Slade Brooks

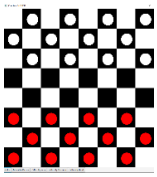
Report

Contents

Slade Brooks

Introduction

Example screenshot of program:



My program contains a board that has clickable input and 5 buttons bellow the board. The “rules” button brings up a page of the rules, the “help” button highlights movable pieces when hovered over with the mouse and the 3 difficulty buttons change the difficulty of the AI between easy, medium and hard depending on which is clicked.

Selection of piece colour was not covered in the specification, so I implemented the board as the AI player always being white. Turn order was not mentioned in the specs so the human Player always starts the game with the first move.

In order to make a move, the player clicks on the piece, available moves are highlighted, then the player clicks on the tile to move to.

The classes I used to implement this are:

Checkers:

This class runs the program when executed, it sets up the javafx application, automatically by creating the board, agent and buttons before adding the board and buttons to an interactable scene.

Board:

Board contains all the associated information needed including tile and pieces. Board is interacted with directly by the Checkers GUI class.

Piece:

Extends javafx stackpane to represent a piece on the board. Stackpane is used so that the piece and its crown can be stacked on eachother.

Tile:

Extends javafx rectangle class to represent a tile on the board.

Move:

Used to group appropriate information about a move.

Agent:

The agent is interacted with through the main checkers class. The agent is given a board and generates the best move it can from that board then returns the move. The checkers main app then feeds this move to the board to move a piece.

Enums:

I created 3 enumerable types to be used for the purpose of game logic. These types are PieceColor, PieceType and MoveType.

Slade Brooks

Description of program functionality

All line numbers refer to the appendix at the bottom of the document referencing code lines. All answer numbers refer to other answer of this section.

Game Internals

1. Interactive gameplay:
   Interactive gameplay is handled using Javafx imports. The game is run as an application, for examples see the human computer interaction sections for the answers 1-6 in the next section. This is done by taking the players clicks as input(can only affect board on their turn) and allowing the AI to move on their turn. The game loop can be seen in the animation timer created on line 51 of the appendix. This is run inside the start method implemented for the application extension of the class.
2. Valid state representation:
   In order to represent valid state representation in the agent. I made all classes except the checkers class so that they had a constructor capable of creating a deep copy of their object types. This allowed me to use the Board itself as a state representation with the getHeuristic() method in Board used as the states value. Using deep copies was necessary in order to perform moves on the Board objects without effecting the main board. This allowed the agent to explore a virtual tree of the boards available moves without changing anything in the main game GUI until needed. I implemented this through the boards method of getMoveResult() on line 376 in the appendix. This lets the Board create a deep copy of itself, then a deep copy of the move using the new board. Moves the piece in the copy then returns the board copy as a separate object representing the state after the move. As seen in all the classes(except checkers) they have constructors that allow for deep copies.
3. Successor function:
   See answer 4 below for pointers to description of the successor function. The successor function starts on line 1038 of the appendix.
4. Valid successor function:
   See answer 13 and 15 below for explanation of agents successor output being ensured as valid and see answer 6 below for explanation of the successor function.
5. Validation of user moves:
   See answer 7 below for explanation of move validation.
6. Successor function generates AI moves:
   Line 1038 is where the successor function starts. As you can see, it uses the minimax method to save the best move into a field of the class, then it creates a deep copy of that move using the original board and returns that copied move. See answer 9 and 11 below for further explanation of the minimax function. The successor function uses the depth saved as a field in the class to start off the minimax function which then recalls itself.
7. Rejection of invalid user moves with explanation:
   Every possibility where the use can click on the board is handled in the method recieveClick() starting on line 542. Every case where the user is not selecting a valid piece to move or a valid move is handled in this method by displaying an alert box explaining why the click was invalid. This by checking the users clicks against legal moves. Only legal moves are obtained through the process of the getAllMoves() method to compare against. See answer 13 below for a further explanation of how only legal valid moves are added to the move list generated by getAllMoves().
8. Rejection of invalid user moves:
   See answer 7 above for explanation.

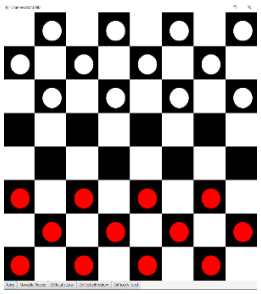
9. Valid minimax evaluation:
   The valid minimax function starts on line 1046. As you can see if follows the minimax algorithm directly with small additions added to the best move for max is saved
10. Minimax evaluation uses elements of pruning:
    See 11 below.
11. Minimax evaluation uses alpha beta pruning:
    This is shown in the minimax method at line 1067 and line 1079 where for either the max or min section where if alpha becomes greater than or equal to beta, the loop breaks and returns the value. Meaning not all parts of the tree representation need to be explored.
12. Variable level of AI cleverness adjustable by the user:
    Variable level of AI cleverness is set by changing the depth of the minimax function, meaning that the AI then looks more moves ahead to decide on the best move. This is shown on the line 1042 where the successor function calls the minimax method using the starting agent depth. This depth is changed by the user using buttons added to the UI, the buttons are created in the Checkers class in the method starting on lines 121. This method is used to create a button that sets the AI depth to a given number. 3 of these buttons are created with depths of 1 for easy, 2 for medium and 3 for hard. This is shown in lines 90, 91 and 92 of the appendix where the buttons are created before being added to the window in line 110 .
13. Entirely valid AI moves:
    Valid AI moves are generated by only generating legal moves as possibilities. As seen on line 1048, the minimax method used the boards method of getAllMoves() which only returns legal moves meaning the AI only has legal moves to pick from. Line 482 and 487 of the get moves method shows that they are only added to the list if they are legal. Legality is handled in the Move class itself through the method on line 994 where all needed legal cases of the move are handled.
14. Multi step user moves:
    See 15 and 16 below for explanation.
15. Multi step AI moves:
    Multi step kill moves are handled for the user and AI the same way. They are handled in the game logic by not ending the turn after a kill if there is another kill move with the same piece. This system also ensures that after a kill move if the piece can kill again then that piece only can move. This implements multi kills and enforced the forced capture rule further. For further explanation see 16(forced capture) below.
16. Forced capture (if there is a capture opportunity, the capturing move must be made):
    The forced capture rule is handled when generating all available moves. If a kill move is present as seen in line 525 of the Board class. Then only kill moves are returned as available moves. Both the Agent and the player pick from this generated move list to make moves. This ensures that they may only perform capture moves when one is present. The logic for this is implemented when a piece is moved and is seen in lines 346 to lines 361 of the board class. Lines 356 to 360 allows the other pieces to move again by setting the killerPiece field to null if no available kill moves exist for the piece after a kill move. This works because if killer piece is not set to null, only moves for that piece can be generated as seen on the method starting on line 449, where moves are only generated if killerPiece is null or the piece being looked at matches the killer piece.
17. Automatic king conversion:

This is demonstrated in line 264 of the board class where during a move, if the piece moves a position where its y value is equal to either 0 or 7(ends of board) then it is automatically converted to a king. For automatic conversion during regicide, see below.

