CIS 163 – Computer Science II
Project 2: Chess Game

Student Name	
Due Date	

Graded Item	Pts	Points Awarded
Javadoc comments and coding style/technique (http://www.cis.gvsu.edu/studentsupport/javaguide) ● Code Indentation (auto format source code in IDE) ● Naming Conventions (see Java style guide) ● Proper access modifiers for fields and methods ● Use of helper (private) methods ● Using good variable names ● Header/class comments ● Every method uses @param and @return ● Every method uses a /********** separator ● Overall layout, readability, No text wrap ● Using /** … / for each Instance variable ● Has many inner "inner" comments	10	
Steps 1 – 5: Base Functionality ▪ Model/View separation ▪ Functioning GUI ▪ Initial chess board is set up correctly ▪ `Pawn` and `Rook` pieces move correctly ▪ `King`, `Queen`, `Knight`, and `Bishop`	20	
Step 6 : Additional functionality ▪ King piece moves correctly ▪ Queen piece moves correctly ▪ Knight piece moves correctly ▪ Bishop piece moves correctly	10	
Step 7: Additional functionality ▪ inCheck() of ChessModel class	10	
Step 8: CheckMate ▪ isComplete() of ChessModel class	10	
Steps 9 – 10: GUI	10	
Step 11: Undo	10	
Step 12: AI	10	
Step 12: JUNIT testing	10	
Total	**100**	

Comments:

```java
package chess;

import java.util.ArrayList;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Joseph Lentine
 * CIS 163 Winter 2022
 * Project 2
 *
 * ChessModel class, uses varoius piece Class logic to play a full game
 * of chess
 *
 */
public class ChessModel implements IChessModel {

    /** array of arrays that represents the chess board */
    private IChessPiece[][] board;

    /** current player */
    private Player player;

    /** keeps track of what turn it is */
    private int turn;

    /** array of SaveStates that holds data for all prior moves */
    private ArrayList<SaveState> prevMoves;

    /** keeps track of if an en passant move can be done this turn */
    private boolean canEnPassant;

    /** holds all data on if castling can be done */
    private CastlingData canCastle;

    /**********************************************************************
     * default constructor. sets up board and instantiates variables
     *
     */
    public ChessModel() {
        board = new IChessPiece[8][8];
        player = Player.WHITE;
        turn = 0;
        prevMoves = new ArrayList<>(0);
        canEnPassant = false;
        canCastle = new CastlingData();

        board[7][0] = new Rook(Player.WHITE);
        board[7][1] = new Knight(Player.WHITE);
        board[7][2] = new Bishop(Player.WHITE);
        board[7][3] = new Queen(Player.WHITE);
        board[7][4] = new King(Player.WHITE);
        board[7][5] = new Bishop(Player.WHITE);
        board[7][6] = new Knight(Player.WHITE);
        board[7][7] = new Rook(Player.WHITE);

        board[6][0] = new Pawn(Player.WHITE);
```

```
 56          board[6][1] = new Pawn(Player.WHITE);
 57          board[6][2] = new Pawn(Player.WHITE);
 58          board[6][3] = new Pawn(Player.WHITE);
 59          board[6][4] = new Pawn(Player.WHITE);
 60          board[6][5] = new Pawn(Player.WHITE);
 61          board[6][6] = new Pawn(Player.WHITE);
 62          board[6][7] = new Pawn(Player.WHITE);
 63
 64          board[0][0] = new Rook(Player.BLACK);
 65          board[0][1] = new Knight(Player.BLACK);
 66          board[0][2] = new Bishop(Player.BLACK);
 67          board[0][3] = new Queen(Player.BLACK);
 68          board[0][4] = new King(Player.BLACK);
 69          board[0][5] = new Bishop(Player.BLACK);
 70          board[0][6] = new Knight(Player.BLACK);
 71          board[0][7] = new Rook(Player.BLACK);
 72
 73          board[1][0] = new Pawn(Player.BLACK);
 74          board[1][1] = new Pawn(Player.BLACK);
 75          board[1][2] = new Pawn(Player.BLACK);
 76          board[1][3] = new Pawn(Player.BLACK);
 77          board[1][4] = new Pawn(Player.BLACK);
 78          board[1][5] = new Pawn(Player.BLACK);
 79          board[1][6] = new Pawn(Player.BLACK);
 80          board[1][7] = new Pawn(Player.BLACK);
 81
 82      }
 83
 84      /**********************************************************************
 85       ** Returns whether the game is complete.
 86       *
 87       * @return {@code true} if complete, {@code false} otherwise.
 88       */
 89      public boolean isComplete() {
 90          boolean valid = false;
 91          Move test;
 92          if (inCheck(player)) {
 93              if (noValidMoves()) {
 94                  return true;
 95              }
 96          }
 97          return valid;
 98      }
 99
100      /**********************************************************************
101       * Returns whether or not there are any legal moves available.
102       *
103       * @return true if complete, false otherwise.
104       */
105      public boolean noValidMoves() {
106          boolean valid = true;
107          Move test;
108          Player p = player;
109
110          // go through rows
```

```
111         for (int a = 0; a < 8; ++a) {
112             // go through columns
113             for (int b = 0; b < 8; ++b) {
114                 // check player for piece in location
115                 if (board[a][b] != null && board[a][b].player() == p) {
116                     // go through rows
117                     for (int c = 0; c < 8; ++c) {
118                         // go through columns
119                         for (int d = 0; d < 8; ++d) {
120                             // make new move
121                             test = new Move(a, b, c, d);
122                             // see if new move is valid
123                             if (isValidMove(test)) {
124                                 valid = false;
125                                 a = b = c = d = 8;
126                             }
127                         }
128                     }
129                 }
130             }
131         }
132 
133         return valid;
134     }
135 
136     /**********************************************************************
137      * Returns whether the piece at location
138      * {@code [move.fromRow, move.fromColumn]} is allowed to move to
139      * location {@code [move.fromRow, move.fromColumn]}. also checks
140      * for if move is a valid en passant or castling move
141      *
142      * @param move a {@link chess.Move} object describing the
143      *                move to be made.
144      * @return {@code true} if the proposed move is valid,
145      *                {@code false} otherwise.
146      * @throws IndexOutOfBoundsException if either
147      *                 {@code [move.fromRow, move.fromColumn]} or
148      *                 {@code [move.toRow, move.toColumn]} don't
149      *                 represent valid locations on the board.
150      */
151     public boolean isValidMove(Move move) {
152         // checks if move is out of bounds
153         if (move.fromRow < 0 || move.fromRow > 7 || move.fromColumn < 0
154                 || move.fromColumn > 7 || move.toRow < 0
155                 || move.toRow > 7 || move.toColumn < 0
156                 || move.toColumn > 7) {
157             throw new IndexOutOfBoundsException();
158         }
159 
160         boolean valid = false;
161 
162         // makes sure from location is not null and piece at from
163         // location is owned by the active player
164         if (board[move.fromRow][move.fromColumn] != null
165                 && board[move.fromRow][move.fromColumn].player()
```

```
166                 == player) {
167             //checks if the move is valid to the piece, an en passant
168             // or castling
169             if (isEnPassant(move, board) ||
170                     board[move.fromRow][move.fromColumn].
171                             isValidMove(move, board) ||
172                     isValidCastling(move, board)) {
173                 // tries move
174                 move(move);
175                 // checks if move will put active player into check
176                 if (!inCheck(player.next())) {
177                     valid = true;
178                 }
179                 // undoes move
180                 undo();
181             }
182         }
183 
184         return valid;
185     }
186 
187     /*****************************************************************
188      * Moves the piece from location {@code [move.fromRow,
189      * move.fromColumn]} to location {@code [move.fromRow,
190      * move.fromColumn]}. handles special movement for en passant and
191      * castling. then updates castling data, turn count, and active
192      * player
193      *
194      * @param move a {@link chess.Move} object describing the
195      *             move to be made.
196      * @throws IndexOutOfBoundsException if either
197      *             {@code [move.fromRow, move.fromColumn]} or
198      *             {@code [move.toRow, move.toColumn]} don't
199      *             represent valid locations on the board.
200      */
201     public void move(Move move) {
202         // checks if move is out of bounds
203         if (move.fromRow < 0 || move.fromRow > 7 || move.fromColumn < 0
204                 || move.fromColumn > 7 || move.toRow < 0
205                 || move.toRow > 7 || move.toColumn < 0
206                 || move.toColumn > 7) {
207             throw new IndexOutOfBoundsException();
208         }
209 
210         // checks if move is en passant
211         if (isEnPassant(move, board)) {
212             // saves move state to prevMoves array
213             prevMoves.add(new SaveState(move, board));
214             prevMoves.get(prevMoves.size()-1).
215                     saveCastlingData(canCastle);
216             // performs en passant move
217             board[move.toRow][move.toColumn] =
218                     board[move.fromRow][move.fromColumn];
219             board[move.fromRow][move.fromColumn] = null;
220             board[move.fromRow][move.toColumn] = null;
```

```
221             // finishes saving board state
222             prevMoves.get(prevMoves.size()-1).setEnPassant(true);
223         }
224         else if (isValidCastling(move, board)) {
225             // saves move state to prevMoves array
226             prevMoves.add(new SaveState(move, board));
227             prevMoves.get(prevMoves.size()-1).
