SimpleChess is an application that allows one or two players to compete in a chess game. Some features of the app include being able to play a chess CPU and additionally, you can battle a more aggressive CPU. A working leaderboard to track wins when you play and also the game allows you to see what moves are available the game has a check system and game-ending conditions. Clean and understandable UI also makes this app *simple*.

Now let's go over some documentation for some important bits of code.

AggroChessBot

```
/**
 * This class represents an aggressive chess bot that extends the abstract ChessBot class.
 * It evaluates the possible moves for its pieces based on a score that is assigned to each move.
 * The move with the highest score is chosen randomly among the highest-scoring moves.
 */
public class AggroChessBot extends ChessBot {

	/**
	* Constructs an instance of the AggroChessBot class by calling the constructor of its superclass.
	*/
	public AggroChessBot(){
	super();
	}

	/**
	* Overrides the make_move method of the ChessBot class.
	* This method evaluates all possible moves for the AggroChessBot's pieces and assigns a score to each move.
	* The move with the highest score is chosen randomly among the highest-scoring moves.
	*
	* @param board a 2D array of Piece objects representing the current state of the chess board.
	* @param kingPos a Coordinates object representing the position of the king on the chess board.
	* @return a List of Coordinates objects representing the start and end positions of the chosen move.
	*/
```

```
    @Override
    public List<Coordinates> make_move(Piece[][] board, Coordinates kingPos) {
    // implementation code here
    }

    /**
    * This method calculates a score for a given move based on the type of the piece being moved.
    * Pawns are worth 1 point, knights and bishops are worth 3 points, rooks are worth 5 points, and queens are worth 9 points.
    *
    * @param board a 2D array of Piece objects representing the current state of the chess board.
    * @param c a Coordinates object representing the position of the piece on the chess board.
    * @return an integer score based on the type of the piece being moved.
    */
    public int get_score(Piece[][] board, Coordinates c){
    // implementation code here
    }

}
```

Bishop

```
/**

    Bishop class, representing a Bishop chess piece.

    Inherits from Piece class. */ package com.example.group12project.ChessComponents;

import java.util.*;

public class Bishop extends Piece {

/**
 * Constructor for Bishop class.
 * @param p The player (white or black) that the Bishop belongs to.
 */
public Bishop(String p){
```

```
    super(p);
    this.type = "B";
}

/**
 * Determines if a Bishop can move from a starting position to an end position on the chess
board.
 * @param board The chess board represented by a 2D array of Pieces.
 * @param start The starting position of the Bishop.
 * @param end The ending position of the Bishop.
 * @param kingPos The current position of the King of the same player as the Bishop.
 * @return true if the move is valid, false otherwise.
 */
public boolean can_move(Piece[][] board, Coordinates start, Coordinates end, Coordinates
kingPos){
    Piece s = board[start.X()][start.Y()];      // get start piece
    Piece e = board[end.X()][end.Y()];          // if end isn't null, get piece

    if(s==null){    // if there is no piece
        return false;
    }

    if(e != null && (Objects.equals(s.get_player(), e.get_player()))){
        // if end piece is same as start piece (can't take own piece)
        return false;
    }

    int xdiff = Math.abs(end.X() - start.X());
    int ydiff = Math.abs(end.Y() - start.Y());
    // check for diagonal movement or no movement
    if((xdiff != ydiff)){
        return false;
    }

    // check for preconditions + pinning
    if(!super.can_move(board, start, end, kingPos)){
        return false;
    }

    // check for obstruction
    // 1. get list of all spaces between start and end
    // 2. check each to see if there is a piece, if so, cannot move to that spot
    List<Coordinates> between = Coordinates.places_between(start, end);
    for(Coordinates i : between){
```

```
        if(board[i.X()][i.Y()] != null){
            return false;
        }
    }
    return true;
}

/**
 * Determines if a Bishop can move from a starting position to an end position on the chess
board.
 * @param board The chess board represented by a 2D array of Pieces.
 * @param start The starting position of the Bishop.
 * @param end The ending position of the Bishop.
 * @return true if the move is valid, false otherwise.
 */
public boolean can_move(Piece[][] board, Coordinates start, Coordinates end){
    if(!super.can_move(board, start, end)){
        return false;
    }

    int xdiff = Math.abs(end.X() - start.X());
    int ydiff = Math.abs(end.Y() - start.Y());

    // check for diagonal movement or no movement
    if(xdiff != ydiff){
        return false;
    }

    // check for obstruction
    // 1. get list of all spaces between start and end
    // 2. check each to see if there is a piece, if so, cannot move to that spot
    List<Coordinates> between = Coordinates.places_between(start, end);
    for(Coordinates i : between){
        if(board[i.X()][i.Y()] != null){
            return false;
        }
    }
    return true;
}

/**
```

Generates a list of all possible moves for the bishop on the board, given the current board state and the bishop's starting position.

@param board the current board state

@param start the starting position of the bishop

@param kingPos the position of the king of the same color as the bishop (used for checking for checks/checkmates)

@return a list of all possible moves for the bishop */

public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start, Coordinates kingPos) { List<Coordinates> poss = new ArrayList<>(); for(int i = 0; i < 8; i++){ for(int j = 0; j < 8; j++){ Coordinates temp = new Coordinates(i,j); if(can_move(board, start, temp, kingPos)){ poss.add(temp); } } } return poss; }

/**

Generates a list of all possible moves for the bishop on the board, given the current board state and the bishop's starting position.