18. Regicide (if a normal piece manages to capture a king, it is itself upgraded to a king):
    As seen in lines 30,351,352 of class Board. During a kill move if the killed piece is a king then the piece moving is turned into a king. This is implemented in the Piece class in line 851 where the method for setting a piece as a king is shown. This method changes the type of the piece to king(adds extra movements) and adds a golden crown to the piece.

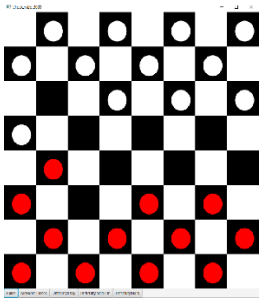
Human Computer Interface

1. Some kind of board representation displayed on screen:

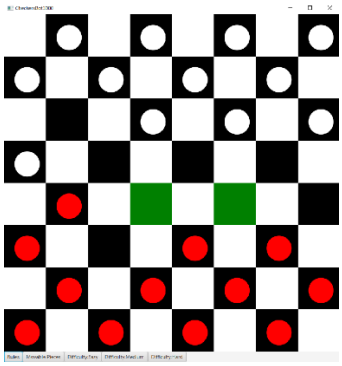


2. The interface properly updates display after completed moves.
   See 3 bellow.

3. . Fully interactive GUI that uses graphics:

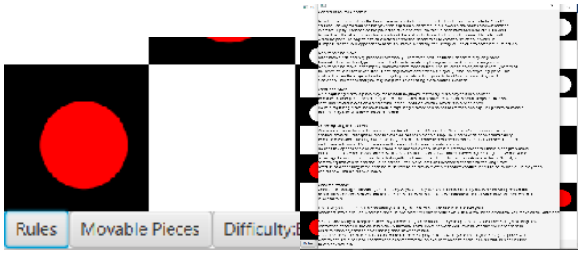


4. Mouse interaction focus, e.g., click to select & click to place or drag and drop (better):



Slade Brooks

5. GUI pauses appropriately to show the intermediate steps in multi-step moves:
   Shown through gameplay.
6. Dedicated display of the rules (e.g. a corresponding button opening a pop-up window):



