ArrayList containing the moves that a pawn can move forward.
     * 0 if frontFree has a size of 0
     * 1 if the pawn has the square in front of it free
     * 2 if the pawn is at its initial position & the squares in front of it are empty
     */

    private ArrayList<Coordinate> pawnForward(Pieces pieces) {

        ArrayList<Coordinate> potentialForward = Move.frontFree(pieces, this, 2);
        ArrayList<Coordinate> actualForward = new ArrayList<>();

        if (potentialForward.size() > 0) {

            Coordinate front1 = potentialForward.get(0);
            Coordinate front2 = Coordinate.emptyCoordinate;

            if (potentialForward.size() == 2) {
```

```
            front2 = potentialForward.get(1);
        }

        if (Move.tileFull(pieces, front1))
            return actualForward;
        else {
            actualForward.add(front1);
        }

        if (pieces.findPiece(this).equals(getOGcoord()) && front2 !=
Coordinate.emptyCoordinate) {
            if (Move.tileFull(pieces, front2))
                return actualForward;
            else
                actualForward.add(front2);
        }
    }
    return actualForward;
}

/**
 * Determines how many squares a pawn can capture en passant
 * @param pieces the board being played in
 * @return an ArrayList containing the moves that a pawn can move to in
order to capture en passant.
 * These are added if:
 * the piece directly to the left/right contains a pawn of the opposite colour
(called potentialPawn)
 * this potentialPawn has just moved 2 squares forward from its initial
position
 * If the en passant capture is possible, enPassantLeft or enPassantRight are
set to true
 */

public ArrayList <Coordinate> enPassant (Pieces pieces) {
    ArrayList<Coordinate> enPassantMoves = new ArrayList<>();
    ArrayList<Coordinate> left = Move.leftFree(pieces,this,1);
    ArrayList<Coordinate> right = Move.rightFree(pieces,this,1);

    if (left.size() == 1) {
        Coordinate leftTile = left.get(0);
        boolean correctPiece = Move.tileFull(pieces,leftTile)
                     && pieces.getPiece(leftTile).getName() == ID.PAWN
                     && pieces.getPiece(leftTile).getColour() ==
COLOUR.not(getColour());
```

```
          if (correctPiece) {
             Pawn potentialPawn = (Pawn) pieces.getPiece(leftTile);
             boolean correctPassantContext = potentialPawn.getHasMovedTwo()
                                    &&
potentialPawn.getPreviousCoordinate().equals(potentialPawn.getOGcoord())
                                    && !
potentialPawn.getPreviousCoordinate().equals(potentialPawn.getCoords());
             if (correctPassantContext) {
                enPassantMoves.addAll(Move.frontLDigFree(pieces, this, 1));
                enPassantLeft = true;
             }
          }
       }

       if (right.size() == 1) {
          Coordinate rightTile = right.get(0);
          boolean correctPiece = Move.tileFull(pieces,rightTile)
                && pieces.getPiece(rightTile).getName() == ID.PAWN
                && pieces.getPiece(rightTile).getColour() ==
COLOUR.not(getColour());
          if (correctPiece) {
             Pawn potentialPawn = (Pawn) pieces.getPiece(rightTile);
             boolean correctPassantContext = potentialPawn.getHasMovedTwo()
                   &&
potentialPawn.getPreviousCoordinate().equals(potentialPawn.getOGcoord())
                   && !
potentialPawn.getPreviousCoordinate().equals(potentialPawn.getCoords());
             if (correctPassantContext) {
                enPassantMoves.addAll(Move.frontRDigFree(pieces, this, 1));
                enPassantRight = true;
             }
          }
       }

       return enPassantMoves;
    }

    //_______________________________________________Pawn Promotion
Methods_______________________________________________

    /**
     * The promotion query for a TBIBoard to determine what the pawn will be
promoted to
     * @param promotionSquare the square to which the pawn moves to
     * @return a Piece (out of Queen, Rook, Bishop or Knight), based on the user
input.
     * Uses a while loop to make sure that a valid piece is provided.
     * Sets promotedPiece to the corect user input
     * @throws NullPointerException if the promotedPiece is not instantiated
     */

    public Piece promotionQuery(Coordinate promotionSquare) {

        Piece promotee = null;
        Scanner sc = new Scanner(System.in);
        boolean correctInput = false;

        System.out.print("Your pawn can be promoted. To what piece? ");

        while (!correctInput) {
            System.out.println("You can choose between: \n" +
                    "· Queen (Q) \n" +
                    "· Rook (R) \n" +
                    "· Bishop (B) \n" +
                    "· Knight (N)");
            String promoted = sc.next();
            switch (promoted) {
                case "Queen":
                case "Q":
                    promotee = new Queen(getColour(), promotionSquare);
                    correctInput = true;
                    break;
                case "Rook":
                case "R":
                    promotee = new Rook(getColour(), promotionSquare);
                    correctInput = true;
                    break;
                case "Bishop":
                case "B":
                    promotee = new Bishop(getColour(), promotionSquare);
                    correctInput = true;
                    break;
                case "Knight":
                case "N":
                    promotee = new Knight(getColour(), promotionSquare);
                    correctInput = true;
                    break;
                default:
                    System.out.println("Incorrect piece entered. Please try again");
            }
        }

        sc.close();