228                     saveCastlingData(canCastle);
229             // checks if black is making move
230             if (move.fromRow == 0) {
231                 // checks if queenside castling
232                 if ((move.fromColumn - move.toColumn) > 0) {
233                     board[0][0] = null;
234                     board[0][4] = null;
235                     board[0][3] = new Rook(Player.BLACK);
236                     board[0][2] = new King(Player.BLACK);
237                 }
238                 // runs kingside castling
239                 else {
240                     board[0][7] = null;
241                     board[0][4] = null;
242                     board[0][5] = new Rook(Player.BLACK);
243                     board[0][6] = new King(Player.BLACK);
244                 }
245             }
246             // checks if white is making move
247             if (move.fromRow == 7) {
248                 // checks if queenside castling
249                 if ((move.fromColumn - move.toColumn) > 0) {
250                     board[7][0] = null;
251                     board[7][4] = null;
252                     board[7][3] = new Rook(Player.WHITE);
253                     board[7][2] = new King(Player.WHITE);
254                 }
255                 // runs kingside castling
256                 else {
257                     board[7][7] = null;
258                     board[7][4] = null;
259                     board[7][5] = new Rook(Player.WHITE);
260                     board[7][6] = new King(Player.WHITE);
261                 }
262             }
263             // finishes saving move state
264             prevMoves.get(prevMoves.size()-1).setWasCastling(true);
265         }
266         // performs regular move if neither en passant or castling
267         else {
268             // saves move state to prevMoves array
269             prevMoves.add(new SaveState(move, board));
270             prevMoves.get(prevMoves.size()-1).
271                     saveCastlingData(canCastle);
272             // performs move
273             board[move.toRow][move.toColumn] =
274                     board[move.fromRow][move.fromColumn];
275             board[move.fromRow][move.fromColumn] = null;
```

```
276         }
277
278         // checks if next move can be an en passant
279         if (board[move.toRow][move.toColumn].type().equals("Pawn")
280                 && Math.abs(move.fromRow - move.toRow) == 2) {
281             canEnPassant = true;
282         }
283         else {
284             canEnPassant = false;
285         }
286
287         updateCastlingData(move, board);
288         setNextPlayer();
289         turn++;
290
291     }
292
293     /**********************************************************************
294      * runs the given move regardless of validity
295      *
296      * @param move given move
297      */
298     private void tryMove(Move move) {
299         board[move.toRow][move.toColumn] =
300                 board[move.fromRow][move.fromColumn];
301         board[move.fromRow][move.fromColumn] = null;
302     }
303
304     /**********************************************************************
305      * Report whether the current player p is in check.
306      * @param  p {@link chess.Move} the Player being checked
307      * @return {@code true} if the current player is in check,
308      * {@code false} otherwise.
309      */
310     public boolean inCheck(Player p) {
311         boolean valid = false;
312         int kingRow = -1;
313         int kingCol = -1;
314         Move test;
315
316         // loop through board to find king
317         for (int a = 0; a < 8; ++a) {
318             for (int b = 0; b < 8; ++b) {
319                 if (board[a][b] != null && board[a][b].player() == p
320                         && board[a][b].type().equals("King")) {
321                     kingRow = a;
322                     kingCol = b;
323                     a = 8;
324                     b = 8;
325                 }
326             }
327         }
328
329         // loop through board checking if any piece has a valid move
330         // to king's location
```

```
331         for (int i = 0; i < 8; ++i) {
332             for (int j = 0; j < 8; ++j) {
333                 test = new Move(i, j, kingRow, kingCol);
334                 if (board[i][j] != null
335                         && board[i][j].isValidMove(test, board)) {
336                     valid = true;
337                     i = 8;
338                     j = 8;
339                 }
340             }
341         }
342 
343         return valid;
344     }
345 
346     /*****************************************************************
347      * undoes the previous action
348      *
349      */
350     public void undo() {
351         // can't undo if no moves have been done
352         if (turn == 0) {
353             return;
354         }
355 
356         // looks up previous move data
357         SaveState save = prevMoves.get(prevMoves.size()-1);
358 
359         //sets canCastle to saved castling data
360         canCastle = save.data;
361 
362         //sets pieces to previous spots
363         board[save.move.fromRow][save.move.fromColumn] =
364                 save.fromPiece;
365         board[save.move.toRow][save.move.toColumn] = save.toPiece;
366 
367         // sets canEnPassant to saved data
368         if (save.wasEnPassant) {
369             canEnPassant = true;
370             if (save.fromPiece.player() == Player.BLACK) {
371                 board[save.move.toRow - 1][save.move.toColumn] =
372                         new Pawn(Player.WHITE);
373             }
374             else {
375                 board[save.move.toRow + 1][save.move.toColumn] =
376                         new Pawn(Player.BLACK);
377             }
378         }
379         else {
380             canEnPassant = false;
381         }
382 
383         // undoes castling if castling was done
384         if (save.wasCastling) {
385             if (save.move.fromRow == 0) {
```

```
386                 if (save.move.fromColumn - save.move.toColumn > 0) {
387                     board[0][0] = new Rook(Player.BLACK);
388                     board[0][3] = null;
389                 }
390                 else {
391                     board[0][7] = new Rook(Player.BLACK);
392                     board[0][5] = null;
393                 }
394             }
395             if (save.move.fromRow == 7) {
396                 if (save.move.fromColumn - save.move.toColumn > 0) {
397                     board[7][0] = new Rook(Player.WHITE);
398                     board[7][3] = null;
399                 }
400                 else {
401                     board[7][7] = new Rook(Player.WHITE);
402                     board[7][5] = null;
403                 }
404             }
405         }
406         // deletes last SaveState in prevMoves array
407         prevMoves.remove(prevMoves.size()-1);
408         // decrements turn count
409         turn--;
410         // switches player
411         setNextPlayer();
412     }
413 
414     /*****************************************************************
415      * checks if there is a promotable pawn in a back row
416      * @return returns true if a pawn can be promoted
417      */
418     public boolean canPromote() {
419         // checks if there is a pawn in a back row
420         for (int i = 0; i < 8; ++i) {
421             if (board[0][i] != null) {
422                 if (board[0][i].type().equals("Pawn") &&
423                         board[0][i].player().equals(Player.WHITE)) {
424                     return true;
425                 }
426             }
427             if (board[7][i] != null) {
428                 if (board[7][i].type().equals("Pawn") &&
429                         board[7][i].player().equals(Player.BLACK)) {
430                     return true;
431                 }
432             }
433         }
434         return false;
435     }
436 
437     /*****************************************************************
438      * promotes the first promotable pawn to the type given
439      *
440      * @param type string with the type to be promoted to
```

```
441      */
442     public void promote(String type) {
443         Player p = player.next();
444         IChessPiece promotion = new Pawn(p);
445 
446         switch (type) {
447             case "Rook":
448                 promotion = new Rook(p);
449                 break;
450             case "Knight":
451                 promotion = new Knight(p);
452                 break;
453             case "Bishop":
454                 promotion = new Bishop(p);
455                 break;
456             case "Queen":
457                 promotion = new Queen(p);
458                 break;
459         }
460         for (int i = 0; i < 8; ++i) {
461             if (board[0][i] != null) {
462                 if (board[0][i].type().equals("Pawn") &&
463                         board[0][i].player().equals(Player.WHITE)) {
464                     board[0][i] = promotion;
465                 }
466             }
467             if (board[7][i] != null) {
468                 if (board[7][i].type().equals("Pawn") &&
469                         board[7][i].player().equals(Player.BLACK)) {
470                     board[7][i] = promotion;
471                 }
472             }
473         }
474 
475     }
476 
477     /*****************************************************************
478      * returns true if the move provided on the board provided is valid
479      * en passant, false otherwise
480      *
481      * @param move given move
482      * @param board given board
483      * @return returns true if move is valid en passant
484      */
485     public boolean isEnPassant(Move move, IChessPiece[][] board) {
486         boolean valid = false;
487         // checks if turn is zero
488         if (turn == 0) {
489             return valid;
490         }
491         // accesses the previous move data
492         SaveState lastTurn = prevMoves.get(prevMoves.size() - 1);
493         // checks if the move can be an en passant
494         if (!canEnPassant) {
495             return valid;
```

```
496         }
497         // checks if the last piece to move was a pawn
498         if (!board[move.fromRow][move.fromColumn].type()
499                 .equals("Pawn")) {
500             return valid;
501         }
502         // checks if player is white
503         if (player == Player.WHITE) {
504             // checks if the moving piece is in row 3 (the only row
505             // white can en passant from)
506             if (move.fromRow == 3) {
507                 // checks if the pawn is moving to a spot behind the
508                 // pawn that moved last turn
509                 if (move.toColumn == lastTurn.move.fromColumn &&
510                         Math.abs(move.fromColumn - move.toColumn) == 1
511                         && (move.fromRow - move.toRow) == 1) {
512                     valid = true;
513                 }
514             }
515         }
516         // checks if player is black
517         else if (player == Player.BLACK) {
518             // checks if the moving piece is in row 4 ( the only row
519             // black can en passant from)
520             if (move.fromRow == 4) {
521                 // checks if the pawn is moving to a spot behind the
522                 // pawn that moved last turn
523                 if (move.toColumn == lastTurn.move.fromColumn &&
524                         Math.abs(move.fromColumn - move.toColumn) == 1
525                         && (move.fromRow - move.toRow) == -1) {
526                     valid = true;
527                 }
528             }
529         }
530         return valid;
531     }
532 
533     /**********************************************************************
534      * returns true if the move provided on the board provided is valid
535      * castling, false otherwise
536      *
537      * @param move move data
538      * @param board board data
539      * @return returns true if move is valid castling
540      */
541     public boolean isValidCastling(Move move, IChessPiece[][] board) {
542         boolean valid = false;