Appendix

Class: Checkers

```
//javafx imports
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.layout.GridPane;
import javafx.animation.AnimationTimer;
import javafx.scene.control.Alert;
import javafx.scene.layout.Pane;
import javafx.scene.control.Button;
import javafx.event.EventHandler;

//file reading imports
import java.nio.file.Files;
import java.nio.file.Paths;
import javafx.scene.input.MouseEvent;

/**
 * The Checkers class creates the GUI using javafx implementations by extending application. It also handles turns. It creates and istance of the board
 * and the agent. Handles mouse input.
 *
 * @author Slade Brooks
```

```
32      * @version 1
33      */
34     public class Checkers extends Application{
35        Board board = new Board();
36        Agent agent = new Agent();
37
38        /**
39        * Start method used in application, is overriden. Takes the Stage, sets it up and impliments a game
40     loop in it.
41        *
42        * @param  primaryStage the primary stage to be shown.
43        */
44        public void start(Stage primaryStage) throws Exception{
45           Scene scene = new Scene(createContent());
46           primaryStage.setTitle("CheckersBot3000");
47           primaryStage.setScene(scene);
48           primaryStage.show();
49
50           //the game loop that handles the Agents turn and checks for a winner
51           AnimationTimer agentTurn = new AnimationTimer()
52           {
53              @Override
54              public void handle(long arg0)
55              {  //checks the board for a winner and prints the winner confirmation if one exists
56                 if(board.winCheck()){
57                    //new Alert(Alert.AlertType.CONFIRMATION, "Winner.").showAndWait();
58                    Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
59                    alert.setTitle("WINNER!");
60                    String winMessage = ((board.getTurn() == PieceColor.RED? PieceColor.WHITE :
61     PieceColor.RED) +" wins!!!");
62                    alert.setHeaderText(winMessage);
63                    alert.setContentText("Start new game?");
```

```
64                  alert.showAndWait();
65                  primaryStage.close();
66              }
67              //if no winner and agents turn then implement agents turn.
68              else if(board.getTurn() == PieceColor.WHITE && !board.winCheck()){
69                  Move agentMove = agent.successor(board);
70                  board.moveIfLegal(agentMove);
71              }
72            }
73        };
74        agentTurn.start();
75    }
76    /**
77     * Creates the content for the UI, this is a part of the application implementation and required for
78 this way of using javafx. This method creates
79     * a flowpane containing a Pane representing the board and buttons with correct utility
80 implemented.
81     *
82     * @return Parent a FlowPane containing the board and all the neccessary buttons.
83     */
84    public Parent createContent(){
85
86        FlowPane window = new FlowPane();
87
88        Button rules = createRuleButton();//displays rules when clicked.
89        Button help = createHelpButton();//highlights movable pieces when hovered.
90        Button hard = createDifficultyButton("Difficulty:Hard", 3);//sets agent depth to 3.
91        Button medium = createDifficultyButton("Difficulty:Medium", 2);//sets agent depth to 2.
92        Button easy = createDifficultyButton("Difficulty:Easy", 1);//sets agent depth to 1.
93
94        Pane grid = new Pane();//contains the board tiles and pieces
95        grid.getChildren().add(board.tiles);
```

```
96          grid.getChildren().add(board.pieces);
97          grid.setPrefSize(800,800);
98          //below when a board tile is clicked, the click co-ords are sent to board to proccess.
99          grid.setOnMouseClicked(e ->{
100             int x = (int)(e.getSceneX()/100);//x co-ord of click.
101             int y = (int)(e.getSceneY()/100);//y co-ord of click.
102             //only sends clicks if there isnt a winner.
103             if(!board.winCheck()){
104                 board.recieveMouseClick(x,y);//send click.
105             }
106         });
107
108
109         window.getChildren().add(grid);//ads board to window
110         window.getChildren().addAll(rules,help,easy,medium,hard);//adds buttons to window
111         return window;
112     }
113
114     /**
115      * Creates a button that when clicked displays the rules in a pop up dialog box. The rules are
116 sources from a text file contained in the project
117      * folder.
118      *
119      * @return Button rule buton that when click displays the rules from the rules.txt file.
120      */
121     private Button createRuleButton(){
122         Button button = new Button("Rules");//creates button.
123         //set the action that occurs when the button is clicked.
124         button.setOnAction(
125                 new EventHandler<ActionEvent>() {
126                     @Override
```

```
127                 public void handle(ActionEvent event) {
128                     final Stage dialog = new Stage();
129                     VBox dialogVbox = new VBox(20);
130 
131                     String content = "";
132                     try{
133                         content = new String ( Files.readAllBytes( Paths.get("rules.txt") ) );//reads the rules
134 file and saves into content.
135                     }catch(Exception e){System.out.println(e);}
136                     dialogVbox.getChildren().add(new Text(content));//adds the content to the dialog box
137                     Scene dialogScene = new Scene(dialogVbox, 700, 850);
138                     dialog.setScene(dialogScene);
139                     dialog.show();
140                 }
141             });
142         return button;
143     }
144     /**
145      * Creates a button that when hovered over, highlights movable pieces green for the player. When
146 mouse leaves button, pieces are unhighlighted.
147      *
148      * @return Button help button.
149      */
150     private Button createHelpButton(){
151         Button button = new Button("Movable Pieces");
152         //adds the mouse enters functionality to highlight pieces.
153         button.addEventHandler(MouseEvent.MOUSE_ENTERED,
154         new EventHandler<MouseEvent>() {
155           @Override
156           public void handle(MouseEvent e) {
157            board.highlightMovablePieces();
158           }
```

```
159         });
160         //adds the mouse leaves functionality to unhighlight pieces.
161         button.addEventHandler(MouseEvent.MOUSE_EXITED,
162         new EventHandler<MouseEvent>() {
163           @Override
164           public void handle(MouseEvent e) {
165            board.unHighlightPieces();
166           }
167         });
168
169         return button;
170       }
171       /**
172       * Creates a button that when clicked changes the minimax depth of the agent(changes difficulty).
173       *
174       * @param buttonName the buttons name when displayed in GUI.
175       * @param the depth that the button sets the agent too.
176       *
177       * @return Button with name that changes difficulty of agent.
178       */
179       private Button createDifficultyButton(String buttonName, int difficulty){
180         Button button = new Button(buttonName);
181         button.setOnAction(
182               new EventHandler<ActionEvent>() {
183                  @Override
184                  public void handle(ActionEvent event) {
185                     agent.setDepth(difficulty);//implements depth change in agent.
186                  }
187               });
188         return button;
189       }
```

```
190
191     /**
192      * Main method used to launch the application.
193      *
194      * @param args not used.
195      */
196     public static void main(String[] args){
197         launch(args);
198     }
199
200 }
```