        Objects.requireNonNull(promotee);

        promotedPiece = promotee;

        return promotee;
    }

    /**
     * The promotion query for a GUIBoard to determine what the pawn will be
promoted to.
     * It creates a JOptionPane (showing an option dialog),
     * with JButtons corresponding to the pieces that the pawn can be promoted
to
     * Each has an ActionListener, which sets the promotedPiece based on which
JButton has been clicked
     * Clicking the OK button confirms the piece chosen
     * If no piece is chosen, a queen is chosen as default.
     * @param promotionSquare the square to which the pawn moves to
     */

    public void GUIPromotionQuery (Coordinate promotionSquare) {

        JButton queenOption;
        JButton rookOption;
        JButton bishopOption;
        JButton knightOption;
        ImageIcon icon;

        if (this.getColour() == COLOUR.B) {
            queenOption = new JButton(new ImageIcon("BQueen.png"));
            rookOption = new JButton(new ImageIcon("BRook.png"));
            bishopOption = new JButton(new ImageIcon("BBishop.png"));
            knightOption = new JButton(new ImageIcon("BKnight.png"));
            promotedPiece = new Queen(COLOUR.B,promotionSquare);
            icon = new ImageIcon("BPawn.png");
        }
        else {
            queenOption = new JButton(new ImageIcon("WQueen.png"));
            rookOption = new JButton(new ImageIcon("WRook.png"));
            bishopOption = new JButton(new ImageIcon("WBishop.png"));
            knightOption = new JButton(new ImageIcon("WKnight.png"));
            promotedPiece = new Queen(COLOUR.W,promotionSquare);
            icon = new ImageIcon("WPawn.png");
        }

        GUIBoard.formatInvisibleButton(queenOption);
        GUIBoard.formatInvisibleButton(rookOption);
        GUIBoard.formatInvisibleButton(bishopOption);
        GUIBoard.formatInvisibleButton(knightOption);

        Object[] options =
{"OK",knightOption,bishopOption,rookOption,queenOption};

        queenOption.addActionListener(actionEvent -> promotedPiece = new
Queen(getColour(),promotionSquare));

        rookOption.addActionListener(actionEvent -> promotedPiece = new
Rook(getColour(),promotionSquare));

        bishopOption.addActionListener(actionEvent -> promotedPiece = new
Bishop(getColour(),promotionSquare));

        knightOption.addActionListener(actionEvent -> promotedPiece = new
Knight(getColour(),promotionSquare));

        UIManager.put("OptionPane.background", GUIBoard.infoColour);
        UIManager.put("Panel.background", GUIBoard.infoColour);
        UIManager.put("OptionPane.messageForeground", Color.white);

        JOptionPane.showOptionDialog(null,
                "Choose a piece to promote. A queen is the default:",
                "Promotion Query",
                JOptionPane.DEFAULT_OPTION,
                JOptionPane.QUESTION_MESSAGE,
                icon,
                options,options[0]);
    }

    /**
     * Determines if a black pawn can promote
     * @param coordinate the coordinate to promote to
     * @return true if the pawn is black and its rank is the first rank (1)
     */

    public boolean canPromoteBlack (Coordinate coordinate) {
        return this.getColour() == COLOUR.B && coordinate.getRank() ==
BOARD.FIRST_RANK.getRankVal();
    }

    /**
     * Determines if a white pawn can promote
     * @param coordinate the coordinate to promote to
     * @return true if the pawn is white and its rank is the last rank (8)
     */

    public boolean canPromoteWhite (Coordinate coordinate) {
        return this.getColour() == COLOUR.W && coordinate.getRank() ==
BOARD.LAST_RANK.getRankVal();
    }

    public Piece getPromotedPiece() {
        return promotedPiece;
    }

    //_______________________________________________Overridden
Methods_______________________________________________

    @Override
    public Pawn makeCopy() {
        return new Pawn(this);
    }

    /**
     * Produces an ArrayList containing all the raw moves available to a Pawn
within a given board
     * @param pieces the board being played in
     * @return an ArrayList containing all the coordinates available to a Pawn,
based on:
     * can it move 0,1 or 2 squares forward (using pawnForward)
     * can it capture on a diagonal? (using canEatLeftDig and canEatRightDig)
     * can it capture en passant? (using enPassant)
     */

    @Override
    public ArrayList<Coordinate> getRawMoves(Pieces pieces) {

        ArrayList<Coordinate> pawnMoves = new ArrayList<>();

        if (canEatLeftDig(pieces))
            pawnMoves.addAll(Move.frontLDigFree(pieces, this, 1));
        pawnMoves.addAll(pawnForward(pieces));

        if (canEatRightDig(pieces))
            pawnMoves.addAll(Move.frontRDigFree(pieces, this, 1));

        pawnMoves.addAll(enPassant(pieces));

        return pawnMoves;
    }

    @Override
    public ImageIcon getImageIcon() {
        return icon;
    }


}
```

PIECES.java

```
import javax.swing.*;
import java.util.*;

/*
1) Class Constructors
2) Getters & Setters
3) Piece Movement Methods
4) toString() Methods
5) Overridden Methods
6) Abstract Methods
*/

/**
 * The Piece class is used to represent a generic chess piece. It acts as a super
class for the 6 distinct pieces from chess.
 * The Piece class is also used to determine all the moves that a given piece
can make within a given board.
 */

public abstract class Piece{
    /**
     * name is the ID of the piece. Determines whether it is a pawn, knight,
bishop, rook, queen or king.
     * colour is the COLOUR of the piece. Determines whether it is black or white.
     * coords is the current coordinate that a piece is occupying within the board
     * OGcoord is the initial coordinate that a piece occupies when a board is
created
     * pieceID is a unique String associated with each piece, created when a
Piece is constructed
     * potentialMoves refers to all the possible coordinates that a piece can go to,
given a certain board
     * dimension is the "size" of a board, which is 8 by default
     * single is the first rank, which is 1 by default
     * hasMoved determines whether a piece has moved at least once within a
given board
     * emptyPiece is a default piece, used as a filler
     */
    private final ID name;
    private final COLOUR colour;
    private Coordinate coords;
    private final Coordinate OGcoord;
    private final String pieceID; //potentially remove
    private HashSet<Coordinate> potentialMoves = new HashSet<>();
    public int dimension = BOARD.LAST_RANK.getRankVal();
    public int single = BOARD.FIRST_RANK.getRankVal();
    private boolean hasMoved = false;
    public static Piece emptyPiece = new
Rook(COLOUR.W,Coordinate.emptyCoordinate);