This method assumes that the king position is not provided.

@param board the current board state

@param start the starting position of the bishop

@return a list of all possible moves for the bishop */ public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start) { List<Coordinates> poss = new ArrayList<>(); for(int i = 0; i < 8; i++){ for(int j = 0; j < 8; j++){ Coordinates temp = new Coordinates(i,j); if(can_move(board, start, temp)){ poss.add(temp); } } } return poss; }

}

ChessBot

/**
 * The abstract ChessBot class represents a player in a chess game.

```
 * It contains a player field indicating which color the bot is playing as,
 * and an abstract method make_move() that takes a game board and king position
 * as input and returns a list of coordinates representing the bot's move.
 */
public abstract class ChessBot {

	/**
	* The player field indicates which color the bot is playing as.
	*/
	String player;

	/**
	* The constructor sets the player field to "W" (white).
	*/
	public ChessBot() {
	player = "W";
	}

	/**
	* The make_move() method is an abstract method that takes a 2D array of
	* Piece objects representing the current state of the game board, and
	* a Coordinates object representing the position of the bot's king on the board.
	* The method returns a List of Coordinates objects representing the bot's move.
	*
	* @param board The 2D array of Piece objects representing the current state of the
game board.
	* @param kingPos The Coordinates object representing the position of the bot's king on
the board.
	* @return A List of Coordinates objects representing the bot's move.
	*/
	public abstract List<Coordinates> make_move(Piece[][] board, Coordinates kingPos);

}
```

Coordinates

```
/**
```

A class representing coordinates on a chess board. */ package com.example.group12project.ChessComponents;

import androidx.annotation.IdRes; import com.example.group12project.R;

import java.util.ArrayList; import java.util.List;

public class Coordinates {

private final int x;
private final int y;

```
/**
 * Constructor for creating a new instance of the Coordinates class.
 * @param x The x coordinate of the position.
 * @param y The y coordinate of the position.
 */
public Coordinates(int x, int y){
        this.x = x;
        this.y = y;
}

/**
 * Returns the x coordinate of the position.
 * @return The x coordinate of the position.
 */
public int X(){
        return x;
}

/**
 * Returns the y coordinate of the position.
 * @return The y coordinate of the position.
 */
public int Y(){
        return y;
}

/**
 * Returns the position of the given view ID.
 * @param n The ID of the view representing the position.
 * @return The Coordinates object representing the position of the view.
 */
public static Coordinates get_pos(@IdRes int n){
        Coordinates pos = null;
```

```
        switch(n){
        case(R.id.i00):{ pos = new Coordinates(0, 0);break;}
        case(R.id.i01):{ pos = new Coordinates(0, 1);break;}
        case(R.id.i02):{ pos = new Coordinates(0, 2);break;}
        case(R.id.i03):{ pos = new Coordinates(0, 3);break;}
        case(R.id.i04):{ pos = new Coordinates(0, 4);break;}
        case(R.id.i05):{ pos = new Coordinates(0, 5);break;}
        case(R.id.i06):{ pos = new Coordinates(0, 6);break;}
        case(R.id.i07):{ pos = new Coordinates(0, 7);break;}
        case(R.id.i10):{ pos = new Coordinates(1, 0);break;}
        case(R.id.i11):{ pos = new Coordinates(1, 1);break;}
        case(R.id.i12):{ pos = new Coordinates(1, 2);break;}
        case(R.id.i13):{ pos = new Coordinates(1, 3);break;}
        case(R.id.i14):{ pos = new Coordinates(1, 4);break;}
        case(R.id.i15):{ pos = new Coordinates(1, 5);break;}
        case(R.id.i16):{ pos = new Coordinates(1, 6);break;}
        case(R.id.i17):{ pos = new Coordinates(1, 7);break;}
        case(R.id.i20):{ pos = new Coordinates(2, 0);break;}
        case(R.id.i21):{ pos = new Coordinates(2, 1);break;}
        case(R.id.i22):{ pos = new Coordinates(2, 2);break;}
        case(R.id.i23):{ pos = new Coordinates(2, 3);break;}
        case(R.id.i24):{ pos = new Coordinates(2, 4);break;}
        case(R.id.i25):{ pos = new Coordinates(2, 5);break;}
        case(R.id.i26):{ pos = new Coordinates(2, 6);break;}
        case(R.id.i27):{ pos = new Coordinates(2, 7);break;}
        case(R.id.i30):{ pos = new Coordinates(3, 0);break;}
        ….
/**
 * Prints the coordinates of this instance in the format [x, y].
 */
public void display_coord(){
        System.out.println("[ " + X() + " , " + Y() + " ]");
}

/**
 * Returns a list of Coordinates that represent the spaces between the given start and end
Coordinates.
 *
 * @param start the starting Coordinate
 * @param end the ending Coordinate
 * @return a list of Coordinates that represent the spaces between the given start and end
Coordinates
 */
public static List<Coordinates> places_between(Coordinates start, Coordinates end){
```

```
List<Coordinates> spaces = new ArrayList<>();
int xdiff = end.X() - start.X();
int ydiff = end.Y() - start.Y();

if(xdiff == 0 && ydiff > 0){       //checking for positive y movement
for(int i = 1; i < ydiff; i++){
spaces.add(new Coordinates(start.X(), start.Y()+i));
}
}
else if(xdiff == 0 && ydiff < 0){   //checking for negative y movement
for(int i = 1; i < Math.abs(ydiff); i++){
spaces.add(new Coordinates(start.X(), start.Y()-i));
}
}
else if(ydiff == 0 && xdiff > 0){          //checking for positive x movement
for(int i = 1; i < xdiff; i++){
spaces.add(new Coordinates(start.X()+i, start.Y()));
}
}
else if(ydiff == 0 && xdiff < 0){          //checking for negative x movement
for(int i = 1; i < Math.abs(xdiff); i++){
spaces.add(new Coordinates(start.X()-i, start.Y()));
}
}
else {
boolean b = Math.abs(xdiff) == Math.abs(ydiff);
if(ydiff > 0 && xdiff > 0 && b){   //checking for bottom right movement
for(int i = 1; i < Math.abs(xdiff); i++){
        spaces.add(new Coordinates(start.X()+i, start.Y()+i));
}
}
else if(ydiff < 0 && xdiff > 0 && b){     //checking for bottom left movement
for(int i = 1; i < Math.abs(xdiff); i++){
        spaces.add(new Coordinates(start.X()+i, start.Y()-i));
}
}
else if(ydiff > 0 && xdiff < 0 && b){     //checking for top right movement
for(int i = 1; i < Math.abs(xdiff); i++){
        spaces.add(new Coordinates(start.X()-i, start.Y()+i));
}
}
else if(ydiff < 0 && xdiff < 0 && b){     //checking for top left movement
for(int i = 1; i < Math.abs(xdiff); i++){
        spaces.add(new Coordinates(start.X()-i, start.Y()-i));
```

```
            }
            }
            }

            return spaces;
}

/**
 * Checks whether this instance is equal to the given Coordinate.
 *
 * @param other the Coordinate to compare with
 * @return true if this instance is equal to the given Coordinate, false otherwise
 */
public boolean equals(Coordinates other){
            return (this.X() == other.X()) && (this.Y() == other.Y());
}

/**
 * Checks whether this instance represents a white square on a chessboard.
 *
 * @return true if this instance represents a white square, false otherwise
 */
public boolean isWhite(){
…
```