Class: Board

```
201 import javafx.scene.Group;
202 import java.lang.Cloneable;
203 import javafx.scene.Node;
204 import javafx.scene.paint.Color;
205 import javafx.scene.control.Alert;//used for alerting the user after unallowed click/move attempt.
206 import java.util.ArrayList;
207 /**
208  * Board represents the board and all associated data, it is designed so that the main Checkers class
209 can interact with this only to effect the displayed
210  * pieces and tiles on the board.
211  *
212  * @author Slade Brooks
213  * @version 1
214  */
215 public class Board{
216     final static int TILE_SIZE = 100;//static tilesize, static to reduce parameters needed for other
217 methods.
218     private Tile[][] board = new Tile[8][8];//The board itself where the tiles are stored.
219     public Group tiles = new Group();//left as public so UI can interact with directly.
220     public Group pieces = new Group();////left as public so UI can interact with directly.
221
```

```
222     private PieceColor currentTurn;//current player turn.
223     private ArrayList<Move> availableMoves = new ArrayList<Move>();//used to set available moves
224 for a piece for highlighting tiles.
225     private Piece killerPiece = null;// used if the last move was a kill move
226     /**
227     * Constructor used when creating a Board as a deep copy of another Board. Copies Tile/Piece
228 locations creating copies of the Tile/Pieces.
229     *
230     * @param  oldBoard the oldBoard to be copied.
231     */
232     public Board(Board oldBoard){
233         currentTurn = oldBoard.currentTurn;//sets current turn to be the same. turn is an enum so deep
234 copy is not needed.
235
236         //iterates through board old board squares
237         for(int y = 0; y < 8; y++){
238             for(int x = 0; x < 8; x++){
239                 Tile tile = new Tile(oldBoard.getTile(x,y));//creates copy of each tile in oldBoard.
240                 board[x][y] = tile;
241                 tiles.getChildren().add(tile);//adds tile to group
242                 if(oldBoard.getTile(x,y).hasPiece()){//if piece exists on tile, creates copy and sets it
243 appropriately.
244                     Piece piece = new Piece(oldBoard.getTile(x,y).getPiece());
245                     board[x][y].addPiece(piece);
246                     pieces.getChildren().add(piece);
247                 }
248             }
249         }
250     }
251     /**
252     * Constructor used when creating a new board with pieces set to default positions.
253     */
```

```
254     public Board(){
255         currentTurn = PieceColor.RED;//Starting player/colour.
256 
257         //iterates through each square in the 8 x 8 board.
258         for(int y = 0; y < 8; y++){
259             for(int x = 0; x < 8; x++){
260                 Tile tile = new Tile(x,y,(((x+y)%2) == 1 ? Color.BLACK : Color.WHITE));//creates a new tile
261 using formula to set colour.
262                 board[x][y] = tile;
263                 tiles.getChildren().add(tile);
264                 if(((y == 0 || y == 2) && ((x+1) % 2) == 0)||((y == 1) && ((x+2) % 2) == 0)){//checks formula
265 to create white pieces appropriately.
266                     Piece piece = new Piece(PieceType.PIECE, PieceColor.WHITE, x,y);
267                     board[x][y].addPiece(piece);
268                     pieces.getChildren().add(piece);
269                 }
270                 else if(((y == 7 || y == 5) && ((x+2) % 2) == 0)||((y == 6) && ((x+1) % 2) == 0)){//checks
271 formula to create red pieces appropriately.
272                     Piece piece = new Piece(PieceType.PIECE, PieceColor.RED, x, y);
273                     board[x][y].addPiece(piece);
274                     pieces.getChildren().add(piece);
275                 }
276             }
277         }
278     }
279     /**
280     * Get method returning the current players turn in the form of a PieceColor enum.
281     *
282     * @return PieceColor turn.
283     */
284     public PieceColor getTurn(){
285         return this.currentTurn;
```

```
286     }
287     /**
288     * Get method for getting a Piece by co-ordinates.
289     *
290     * @return Piece piece at co-ords
291     */
292     public Piece getPiece(int x, int y){
293        return board[x][y].getPiece();
294     }
295     /**
296     * Get method for getting a Tile by co-ordinates.
297     *
298     * @return Tile tile at co-ords
299     */
300     public Tile getTile(int x, int y){
301        return board[x][y];
302     }
303
304     /**
305     * Moves piece after checking move legality.Used by players to move pieces. Applies delay to the
306  movePiece() method because this will only be used
307     * to take a turn not for internals of the agent. try catch statement used because the delay needs
308  to throw exception to be used.
309     *
310     * @param move the move that needs to be performed.
311     */
312     public void moveIfLegal(Move move) {
313        if(move.isLegal(currentTurn)){
314           try{
315              movePiece(move,true);
316           }catch(Exception e){}
317        }
```

```
318     }
319     /**
320     * Moves piece, uses delay optionaly to segment kill moves. Delay is optional so this method can be
321 used by the agents internals without using the
322     * delay in calculations.
323     *
324     * @param move the move that needs to be performed.
325     * @param sleep boolean used to decide if the delay is needed
326     */
327     public void movePiece(Move move, Boolean sleep) throws InterruptedException{
328         int x = move.getPiece().getX();//gets pieces x.
329         int y = move.getPiece().getY();//gets pieces y
330
331         int toX = (int)move.getTo().getX();//gets destinations x.
332         int toY = (int)move.getTo().getY();//gets destinations y.
333
334         //if piece reaches either end of board it is set to king.
335         if(toY == 0 || toY == 7){
336             move.getPiece().setKing();
337         }
338
339         move.getPiece().setXY(toX,toY);//sets pieces new coords
340         this.getTile(x,y).removePiece();//removes piece from old tile
341         move.getTo().addPiece(move.getPiece());//places piece on new tile
342         move.getPiece().move(move.getPiece().getX(),move.getPiece().getY());//move piece on GUI,
343 gets piece from Move then uses the pieces move().
344
345         //different game logic is the move is a kill move, ensuring multi kill moves are possible.
346         if(move.getType() == MoveType.KILL) {
347             if(sleep){
348                 Thread.sleep(500);//used to delay between kills when not performing agents internals.
349             }
```

```
350             if(move.getKill().getPiece().type() == PieceType.KING){
351                 move.getPiece().setKing();//implements regicide, becomes king if king is killed.
352             }
353             pieces.getChildren().remove(move.getKill().getPiece());//removes killed piece from group
354             move.getKill().removePiece();//removes killed piece from tile
355             killerPiece = move.getPiece();//ensures that double kill possibility is checked.
356             if(getAllMoves().size() < 1){
357               currentTurn = (currentTurn == PieceColor.WHITE ? PieceColor.RED : PieceColor.WHITE);
358 //changes turn only if another kill is not possible.
359               killerPiece = null;//sets current kill piece to null if no kill is possible.
360             }
361         }
362         //runs when move is not a kill move.
363         else{
364             currentTurn = (currentTurn == PieceColor.WHITE ? PieceColor.RED :
365 PieceColor.WHITE);//change turn.
366             killerPiece = null;
367         }
368     }
369
370     /**
371      * Creates a deep copy of a board then performes a deep copied move on the board. This is only
372 used by agent internals as a gamestate node.
373      *
374      * @param move the move that needs to be performed.
375      */
376     public Board getMoveResults(Move move){
377         Board newBoard = null;
378         newBoard = new Board(this);//deep copies current board
379         Move newMove = new Move(move, newBoard);//deep copies move to be performed with
380 associated objects from new copied board.
381
```

```
382         //try statement needed for delay even though not used.
383         try{
384             newBoard.movePiece(newMove,false);//moves piece on the new board.
385         }catch(Exception e){}
386 
387         return newBoard;
388     }
389     /**
390      * Returns a value for the board to be used by the agent. Because agent colour doesnt change,
391 white will always be the max in a minimax algorithm
392      * therefore this method returns a higher value based on white having a higher score. The heuristic
393 type greatly effects the agent playstyle,
394      * this is a simple heuristic causing the agent to only value capturing pieces and creating Kings
395 while minimising enemy Kings.
396      *
397      * @return int value that is higher if white is performing well in terms of piece numbers otherwise
398 lower.
399      */
400     public int getHeuristic(){
401         int score = 0;
402         //counts through all children adding 1 point for a white PIECE and 2 for a white KING, -1 point
403 for red piece and -2 for red king.
404         for(Node p: pieces.getChildren()){
405             if(((Piece)p).getColor() == PieceColor.WHITE){
406                 if(((Piece)p).type() == PieceType.KING){
407                     score++;
408                 }
409                 score++;
410             }
411             if(((Piece)p).getColor() == PieceColor.RED){
412                 if(((Piece)p).type() == PieceType.KING){
413                     score--;
414                 }
```

```
415                 score++;
416             }
417         }
418         return score;
419     }
420     /**
421     * This method is used in highlighting available moves for a piece when clicked. Gets all available
422 moves then stores all of those moves that
423     * correspond to a specific piece in a field in this Board.
424     *
425     *@param Piece sets all available moves for this piece.
426     */
427     public void getAvailableMoves(Piece piece){
428         ArrayList<Move> allMoves = getAllMoves();//get all possible moves for all pieces.
429         availableMoves.clear();//clear old available moves.
430
431         //iterates through all possible moves.
432         for(Move m: allMoves){
433             if(m.getPiece() == piece){
434                 availableMoves.add(m);//if that move starts with the chosen piece then move is added to
435 available moves.
436                 m.getTo().highlight();//highlights the tile of the move.
437             }
438
439         }
440     }
441     /**
442     * This method returns all possible move for the current player. Only kill moves if any are available
443 otherwise all normal moves. Piece type is taken
444     * into account in the move list. if the game is mid multi kill move then turn doesnt change and
445 only those multi kill moves are allowed next.
446     *
```

```
447     * @return ArrayList<Move> arraylist containing all possible and legal moves for current player.
448     */
449     public ArrayList<Move> getAllMoves(){
450
451         PieceColor col = this.currentTurn;
452         ArrayList<Move> killMoves = new ArrayList<Move>();//where possible kill moves are stored
453         ArrayList<Move> normalMoves = new ArrayList<Move>();
454         for(Node p: pieces.getChildren()){//iterates through all node pieces.
455             Piece piece = (Piece)p;//castes the node to a piece.
456             //checks if piece matches current turn and if the game is midway through a multi kill move.
457             if(col == piece.getColor() && (killerPiece == null ||piece == killerPiece)){
458                 //this if statement handles white moves or kings moving backwards.
459                 if(piece.type() == PieceType.KING || piece.getColor() == PieceColor.WHITE){
460                     ArrayList<Move> tempKill = new ArrayList<Move>();//only stores kill moves for 1 piece.
461                     ArrayList<Move> tempNorm = new ArrayList<Move>();//only stores normal moves for 1
462 piece.
463                     //the below 4 if statements only adds the moves if they are within bounds of the board.
464                     if((piece.getX()-1) >= 0 && (piece.getY()+1) < 8 && killerPiece == null){
465                         tempNorm.add( new Move(MoveType.NORMAL, piece, getTile(piece.getX()-1,
466 piece.getY()+1)));
467                     }
468                     if((piece.getX()+1) < 8 && (piece.getY()+1) < 8 && killerPiece == null){
469                         tempNorm.add( new Move(MoveType.NORMAL, piece, getTile(piece.getX()+1,
470 piece.getY()+1)));
471                     }
472                     if((piece.getX()-2) >= 0 && (piece.getY()+2) < 8){
473                         tempKill.add( new Move(MoveType.KILL, piece, getTile(piece.getX()-2, piece.getY()+2),
474 getTile(piece.getX()-1, piece.getY()+1)));
475                     }
476                     if((piece.getX()+2) < 8 && (piece.getY()+2) < 8){
477                         tempKill.add( new Move(MoveType.KILL, piece, getTile(piece.getX()+2, piece.getY()+2),
478 getTile(piece.getX()+1, piece.getY()+1)));
479                     }
```

```
480                 //checks each move is legal before adding them to the master lists.
481                 for(Move m: tempKill){
482                     if(m.isLegal(this.currentTurn)){
483                         killMoves.add(m);
484                     }
485                 }
486                 for(Move m: tempNorm){
487                     if(m.isLegal(this.currentTurn)){
488                         normalMoves.add(m);
489                     }
490                 }
491             }
492             //this if statement handes red moves or kings moving forward. performs the same logic as
493 above but different directional moves.
494             if(piece.type() == PieceType.KING || piece.getColor() == PieceColor.RED){
495                 ArrayList<Move> tempKill = new ArrayList<Move>();
496                 ArrayList<Move> tempNorm = new ArrayList<Move>();
497                 if((piece.getX()-1) >= 0 && (piece.getY()-1) >= 0 && killerPiece == null){
498                     tempNorm.add( new Move(MoveType.NORMAL, piece, getTile(piece.getX()-1,
499 piece.getY()-1)));
500                 }
501                 if((piece.getX()+1) < 8 && (piece.getY()-1) >=0 && killerPiece == null){
502                     tempNorm.add( new Move(MoveType.NORMAL, piece, getTile(piece.getX()+1,
503 piece.getY()-1)));
504                 }
505                 if((piece.getX()-2) >= 0 && (piece.getY()-2) >= 0){
506                     tempKill.add( new Move(MoveType.KILL, piece, getTile(piece.getX()-2, piece.getY()-2),
507 getTile(piece.getX()-1, piece.getY()-1)));
508                 }
509                 if((piece.getX()+2) < 8 && (piece.getY()-2) >= 0){
510                     tempKill.add( new Move(MoveType.KILL, piece, getTile(piece.getX()+2, piece.getY()-2),
511 getTile(piece.getX()+1, piece.getY()-1)));
512                 }
```

```
513                 for(Move m: tempKill){
514                     if(m.isLegal(this.currentTurn)){killMoves.add(m);}
515                 }
516                 for(Move m: tempNorm){
517
518                     if(m.isLegal(this.currentTurn)){normalMoves.add(m);}
519                 }
520               }
521             }
522         }
523         //this if statement implements kill moves having to be taken by only returning kill moves if
524 atleast 1 is possible.
525         if(killMoves.size() > 0){
526             return killMoves;
527         }
528         //if no legal kill moves are available then return legal normal moves
529         else{
530             //System.out.println(normalMoves.size() +" normal moves");
531             return normalMoves;
532         }
533     }
534     /**
535      * This method performs the logic for the board recieving a mouse click and creates an appropriate
536 error message if the click is to move illegaly.
537      *
538      *
539      * @param x board co-ordinate of the click.
540      * @param y board -co-ordinate of the click.
541      */
542     public void recieveMouseClick(int x, int y){
543         //Thread.sleep(300);
544         Tile tileClicked = board[x][y];
```

```
545         //if no moves are highlighted and no piece to move on tile
546         if(tileClicked.getPiece() == null && availableMoves.size() < 1){
547             new Alert(Alert.AlertType.ERROR, "No piece to move on this tile.").showAndWait();//informs
548 player.
549         }
550         //player clicks on empty tile after clicking on own piece to move to tile.
551         else if(tileClicked.getPiece() == null && availableMoves.size() > 0){
552             boolean found = false;
553             //checks available moves to see if move is legal.
554             for(Move m: availableMoves){
555                 Tile tile = m.getTo();
556                 if(tileClicked == tile){
557                     try{
558                         movePiece(m,false);
559                     }catch(Exception e){}
560                     unHighlightAvailable();//unhighlights available moves
561                     found = true;
562                 }
563             }
564             //move is legal so available moves are cleared.
565             if(found){
566                 availableMoves.clear();
567             }
568             //move isnt legal so player is informed.
569             else{
570                 new Alert(Alert.AlertType.ERROR, "Cannot move piece to that tile, select highlighted tile or
571 a new piece to move.").showAndWait();
572             }
573         }
574         //if no available moves shown and user clicks on enemy piece.
575         else if(tileClicked.getPiece().getColor() != PieceColor.RED && availableMoves.size() < 1){
```

```
576             new Alert(Alert.AlertType.ERROR, "This is not your piece, click on a red piece during your turn
577     to see moves.").showAndWait();
578         }
579         //clicks on enemy piece after highlighting available moves.
580         else if(tileClicked.getPiece().getColor() != PieceColor.RED && availableMoves.size() > 0){
581             new Alert(Alert.AlertType.ERROR, "Invalid move location, click on one of the highlighted tiles
582     to move or select a new piece.").showAndWait();
583         }
584         //player clicks on own piece, available moves are highlighted for that piece.
585         else if(tileClicked.getPiece().getColor() == PieceColor.RED ){
586             unHighlightAvailable();
587             availableMoves.clear();
588             getAvailableMoves(tileClicked.getPiece());
589         }
590     }
591     /**
592      * This method un highlights all the destination tiles in available moves for a piece when the piece
593     is either moved or a new piece is selected.
594      */
595     private void unHighlightAvailable(){
596         for(Move m:availableMoves){
597             m.getTo().unHighlight();
598         }
599     }
600     /**
601      * This method iterates through the pieces and if each player has atleast one piece, the returns
602     false otherwise if a player has no pieces returns true.
603      *
604      * @return boolean true if a player has won otherwise false.
605      */
606     public boolean winCheck(){
607         boolean whiteWin = true;
```

```
608         boolean redWin = true;
609         for(Node p: pieces.getChildren()){
610             if(((Piece)p).getColor() == PieceColor.WHITE){
611                 redWin = false;
612             }
613             if(((Piece)p).getColor() == PieceColor.RED){
614                 whiteWin = false;
615             }
616         }
617         if(whiteWin || redWin){
618             return true;
619         }
620         else{
621             return false;
622         }
623     }
624     /**
625     * Highlights all pieces with available moves.
626     */
627     public void highlightMovablePieces(){
628         ArrayList<Move> allMoves = getAllMoves();
629         for(Move m: allMoves){
630             m.getPiece().highlight();
631         }
632     }
633     /**
634     * Unhighlights all pieces with available moves.
635     */
636     public void unHighlightPieces(){
637        ArrayList<Move> allMoves = getAllMoves();
638         for(Move m: allMoves){
```

```
639             m.getPiece().unHighlight();
640         }
641     }
642 }
```