    //______________________________________________________Class
Constructors_______________________________________________________

    public Piece (ID name, COLOUR colour, Coordinate OGcoord) {

        Objects.requireNonNull(name, "The piece must be correctly identified with
an ID.");
        Objects.requireNonNull(colour, "The piece must be either white or black.");
        Objects.requireNonNull(OGcoord, "The piece must have an origin
coordinate to be correctly initiallised.");

        this.name = name;
        this.colour = colour;
        this.OGcoord = OGcoord;
        coords = OGcoord;
        pieceID = "*"+name.toString()+"*"+colour.toString()+"*"+OGcoord.getFile()
+"*";
    }

    public Piece (Piece original) {
        Objects.requireNonNull(original,"You can't copy a null piece");
        this.name = original.name;
        this.colour = original.colour;
        this.OGcoord = new Coordinate(original.OGcoord);
        this.coords = new Coordinate(original.coords);
        this.pieceID = original.pieceID;
        this.potentialMoves = new HashSet<>();

        for (Coordinate coord : original.getPotentialMoves()) {
            this.potentialMoves.add(new Coordinate(coord));
        }

        this.dimension = original.dimension;
        this.single = original.single;
        this.hasMoved = original.hasMoved;
    }

    //_______________________________________________Getters &
Setters_______________________________________________

    public Coordinate getCoords() {
        return coords;
    }

    public char getFile() {
        return getCoords().getFile();
    }

    public int getRank() {
        return getCoords().getRank();
    }

    public COLOUR getColour() {
        return colour;
    }

    public ID getName() {
        return name;
    }
    public Coordinate getOGcoord() {
        return OGcoord;
    }

    public String getPieceID() {
        return pieceID;
    }

    public void setCoords(Coordinate coords) {
        this.coords = coords;
    }

    public boolean getHasMoved() {return hasMoved;}

    public void setHasMoved() {hasMoved = true;}

    public void addMoves(ArrayList<Coordinate> someMoves) {
        potentialMoves.addAll(someMoves);
    }

    public void clearMoves() {
        potentialMoves.clear();
    }

    public HashSet<Coordinate> getPotentialMoves() {
        return potentialMoves;
    }

    //________________________________________________Piece Movement
Methods________________________________________________

    /**
     * Given a set of raw moves available to a piece, it removes those moves
(coordinates) that would lead to check,
     * and so, would be illegal moves.
     * It uses an iterator to iterate through all raw moves of a piece. Then, it
creates a temporary board (copying the current one),
     * and moves the piece to the raw move provided. If this causes check, then it
removes the coordinate from the raw moves.
     * The resulting ArrayList will be used to update the potential moves of the
piece.
     * @param pieces an instance of Pieces containing the information of the
current playing board
     * @return an ArrayList of coordinates containing all legal moves that can be
made by a Piece in a given board
```

```
     */

    public ArrayList<Coordinate> removeOwnCheck(Pieces pieces) {

       King potentialKing = null;
       boolean removeKingCastle = false;
       boolean removeQueenCastle = false;

       ArrayList<Coordinate> potentials = getRawMoves(pieces);

       if (potentials.size() == 0)
          return potentials;

       Iterator<Coordinate> it = potentials.iterator();

       while (it.hasNext()) {
          Coordinate nextMove = it.next();
          Pieces p = new Pieces(pieces);
          p.pieceMove(nextMove, this.makeCopy());
          Coordinate kingPosition = p.findKing(getColour());
          HashSet<Coordinate> dangerMoves =
p.allColouredRaws(COLOUR.not(getColour()));
          if (dangerMoves.contains(kingPosition))
             if (this.getName() == ID.KING) { // we need to remove the possibility of
castling if the King can be put in check
                potentialKing = (King) this;
                if (nextMove.equals(potentialKing.getTransitionCoordKingK())) {
                   it.remove();
                   removeKingCastle = true;
                }
                else if (nextMove.equals(potentialKing.getTransitionCoordKingQ()))
{
                   it.remove();
                   removeQueenCastle = true;
                }
                else
                   it.remove();
             }
             else
                it.remove();
       }
       if (potentialKing != null) {
          if (removeKingCastle)
             potentials.remove(potentialKing.getCastleCoordKingK());
          if (removeQueenCastle)
```

```
            potentials.remove(potentialKing.getCastleCoordKingQ());
        }

        return potentials;
    }

    /**
     * Updates the potential moves for a piece
     * @param pieces an instance of Pieces containing the information of the
current playing board
     */
    public void updatePotentialMoves(Pieces pieces)
{addMoves(removeOwnCheck(pieces));
    }

    /**
     * Determines whether a coordinate represents a valid move for the current
instance of a piece. It checks that
     * the coordinate provided is within the piece's potential moves, and
ascertains that the piece moving is of the colour
     * of the current turn of play.
     * @param destination a Coordinate representing a move destination
     * @param colour the colour corresponding to the side that is meant to play
     * @return true if it is the piece's current turn & the move is within it's
potential moves
     */
    public boolean isValidMove(Coordinate destination, COLOUR colour) {
        return getPotentialMoves().contains(destination) && getColour() == colour;
    }

    //______________________________________________toString()
Methods______________________________________________