543         // checks if player is in check (cannot castle if in check)
544         if (inCheck(player)) {
545             return valid;
546         }
547         // checks if the move is two spaces to the right or left
548         if (!(Math.abs(move.fromColumn - move.toColumn) == 2 &&
549                 move.fromRow == move.toRow)) {
550             return valid;
```

```
551         }
552         // checks white logic
553         if (player == Player.WHITE) {
554             // checks if the white king has moved this game
555             if (canCastle.whiteKingMoved) {
556                 return valid;
557             }
558             // logic if movement is to right
559             if ((move.fromColumn - move.toColumn) < 0) {
560                 // checks if white rook has moved this game
561                 if (!canCastle.whiteRightRookMoved) {
562                     // checks that the spaces between the rook and king
563                     // are empty
564                     if (board[7][5] != null || board[7][6] != null) {
565                         return valid;
566                     }
567                     // checks that the king would not be in check in
568                     // either of the spaces it is moving through
569                     Move move1 = new Move(7, 4, 7, 5);
570                     boolean move1Valid = false;
571                     Move move2 = new Move(7, 4, 7, 6);
572                     boolean move2Valid = false;
573                     tryMove(move1);
574                     if (!inCheck(player)) {
575                         move1Valid = true;
576                     }
577                     board[move1.fromRow][move1.fromColumn] =
578                             new King(Player.WHITE);
579                     board[move1.toRow][move1.toColumn] = null;
580                     tryMove(move2);
581                     if (!inCheck(player)) {
582                         move2Valid = true;
583                     }
584                     board[move2.fromRow][move2.fromColumn] =
585                             new King(Player.WHITE);
586                     board[move2.toRow][move2.toColumn] = null;
587                     if (move1Valid && move2Valid) {
588                         valid = true;
589                     }
590                 }
591             }
592             // logic if movement is to left
593             if ((move.fromColumn - move.toColumn) > 0) {
594                 // checks if left rook has moved this game
595                 if (!canCastle.whiteLeftRookMoved) {
596                     // checks that the spaces between the rook and king
597                     // are empty
598                     if (board[7][3] != null || board[7][2] != null
599                             || board[7][1] != null) {
600                         return valid;
601                     }
602                     // checks that the king would not be in check in
603                     // either of the spaces it is moving through
604                     Move move1 = new Move(7, 4, 7, 3);
605                     boolean move1Valid = false;
```

```
606                     Move move2 = new Move(7, 4, 7, 2);
607                     boolean move2Valid = false;
608                     tryMove(move1);
609                     if (!inCheck(player)) {
610                         move1Valid = true;
611                     }
612                     board[move1.fromRow][move1.fromColumn] =
613                             new King(Player.WHITE);
614                     board[move1.toRow][move1.toColumn] = null;
615
616                     tryMove(move2);
617                     if (!inCheck(player)) {
618                         move2Valid = true;
619                     }
620                     board[move2.fromRow][move2.fromColumn] =
621                             new King(Player.WHITE);
622                     board[move2.toRow][move2.toColumn] = null;
623
624                     if (move1Valid && move2Valid) {
625                         valid = true;
626                     }
627                 }
628             }
629         }
630         // logic for if player is black
631         if (player == Player.BLACK) {
632             // checks if king has moved this game
633             if (canCastle.blackKingMoved) {
634                 return valid;
635             }
636             // logic if movement is to right
637             if ((move.fromColumn - move.toColumn) < 0) {
638                 // checks if right rook has moved this game
639                 if (!canCastle.blackRightRookMoved) {
640                     // checks that the spaces between the king and rook
641                     // are empty
642                     if (board[0][5] != null || board[0][6] != null) {
643                         return valid;
644                     }
645                     // checks that the king would not be in check in
646                     // either of the spaces it is moving through
647                     Move move1 = new Move(0, 4, 0, 5);
648                     boolean move1Valid = false;
649                     Move move2 = new Move(0, 4, 0, 6);
650                     boolean move2Valid = false;
651                     tryMove(move1);
652                     if (!inCheck(player)) {
653                         move1Valid = true;
654                     }
655                     board[move1.fromRow][move1.fromColumn] =
656                             new King(Player.BLACK);
657                     board[move1.toRow][move1.toColumn] = null;
658                     tryMove(move2);
659                     if (!inCheck(player)) {
660                         move2Valid = true;
```

```
661                     }
662                     board[move2.fromRow][move2.fromColumn] =
663                             new King(Player.BLACK);
664                     board[move2.toRow][move2.toColumn] = null;
665                     if (move1Valid && move2Valid) {
666                         valid = true;
667                     }
668                 }
669             }
670             // logic if movement is to left
671             if ((move.fromColumn - move.toColumn) > 0) {
672                 // checks if left rook has moved this game
673                 if (!canCastle.blackLeftRookMoved) {
674                     // checks that the spaces between the king and rook
675                     // are empty
676                     if (board[0][3] != null || board[0][2] != null ||
677                             board[0][1] != null) {
678                         return valid;
679                     }
680                     // checks that the king would not be in check in
681                     // either of the spaces it is moving through
682                     Move move1 = new Move(0, 4, 0, 3);
683                     boolean move1Valid = false;
684                     Move move2 = new Move(0, 4, 0, 2);
685                     boolean move2Valid = false;
686                     tryMove(move1);
687                     if (!inCheck(player)) {
688                         move1Valid = true;
689                     }
690                     board[move1.fromRow][move1.fromColumn] =
691                             new King(Player.BLACK);
692                     board[move1.toRow][move1.toColumn] = null;
693                     tryMove(move2);
694                     if (!inCheck(player)) {
695                         move2Valid = true;
696                     }
697                     board[move2.fromRow][move2.fromColumn] =
698                             new King(Player.BLACK);
699                     board[move2.toRow][move2.toColumn] = null;
700
701                     if (move1Valid && move2Valid) {
702                         valid = true;
703                     }
704                 }
705             }
706         }
707
708         return valid;
709     }
710
711     /**********************************************************************
712      * updates if the kings or rooks have moved from their starting
713      * position
714      *
715      * @param move move data
```

```
716      * @param board board state
717      */
718     public void updateCastlingData(Move move, IChessPiece[][] board) {
719         if (move.fromRow == 0 && move.fromColumn == 0 &&
720                 board[move.toRow][move.toColumn].type().equals("Rook"))
721         { canCastle.setBlackLeftRookMoved(true); }
722         else if (move.fromRow == 0 && move.fromColumn == 7 &&
723                 board[move.toRow][move.toColumn].type().equals("Rook"))
724         { canCastle.setBlackRightRookMoved(true); }
725         else if (move.fromRow == 0 && move.fromColumn == 4 &&
726                 board[move.toRow][move.toColumn].type().equals("King"))
727         { canCastle.setBlackKingMoved(true); }
728         else if (move.fromRow == 7 && move.fromColumn == 0 &&
729                 board[move.toRow][move.toColumn].type().equals("Rook"))
730         { canCastle.setWhiteLeftRookMoved(true); }
731         else if (move.fromRow == 7 && move.fromColumn == 7 &&
732                 board[move.toRow][move.toColumn].type().equals("Rook"))
733         { canCastle.setWhiteRightRookMoved(true); }
734         else if (move.fromRow == 7 && move.fromColumn == 4 &&
735                 board[move.toRow][move.toColumn].type().equals("King"))
736         { canCastle.setWhiteKingMoved(true); }
737     }
738
739     /**********************************************************************
740      * Return the current player.
741      *
742      * @return the current player
743      */
744     public Player currentPlayer() {
745         return player;
746     }
747
748     /**********************************************************************
749      * Return the number of rows
750      *
751      * @return 8
752      */
753     public int numRows() {
754         return 8;
755     }
756
757     /**********************************************************************
758      * Return the number of columns
759      *
760      * @return 8
761      */
762     public int numColumns() {
763         return 8;
764     }
765
766     /**********************************************************************
767      * return piece at specified location
768      *
769      * @param row specified row
770      * @param column specified column
```

```
771      *
772      * @return returns piece at row, column
773      */
774     public IChessPiece pieceAt(int row, int column) {
775         return board[row][column];
776     }
777 
778     /*****************************************************************
779      * changes active player
780      *
781      */
782     public void setNextPlayer() {
783         player = player.next();
784     }
785 
786     /*****************************************************************
787      * places a piece at specified location
788      *
789      * @param row specified row
790      * @param column specified column
791      * @param piece specified piece
792      */
793     public void setPiece(int row, int column, IChessPiece piece) {
794         board[row][column] = piece;
795     }
796 
797     /*****************************************************************
798      * AI for black, holds unique moves for first and second turns, and
799      * logic to follow for later turns
800      *
801      */
802     public void AI() {
803         /*
804          * Write a simple AI set of rules in the following order.
805          * a. Check to see if you are in check.
806          *      i. If so, get out of check by moving the king or
807          *         placing a piece to block the check
808          *
809          * b. Attempt to put opponent into check (or checkmate).
810          *      i. Attempt to put opponent into check without losing
811          *         your piece
812          *      ii. Perhaps you have won the game.
813          *
814          *c. Determine if any of your pieces are in danger,
815          *      i. Move them if you can.
816          *      ii. Attempt to protect that piece.
817          *
818          *d. Move a piece (pawns first) forward toward opponent king
819          *      i. check to see if that piece is in danger of being
820          *         removed, if so, move a different piece.