Class: Tile

```
643 import javafx.scene.shape.Rectangle;
644 import javafx.scene.paint.Color;
645 
646 /**
647 * Used to represent a tile on the board, extends rectangle to create a grid system.
648 *
649 * @author     Slade Brooks
650 * @version    1
651 */
652 public class Tile extends Rectangle
653 {
654     private Piece piece;//Piece stored in tile
655     public Color c;//Colour of tile
656     final int TILE_SIZE = Board.TILE_SIZE;
657 
658     /**
659     * Constructor used when creating a deep copy.
660     *
661     * @param  tile the tile param provided is deep copied to create this tile.
662     */
663     public Tile(Tile tile){
664         this(tile.getXInt(),tile.getYInt(),tile.getColor());
665     }
666     /**
667     * Constructor used for creating a tile.
668     *
669 
```

```
670         * @param  x used to set the x value of the rectangle and the xInt value of the Tile.
671         * @param  y used to set the y value of the rectangle and the yInt value of the Tile.
672         * @param  c of type Color, used to fill the rectangles colour.
673         */
674         public Tile(int x, int y,Color c)
675         {
676            piece = null;//tiles are created with no pieces.
677            this.setX(x);//sets x co ordinate of tile on board.
678            this.setY(y);//sets y co ordinate of tile on board.
679            this.c = c;
680
681            setWidth(TILE_SIZE);//sets width of tile to 100.
682            setHeight(TILE_SIZE);//sets height of tile to 100.
683
684            //tile is set to the position of its x and y multiplied by tile size.
685            relocate(this.getX() * TILE_SIZE, this.getY() * TILE_SIZE);
686            setFill(c);//fills the colour of the tile on board.
687         }
688
689         /**
690         * Method is used to determine if the tiles contains a piece or if its Piece field is set to null.
691         *
692         * @return True if Tile contains piece, otherwise false.
693         */
694         public boolean hasPiece()
695         {
696            return piece != null;
697         }
698         /**
699         * Get method for the tiles stored Piece.
700         *
```

```
701         * @return Piece returns the Tiles associated piece field.
702         */
703         public Piece getPiece(){
704             return piece;
705         }
706         /**
707         * A set method for the tiles Piece field.
708         *
709         * @param p sets the Piece field of tile to equal p.
710         */
711         public void addPiece(Piece p){
712             this.piece = p;
713         }
714         /**
715         * Sets the Piece field of Tile to equal null.
716         */
717         public void removePiece(){
718             piece = null;
719         }
720         /**
721         * Method is used to highlight the tile as green in the GUI.
722         */
723         public void highlight(){
724             setFill(Color.GREEN);
725         }
726         /**
727         * Method is used to remove GUI highlight and set tile back to origional colour.
728         */
729         public void unHighlight(){
730             setFill(this.c);
731         }
```

```
732     /**
733      * The getX() method for rectangle is final and cannot be overwritten, this is used as an alternative
734     that returns an int, used instead of casting.
735      *
736      * return x int representation of the tiles x position.
737      */
738     public int getXInt(){
739         return (int)this.getX();
740     }
741     /**
742      * The getY() method for rectangle is final and cannot be overwritten, this is used as an alternative
743     that returns an int, used instead of casting.
744      *
745      * return y int representation of the tiles y position.
746      */
747     public int getYInt(){
748         return (int)this.getY();
749     }
750     /**
751      * Get method for the colour of the tile.
752      *
753      * return c returns the default colour of the tile in type Color.
754      */
755     public Color getColor(){
756         return this.c;
757     }
758 }
```