    @Override
    public String toString() { //standard PNG String
        return name.toString() + coords.toString();
    }

    public String toBoardString() { // for String in the TBIBoard
        if (name == ID.PAWN)
            return "p" + colour.toSmallString();
        else
            return name.toString() + colour.toSmallString();
    }
    public String toFancyString() { // for String that clarifies the piece's position
        return name.toFullString() + " is at " + coords.toString();
    }

    //_______________________________________________Overridden
Methods_______________________________________________

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Piece piece = (Piece) o;
        return name == piece.name &&
                colour == piece.colour &&
                OGcoord.equals(piece.OGcoord) &&
                pieceID.equals(piece.pieceID);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, colour, OGcoord, pieceID);
    }


    //_______________________________________________Abstract
Methods_______________________________________________


    /**
     * Provides all the raw moves (all moves, without considering potential
cehcks) possible for a given piece within a certain board
     * The raw moves are calculated individually within a given piece
     * @param pieces the board being played in
     * @return an ArrayList containing all the raw move coordinates
     */

    public abstract ArrayList<Coordinate> getRawMoves(Pieces pieces);

    /**
     * @return the ImageIcon associated with a certain piece. This is assigned
when the Piece is instantiated.
     * Used to assign the icons for the GUIBoard
     */

    public abstract ImageIcon getImageIcon();
    /**
     * Creates an exact copy of the instance of the piece
     * This is used to create copies of HashMaps, and for calculating potential
moves
     * @return a copy of the piece instance
     */

    public abstract Piece makeCopy();
}
```

```
import java.util.*;

/*
1) Class Constructors
2) Collection Copying
3) Getters & Setters
4) Piece Related Methods
5) Board Distribution Related Methods
6) Game Logic Methods
7) Piece Movement Methods
8) Overridden Methods
*/

/**
 * The Pieces class is used to represent the board that is being played with.
 * It contains the position of all pieces, alongside all the methods that handle
game logic,
 * such as checking when a game ends or handling the movement of a piece.
 */

public class Pieces {

    /**
     * pieces is a HashMap which uses a Coordinate-Piece key-value pair,
     * with the Coordinate representing the current position of the Piece
     *
     * previousPieces is a HashMap which uses a Coordinate-Piece key-value
pair, used of a representation of the previous board
     * isCapture is a boolean value, used to determine whether a given move has
led to a capture
```

```
     * isGUIGame is a boolean value, used to determine whether the current
game being played is via a GUIBoard or a TBIBoard
     *
     * gameProgress is an ArrayList, containg HashMaps with Coordinate-Piece
key-value pairs,
     * used to store the board's development as a game is played.
     * This is used to determine whether there is a draw by threefold repetition
     */

    private HashMap<Coordinate, Piece> pieces;
    private HashMap<Coordinate, Piece> previousPieces;
    private boolean isCapture;
    private boolean isGUIGame;
    private ArrayList<HashMap<Coordinate,Piece>> gameProgress = new
ArrayList<>();

    //________________________________________________Class
Constructors________________________________________________

    public Pieces() {
        pieces = Boards.getChessBoard();
        previousPieces = copyHashMap(pieces);
        gameProgress.add(copyHashMap(pieces));
        updatePotentials(); // when we instantiate Pieces, we immediately calculte
the potential moves to begin the game
    }

    public Pieces(HashMap<Coordinate, Piece> newBoard) {
        pieces = newBoard;
        previousPieces = copyHashMap(pieces);
        gameProgress.add(copyHashMap(pieces));
        updatePotentials();
    }

    public Pieces (Pieces original) { // we create a copy constructor to help
calculate potential moves
        this.pieces = copyHashMap(original.getPieces());
        this.previousPieces = original.previousPieces;
        this.isCapture = original.isCapture;
        this.isGUIGame = original.isGUIGame;
        this.gameProgress = copyArrayHash(original.getGameProgress());
    }

    //________________________________________________Collection
Copying________________________________________________
```

```
    private HashMap<Coordinate, Piece> copyHashMap (HashMap<Coordinate,
Piece> original) { // creates a copy of a pieces HashMap

        HashMap<Coordinate, Piece> copyMap = new HashMap<>();
        for (Coordinate key : original.keySet()) {
            Coordinate newKey = new Coordinate(key);
            Piece newPiece = original.get(key).makeCopy();
            copyMap.put(newKey,newPiece);
        }

        return copyMap;
    }

    private ArrayList<HashMap<Coordinate,Piece>> copyArrayHash
(ArrayList<HashMap<Coordinate,Piece>> original) { // creates a copy of a
gameProgress ArrayList
        ArrayList<HashMap<Coordinate,Piece>> copyList = new ArrayList<>();
        for (HashMap<Coordinate,Piece> game : original) {
            copyList.add(copyHashMap(game));
        }

        return copyList;
    }

    //________________________________________________Getters &
Setters________________________________________________

    public HashMap<Coordinate, Piece> getPieces() {
        return pieces;
    }

    public void setPieces(HashMap<Coordinate,Piece> pieces) {this.pieces =
pieces;}

    public boolean getIsCapture() {
        return isCapture;
    }

    public void setIsCapture(boolean captureStatus) {
        this.isCapture = captureStatus;
    }

    public HashMap<Coordinate, Piece> getPreviousPieces() {
        return previousPieces;
    }

    public void setPreviousPieces(HashMap<Coordinate, Piece> previousPieces)
{
        this.previousPieces = copyHashMap(previousPieces);
    }

    public ArrayList<HashMap<Coordinate, Piece>> getGameProgress() {
        return gameProgress;
    }

    public void setGUIGame (boolean GUIStatus) {
        isGUIGame = GUIStatus;
    }

    //_______________________________________________Piece Related
Methods_______________________________________________