821          */
822 
823         // check if it's black's first move
824         if (prevMoves.size() == 1) {
825             // grab the location of the piece that white just moved
```

```
826             int fromRow = prevMoves.get(0).move.fromRow;
827             int fromColumn = prevMoves.get(0).move.fromColumn;
828             // if they moved queenside pawn, respond with same
829             if (fromRow ==6 && fromColumn == 3) {
830                 move(new Move(1, 3, 3, 3));
831                 return;
832             }
833             // else move kingside pawn up one
834             else {
835                 move(new Move(1, 4, 2, 4));
836                 return;
837             }
838         }
839 
840         // check if it's black's second move
841         if (prevMoves.size() == 3) {
842             // get ints for white's last move
843             int fromRow = prevMoves.get(2).move.fromRow;
844             int fromColumn = prevMoves.get(2).move.fromColumn;
845             int toRow = prevMoves.get(2).move.toRow;
846             int toColumn = prevMoves.get(2).move.toColumn;
847             // if white continues into queen's gambit, respond with
848             // slab defense
849             if (fromRow == 6 && fromColumn == 2 && toRow == 4
850                     && toColumn == 2) {
851                 move(new Move(1, 2, 2, 2));
852                 return;
853             }
854         }
855 
856         if (!dodgeCheck()) {
857             if (!protectPiece()) {
858                 if (!attackOpponent()) {
859                     if (!releaseTheQueen()) {
860                         if (!pushTheRanks()) {
861                             if (!moveSomethingOtherThanTheKing()) {
862                                 if (!okayGottaMakeAnActualMove()) {
863                                     moveKing();
864                                 }
865                             }
866                         }
867                     }
868                 }
869             }
870         }
871 
872     }
873 
874     /*****************************************************************
875      * logic to get out of check if black king is in check
876      *
877      * @return returns true if move was performed
878      */
879     private boolean dodgeCheck() {
880         // see if black king is in check
```

```
881         if (inCheck(player)) {
882             // see if attacking will take the king out of check
883             // be aggressive
884             if (attackOpponent()) {
885                 return true;
886             }
887             Move test;
888             // go through rows
889             for (int a = 0; a < 8; ++a) {
890                 // go through columns
891                 for (int b = 0; b < 8; ++b) {
892                     // check for black piece
893                     if (board[a][b] != null && board[a][b].player()
894                             == player) {
895                         // go through rows
896                         for (int c = 0; c < 8; ++c) {
897                             // go through columns
898                             for (int d = 0; d < 8; ++d) {
899                                 // make new move
900                                 test = new Move(a, b, c, d);
901                                 // see if new move is valid
902                                 if (isValidMove(test)) {
903                                     // perform move and return true
904                                     move(test);
905                                     return true;
906                                 }
907                             }
908                         }
909                     }
910                 }
911             }
912         }
913         // return false if no move was done
914         return false;
915     }
916 
917     /*****************************************************************
918      * logic for if black piece is under attack. will perform a move if
919      * the piece being attacked can be protected without sacrificing
920      * the defending piece
921      *
922      * @return returns true if move was performed
923      */
924     private boolean protectPiece() {
925         Move test;
926         // go through rows
927         for (int a = 0; a < 8; ++a) {
928             // go through columns
929             for (int b = 0; b < 8; ++b) {
930                 // find white piece
931                 if (board[a][b] != null && board[a][b].player()
932                         == player.next()) {
933                     // go through rows
934                     for (int c = 0; c < 8; ++c) {
935                         // go through columns
```

```
936                         for (int d = 0; d < 8; ++d) {
937                             // find black piece that isn't a pawn
938                             if (board[c][d] != null && !board[c][d].
939                                     type().equals("Pawn") &&
940                                     board[c][d].player() == player) {
941                                 // make move for the white piece
942                                 // capturing the black piece
943                                 test = new Move(a, b, c, d);
944                                 // see if new move is valid. player
945                                 // needs to change as isValidMove
946                                 // checks to make sure the piece being
947                                 // moved is owned by the current player
948                                 player = player.next();
949                                 if (isValidMove(test)) {
950                                     // check if a move can be done to
951                                     // keep that piece from being
952                                     // captured
953                                     if (canProtect(test)) {
954                                         // return true if canProtect
955                                         // kept a move
956                                         return true;
957                                     }
958                                 }
959                                 // set player back to black
960                                 player = player.next();
961                             }
962                         }
963                     }
964                 }
965             }
966         }
967         // return false if no move is kept
968         return false;
969     }
970 
971     /**********************************************************************
972      * logic to check if potential white move can be protected against
973      *
974      * @param move move that white will perform to take piece
975      * @return returns true if move will successfully protect piece
976      *      without putting new piece at risk from the attacking white
977      *      piece
978      */
979     private boolean canProtect(Move move) {
980         Move test;
981         // go through rows
982         for (int a = 0; a < 8; ++a) {
983             // go through columns
984             for (int b = 0; b < 8; ++b) {
985                 // find black piece
986                 if (board[a][b] != null && board[a][b].player()
987                         == player.next()) {
988                     // go through rows
989                     for (int c = 0; c < 8; ++c) {
990                         // go through columns
```

```
 991                         for (int d = 0; d < 8; ++d) {
 992                             // switch to next player so isValidMove
 993                             // will check for moves valid to black
 994                             player = player.next();
 995                             // check all moves that black can do
 996                             test = new Move(a, b, c, d);
 997                             // see if new move is valid
 998                             if (isValidMove(test)) {
 999                                 // try out the move
1000                                 move(test);
1001                                 // see if the parameter move that was
1002                                 // given is no longer valid
1003                                 if (!isValidMove(move)) {
1004                                     // make a new move to see if white
1005                                     // piece that was threatening
1006                                     // will be able to capture the
1007                                     // black piece that just moved
1008                                     // to protect
1009                                     Move try2 = new Move(move.fromRow,
1010                                             move.fromColumn,
1011                                             test.toRow, test.toColumn);
1012                                     if (!isValidMove(try2)) {
1013                                         // return true if move will be
1014                                         // kept
1015                                         return true;
1016                                     }
1017                                 }
1018                                 // undo move if it doesn't meet all
1019                                 // criteria
1020                                 undo();
1021                             }
1022                             // set player back to white
1023                             player = player.next();
1024                         }
1025                     }
1026                 }
1027             }
1028         }
1029         // return false if no move was kept
1030         return false;
1031     }
1032
1033     /**********************************************************************
1034      * logic if black can take a piece. will not make the attack if the
1035      * piece will then be under attack
1036      *
1037      * @return returns true if move is made
1038      */
1039     private boolean attackOpponent() {
1040
1041         Move test;
1042         // go through rows
1043         for (int a = 0; a < 8; ++a) {
1044             // go through columns
1045             for (int b = 0; b < 8; ++b) {
```

```
1046                // check for black piece
1047                if (board[a][b] != null && board[a][b].player()
1048                        == player) {
1049                    // go through rows
1050                    for (int c = 0; c < 8; ++c) {
1051                        // go through columns
1052                        for (int d = 0; d < 8; ++d) {
1053                            // check for white piece
1054                            if (board[c][d] != null && board[c][d]
1055                                    .player() == player.next()) {
1056                                // make new move
1057                                test = new Move(a, b, c, d);
1058                                // see if new move is valid
1059                                if (isValidMove(test)) {
1060                                    // take the piece
1061                                    move(test);
1062                                    // check if piece moved into threat
1063                                    if (!movedIntoThreat(test)) {
1064                                        // return true if piece didn't move
1065                                        // into threat
1066                                        return true;
1067                                    }
1068                                    // undo if piece moved into threat
1069                                    undo();
1070                                }
1071                            }
1072                        }
1073                    }
1074                }
1075            }
1076        }
1077        // return false if no move was done
1078        return false;
1079    }
1080
1081    /*****************************************************************
1082     * logic to move the queen as far down-right as possible, without
1083     * it being under attack after the move.
1084     *
1085     * @return returns true if move is made
1086     */
1087    private boolean releaseTheQueen() {
1088        Move test;
1089        //go through rows
1090        for (int a = 7; a >= 0; --a) {
1091            // go through columns
1092            for (int b = 7; b >= 0; --b) {
1093                // find the black king
1094                if (board[a][b] != null && board[a][b].player()
1095                        == player
1096                        && board[a][b].type().equals("Queen")) {
1097                    // go through rows
1098                    for (int c = 7; c >= 0; --c) {
1099                        // go through columns
1100                        for (int d = 7; d >= 0; --d) {
```

```
1101                            // make new move
1102                            test = new Move(a, b, c, d);
1103                            // see if new move is valid
1104                            if (isValidMove(test)) {
1105                                // perform move
1106                                move(test);
1107                                // check if piece moved into threat
1108                                if (!movedIntoThreat(test)) {
1109                                    // return true if piece didn't move
1110                                    // into threat
1111                                    return true;
1112                                }
1113                                // undo if piece moved into threat
1114                                undo();
1115                            }
1116                        }
1117                    }
1118                }
1119            }
1120        }
1121        return false;
1122    }
1123
1124    /*****************************************************************
1125     * logic to find a pawn in the starting pawn row and move it
1126     * forward. starts checking on the right side, moves to left, will
1127     * move the piece as far down as possible
1128     *
1129     * @return returns true if move is made
1130     */
1131    private boolean pushTheRanks() {
1132        Move test;
1133        // go through columns
1134        for (int b = 7; b >= 0; --b) {
1135            // check player for non-king black piece
1136            if (board[1][b] != null && board[1][b].player()
1137                    == player && board[1][b].type()
1138                    .equals("Pawn")) {
1139                // go through rows
1140                for (int c = 7; c >= 0; --c) {
1141                    // go through columns
1142                    for (int d = 7; d >= 0; --d) {
1143                        // make new move
1144                        test = new Move(1, b, c, d);
1145                        // see if new move is valid
1146                        if (isValidMove(test)) {
1147                            // perform move
1148                            move(test);
1149                            // check if piece moved into threat
1150                            if (!movedIntoThreat(test)) {
1151                                // return true if piece didn't move
1152                                // into threat
1153                                return true;
1154                            }
1155                            // undo if piece moved into threat
```

```
1156                            undo();
1157                        }
1158                    }
1159                }
1160            }
1161        }
1162    // return false if no move was done
1163    return false;
1164    }
1165
1166    /*****************************************************************
1167     * moves the upper-left most piece as far down-right as possible.