Class: Piece

```
759 import javafx.scene.layout.StackPane;
760 import javafx.scene.paint.Color;
761 import javafx.scene.shape.Ellipse;
762
763
```

```
764 import java.util.ArrayList;
765
766 /**
767 *The piece class extends StackPane so that multiple ellipses can be stacked ontop of eachother to
768 represent a King piece.
769 *
770 * @author     Slade Brooks
771 * @version    1
772 */
773 public class Piece extends StackPane
774 {
775     final int TILE_SIZE = Board.TILE_SIZE;
776     private PieceType type;
777     private PieceColor color;
778     private int x, y;
779
780
781     /**
782     * Constructor used when creating a deep copy from another Piece.
783     *
784     * @param  piece the piece param provided is deep copied to create this Piece.
785     */
786     public Piece(Piece piece){
787         this(piece.type(),piece.getColor(),piece.getX(),piece.getY());
788     }
789     /**
790     * Constructor used for creating a Piece.
791     *
792     * @param  type used to set the x value of the rectangle and the xInt value of the Tile.
793     * @param  color used to set the y value of the rectangle and the yInt value of the Tile.
794     * @param  x position of piece in terms of board tiles.
```

```
795     * @param  y of type Color, used to fill the rectangles colour.
796     */
797     public Piece(PieceType type,PieceColor color, int x, int y)
798     {
799         this.type = type;//type of piece, either PIECE type or KING type
800         this.x = x;//x co ordinate on the board.
801         this.y = y;//y co ordinate on the board.
802         this.color = color;//colour  of piece, either RED or WHITE
803 
804         this.move(x, y);//sets the piece over the appropriate tile
805 
806         Ellipse ellipse = new Ellipse(TILE_SIZE * 0.3, TILE_SIZE * 0.3);//This ellipse represents the bottom
807     piece itself.
808         ellipse.setId("ellipse");//used to lookup the bottom piece(kings crown doesnt need to be looked
809     up).
810         ellipse.setRadiusX(30);
811         ellipse.setRadiusY(30);
812         ellipse.setFill(color == PieceColor.WHITE ? Color.WHITE : Color.RED);//fils the ellipse to match
813     the appropriate PieceColor
814 
815         getChildren().addAll(ellipse);//adds the ellipse to the StackPane
816 
817         //Implements king type if type is set as king
818         if(type == PieceType.KING){
819             this.setKing();
820         }
821     }
822     /**
823     * Sets the piece to type KING and creates a golden ellipse to be placed on the top to distinguish it.
824     */
825     public void setKing(){
826         //implements king if not already a king.
```

```
827             if(this.type != PieceType.KING){
828                 this.type = PieceType.KING;//new PieceType.
829                 Ellipse crown = new Ellipse(20 * 0.3, 20 * 0.3);//creates crown ellipse.
830                 crown.setRadiusX(10);//crown is smaller than origional piece ellipse.
831                 crown.setRadiusY(10);
832                 crown.setFill(Color.GOLD);//colours crown gold.
833                 getChildren().addAll(crown);//adds crown to the Piece stackpane.
834             }
835         }
836         /**
837         * Relocates the stackpane to x,y co-ords on the board.
838         */
839         public void move(int x, int y) {
840             relocate(x * TILE_SIZE + 25, y * TILE_SIZE + 25);
841         }
842         /**
843         * Get method for PieceType enum stored in Piece.
844         *
845         * @return returns the PieceType of the piece, e.g. PIECE, KING.
846         */
847         public PieceType type(){
848             return type;
849         }
850         /**
851         * Set method for the x and y co-ords of the piece.
852         *
853         * @param x sets pieces x to this.
854         * @param y sets pieces y to this.
855         */
856         public void setXY(int x, int y){
857             this.x = x;
```

```
858             this.y = y;
859         }
860         /**
861         * Get method for x co-ord of piece on board.
862         *
863         * @return x int co-ord of piece.
864         */
865         public int getX(){
866             return x;
867         }
868         /**
869         * Get method for y co-ord of piece on board.
870         *
871         * @return y int co-ord of piece.
872         */
873         public int getY(){
874             return y;
875         }
876         /**
877         * Get method for PieceColor of the Piece e.g. RED, WHITE.
878         *
879         * @return PieceColor field.
880         */
881         public PieceColor getColor(){
882             return this.color;
883         }
884         /**
885         * Looks up the bottom ellipse and colours it green to represent highlighting.
886         */
887         public void highlight(){
888             Ellipse ellipse = (Ellipse)lookup("#ellipse");
```

```
889         ellipse.setFill(Color.GREEN);
890     }
891     /**
892     * Used to remove highlighting from a highlighted piece.
893     */
894     public void unHighlight(){
895         Ellipse ellipse = (Ellipse)lookup("#ellipse");
896         ellipse.setFill((color == PieceColor.WHITE ? Color.WHITE : Color.RED));
897     }
898 }
```