    /**
     * Adds a Coordinate-Piece key-value pair to the pieces HashMap (used when
handling a piece movement)
     * @param coordinate the destination coordinate for a given move
     * @param piece the piece making the move
     */

    public void addPiece(Coordinate coordinate, Piece piece) {
        pieces.put(coordinate,piece);
    }

    /**
     * Finds a given piece within the current instance of Pieces
     * @param piece the piece that is looked for
     * @return the Coordinate that a given piece occupies. If the piece isn't in the
board, returns the emptyCoordinate
     * @throws NullPointerException if the piece provided isn't instantiated
     */

    public Coordinate findPiece(Piece piece) {

        Objects.requireNonNull(piece, "Provide an existing piece. It can't be
null.");

        for (Coordinate key : pieces.keySet()) {
            if (pieces.get(key).equals(piece))
                return key;
        }
        System.err.println(piece.getName().toFullString() +" not found.");
        return Coordinate.emptyCoordinate;
    }

    /**
     * Finds a given king within the current instance of Pieces
     * @param colour the colour of the king that is being looked for
     * @return the Coordinate that the King occupies. If the King isn't in the
board, returns the emptyCoordinate
     */

    public Coordinate findKing(COLOUR colour) {
        for (Coordinate key : pieces.keySet()) {
            Piece potentialKing = pieces.get(key);
            if (potentialKing.getName().equals(ID.KING) &&
(potentialKing.getColour() == colour))
                return key;
        }
        String pieceNotInBoard = "King not found. Assuming it isn't in board.
Empty coordinate provided.";
        System.err.println(pieceNotInBoard);
        return Coordinate.emptyCoordinate;
    }

    /**
     * Returns a piece within the current instance of Pieces, given a Coordinate
     * @param coordinate the coordinate that is being looked for within the board
     * @return the Piece occupying the given coordinate. If there is no piece at
the coordinate, returns the emptyPiece
     * @throws NullPointerException if the Coordinate provided isn't instantiated
     */

    public Piece getPiece(Coordinate coordinate) {

        Objects.requireNonNull(coordinate, "Provide an existing coordinate. It
can't be null.");

        for (Coordinate key : pieces.keySet()) {
            if (key.equals(coordinate))
                return pieces.get(key);
        }
        System.err.println("There is no piece in this coordinate. Empty piece
provided.");
        return Piece.emptyPiece;
    }

    //_______________________________________________Board Distribution
Related Methods_______________________________________________

    /**
     * Provides a HashMap containing the coordinates and pieces of a given
colour
     * @param colour the colour of the pieces of interest
     * @return a HashMap of the form of pieces, containing only those pieces of
a given colour
     */

    public HashMap<Coordinate, Piece> getColourPieces(COLOUR colour) {
        HashMap<Coordinate,Piece> colours = new HashMap<>();
        for (Coordinate key : pieces.keySet()) {
            Piece piece = pieces.get(key);
            if (piece.getColour() == colour)
                colours.put(key,piece);
        }
        return colours;
    }

    /**
     * Provides a HashSet of all the potential moves of the pieces of a given
colour
     * @param colour the colour of the pieces of interest
     * @return a HashSet of Coordinates, containing all the coordinates that
pieces of a given colour can go to
     */

    public HashSet<Coordinate> allColouredPotentials (COLOUR colour) {
        HashSet<Coordinate> allMoves = new HashSet<>();
        HashMap<Coordinate, Piece> allColoured = getColourPieces(colour);
        for (Piece piece : allColoured.values()){
            allMoves.addAll(piece.getPotentialMoves());
        }
        return allMoves;
    }

    /**
     * Provides a HashSet of all the raw moves of the pieces of a given colour
     * @param colour the colour of the pieces of interest
     * @return a HashSet of Coordinates, containing all the raw coordinates that
pieces of a given colour can go to
```

```
     */

    public HashSet<Coordinate> allColouredRaws (COLOUR colour) {
        HashSet<Coordinate> allMoves = new HashSet<>();
        HashMap<Coordinate, Piece> allColoured = getColourPieces(colour);
        for (Piece piece : allColoured.values()){
            allMoves.addAll(piece.getRawMoves(this));
        }
        return allMoves;
    }

    /**
     * For all pawns in the current board, updates the previous coordinate that
they occupied after a move is made.
     * This is used to validate en passant captures.
     */

    public void updatePreviousMovePawns () {
        for (Piece potentialPawn : pieces.values()){
            if (potentialPawn.getName() == ID.PAWN) {
                Pawn pawn = (Pawn) potentialPawn;
                pawn.setPreviousCoordinate(pawn.getCoords());
            }
        }
    }

    /**
     * For a given piece, determines whether there is a piece in the same file of
the same type (i.e same ID)
     * @param piece the piece that is being considered
     * @return true if and only if there is a piece, of the same colour, of the same
type in the same file as the argument piece
     * within the given board
     */

    public boolean pieceInSameFile (Piece piece) {

        if (piece.getName() == ID.KING)
            return false;

        HashMap <Coordinate, Piece> coloured =
getColourPieces(piece.getColour());
        for (Piece value : coloured.values()) {
            if (value.getName() == piece.getName() && value.getFile() ==
piece.getFile() && !value.equals(piece))
```

```
            return true;
        }
        return false;
    }

    /**
     * For a given piece, determines whether there is a piece in the same rank of
the same type (i.e same ID)
     * @param piece the piece that is being considered
     * @return true if and only if there is a piece, of the same colour, of the same
type in the same rank as the argument piece
     * within the given board
     */

    public boolean pieceInSameRank (Piece piece) {

        if (piece.getName() == ID.KING)
            return false;