1168     * does not attempt the move if it will be captured
1169     *
1170     * @return returns true if move is made
1171     */
1172    private boolean moveSomethingOtherThanTheKing() {
1173        Move test;
1174        for (int a = 0; a < 8; ++a) {
1175            // go through columns
1176            for (int b = 0; b < 8; ++b) {
1177                // check player for non-king black piece
1178                if (board[a][b] != null && board[a][b].player()
1179                        == player && !board[a][b].type()
1180                        .equals("King")) {
1181                    // go through rows
1182                    for (int c = 7; c >= 0; --c) {
1183                        // go through columns
1184                        for (int d = 7; d >= 0; --d) {
1185                            // make new move
1186                            test = new Move(a, b, c, d);
1187                            // see if new move is valid
1188                            if (isValidMove(test)) {
1189                                // perform move
1190                                move(test);
1191                                // check if piece moved into threat
1192                                if (!movedIntoThreat(test)) {
1193                                    // return true if piece didn't move
1194                                    // into threat
1195                                    return true;
1196                                }
1197                                // undo if piece moved into threat
1198                                undo();
1199                            }
1200                        }
1201                    }
1202                }
1203            }
1204        }
1205        // return false if no move was done
1206        return false;
1207    }
1208
1209    /*****************************************************************
1210     * logic to move a piece other than the king, regardless of if it
```

```
1211      * will get captured. the ai needs to make a move, in the end. will
1212      * move the upper-left most piece as far down-right as possible
1213      *
1214      * @return returns true if move is made
1215      */
1216     private boolean okayGottaMakeAnActualMove() {
1217         Move test;
1218         for (int a = 0; a < 8; ++a) {
1219             // go through columns
1220             for (int b = 0; b < 8; ++b) {
1221                 // check player for non-king black piece
1222                 if (board[a][b] != null && board[a][b].player()
1223                         == player && !board[a][b].type()
1224                         .equals("King")) {
1225                     // go through rows
1226                     for (int c = 7; c >= 0; --c) {
1227                         // go through columns
1228                         for (int d = 7; d >= 0; --d) {
1229                             // make new move
1230                             test = new Move(a, b, c, d);
1231                             // see if new move is valid
1232                             if (isValidMove(test)) {
1233                                 // perform move
1234                                 move(test);
1235                                 return true;
1236                             }
1237                         }
1238                     }
1239                 }
1240             }
1241         }
1242         // return false if no move was done
1243         return false;
1244     }
1245
1246     /**********************************************************************
1247      * logic to move the king if no other piece can move. this will
1248      * move the king to the top left of the board. will move the king
1249      * to the upper-left corner
1250      *
1251      */
1252     public void moveKing() {
1253         Move test;
1254         //go through rows
1255         for (int a = 7; a >= 0; --a) {
1256             // go through columns
1257             for (int b = 7; b >= 0; --b) {
1258                 // find the black king
1259                 if (board[a][b] != null && board[a][b].player()
1260                         == player
1261                         && board[a][b].type().equals("King")) {
1262                     // go through rows
1263                     for (int c = 0; c < 8; ++c) {
1264                         // go through columns
1265                         for (int d = 0; d < 8; ++d) {
```

```
1266                            // make new move
1267                            test = new Move(a, b, c, d);
1268                            // see if new move is valid
1269                            if (isValidMove(test)) {
1270                                // perform move
1271                                move(test);
1272                                return;
1273                            }
1274                        }
1275                    }
1276                }
1277            }
1278        }
1279    }
1280
1281    /*****************************************************************
1282     * checks to see if given move puts piece in risk of being captured
1283     *
1284     * @param move move just performed
1285     * @return returns true if piece is in threat
1286     */
1287    private boolean movedIntoThreat(Move move) {
1288        Move test;
1289        // go through rows
1290        for (int a = 0; a < 8; ++a) {
1291            // go through columns
1292            for (int b = 0; b < 8; ++b) {
1293                // find black piece
1294                if (board[a][b] != null && board[a][b].player()
1295                        == player) {
1296                    test = new Move(a, b, move.toRow, move.toColumn);
1297                    if (isValidMove(test)) {
1298                        return true;
1299                    }
1300                }
1301            }
1302        }
1303    return false;
1304    }
1305 }
```

File - C:\Users\Admin\IdeaProjects\CIS163Projects\src\chess\ChessPiece.java

```java
package chess;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Joseph Lentine
 * CIS 163 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * ChessPiece class that is extended by all other piece classes
 *
 */
public abstract class ChessPiece implements IChessPiece {

    /** the owner of the piece */
    private Player owner;

    /**********************************************************************
     * constructs a Chess piece and sets owner to the given parameter
     *
     * @param player the owner of the chess piece
     */
    protected ChessPiece(Player player) {
        this.owner = player;
    }

    /**********************************************************************
     * Return the type of this piece ("King", "Queen", "Rook", etc.).
     * Note:  In this case "type" refers to the game of chess, not
     * the type of the Java class.
     *
     * @return the type of this piece
     */
    @Override
    public abstract String type();

    /**********************************************************************
     * Return the player that owns this piece.
     *
     * @return the player that owns this piece.
     */
    @Override
    public Player player() {
        return owner;
    }

    /**********************************************************************
     * Returns whether the piece at location
     * {@code [move.fromRow, move.fromColumn]} is allowed to move to
     * location {@code [move.fromRow, move.fromColumn]}.
     *
     * Note:  Pieces don't store their own location
     * (because doing so would be redundant).  Therefore, the
     * {@code [move.fromRow, move.fromColumn]} component of
     * {@code move} is necessary. {@code this} object must be the piece
     * at location {@code [move.fromRow, move.fromColumn]}.
```

```
 56      * (This method makes no sense otherwise.)
 57      *
 58      * @param move  a {@link chess.Move} object describing the
 59      *                  move to be made.
 60      * @param board the {@link chess.IChessPiece} in which this
 61      *                  piece resides.
 62      * @return {@code true} if the proposed move is valid,
 63      *                  {@code false}otherwise.
 64      * @throws IndexOutOfBoundsException if either {@code
 65      *          [move.fromRow, move.fromColumn]} or {@code [move.toRow,
 66      *          move.toColumn]} don't represent valid locations on the
 67      *          board.
 68      * @throws IllegalArgumentException  if {@code this} object isn't
 69      *          the piece at location {@code [move.fromRow,
 70      *          move.fromColumn]}.
 71      */
 72     @Override
 73     public boolean isValidMove(Move move, IChessPiece[][] board) {
 74         boolean valid = true;
 75
 76         if (move.fromRow < 0 || move.fromRow > 7 || move.fromColumn < 0
 77                 || move.fromColumn > 7 || move.toRow < 0
 78                 || move.toRow > 7 || move.toColumn < 0
 79                 || move.toColumn > 7) {
 80             throw new IndexOutOfBoundsException();
 81         }
 82
 83         if (((move.fromRow == move.toRow) &&
 84                 (move.fromColumn == move.toColumn))) {
 85             valid = false;
 86         }
 87
 88         if (board[move.fromRow][move.fromColumn] != this) {
 89             throw new IllegalArgumentException();
 90         }
 91
 92         if (board[move.toRow][move.toColumn] != null
 93                 && board[move.toRow][move.toColumn].player()
 94                 == this.player())
 95         {
 96             valid = false;
 97         }
 98
 99         return valid;
100     }
101 }
```

```java
package chess;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Joseph Lentine
 * CIS 163 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * Pawn class, implements the functions of a pawn in chess
 */

public class Pawn extends ChessPiece{

    /*****************************************************************
     * constructs a Pawn and sets owner to the given parameter
     *
     * @param player the owner of the chess piece
     */
    public Pawn(Player player) {
        super(player);
    }

    /*****************************************************************
     * Return the type of this piece ("King", "Queen", "Rook", etc.).
     * Note:  In this case "type" refers to the game of chess, not
     * the type of the Java class.
     *
     * @return the type of this piece
     */
    @Override
    public String type() {
        return "Pawn";
    }

    /*****************************************************************
     * Returns whether the piece at location
     * {@code [move.fromRow, move.fromColumn]} is allowed to move to
     * location {@code [move.fromRow, move.fromColumn]}.
     *
     * Note:  Pieces don't store their own location
     * (because doing so would be redundant).  Therefore, the
     * {@code [move.fromRow, move.fromColumn]} component of
     * {@code move} is necessary. {@code this} object must be the piece
     * at location {@code [move.fromRow, move.fromColumn]}.
     * (This method makes no sense otherwise.)
     *
     * @param move  a {@link chess.Move} object describing the
     *                  move to be made.
     * @param board the {@link chess.IChessPiece} in which this
     *                  piece resides.
     *
     * @return {@code true} if the proposed move is valid,
     *                  {@code false}otherwise.