Class: Move

```
899 /**
900  * Move is a type that contains all the required information to perform a move. It gets saved as a
901 type to make checking legality easier as well
902  * as grouping all neccessary information for easier retrieval.
903  *
904  * @author Slade Brooks
905  * @version 1
906  */
907 public class Move
908 {
909     private Piece piece;//the piece to be moved.
910     private Tile to;//destination of move.
911     private Tile killTile;//If move is of type kill then the tile of the kill is saved
912     private MoveType type;// Can be either NORMAL or KILL
913
914     /**
915     * Constructor used when creating a deep copy from another Move, the associated co-ords from
916 the old board are used to generate a depp copy
917     * of a move with co-ords relating to a new board.
918     *
919     * @param  oldMove the old move to be copied.
920
```

```
921     * @param newBoard the new board for the new objects to be linked to.
922     */
923     public Move(Move oldMove, Board newBoard){
924        this(oldMove.getType(), //type
925           newBoard.getPiece(oldMove.getPiece().getX(),oldMove.getPiece().getY()), //piece
926           newBoard.getTile(oldMove.getTo().getXInt(),oldMove.getTo().getYInt()),  //to
927           (oldMove.getKill() == null ? null :
928 newBoard.getTile(oldMove.getKill().getXInt(),oldMove.getKill().getYInt())) //killTile
929        );
930     }
931     /**
932     * Constructor used for creating a Move of type KILL.
933     *
934     * @param  type used to represent if the move is of type KILL or type NORMAL.
935     * @param  piece links to a Piece object to be moved.
936     * @param  to links to a Tile object as the destination of the piece.
937     * @param  killTile links to a Tile for kill move.
938     */
939     public Move(MoveType type, Piece piece, Tile to, Tile killTile)
940     {
941        this(type,piece,to);
942        this.killTile = killTile;
943     }
944     /**
945     * Constructor used for creating a Move without a kill Tile.
946     *
947     * @param  type used to represent if the move is of type KILL or type NORMAL.
948     * @param  piece links to a Piece object to be moved.
949     * @param  to links to a Tile object as the destination of the piece.
950     */
951     public Move(MoveType type, Piece piece, Tile to)
```

```
952         {
953             this.piece = piece;
954             this.to = to;
955             this.type = type;
956         }
957         /**
958         * Get method for Piece to be moved.
959         *
960         * @return piece to be moved.
961         */
962         public Piece getPiece(){
963             return piece;
964         }
965         /**
966         * Get method for the destination Tile.
967         *
968         * @return destination Tile.
969         */
970         public Tile getTo(){
971             return to;
972         }
973         /**
974         * Get method for the kill Tile.
975         *
976         * @return kill Tile.
977         */
978         public Tile getKill(){
979             return killTile;
980         }
981         /**
982         * Get method for the MoveType e.g. KILL, NORMAL.
```

```
983         *
984         * @return MoveType of Move.
985         */
986         public MoveType getType(){
987             return type;
988         }
989         /**
990         * Checks legality of move using the internals stored in Move.
991         *
992         * @return boolean true if legal, otherwise false.
993         */
994         public boolean isLegal(PieceColor currentTurn){
995             //todo handleillegal moves
996             if(currentTurn!= piece.getColor()){
997                 return false;
998             }
999             else if(to.hasPiece() == true){
1000                return false;
1001            }
1002            if(type == MoveType.KILL && killTile.hasPiece() == false){
1003                return false;
1004            }
1005            else if(type == MoveType.KILL && killTile.getPiece().getColor() == currentTurn){
1006                return false;
1007            }
1008            return true;
1009        }
1010    }
```