        HashMap <Coordinate, Piece> coloured =
getColourPieces(piece.getColour());
        for (Piece value : coloured.values()) {
            if (value.getName() == piece.getName() && value.getRank() ==
piece.getRank() && !value.equals(piece))
                return true;
        }
        return false;
    }

    /**
     * Determines whether there are 2 pieces of the same type within the board
that can go to the same coordinate
     * @param coordinate the coordinate that is being considered
     * @param piece the piece that is being considered
     * @return true if and only if there is another piece of the same type and
colour as the argument piece,
     * and if both can go to the given argument coordinate.
     * @throws AssertionError if the coordinate provided isn't within the
argument piece's potential moves
     */

    public boolean pieceToSameCoordinate (Coordinate coordinate, Piece piece)
{
        assert piece.getPotentialMoves().contains(coordinate);
        if (piece.getName() == ID.KING)
            return false;

        HashMap <Coordinate, Piece> coloured =
getColourPieces(piece.getColour());
        for (Piece value : coloured.values()) {
            if (value.getName() == piece.getName() &&
value.getPotentialMoves().contains(coordinate) && !value.equals(piece))
                return true;
        }
        return false;
    }

    //_______________________________________________Game Logic
Methods________________________________________________

    /**
     * Determines whether there is check on the board.
     * is in the potential moves of the pieces of the opposite colour
     * @param colour the colour that could be in check
     * @return true if the king of the given colour is in check.
     * Calculated by considering whether the coordinate of the king of a given
colour
     */

    public boolean isCheck(COLOUR colour) { //check against the given colour
        Coordinate kingPosition = findKing(colour);

        if (kingPosition.equals(Coordinate.emptyCoordinate))
            throw new IllegalArgumentException("There is no king in the board.
This is an illegal game!");

        HashSet<Coordinate> dangerMoves =
allColouredPotentials(COLOUR.not(colour));
        return (dangerMoves.contains(kingPosition));
    }

    /**
     * Determines whether there is mate on the board.
     * @param colour the colour that could be in mate
     * @return true if the King of the given colour is in check, and all the pieces
of that colour have no possible moves (no potential moves)
     */

    public boolean isMate(COLOUR colour) {
        HashSet<Coordinate> allMoves = allColouredPotentials(colour);
        return isCheck(colour) && (allMoves.size() == 0);
    }

    /**
     * Determines whether there is a draw on the board.
     * @return true if:
     * there are only 2 kings in the board
     * there are only two kings and a bishop/knight
     * there are only two kings and two bishops of opposite sides but same
diagonal colour (bishops can only move on black or white squares)
     * there is threefold repetition (the same position is repeated 3 times, at any
time during the game).
     * This is ascertained by looping through gameProgress, and checking
whether there are 3 equal HashMaps.
     */

    public boolean isDraw() {

        int n = gameProgress.size();

        boolean twoKings = !
findKing(COLOUR.B).equals(Coordinate.emptyCoordinate) && !
findKing(COLOUR.W).equals(Coordinate.emptyCoordinate);

        if (getPieces().size() == 2) // only 2 kings
            return twoKings;
        else if (getPieces().size() == 3) { // 2 kings and a bishop/knight
            int counter = 0;
            for (Piece piece : this.getPieces().values()) {
                if (piece.getName() == ID.BISHOP || piece.getName() == ID.KNIGHT)
                    counter++;
            }
            return twoKings && counter == 1;
        }
        else if (getPieces().size() == 4) { // 2 kings and 2 bishops with same
diagonal colour
            int counterB = 0;
            Bishop bishopB = null;
            int counterW = 0;
            Bishop bishopW = null;
            for (Piece piece : this.getPieces().values()) {
                if (piece.getName() == ID.BISHOP) {
                    if (piece.getColour() == COLOUR.B) {
                        bishopB = (Bishop) piece;
                        counterB++;
                    }
                    else {
                        bishopW = (Bishop) piece;
                        counterW++;
                    }
                }
            }

            boolean sameColourBishops = counterB == 1 &&
                                  counterW == 1 &&
                                  bishopB.getOGcoord().getFile() !=
bishopW.getOGcoord().getFile();

            return twoKings && sameColourBishops;
        }
        else if (n >= 3){ // threefold repetition
            for (HashMap<Coordinate, Piece> currentGame : gameProgress) {
                int counter = 0;
                for (HashMap<Coordinate, Piece> checkGame : gameProgress) {
                    if (currentGame.equals(checkGame)) {
                        counter++;
                    }
                }
                if (counter == 3)
                    return true;
            }

        }

        return false;

    }

    /**
     * Determines whether there is a stalemate on the board.
     * @param colour the colour of the turn in which a potential stalemating
move has been made
     * @return true if there isn't check in the board, but the pieces of the opposite
colour have no potential moves
     */

    public boolean isStalemate(COLOUR colour) {
        HashSet<Coordinate> allMoves =
allColouredPotentials(COLOUR.not(colour));
```

```
        return allMoves.size() == 0 && !isCheck(COLOUR.not(colour));

    }

    //______________________________________________Piece Movement
Methods______________________________________________

    /**
     * Given a destination coordinate and a piece, moves the piece to that
coordinate.
     * This is done by adding the piece and the coordinate to the pieces
HashMap.
     * It sets the new coordinates for the pieces, sets "hasMoved" to true, and
removes the previous coordinate of the piece
     * within the pieces HashMap.
     * @param coordinate the destination coordinate for a given move
     * @param piece the piece that is making the move
     */

    public void pieceMove (Coordinate coordinate, Piece piece) {
        Coordinate pieceCoord = findPiece(piece);
        addPiece(coordinate, piece);
        piece.setCoords(coordinate);
        piece.setHasMoved();
        pieces.remove(pieceCoord);
    }