     */
    @Override
```

```
 56     public boolean isValidMove(Move move, IChessPiece[][] board) {
 57         boolean valid = super.isValidMove(move, board);
 58         if (valid == false) { return valid; }
 59 
 60         // start with checking player. white moves upwards
 61         // fromRow - toRow will be positive
 62         if (player() == Player.WHITE) {
 63             // check if piece is staying in the same column
 64             if (move.fromColumn == move.toColumn) {
 65                 //check if there is a piece controlled by opponent in
 66                 //desired move location
 67                 if (board[move.toRow][move.toColumn] != null) {
 68                     valid = false;
 69                 }
 70                 // check if pawn is in starting row
 71                 if (move.fromRow == 6) {
 72                     // pawn can move one or two spaces from
 73                     // starting row
 74                     if (move.fromRow - move.toRow < 1
 75                             || move.fromRow - move.toRow > 2) {
 76                         valid = false;
 77                     }
 78                     // check that if pawn moves two spaces, it doesn't
 79                     // go through another piece
 80                     if (move.fromRow - move.toRow == 2 &&
 81                             board[move.fromRow - 1][move.toColumn]
 82                                     != null) {
 83                         valid = false;
 84                     }
 85 
 86                 }
 87                 // pawn is not in starting row
 88                 else {
 89                     // pawn can only move forward once
 90                     if (move.fromRow - move.toRow != 1) {
 91                         valid = false;
 92                     }
 93                 }
 94             }
 95             // pawn not staying in same column
 96             else {
 97                 // pawn can only move diagonally forward one space
 98                 if (Math.abs(move.fromColumn - move.toColumn) == 1
 99                         && move.fromRow - move.toRow == 1) {
100                     // check to make sure to location has a piece
101                     // controlled by other player
102                     if (board[move.toRow][move.toColumn] == null) {
103                         valid = false;
104                     }
105                 }
106                 else {
107                     valid = false;
108                 }
109             }
110         }
```

```
111         // pawn is black, which moves downwards
112         // fromRow - toRow will be negative
113         else {
114             // check if piece is staying in the same column
115             if (move.fromColumn == move.toColumn) {
116                 //check if there is a piece controlled by opponent in
117                 //desired move location
118                 if (board[move.toRow][move.toColumn] != null) {
119                     valid = false;
120                 }
121                 // check if pawn is in starting row
122                 if (move.fromRow == 1) {
123                     // pawn can move one or two spaces from
124                     // starting row
125                     if (move.fromRow - move.toRow > -1
126                             || move.fromRow - move.toRow < -2) {
127                         valid = false;
128                     }
129                     // check that if pawn moves two spaces, it doesn't
130                     // go through another piece
131                     if (move.fromRow - move.toRow == -2) {
132                         if (board[move.fromRow + 1][move.toColumn]
133                                 != null) {
134                             valid = false;
135                         }
136                     }
137                 }
138                 // pawn is not in starting row
139                 else {
140                     // pawn can only move forward once
141                     if (move.fromRow - move.toRow != -1) {
142                         valid = false;
143                     }
144                 }
145             }
146             // pawn not staying in same column
147             else {
148                 // pawn can only move diagonally forward one space
149                 if (Math.abs(move.fromColumn - move.toColumn) == 1
150                         && move.fromRow - move.toRow == -1) {
151                     // check to make sure move location has a piece
152                     // controlled by other player
153                     if (board[move.toRow][move.toColumn] == null) {
154                         valid = false;
155                     }
156                 }
157                 else {
158                     valid = false;
159                 }
160             }
161         }
162 
163         return valid;
164     }
165 
```

```
166 }
167
168
```

```java
package chess;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Joshep Lentine
 * CIS 163 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * Bishop class, implemets the functions of a bishop in chess
 */

public class Bishop extends ChessPiece {

    /*****************************************************************
     * Constructs a Bishop and sets owner to given parameter
     *
     * @param player
     */
    public Bishop(Player player) {
        super(player);
    }

    /*****************************************************************
     * Return the type of this piece ("King", "Queen", "Rook", etc.).
     * Note:  In this case "type" refers to the game of chess, not
     * the type of the Java class.
     *
     * @return the type of chess piece
     */
    @Override
    public String type() {
        return "Bishop";
    }

    /*****************************************************************
     * Returns whether the piece at location
     * {@code [move.fromRow, move.fromColumn]} is allowed to move to
     * location {@code [move.fromRow, move.fromColumn]}
     *
     * Note:  Pieces don't store their own location
     * (because doing so would be redundant).  Therefore, the
     * {@code [move.fromRow, move.fromColumn]} component of
     * {@code move} is necessary.{@code this} object must be the piece
     * at location {@code [move.fromRow, move.fromColumn]}.
     * (This method makes no sense otherwise.)
     *
     * @param move  a {@link chess.Move} object describing the
     *                  move to be made.
     * @param board the {@link chess.IChessPiece} in which this
     *                  piece resides.
     * @return {@code true} if the proposed move is valid,
     *              {@code false}otherwise.
     */
    @Override
    public boolean isValidMove(Move move, IChessPiece[][] board) {
```

```
 56          boolean valid = super.isValidMove(move, board);
 57          if(valid == false){
 58              return valid;
 59          }
 60
 61          //Bishop can only move diagonally unlimited spaces
 62          //meaning the amount of rows and columns moved should be equal
 63          if(Math.abs(move.fromRow - move.toRow) !=
 64                  Math.abs(move.fromColumn - move.toColumn)){
 65              return false;
 66          }
 67          //Checks if bishop is moving up left
 68          if(move.fromRow - move.toRow < 0 &&
 69                  move.fromColumn - move.toColumn > 0){
 70              for(int i = 1;
 71                  i < Math.abs(move.fromRow - move.toRow); i++){
 72                  if(board[move.fromRow + i][move.fromColumn - i]
 73                          != null){
 74                      valid = false;
 75                  }
 76              }
 77          }
 78          //Checks if bishop is move up right
 79          if(move.fromRow - move.toRow > 0 &&
 80                  move.fromColumn - move.toColumn > 0){
 81              for(int i = 1;
 82                  i < Math.abs(move.fromRow - move.toRow); ++i){
 83                  if(board[move.fromRow - i][move.fromColumn - i]
 84                          != null){
 85                      valid = false;
 86                  }
 87              }
 88          }
 89          //Checks if bishop is moving down left
 90          if(move.fromRow - move.toRow < 0 &&
 91                  move.fromColumn - move.toColumn < 0){
 92              for(int i = 1;
 93                  i < Math.abs(move.fromRow - move.toRow); i++){
 94                  if(board[move.fromRow + i][move.fromColumn + i]
 95                          != null){
 96                      valid = false;
 97                  }
 98              }
 99          }
100          //Bishop is moving down right
101          if(move.fromRow - move.toRow > 0 &&
102                  move.fromColumn - move.toColumn < 0){
103              for(int i = 1;
104                  i < Math.abs(move.fromRow - move.toRow); i++){
105                  if(board[move.fromRow - i][move.fromColumn + i]
106                          != null){
107                      valid = false;
108                  }
109              }
110          }
```

```
111         return valid;
112
113     }
114 }
115
```

```java
package chess;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Joseph Lentine
 * CIS 162 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * Rook class, implements the functions of Rook in chess
 */

public class Rook extends ChessPiece {

    /*****************************************************************
     * constructs a Rook and sets owner to the given parameter
     *
     * @param player the owner of the chess piece
     */
    public Rook(Player player) {
        super(player);
    }

    /*****************************************************************
     * Return the type of this piece ("King", "Queen", "Rook", etc.).
     * Note:  In this case "type" refers to the game of chess, not
     * the type of the Java class.
     *
     * @return the type of this piece
     */
    @Override
    public String type() {
        return "Rook";
    }

    /*****************************************************************
     * Returns whether the piece at location
     * {@code [move.fromRow, move.fromColumn]} is allowed to move to
     * location {@code [move.fromRow, move.fromColumn]}.
     *
     * Note:  Pieces don't store their own location
     * (because doing so would be redundant).  Therefore, the
     * {@code [move.fromRow, move.fromColumn]} component of
     * {@code move} is necessary. {@code this} object must be the piece
     * at location {@code [move.fromRow, move.fromColumn]}.
     * (This method makes no sense otherwise.)
     *
     * @param move  a {@link chess.Move} object describing the
     *                  move to be made.
     * @param board the {@link chess.IChessPiece} in which this
     *                  piece resides.
     *
     * @return {@code true} if the proposed move is valid,
     *                  {@code false}otherwise.
     */
    @Override
```

```
    public boolean isValidMove(Move move, IChessPiece[][] board) {
        // run ChessPiece's isValidMove first, if false return that
        boolean valid = super.isValidMove(move, board);
        if (!valid) { return valid; }

        // rook can only move in the same row or the same column
        // if both row and column have changed, the movement is invalid
        if ((move.toRow != move.fromRow)
                && (move.toColumn != move.fromColumn)) {
            valid = false;
        }

        // rook is moving in a column
        if (move.toRow == move.fromRow) {
            // if negative, piece is moving right
            if (Integer.signum((move.fromColumn - move.toColumn))
                    == -1) {
                // loop through all spaces between start and finish
                // to check for pieces in the way
                for (int i = move.fromColumn + 1; i < move.toColumn;
                     ++i){
                    if (board[move.toRow][i] != null) {
                        valid = false;
                        break;
                    }
                }
            }
            // if positive, piece is moving left
            else {
                // loop through all spaces between start and finish
                // to check for pieces in the way
                for (int i = move.fromColumn - 1; i > move.toColumn;
                     --i){
                    if (board[move.toRow][i] != null) {
                        valid = false;
                        break;
                    }
                }
            }
        }


        // rook is moving in a row
        if (move.toColumn == move.fromColumn) {
            //if negative, piece is moving down
            if (Integer.signum((move.fromRow - move.toRow)) == -1) {
                // loop through all spaces between start and finish
                // to check for pieces in the way
                for (int i = (move.fromRow + 1); i < move.toRow; ++i){
                    if (board[i][move.toColumn] != null) {
                        valid = false;
                        break;
                    }
                }
            }
```

```
111             // if positive, piece is moving up
112             else {
113                 // loop through all spaces between start and finish
114                 // to check for pieces in the way
115                 for (int i = (move.fromRow - 1); i > move.toRow; --i){
116                     if (board[i][move.toColumn] != null) {
117                         valid = false;
118                         break;
119                     }
120                 }
121             }
122         }
123 
124         return valid;
125 
126     }
127 
128 }
129 
```

```java
package chess;

/**********************************************************************
 * @author Dietsche, Kyle Scott, Joseph Lentine
 * CIS 163 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * Knight class implements the functions of a knight in chess
 */

public class Knight extends ChessPiece {

    /*****************************************************************
     * Constructs a Knight and sets owner to given parameter
     *
     * @param player the owner of the chess piece
     */
    public Knight(Player player) {
        super(player);
    }

    /*****************************************************************
     * Return the type of this piece ("King", "Queen", "Rook", etc.).
     * Note:  In this case "type" refers to the game of chess, not
     * the type of the Java class.
     *
     * @return the type of this piece
     */
    @Override
    public String type() {
        return "Knight";
    }

    /*****************************************************************
     * Returns whether the piece at location
     * {@code [move.fromRow, move.fromColumn]} is allowed to move to
     * location {@code [move.fromRow, move.fromColumn]}.