King

```
/**
 * Represents a King piece in chess.
 */
public class King extends Piece {

    /**
     * Indicates whether the king has moved or not.
     */
    public boolean hasMoved = false;

    /**
     * Constructs a new King piece with the specified player color.
     * @param p the player color of the piece
     */
    public King(String p){
        super(p);
```

```
        this.type = "K";
    }

    /**
     * Determines if the piece can move from the start to the end coordinates.
     * @param board the current chess board state
     * @param start the starting position of the piece
     * @param end the ending position of the piece
     * @param kingPos the current position of the king
     * @return true if the move is valid, false otherwise
     */
    public boolean can_move(Piece[][] board, Coordinates start, Coordinates end, Coordinates
kingPos) {
        if (!super.can_move(board, start, end, kingPos)) {
            return false;
        }

        int xdiff = Math.abs(end.X() - start.X());
        int ydiff = Math.abs(end.Y() - start.Y());

        // cannot move more than one square unless castling
        if (xdiff > 1 || ydiff > 1) {
            // add castling logic
            return false;
        }

        return !this.kingInCheck(board, end);
    }

    /**
     * Determines if the piece can move from the start to the end coordinates.
     * @param board the current chess board state
     * @param start the starting position of the piece
     * @param end the ending position of the piece
     * @return true if the move is valid, false otherwise
     */
    public boolean can_move(Piece[][] board, Coordinates start, Coordinates end) {
        if (!super.can_move(board, start, end)) {
            return false;
        }

        int xdiff = Math.abs(end.X() - start.X());
        int ydiff = Math.abs(end.Y() - start.Y());
```

```
        // cannot move more than one square unless castling
        if (xdiff > 1 || ydiff > 1) {
            // add castling logic
            return false;
        }

        return !this.kingInCheck(board, end);
    }

    /**
     * Returns a list of all possible moves for the piece on the board.
     * @param board the current chess board state
     * @param start the starting position of the piece
     * @param kingPos the current position of the king
     * @return a list of all possible moves for the piece on the board
     */
    public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start, Coordinates
kingPos) {
        List<Coordinates> poss = new ArrayList<>();
        for(int i = 0; i < 8; i++){
            for(int j = 0; j < 8; j++){
                Coordinates temp = new Coordinates(i,j);
                if(can_move(board, start, temp, kingPos)){
                    poss.add(temp);
                }
            }
        }
        return poss;
    }

    /**
     * Returns a list of all possible moves for the piece on the board.
     * @param board the current chess board state
     * @param start the starting position of the piece
     * @return a list of all possible moves for the piece on the board
     */
/**
     * Returns a list of all possible moves for a given piece on the board.
     * @param board the current state of the chess board
     * @param start the coordinates of the piece to be moved
     * @return a list of all possible coordinates where the piece can move
     */
    public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start) {
        // method body
```

```
    }

    /**
     * Checks if the given king is in check by any of the opponent's pieces.
     * @param chessboard the current state of the chess board
     * @param king_square the coordinates of the king to be checked
     * @return true if the king is in check, false otherwise
     */
    public boolean kingInCheck(Piece[][] chessboard, Coordinates king_square){
        // method body
    }

    /**
     * Returns the coordinates of the piece that has put the given king in check.
     * @param chessboard the current state of the chess board
     * @param king_square the coordinates of the king to be checked
     * @return the coordinates of the piece that has put the king in check, null if no such piece
exists
     */
    public Coordinates checkByWho(Piece[][] chessboard, Coordinates king_square){
        // method body
    }
}
```