    /**
     * Given a destination coordinate and a piece, changes the pieces HashMap
accordingly.
     * It sets the previous pieces and determines whether there has been a
capture.
     * After making the move, the potential moves of the new pieces are
recalculated,
     * and the given pieces HashMap are added to gameProgress.
     * For all pieces except the King and the Pawn, pieceMove is used.
     * For King, we consider whether castling is possible. If so, the king and
corresponding rook are moved.
     * For Pawn, we consider promotion (executing a promotion query) and en
passant capture.
     * @param coordinate the destination coordinate for a given move
     * @param piece the piece that is making the move
     */

    public void makeMove (Coordinate coordinate, Piece piece) {
        if (piece.isValidMove(coordinate, piece.getColour())) {
            setPreviousPieces(this.getPieces());
            isCapture = Move.tileFull(this, coordinate) &&
Move.isNotTileColour(this,coordinate, piece.getColour());
            if (piece.getName() == ID.KING) {
                King castleKing = (King) piece;
                if (castleKing.canCastleQueen(this) &&
coordinate.equals(castleKing.getCastleCoordKingQ()) && !
isCheck(castleKing.getColour())) {
                    assert !
findPiece(castleKing.getRookQueen()).equals(Coordinate.emptyCoordinate);
                    pieceMove(coordinate,castleKing);

pieceMove(castleKing.getRookQueen().getCastleCoordRook(),castleKing.getRo
okQueen());
                }
                else if (castleKing.canCastleKing(this) &&
coordinate.equals(castleKing.getCastleCoordKingK()) && !
isCheck(castleKing.getColour())) {
                    assert !
findPiece(castleKing.getRookKing()).equals(Coordinate.emptyCoordinate);
                    pieceMove(coordinate,castleKing);

pieceMove(castleKing.getRookKing().getCastleCoordRook(),castleKing.getRoo
kKing());
                }
                else {
                    pieceMove(coordinate, castleKing);
                }
            }
            else if (piece.getName() == ID.PAWN) {
                Pawn pawn = (Pawn) piece;

                updatePreviousMovePawns();
                if (Math.abs(coordinate.getRank() - pawn.getRank()) == 2)
                    pawn.setHasMovedTwo();

                if (pawn.canPromoteBlack(coordinate) ||
pawn.canPromoteWhite(coordinate)) {
                    Piece toPromote;

                    if (isGUIGame) {
                        pawn.GUIPromotionQuery(coordinate);
                        toPromote = pawn.getPromotedPiece();
                    }
                    else {
                        toPromote = pawn.promotionQuery(coordinate);
                    }
                    Coordinate pieceCoord = findPiece(piece);
                    addPiece(coordinate, toPromote);
                    pieces.remove(pieceCoord);
                }
                else if (pawn.getEnPassantLeft()) {
                    Coordinate left = Move.leftFree(this,pawn,1).get(0);
                    setIsCapture(true);
                    pieces.remove(left);
                    pieceMove(coordinate,pawn);
                }
                else if (pawn.getEnPassantRight()) {
                    Coordinate right = Move.rightFree(this,pawn,1).get(0);
                    setIsCapture(true);
                    pieces.remove(right);
                    pieceMove(coordinate,pawn);
                }
                else {
                    pieceMove(coordinate, pawn);
                }
            }
            else {
                pieceMove(coordinate, piece);
            }
        }
        else
            System.err.println(piece.getName().toFullString() + " to " +
coordinate.toString() + " is an invalid move.");

        gameProgress.add(copyHashMap(pieces));
        updatePotentials();

}

/**
 * Given a new pieces HashMap, recalculates all potential moves of all the
pieces in the HashMap
 */

public void updatePotentials() {

        for (Piece value : pieces.values()) {
            value.clearMoves();
            value.updatePotentialMoves(this);
        }
    }

    //_______________________________________________Overridden
Methods_______________________________________________

    @Override
    public String toString() {
        StringBuilder str = new StringBuilder();

        pieces.forEach((coord, piece) -> str.append(piece.getPieceID())
                        .append(" is at ")
                        .append(coord.toString())
                        .append("\n"));

        return str.toString();
        }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Pieces pieces1 = (Pieces) o;
        return Objects.equals(pieces, pieces1.pieces);
    }

    @Override
    public int hashCode() {
        return Objects.hash(pieces);
    }
}
```

QUEEN.java

```
import javax.swing.*;
import java.util.ArrayList;

/*
1) Class Constructors
```

```
2) Overridden Methods
*/

public class Queen extends Piece{

   private ImageIcon icon;

   //________________________________________________Class
Constructors________________________________________________

   public Queen(COLOUR colour, Coordinate OGcoord) {
      super(ID.QUEEN, colour, OGcoord);
      if (getColour() == COLOUR.B)
         icon = new ImageIcon("BQueen.png");
      else if (getColour() == COLOUR.W)
         icon = new ImageIcon("WQueen.png");
   }

   public Queen(Queen original) {
      super(original);
   }

   //________________________________________________Overridden
Methods________________________________________________

   @Override
   public Queen makeCopy() {
      return new Queen(this);
   }

   /**
    * Produces an ArrayList containing all the raw moves available to a Queen
within a given board
    * @param pieces the board being played in
    * @return an ArrayList containing all the coordinates produced from the
Move class
    * (all the diagonals, all verticals and all horizontals)
    */

   @Override
   public ArrayList<Coordinate> getRawMoves(Pieces pieces) {

      ArrayList<Coordinate> front = Move.frontFree(pieces,this,dimension);
      ArrayList<Coordinate> right = Move.rightFree(pieces,this,dimension);
      ArrayList<Coordinate> back = Move.backFree(pieces,this,dimension);
```