     *
     * Note:  Pieces don't store their own location
     * (because doing so would be redundant).  Therefore, the
     * {@code [move.fromRow, move.fromColumn]} component of
     * {@code move} is necessary. {@code this} object must be the piece
     * at location {@code [move.fromRow, move.fromColumn]}.
     * (This method makes no sense otherwise.)
     *
     * @param move  a {@link chess.Move} object describing the
     *                  move to be made.
     * @param board the {@link chess.IChessPiece} in which this
     *                  piece resides.
     *
     * @return {@code true} if the proposed move is valid,
     *                  {@code false}otherwise.
     */
    @Override
```

```
56      public boolean isValidMove(Move move, IChessPiece[][] board){
57
58          boolean valid = super.isValidMove(move, board);
59          if (valid == false) { return valid; }
60
61          valid = false;
62
63          if (Math.abs(move.fromRow - move.toRow) == 2
64                  && Math.abs(move.fromColumn - move.toColumn) == 1) {
65              valid = true;
66          }
67          if (Math.abs(move.fromRow - move.toRow) == 1
68                  && Math.abs(move.fromColumn - move.toColumn) == 2) {
69              valid = true;
70          }
71
72          return valid;
73
74      }
75
76  }
77
```

```java
package chess;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Joseph Lentine
 * CIS 163 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * King class implements the functions of a king in chess
 */

public class King extends ChessPiece {

    /**********************************************************************
     * constructs a King and sets owner to the given parameter
     *
     * @param player the owner of the chess piece
     */
    public King(Player player) {
        super(player);
    }

    /**********************************************************************
     * Return the type of this piece ("King", "Queen", "Rook", etc.).
     * Note:  In this case "type" refers to the game of chess, not
     * the type of the Java class.
     *
     * @return the type of this piece
     */
    @Override
    public String type() {
        return "King";
    }

    /**********************************************************************
     * Returns whether the piece at location
     * {@code [move.fromRow, move.fromColumn]} is allowed to move to
     * location {@code [move.fromRow, move.fromColumn]}.
     *
     * Note:  Pieces don't store their own location
     * (because doing so would be redundant).  Therefore, the
     * {@code [move.fromRow, move.fromColumn]} component of
     * {@code move} is necessary. {@code this} object must be the piece
     * at location {@code [move.fromRow, move.fromColumn]}.
     * (This method makes no sense otherwise.)
     *
     * @param move  a {@link chess.Move} object describing the
     *                  move to be made.
     * @param board the {@link chess.IChessPiece} in which this
     *                  piece resides.
     * @return {@code true} if the proposed move is valid,
     *                  {@code false}otherwise.
     */
    @Override
    public boolean isValidMove(Move move, IChessPiece[][] board) {
```

```
56          boolean valid = super.isValidMove(move, board);
57          if (valid == false) {
58              return valid;
59          }
60          if (move.fromColumn == move.toColumn) {
61              //King can move 1 space back and forward
62              // if staying in same column
63              if (Math.abs(move.fromRow - move.toRow) != 1) {
64                  valid = false;
65              }
66          }
67          //Checks if player is staying in same row
68          else if(move.fromRow == move.toRow){
69              //King can move 1 space left and right
70              // if staying in same row
71              if (Math.abs(move.fromColumn - move.toColumn) != 1) {
72                  valid = false;
73              }
74          }
75          //Checks if player is moving diagonally
76          else{
77              //King can move 1 space left and right
78              // and 1 space forward and back
79              if(Math.abs(move.fromRow - move.toRow) != 1
80                      || Math.abs(move.fromColumn - move.toColumn) != 1){
81                  valid = false;
82              }
83          }
84
85          return valid;
86      }
87  }
88
```

```java
package chess;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Joshep Lentine
 * CIS 163 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * Queen class implements the functions of a queen in chess
 */

public class Queen extends ChessPiece {

    /**********************************************************************
     * constructs a Queen and sets owner to the given parameter
     *
     * @param player the owner of the chess piece
     */
    public Queen(Player player) {
        super(player);
    }

    /**********************************************************************
     * Return the type of this piece ("King", "Queen", "Rook", etc.).
     * Note:  In this case "type" refers to the game of chess, not
     * the type of the Java class.
     *
     * @return the type of this piece
     */
    @Override
    public String type() {
        return "Queen";

    }

    /**********************************************************************
     * Returns whether the piece at location
     * {@code [move.fromRow, move.fromColumn]} is allowed to move to
     * location {@code [move.fromRow, move.fromColumn]}.
     *
     * Note:  Pieces don't store their own location
     * (because doing so would be redundant).  Therefore, the
     * {@code [move.fromRow, move.fromColumn]} component of
     * {@code move} is necessary. {@code this} object must be the piece
     * at location {@code [move.fromRow, move.fromColumn]}.
     * (This method makes no sense otherwise.)
     *
     * @param move  a {@link chess.Move} object describing the
     *                  move to be made.
     * @param board the {@link chess.IChessPiece} in which this
     *                  piece resides.
     *
     * @return {@code true} if the proposed move is valid,
     *                  {@code false}otherwise.
     */
```

```
56      @Override
57      public boolean isValidMove(Move move, IChessPiece[][] board) {
58          // create a bishop piece owned by the queen's player
59          Bishop move1 = new Bishop
60                  (board[move.fromRow][move.fromColumn].player());
61          // create a rook piece owned by the queen's player
62          Rook move2 = new Rook
63                  (board[move.fromRow][move.fromColumn].player());
64          // place the bishop on the board where the queen was;
65          board[move.fromRow][move.fromColumn] = move1;
66          // check if the bishop move logic could move to the desired
67          // location
68          boolean bishopIsValid = move1.isValidMove(move, board);
69          // place the rook on the board where the queen was
70          board[move.fromRow][move.fromColumn] = move2;
71          // check if the rook move logic could move to the desired
72          // location
73          boolean rookIsValid = move2.isValidMove(move, board);
74          // place the queen back on the board
75          board[move.fromRow][move.fromColumn] = this;
76          // return if either the bishop or the rook could perform the
77          // move
78          return (bishopIsValid || rookIsValid);
79
80
81
82      }
83  }
84
```

```java
package chess;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Joseph Lentine
 * CIS 163 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * holds move data, copies of the to and from pieces, if the move was
 * en passant or castling, and data on if castling can be done. this is
 * used in ChessModel to be able to undo moves.
 *
 */
public class SaveState {

    /** move stored */
    public Move move;

    /** piece stored in the move from location */
    public IChessPiece fromPiece;

    /** piece stored in the move to location */
    public IChessPiece toPiece;

    /** if the move stored was en passant */
    public boolean wasEnPassant;

    /** if the move stored was castling */
    public boolean wasCastling;

    /** what the CastlingData of the model was right before the move */
    public CastlingData data;

    /**********************************************************************
     * constructor that saves the given move and pieces at the to and
     * from locations
     *
     * @param move given move
     * @param board given board
     */
    public SaveState (Move move, IChessPiece[][] board) {
        this.move = move;
        this.fromPiece = copy(move.fromRow, move.fromColumn, board);
        this.toPiece = copy(move.toRow, move.toColumn, board);
        wasEnPassant = false;
        wasCastling = false;
        data = new CastlingData();
    }

    /**********************************************************************
     * returnss a new IChessPiece of the same piece type and player as
     * the piece in the given row and column, so that the piece stored
     * in this save state doesn't point at the old location
     *
     * @param row given row in the board
```

```
 56      * @param col given column in the board
 57      * @param board given board
 58      * @return returns new version of piece
 59      */
 60     public IChessPiece copy(int row, int col, IChessPiece[][] board) {
 61         if (board[row][col] != null) {
 62             String type = board[row][col].type();
 63             Player player = board[row][col].player();
 64             switch (type) {
 65                 case "Pawn":
 66                     return new Pawn(player);
 67                 case "Rook":
 68                     return new Rook(player);
 69                 case "Knight":
 70                     return new Knight(player);
 71                 case "Bishop":
 72                     return new Bishop(player);
 73                 case "Queen":
 74                     return new Queen(player);
 75                 case "King":
 76                     return new King(player);
 77             }
 78         }
 79         return null;
 80     }
 81 
 82     /**********************************************************************
 83      * sets wasCastling to the given boolean value
 84      *
 85      * @param wasCastling was the move being stored a castling move
 86      */
 87     public void setWasCastling(boolean wasCastling) {
 88         this.wasCastling = wasCastling;
 89     }
 90 
 91     /**********************************************************************
 92      * sets wasEnPassant to the given boolean value
 93      * @param wasEnPassant was the move being stored an en passant move
 94      */
 95     public void setEnPassant(boolean wasEnPassant) {
 96         this.wasEnPassant = wasEnPassant;
 97     }
 98 