Knight

```
/**
 * Represents a Knight chess piece that extends the abstract Piece class.
 * This class contains methods to check if the piece can move to a given coordinate and
 * to retrieve a list of all possible moves for the piece on the current board.
 */
public class Knight extends Piece {

        /**
        * Constructor for a Knight object that takes a player string as parameter.
        * Sets the type of the piece to "N".
        *
        * @param p A string representing the player that owns the piece ("white" or "black").
        */
        public Knight(String p){
        super(p);
```

```
        this.type = "N";
        }

        /**
        * Checks if the Knight can move from its starting position to a given end coordinate.
        * Overrides the can_move method in the Piece class.
        *
        * @param board The current game board as a 2D Piece array.
        * @param start The starting coordinate of the piece on the board.
        * @param end The end coordinate that the piece wants to move to.
        * @param kingPos The position of the player's king piece.
        * @return A boolean indicating if the move is valid or not.
        */
        @Override
        public boolean can_move(Piece[][] board, Coordinates start, Coordinates end,
Coordinates kingPos){
        Piece s = board[start.X()][start.Y()];   // get start piece
        Piece e = board[end.X()][end.Y()];               // if end isn't null, get piece

        if(s==null){      // if there is no piece
        return false;
        }

        if(e != null && (Objects.equals(s.get_player(), e.get_player()))){
        // if end piece is same as start piece (can't take own piece)
        return false;
        }

        int xdiff = Math.abs(end.X() - start.X());
        int ydiff = Math.abs(end.Y() - start.Y());

        if(!((xdiff == 2 && ydiff == 1) || (xdiff == 1 && ydiff == 2))){
        return false;
        }

        return super.can_move(board, start, end, kingPos);
        }

        /**
        * Checks if the Knight can move from its starting position to a given end coordinate.
        * Overrides the can_move method in the Piece class.
        *
        * @param board The current game board as a 2D Piece array.
        * @param start The starting coordinate of the piece on the board.
```

```
        * @param end The end coordinate that the piece wants to move to.
        * @return A boolean indicating if the move is valid or not.
        */
        @Override
        public boolean can_move(Piece[][] board, Coordinates start, Coordinates end){
        if(!super.can_move(board, start, end)){
        return false;
        }

        int xdiff = Math.abs(end.X() - start.X());
        int ydiff = Math.abs(end.Y() - start.Y());

        return (xdiff == 2 && ydiff == 1) || (xdiff == 1 && ydiff == 2);
        }

        /**
        * Retrieves a list of all possible moves for the Knight piece on the current board.
        * Overrides the allPossibleMoves method in the Piece class.
        *
        * @param board The current game board as a 2D Piece array.
        * @param start The starting coordinate of the piece on the board.
        * @param kingPos The position of the player's king piece.
        * @return A list of all possible Coordinates that the piece can move to.
        */
        /**
 * Returns a list of all possible moves that a Knight can make on the given board from the given starting position,
 * considering the position of the opponent's King.
 *
 * @param board the current state of the chess board
 * @param start the starting position of the Knight
 * @param kingPos the current position of the opponent's King
 * @return a list of all possible moves for the Knight
 */
public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start, Coordinates kingPos) {
        // implementation
}

/**
 * Returns a list of all possible moves that a Knight can make on the given board from the given starting position.
 *
 * @param board the current state of the chess board
```

```
 * @param start the starting position of the Knight
 * @return a list of all possible moves for the Knight
 */
public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start) {
        // implementation
}
```

Pawn

```
/**
 * Represents a pawn piece in a game of chess.
 */
public class Pawn extends Piece {

        /**
        * Creates a new pawn with the specified player.
        * @param p The player that the pawn belongs to (either "W" or "B").
        */
        public Pawn(String p){
        super(p);
        this.type = "P";
        }

        /**
        * Determines whether the pawn can move from the starting position to the ending
position on the specified board.
        * @param board The game board.
        * @param start The starting position of the pawn.
        * @param end The ending position of the pawn.
        * @param kingPos The position of the king of the same color as the pawn.
        * @return true if the pawn can move to the ending position, false otherwise.
        */
        public boolean can_move(Piece[][] board, Coordinates start, Coordinates end,
Coordinates kingPos){
        // implementation details...
        }

        /**
        * Determines whether the pawn can move from the starting position to the ending
position on the specified board.
```

```
        * @param board The game board.
        * @param start The starting position of the pawn.
        * @param end The ending position of the pawn.
        * @return true if the pawn can move to the ending position, false otherwise.
        */
        public boolean can_move(Piece[][] board, Coordinates start, Coordinates end){
        // implementation details...
        }

        /**
        * Returns a list of all possible moves for the pawn from the specified starting position on
the specified board.
        * @param board The game board.
        * @param start The starting position of the pawn.
        * @param kingPos The position of the king of the same color as the pawn.
        * @return A list of all possible moves for the pawn from the specified starting position.
        */
        public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start,
Coordinates kingPos) {
        // implementation details...
        }

        /**
        * Returns a list of all possible moves for the pawn from the specified starting position on
the specified board.
        * @param board The game board.
        * @param start The starting position of the pawn.
        * @return A list of all possible moves for the pawn from the specified starting position.
        */
        public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start) {
        // implementation details...
        }
}
```