```
        ArrayList<Coordinate> left = Move.leftFree(pieces,this,dimension);
        ArrayList<Coordinate> frontRDig = Move.frontRDigFree(pieces,
this,dimension);
        ArrayList<Coordinate> backRDig = Move.backRDigFree(pieces, this,
dimension);
        ArrayList<Coordinate> backLDig = Move.backLDigFree(pieces,
this,dimension);
        ArrayList<Coordinate> frontLDig = Move.frontLDigFree(pieces, this,
dimension);

        front.addAll(right);
        back.addAll(left);
        front.addAll(back);

        frontRDig.addAll(backRDig);
        backLDig.addAll(frontLDig);
        frontRDig.addAll(backLDig);

        front.addAll(frontRDig);

        return front;

    }

    @Override
    public ImageIcon getImageIcon() {
        return icon;
    }

}
```

ROOK.java

```
import javax.swing.*;
import java.util.ArrayList;

/*
1) Class Constructors
2) Getters & Setters
3) Overridden Methods
*/
```

```
public class Rook extends Piece{

   private Coordinate castleCoordRook;

   private ImageIcon icon;

   //________________________________________________________Class
Constructors________________________________________________________

   public Rook(COLOUR colour, Coordinate OGcoord) {
      super(ID.ROOK, colour, OGcoord);
      if (getColour() == COLOUR.B)
         icon = new ImageIcon("BRook.png");
      else if (getColour() == COLOUR.W)
         icon = new ImageIcon("WRook.png");
   }

   public Rook (Rook original) {
      super(original);
   }

   //________________________________________________________Getters &
Setters________________________________________________________

   public Coordinate getCastleCoordRook() {
      return castleCoordRook;
   }

   public void setCastleCoordRook(Coordinate castleCoordRook) {
      this.castleCoordRook = castleCoordRook;
   }

   //________________________________________________________Overridden
Methods________________________________________________________

   @Override
   public Rook makeCopy() {
      return new Rook(this);
   }

   /**
    * Produces an ArrayList containing all the raw moves available to a Rook
within a given board
    * @param pieces the board being played in
```

```
     * @return an ArrayList containing all the coordinates produced from the
Move class (all the verticals and all the horizontals)
     */

    @Override
    public ArrayList<Coordinate> getRawMoves(Pieces pieces) {
        ArrayList<Coordinate> front = Move.frontFree(pieces,this,dimension);
        ArrayList<Coordinate> right = Move.rightFree(pieces,this,dimension);
        ArrayList<Coordinate> back = Move.backFree(pieces,this,dimension);
        ArrayList<Coordinate> left = Move.leftFree(pieces,this,dimension);

        front.addAll(right);
        back.addAll(left);
        front.addAll(back);

        return front;
    }

    @Override
    public ImageIcon getImageIcon() {
        return icon;
    }


}
```

TBI.java

```
import java.nio.file.Path;
import java.util.Scanner;

public class TBIBoard {

    public static void gameLoop(Pieces pieces) {

        boolean exit = false;
        COLOUR turn = COLOUR.W;
        Scanner sc = new Scanner(System.in);
        StringBuilder str = new StringBuilder();
        int numberOfTurns = 0;
    while (!exit) {

        String[] move = Move.moveQuery(sc);

        if (!Coordinate.inBoard(new Coordinate(move[0])) || !
Coordinate.inBoard(new Coordinate(move[1]))) {
            System.out.println("At least one of the given coordinates isn't in the
board. Please try again!");
        }
        else {
            Coordinate origin = new Coordinate(move[0]);
            Coordinate destination = new Coordinate(move[1]);

            Piece piece = pieces.getPiece(origin);

            if (piece.equals(Piece.emptyPiece)) {
                System.out.println("The origin coordinate doesn't contain a piece.
Please try again!");
            } else {
                if (piece.isValidMove(destination, turn)) {
                    pieces.makeMove(destination, piece);
                    if (turn == COLOUR.W) {
                        numberOfTurns++;
                        str.append(numberOfTurns).append(".
").append(ChessIO.moveString(pieces, destination, piece)).append(" ");
                    }
                    else

str.append(ChessIO.moveString(pieces,destination,piece)).append(" ");
                    System.out.println(Boards.displayBoard(pieces));
                    if (pieces.isMate(COLOUR.not(turn))) {
                        System.out.println(turn.toString() + " win.");
                        exit = true;
                    }
                    else if (pieces.isStalemate(turn)) {
                        System.out.println("It's a draw by stalemate.");
                        exit = true;
                    }
                    else if (pieces.isDraw()){
                        System.out.println("It's a draw.");
                        exit = true;
                    }
                    else{
                            System.out.println("Enter \"exit\" to end the game, or \"save\"
to save the current state of the game.");
                            String input = sc.nextLine();
                            switch (input) {
                                case "exit":
                                    exit = true;
                                    break;
                                case "save":
                                    Path filePath = ChessIO.fileQuery(sc);
                                    if (ChessIO.saveGame(str.toString(), filePath))
                                        System.out.println("Game saved succesfuly on path " +
filePath.toString());
                                    else
                                        System.out.println("There was an error saving the file
on the path " + filePath.toString());
                                    break;
                                default:
                                    break;
                            }
                            if (!exit) {
                                turn = COLOUR.not(turn);
                                System.out.println(turn.toString() + " to move.");
                            }
                        }
                    } else {
                        System.out.println("Invalid move provided.");
                    }
                }
            }
        }
    }

    public static void main(String[] args) {
        Pieces pieces = new Pieces();
        pieces.setGUIGame(false);
        gameLoop(pieces);
    }

}
```