Class: Agent

```
1011    import java.lang.Math;
1012    import java.util.ArrayList;
1013
```

```
1014  /**
1015   * Write a description of class Agent here.
1016   *
1017   * @author (your name)
1018   * @version (a version number or a date)
1019   */
1020  public class Agent
1021  {
1022     Move bestMove= null;
1023     int agentDepth;
1024
1025     int pieceX;
1026     int pieceY;
1027     int toX;
1028     int toY;
1029     MoveType type;
1030     public Agent()
1031     {
1032        this.agentDepth = 1;
1033     }
1034     public void setDepth(int depth){
1035        this.agentDepth = depth;
1036     }
1037
1038     public Move successor(Board board){
1039        bestMove = null;
1040        int alpha = -10000;
1041        int beta = 10000;
1042        int bestScore = minimax(new Board(board), this.agentDepth, alpha, beta, true);
1043
1044        return new Move(bestMove, board);
```

```
1045        }
1046        public int minimax(Board board, int depth, int alpha, int beta, Boolean isWhite){
1047
1048            ArrayList<Move> legalMoves = board.getAllMoves();
1049
1050            if(depth <= 0 || legalMoves.size() < 1){
1051                return board.getHeuristic();
1052            }
1053            else if(isWhite){
1054                int v = -10000;
1055                for(Move m: legalMoves){
1056                    Move move = new Move(m,new Board(board));
1057
1058                    int max = minimax(board.getMoveResults(m),depth-1,alpha,beta,!isWhite);
1059                    if(max > v && this.agentDepth == depth){
1060
1061                        bestMove = move;
1062
1063                    }
1064                    v = Math.max(v,max);
1065                    alpha = Math.max(alpha, v);
1066                    if(alpha >= beta){
1067                        break;
1068                    }
1069                }
1070                return v;
1071            }
1072            else{
1073                int v = 10000;
1074                for(Move m: legalMoves){
1075                    int min = minimax(board.getMoveResults(m),depth-1,alpha,beta,!isWhite);
```

```
1076                v = Math.min(v,min);
1077                beta = Math.min(beta, v);
1078                if(alpha >= beta){
1079                    break;
1080                }
1081            }
1082            return v;
1083        }
1084    }
1085 }
```

Enum: PieceType

```
1086 /**
1087  * Types a piece can be.
1088  */
1089 enum PieceType{
1090     /**
1091     * Used when a Piece is not a king.
1092     */
1093     PIECE,
1094     /**
1095     * Used when a piece is a king.
1096     */
1097     KING;
1098 }
```

Enum: PieceColor

```
1099 /**
1100  * Colours a piece can be.
1101  */
1102 public enum PieceColor
1103 {
1104     /**
1105     * Used when a piece is white.
1106     */
```

```
    * Used for red players pieces.
    */
    RED,
    /**
    * Used for white players pieces.
    */
    WHITE
}
```

Enum: MoveType

```
/**
 * Used to seperate normal and kill moves.
 */
public enum MoveType
{
    /**
    * Used for normal moves.
    */
    NORMAL,
    /**
    * Used for moves with a kill included.
    */
    KILL
}
```