Piece

```
package com.example.group12project.ChessComponents;

import android.util.Log;
```

```
import java.util.*;

/**
 * The abstract Piece class represents a chess piece on the board.
 * It defines common properties and behaviors of all chess pieces.
 */
public abstract class Piece {

	private final String player;
	protected String type;

	/**
	* Constructor for the Piece class.
	* @param pl the player ("white" or "black") that owns this piece
	*/
	public Piece(String pl){
	player = pl;
	}

	/**
	* Returns the player that owns this piece.
	* @return the player ("white" or "black") that owns this piece
	*/
	public String get_player(){
	return player;
	}

	/**
	* Returns the type of this piece.
	* @return the type of this piece (e.g. "pawn", "rook", etc.)
	*/
	public String get_type(){
	return type;
	}

	/**
	* Determines if this piece can move from the given start position to the given end
position.
	* @param board the chess board, represented as a 2D array of Piece objects
	* @param start the starting position of the move
	* @param end the ending position of the move
	* @param kingPos the position of the king on the board
	* @return true if the move is valid, false otherwise
	*/
```

```
	public boolean can_move(Piece[][] board, Coordinates start, Coordinates end,
Coordinates kingPos){
	Piece s = board[start.X()][start.Y()];   // get start piece
	Piece e = board[end.X()][end.Y()];               // if end isn't null, get piece
	// check first to make sure start and end coordinates are not equal
	if((start.X() == end.X()) && (start.Y() == end.Y())){
	return false;
	}

	if(e == null){
	if(s!=null){
		if(!(this instanceof King)){
		Coordinates pinChecker = isPinned(board, kingPos, start);
		if(pinChecker != null) {
		//if pinned, can only move along axis of checker
		if (end.equals(pinChecker)) {
			return true;
		}
		List<Coordinates> bet = Coordinates.places_between(kingPos, pinChecker);
		for (Coordinates b : bet) {
			if (!b.equals(start) && b.equals(end)) {
			return true;	// if end pos is inline, then we can move
			}
		}
		return false; // else we are pinned and cannot move to end
		}
		}
	}
	}
	else if(s == null || Objects.equals(s.get_player(), e.get_player())){
	// check if starting Piece is null, and check if starting player is same as ending player
(cannot take own piece)
	return false;
	}
	else{
	//checking if pinned
	if(!(this instanceof King)){
		Coordinates pinChecker = isPinned(board, kingPos, start);
		if(pinChecker != null){
		//if pinned, can only move along axis of checker
		if(end.equals(pinChecker)){
		return true;
		}
		List<Coordinates> bet = Coordinates.places_between(kingPos, pinChecker);
```

```
                for(Coordinates b : bet){
                if(!b.equals(start) && b.equals(end)){
                        return true;      // if end pos is inline, then we can move
                }
                }
                return false; // else we are pinned and cannot move to end
                }
        }

        }
        return true;
        }


        /**
     * Determines if a piece can move from start to end coordinate on the board.
     *
     * @param board the current chess board
     * @param start the starting position
     * @param end the end position
     * @return true if the piece can move from start to end, false otherwise
     */
    public boolean can_move(Piece[][] board, Coordinates start, Coordinates end){
        Piece s = board[start.X()][start.Y()];      // get start piece
        Piece e = board[end.X()][end.Y()];          // if end isn't null, get piece

        // check first to make sure start and end coordinates are not equal
        if((start.X() == end.X()) && (start.Y() == end.Y())){
            return false;
        }

        // check if starting Piece is null, and check if starting player is same as ending player
(cannot take own piece)
        if(e == null){
            return (s != null);
        }
        else return s != null && !Objects.equals(s.get_player(), e.get_player());
    }

    /**
     * Returns a list of all possible moves for a piece on the board given a starting position and
the position of the king.
     *
     * @param board the current chess board
```

```
     * @param start the starting position of the piece
     * @param kingPos the position of the king
     * @return a list of all possible moves for the piece
     */
    public abstract List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start,
Coordinates kingPos);

    /**
     * Returns a list of all possible moves for a piece on the board given a starting position.
     *
     * @param board the current chess board
     * @param start the starting position of the piece
     * @return a list of all possible moves for the piece
     */
    public abstract List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start);

    /**
     * Determines if a piece is pinned by another piece on the board.
     *
     * @param board the current chess board
     * @param kingPos the position of the king
     * @param currPos the position of the piece to check if it is pinned
     * @return the position of the piece that is checking if the piece is pinned, null if the piece is
not pinned
     */
    public Coordinates isPinned(Piece[][] board, Coordinates kingPos, Coordinates currPos){
        if(board[currPos.X()][currPos.Y()] instanceof King || kingPos.equals(currPos)){
            return null;
        }
        if(board[kingPos.X()][kingPos.Y()] == null || !(board[kingPos.X()][kingPos.Y()] instanceof
King)){
            return null;
        }
        board[currPos.X()][currPos.Y()] = null;
        Coordinates checker = ((King)board[kingPos.X()][kingPos.Y()]).checkByWho(board,
kingPos);
        board[currPos.X()][currPos.Y()] = this;
        return checker;
    }

    /**
     * Prints the type and player of the piece.
     */
    public void print_piece(){
```

```
        Log.d("Piece:", get_type() + get_player());
    }
}
```

Queen

```
package com.example.group12project.ChessComponents;

import java.util.*;

/**
 * Represents a Queen chess piece.
 */
public class Queen extends Piece{

        /**
        * Constructor for Queen object.
        * @param p The player that owns the Queen (either "W" or "B").
        */
        public Queen(String p){
        super(p);
        this.type = "Q";
        }

        /**
        * Determines if a move is valid for the Queen.
        * @param board The chess board.
        * @param start The starting coordinates of the Queen.
        * @param end The ending coordinates of the Queen.
        * @param kingPos The coordinates of the player's King.
        * @return True if the move is valid, false otherwise.
        */
        public boolean can_move(Piece[][] board, Coordinates start, Coordinates end,
Coordinates kingPos){
        if(!super.can_move(board, start, end, kingPos)){
        return false;
        }

        // Check for diagonal or horizontal/vertical move
        int xdiff = Math.abs(end.X() - start.X());
        int ydiff = Math.abs(end.Y() - start.Y());
```

```
if(xdiff != ydiff){
if(xdiff != 0 && ydiff != 0){
        return false;
}
}

// Check for obstruction
List<Coordinates> between = Coordinates.places_between(start, end);
for(Coordinates i : between){
if(board[i.X()][i.Y()] != null){
        return false;
}
}

return true;
}

/**
* Determines if a move is valid for the Queen.
* @param board The chess board.
* @param start The starting coordinates of the Queen.
* @param end The ending coordinates of the Queen.
* @return True if the move is valid, false otherwise.
*/
public boolean can_move(Piece[][] board, Coordinates start, Coordinates end){
if(!super.can_move(board, start, end)){
return false;
}

// Check for diagonal or horizontal/vertical move
int xdiff = Math.abs(end.X() - start.X());
int ydiff = Math.abs(end.Y() - start.Y());

if(xdiff != ydiff){
if(xdiff != 0 && ydiff != 0){
        return false;
}
}

// Check for obstruction
List<Coordinates> between = Coordinates.places_between(start, end);
for(Coordinates i : between){
if(board[i.X()][i.Y()] != null){
```

```
                return false;
        }
        }

        return true;
        }

        /**
        * Generates a list of all possible moves for the Queen.
        * @param board The chess board.
        * @param start The starting coordinates of the Queen.
        * @param kingPos The coordinates of the player's King.
        * @return A list of all possible moves for the Queen.
        */
        public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start,
Coordinates kingPos) {
        List<Coordinates> poss = new ArrayList<>();
        for(int i = 0; i < 8; i++){
        for(int j = 0; j < 8; j++){
                Coordinates temp = new Coordinates(i,j);
                if(can_move(board, start, temp, kingPos)){
                poss.add(temp);
                }
        }
        }
        return poss;
        }

        /**
        * Generates a list of all possible moves for the Queen.
        * @param board The chess board.
        * @param start The starting coordinates of the Queen.
        * @return A list of all possible moves

/**
 * Returns a list of all possible coordinates that the queen can move to from the specified start
coordinate on the given board.
 * @param board the current state of the chess board
 * @param start the starting coordinate of the queen
 * @return a list of all possible coordinates that the queen can move to
 */
public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start) {
        List<Coordinates> poss = new ArrayList<>();
        for(int i = 0; i < 8; i++){
```

```
        for(int j = 0; j < 8; j++){
        Coordinates temp = new Coordinates(i,j);
        if(can_move(board, start, temp)){
                poss.add(temp);
        }
        }
        }
        return poss;
}
```

RandomChessBot

```
package com.example.group12project.ChessComponents;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Objects;
import java.util.Random;

/**
 * Represents a random chess bot that makes random moves.
 */
public class RandomChessBot extends ChessBot {

        /**
        * Initializes a new instance of the {@link RandomChessBot} class.
        */
        public RandomChessBot(){
        super();
        }

        /**
        * {@inheritDoc}
        */
        @Override
        public List<Coordinates> make_move(Piece[][] board, Coordinates kingPos) {
        List<Coordinates> end_pos = new ArrayList<>();
```

```
        List<Coordinates> start_pos = new ArrayList<>();
        for(int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
                if(board[i][j] != null && Objects.equals(board[i][j].get_player(), this.player)) {
                List<Coordinates> possible = board[i][j].allPossibleMoves(board, new
Coordinates(i,j), kingPos);
                for(Coordinates c : possible){
                //pretend as if move was made, then check if king is NOW in check?

                end_pos.add(new Coordinates(i,j));
                start_pos.add(c);
                }
                }
        }
        }

        Random rand = new Random();
        int choice = rand.nextInt(end_pos.size());
        return Arrays.asList(start_pos.get(choice), end_pos.get(choice));
        }
}
```

Rook

```
/**
 * The Rook class represents a rook in a game of chess. It extends the Piece class and
implements
 * the can_move and allPossibleMoves methods.
 *
 * This class also has a boolean variable called "hasMoved" that is used to determine whether or
not
 * the rook has moved before. This is used for castling.
 *
 */
public class Rook extends Piece {

        /**
        * Constructor for the Rook class.
        *
        * @param p a string representing the player that owns the piece ("white" or "black")
        */
```

```
    public Rook(String p){
    // ...
    }

    /**
    * Determines if the rook can move from the start position to the end position on the
board.
    *
    * @param board the current state of the chess board
    * @param start the starting coordinates of the piece
    * @param end the ending coordinates of the piece
    * @param kingPos the current position of the player's king
    * @return true if the move is legal, false otherwise
    */
    public boolean can_move(Piece[][] board, Coordinates start, Coordinates end,
Coordinates kingPos){
    // ...
    }

    /**
    * Determines if the rook can move from the start position to the end position on the
board.
    *
    * @param board the current state of the chess board
    * @param start the starting coordinates of the piece
    * @param end the ending coordinates of the piece
    * @return true if the move is legal, false otherwise
    */
    public boolean can_move(Piece[][] board, Coordinates start, Coordinates end){
    // ...
    }

    /**
    * Returns a list of all possible moves for the rook.
    *
    * @param board the current state of the chess board
    * @param start the starting coordinates of the piece
    * @param kingPos the current position of the player's king
    * @return a list of all possible moves for the rook
    */
    public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start,
Coordinates kingPos) {
    // ...
    }
```

```
	/**
	* Returns a list of all possible moves for the rook.
	*
	* @param board the current state of the chess board
	* @param start the starting coordinates of the piece
	* @return a list of all possible moves for the rook
	*/
	public List<Coordinates> allPossibleMoves(Piece[][] board, Coordinates start) {
	// ...
	}
}
```

Leaderboard

```
// Variables to hold the winner and loser names
public String winner, loser;

/**
 * Checks whether a username is already in the leaderboard
 * @param name The username to check
 * @param leaderboard The TableLayout that contains the leaderboard
 * @return Boolean true if the username is found in the leaderboard, false otherwise
 */
public boolean check_name(String name, TableLayout leaderboard){
	// Iterate through the rows of the table
	for (int i = 0; i < leaderboard.getChildCount(); i++) {
	View row = leaderboard.getChildAt(i);
	// Check if the row is a TableRow
	if (row instanceof TableRow) {
	TableRow tableRow = (TableRow) row;
	// Get the first TextView in the row
	TextView nameTextView = (TextView) tableRow.getChildAt(0);
	// Get the text from the TextView
	String test_name = nameTextView.getText().toString();
	if (name == test_name) {
		return true;
	}
	}
	}
	return false;
}
```

```
/**
 * Adds a win to a user already in the leaderboard
 * @param username The username to add a win to
 * @param leaderboard The TableLayout that contains the leaderboard
 */
public void add_win(String username, TableLayout leaderboard){
        // Iterate through the rows of the table
        for (int i = 0; i < leaderboard.getChildCount(); i++) {
        View row = leaderboard.getChildAt(i);
        // Check if the row is a TableRow
        if (row instanceof TableRow) {
        TableRow tableRow = (TableRow) row;
        // Get the first TextView in the row
        TextView nameTextView = (TextView) tableRow.getChildAt(0);
        // Get the text from the TextView
        String test_name = nameTextView.getText().toString();
        // Compare the name to the test name
        if ( username == test_name ) {
                TextView winsTextView = (TextView) tableRow.getChildAt(1);
                int wins = Integer.parseInt(winsTextView.getText().toString());
                wins++;
                winsTextView.setText(Integer.toString(wins));
        }
        }
        }
}

/**
 * Updates the leaderboard with either a new entry or by adding a win to
 * an existing winner
 * @param username The username to update in the leaderboard
 * @param winner Boolean indicating whether the user won the game
 * @param leaderboard The TableLayout that contains the leaderboard
 */
/**
 * Updates the leaderboard with the specified user's name and outcome.
 * If the user's name is already on the leaderboard, their win count is incremented.
 * Otherwise, a new row is added to the leaderboard with their name and outcome.
 *
 * @param username the name of the user to add to the leaderboard
 * @param winner true if the user won the game, false otherwise
 * @param leaderboard the TableLayout representing the leaderboard
 */
public void update_leaderboard(String username, boolean winner, TableLayout leaderboard){
```

```
        // implementation here
}

/**
 * Returns the user to the home page.
 */
public void goToHome(){
        // implementation here
}
```

MainActivity

```
/**
 * The main activity of the app that displays the main menu.
 */
public class MainActivity extends AppCompatActivity {

        /**
        * Vector to store entries with a name and number of wins for the leaderboard.
        */
        private static final Vector<Vector<String>> names_wins = new Vector<>();

        /**
        * Returns the leaderboard vector to be accessed across activities.
        *
        * @return The leaderboard vector.
        */
        public static Vector<Vector<String>> getLeaderboardVector() {
        return names_wins;
        }

        /**
        * Switches from main activity to user_names activity.
        */
        public void goToUserNames() {
        Intent intent_1 = new Intent(this, user_names.class);
        startActivity(intent_1);
        }

        /**
        * Starts a new bot game.
        */
```

```
        public void newBot() {
        Intent intent = new Intent(this, user_names_bot.class);
        startActivity(intent);
        }

        /**
        * Switches from main activity to leaderboard activity.
        */
        public void goToLeaderboard() {
        Intent intent = new Intent(this, Leaderboard.class);
        startActivity(intent);
        }

        @Override
        protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main); // Display main menu

        Button new_game = (Button) findViewById(R.id.new_game);
        Button leaderboard = (Button) findViewById(R.id.leaderboard);
        Button new_vs_bot = (Button) findViewById(R.id.Bot);

        new_game.setOnClickListener(view -> goToUserNames());
        new_vs_bot.setOnClickListener(view -> newBot());
        leaderboard.setOnClickListener(view -> goToLeaderboard());
        }

        /**
        * Makes a toast every 5 seconds.
        */
        private void makeToast() {
        final Handler handler = new Handler();
        handler.post(new Runnable() {
        @Override
        public void run() {
                Toast.makeText(MainActivity.this, "Toast every 5s",
Toast.LENGTH_SHORT).show();
                handler.postDelayed(this, 5000);
        }
        });
        }

}
```

SettingsActivity

```
package com.example.group12project;

import android.os.Bundle;

import androidx.appcompat.app.ActionBar;
import androidx.appcompat.app.AppCompatActivity;
import androidx.preference.PreferenceFragmentCompat;

/**
 * This activity displays the settings fragment, which allows the user to view and modify app settings.
 */
public class SettingsActivity extends AppCompatActivity {

        /**
        * Creates the activity and sets up the settings fragment.
        *
        * @param savedInstanceState a Bundle containing saved state information
        */
        @Override
        protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_settings);

        // Load the settings fragment
        if (savedInstanceState == null) {
        getSupportFragmentManager()
                .beginTransaction()
                .replace(R.id.settings, new SettingsFragment())
                .commit();
        }

        // Set up the action bar to display a back button
        ActionBar actionBar = getSupportActionBar();
        if (actionBar != null) {
        actionBar.setDisplayHomeAsUpEnabled(true);
        }
        }

        /**
        * A fragment that displays the app's settings UI.
        */
```

```
        public static class SettingsFragment extends PreferenceFragmentCompat {

        /**
        * Creates the preferences UI from an XML resource.
        *
        * @param savedInstanceState a Bundle containing saved state information
        * @param rootKey                the key for the root of the preference hierarchy
        */
        @Override
        public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
        setPreferencesFromResource(R.xml.root_preferences, rootKey);
        }
        }
}
```

user_names

```
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import androidx.preference.PreferenceFragmentCompat;
import androidx.appcompat.app.ActionBar;

/**
 * Activity for displaying the application settings using the SettingsFragment.
 */
public class SettingsActivity extends AppCompatActivity {

        /**
        * Initializes the activity, sets the content view to the activity_settings layout,
        * and adds the SettingsFragment to the activity.
        * If the savedInstanceState is null, then a new instance of the SettingsFragment is
created.
        * The up button is displayed in the action bar.
        */
        @Override
        protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_settings);
```

```
        if (savedInstanceState == null) {
        getSupportFragmentManager()
                .beginTransaction()
                .replace(R.id.settings, new SettingsFragment())
                .commit();
        }

        ActionBar actionBar = getSupportActionBar();
        if (actionBar != null) {
        actionBar.setDisplayHomeAsUpEnabled(true);
        }
        }

        /**
        * Fragment that displays the preferences for the application.
        */
        public static class SettingsFragment extends PreferenceFragmentCompat {

        /**
        * Loads the preferences from the root_preferences.xml file and adds them to the
fragment.
        *
        * @param savedInstanceState the saved state of the fragment
        * @param rootKey the root key for the preferences
        */
        @Override
        public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
        setPreferencesFromResource(R.xml.root_preferences, rootKey);
        }
        }
}
```

user_names_bot

```
import android.content.Intent;
import android.os.Bundle;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Switch;
```

```
import androidx.appcompat.app.AppCompatActivity;

/**
 * Activity for obtaining the name of the user who will play against the bot.
 */
public class user_names_bot extends AppCompatActivity {

        /**
        * A boolean indicating whether the bot should play aggressively or not.
        */
        boolean random;

        /**
        * Starts the game by launching the {@link Chessboard} activity.
        */
        public void goToBoard() {
        Intent intent_1 = new Intent(this, Chessboard.class);

        EditText t1 = (EditText) findViewById(R.id.player_one_name);
        Switch s1 = (Switch) findViewById(R.id.aggr_switch);
        Boolean random = !(s1.isChecked());

        intent_1.putExtra("BlackName", String.valueOf(t1.getText()));
        intent_1.putExtra("Bot", true);
        intent_1.putExtra("Random", random);

        startActivity(intent_1);
        }

        /**
        * Initializes the activity and sets the onClickListener for the start game button.
        *
        * @param savedInstanceState A Bundle containing the saved state of the activity.
        */
        @Override
        protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_bot_names);

        Button start_game = (Button) findViewById(R.id.start_game);
        start_game.setOnClickListener(view -> goToBoard());
        }
}
```