 99     /**********************************************************************
100      * copies all the data from the given CastlingData into the
101      * CastlingData stored in this method
102      *
103      * @param d CastlingData being copied from
104      */
105     public void saveCastlingData(CastlingData d) {
106         data.setBlackRightRookMoved(d.blackRightRookMoved);
107         data.setBlackLeftRookMoved(d.blackLeftRookMoved);
108         data.setBlackKingMoved(d.blackKingMoved);
109         data.setWhiteKingMoved(d.whiteKingMoved);
110         data.setWhiteLeftRookMoved(d.whiteLeftRookMoved);
```

```
111         data.setWhiteRightRookMoved(d.whiteRightRookMoved);
112     }
113
114 }
```

```java
package chess;

/**********************************************************************
 * @author Dan Dietsche, Kyle Scott, Josheph Lentine
 * CIS 163 Winter 2022
 * 2/21/2022
 * Project 2
 *
 * holds data on if any of the kings or rooks have moved yet this game
 *
 */
public class CastlingData {

    /** if the black king has moved from its starting location */
    public boolean blackKingMoved;

    /** if the black right rook has moved from its starting location */
    public boolean blackRightRookMoved;

    /** if the black left rook has moved from its starting location */
    public boolean blackLeftRookMoved;

    /** if the white king has moved from its starting location */
    public boolean whiteKingMoved;

    /** if the white right rook has moved from its starting location */
    public boolean whiteRightRookMoved;

    /** if the white left rook has moved from its starting location */
    public boolean whiteLeftRookMoved;

    /**********************************************************************
     * default constructor that sets all variables to false
     *
     */
    public CastlingData() {
        blackKingMoved = false;
        blackRightRookMoved = false;
        blackLeftRookMoved = false;
        whiteKingMoved = false;
        whiteLeftRookMoved = false;
        whiteRightRookMoved = false;
    }

    /**********************************************************************
     * sets blackKingMoved to given boolean parameter
     *
     * @param blackKingMoved has the black king moved
     */
    public void setBlackKingMoved(boolean blackKingMoved) {
        this.blackKingMoved = blackKingMoved;
    }

    /**********************************************************************
     * sets blackRightRookMoved to given boolean parameter
```

```
56      *
57      * @param blackRightRookMoved has the black right rook moved
58      */
59     public void setBlackRightRookMoved(boolean blackRightRookMoved) {
60         this.blackRightRookMoved = blackRightRookMoved;
61     }
62
63     /**********************************************************************
64      * sets blackLeftRookMoved to given boolean parameter
65      *
66      * @param blackLeftRookMoved has the black left rook moved
67      */
68     public void setBlackLeftRookMoved(boolean blackLeftRookMoved) {
69         this.blackLeftRookMoved = blackLeftRookMoved;
70     }
71
72     /**********************************************************************
73      * sets whiteKingMoved to given boolean parameter
74      *
75      * @param whiteKingMoved has the white king moved
76      */
77     public void setWhiteKingMoved(boolean whiteKingMoved) {
78         this.whiteKingMoved = whiteKingMoved;
79     }
80
81     /**********************************************************************
82      * sets whiteRightRookMoved to given boolean parameter
83      *
84      * @param whiteRightRookMoved has the white right rook moved
85      */
86     public void setWhiteRightRookMoved(boolean whiteRightRookMoved) {
87         this.whiteRightRookMoved = whiteRightRookMoved;
88     }
89
90     /**********************************************************************
91      * sets whiteLeftRookMoved to given boolean parameter
92      *
93      * @param whiteLeftRookMoved has the white left rook moved
94      */
95     public void setWhiteLeftRookMoved(boolean whiteLeftRookMoved) {
96         this.whiteLeftRookMoved = whiteLeftRookMoved;
97     }
98 }
```

```java
package chess;

import org.junit.Test;

import static org.junit.Assert.*;

public class TestChess {


    // tests ChessPiece isValidMove logic
    @Test
    public void testChessPiece() {
        IChessPiece[][] board = new IChessPiece[8][8];

        board[6][2] = new Pawn(Player.WHITE);

        Move move = new Move(6, 2, 6, 2);

        assertFalse(board[6][2].isValidMove(move, board));

        move = new Move(6, 2, 5, 6);

        board[5][6] = new Queen(Player.WHITE);

        assertFalse(board[6][2].isValidMove(move, board));

    }

    // tests ChessPiece giving oob logic
    @Test(expected = IndexOutOfBoundsException.class)
    public void testChessPieceOOB() {
        IChessPiece[][] board = new IChessPiece[8][8];

        board[6][2] = new Pawn(Player.WHITE);

        Move move = new Move(5, 2, 6, -1);

        board[6][2].isValidMove(move, board);
    }

    //tests ChessPiece giving illegal argument
    @Test(expected = IllegalArgumentException.class)
    public void testChessPieceIllegalArg() {
        IChessPiece[][] board = new IChessPiece[8][8];

        board[6][2] = new Pawn(Player.WHITE);

        Move move = new Move(5, 2, 6, 2);

        board[6][2].isValidMove(move, board);
    }

    // tests that Pawn's white move logic is working
    @Test
    public void testPawnIsValidMoveWhite() {
```

```
 56          IChessPiece[][] board = new IChessPiece[8][8];
 57
 58          board[6][2] = new Pawn(Player.WHITE);
 59
 60          assertSame(Player.WHITE, board[6][2].player());
 61
 62          assertSame("Pawn", board[6][2].type());
 63
 64          Move move = new Move(6, 2, 5, 2);
 65
 66          assertTrue(board[6][2].isValidMove(move, board));
 67
 68          board[5][2] = new Pawn(Player.BLACK);
 69
 70          assertFalse(board[6][2].isValidMove(move, board));
 71
 72          board[5][2] = null;
 73
 74          move = new Move(6, 2, 4, 2);
 75
 76          assertTrue( board[6][2].isValidMove(move, board));
 77
 78          board[5][2] = new Pawn(Player.BLACK);
 79
 80          assertFalse(board[6][2].isValidMove(move, board));
 81
 82          board[5][2] = null;
 83
 84          board[4][2] = new Pawn(Player.BLACK);
 85
 86          assertFalse(board[6][2].isValidMove(move, board));
 87
 88          board[4][2] = null;
 89
 90          move = new Move(6, 2, 3, 2);
 91
 92          assertFalse(board[6][2].isValidMove(move, board));
 93
 94          move = new Move(6, 2, 4, 3);
 95
 96          assertFalse(board[6][2].isValidMove(move, board));
 97
 98          move = new Move(6, 2, 5, 3);
 99
100          assertFalse(board[6][2].isValidMove(move, board));
101
102          board[5][3] = new Pawn(Player.BLACK);
103
104          assertTrue(board[6][2].isValidMove(move, board));
105
106          board[5][3] = null;
107
108          move = new Move(6, 2, 5, 1);
109
110          assertFalse(board[6][2].isValidMove(move, board));
```

```
111
112         board[5][1] = new Pawn(Player.BLACK);
113
114         assertTrue(board[6][2].isValidMove(move, board));
115
116         board[5][1] = null;
117
118         board[6][2] = null;
119
120         board[4][2] = new Pawn(Player.WHITE);
121
122         move = new Move(4, 2, 3, 2);
123
124         assertTrue(board[4][2].isValidMove(move, board));
125
126         board[3][2] = new Pawn(Player.BLACK);
127
128         assertFalse(board[4][2].isValidMove(move, board));
129
130         board[3][2] = null;
131
132         move = new Move(4, 2, 2, 2);
133
134         assertFalse(board[4][2].isValidMove(move, board));
135     }
136
137     // tests that Pawn's black move logic is working
138     @Test
139     public void testPawnIsValidMoveBlack() {
140         IChessPiece[][] board = new IChessPiece[8][8];
141
142         board[1][2] = new Pawn(Player.BLACK);
143
144         assertSame(Player.BLACK, board[1][2].player());
145
146         Move move = new Move(1, 2, 2, 2);
147
148         assertTrue(board[1][2].isValidMove(move, board));
149
150         board[2][2] = new Pawn(Player.WHITE);
151
152         assertFalse(board[1][2].isValidMove(move, board));
153
154         board[2][2] = null;
155
156         move = new Move(1, 2, 3, 2);
157
158         assertTrue(board[1][2].isValidMove(move, board));
159
160         board[2][2] = new Pawn(Player.WHITE);
161
162         assertFalse(board[1][2].isValidMove(move, board));
163
164         board[2][2] = null;
165
```

```
166         board[3][2] = new Pawn(Player.WHITE);
167
168         assertFalse(board[1][2].isValidMove(move, board));
169
170         board[3][2] = null;
171
172         move = new Move(1, 2, 4, 2);
173
174         assertFalse(board[1][2].isValidMove(move, board));
175
176         move = new Move(1, 2, 3, 3);
177
178         assertFalse(board[1][2].isValidMove(move, board));
179
180         move = new Move(1, 2, 2, 3);
181
182         assertFalse(board[1][2].isValidMove(move, board));
183
184         board[2][3] = new Pawn(Player.WHITE);
185
186         assertTrue(board[1][2].isValidMove(move, board));
187
188         board[2][3] = null;
189
190         move = new Move(1, 2, 2, 1);
191
192         assertFalse(board[1][2].isValidMove(move, board));
193
194         board[2][1] = new Pawn(Player.WHITE);
195
196         assertTrue(board[1][2].isValidMove(move, board));
197
198         board[2][1] = null;
199
200         board[1][2] = null;
201
202         board[3][2] = new Pawn(Player.BLACK);
203
204         move = new Move(3, 2, 4, 2);
205
206         assertTrue(board[3][2].isValidMove(move, board));
207
208         board[4][2] = new Pawn(Player.WHITE);
209
210         assertFalse(board[3][2].isValidMove(move, board));
211
212         board[4][2] = null;
213
214         move = new Move(3, 2, 5, 2);
215
216         assertFalse(board[3][2].isValidMove(move, board));
217     }
218
219     // tests that Rook's move logic is working
220     @Test
```

```
221     public void testRookIsValidMove() {
222         IChessPiece[][] board = new IChessPiece[8][8];
223 
224         board[3][2] = new Rook(Player.WHITE);
225 
226         Move move = new Move(3,2,5,2);
227 
228         assertTrue(board[3][2].isValidMove(move, board));
229 
230         board[4][2] = new Pawn(Player.BLACK);
231 
232         assertFalse(board[3][2].isValidMove(move, board));
233 
234         move = new Move(3, 2, 3, 0);
235 
236         assertTrue(board[3][2].isValidMove(move, board));
237 
238         board[3][1] = new Pawn(Player.BLACK);
239 
240         board[4][2] = null;
241 
242         assertFalse(board[3][2].isValidMove(move, board));
243 
244         move = new Move(3, 2, 3, 7);
245 
246         assertTrue(board[3][2].isValidMove(move, board));
247 
248         board[3][1] = null;
249 
250         board[3][6] = new Pawn(Player.BLACK);
251 
252         assertFalse(board[3][2].isValidMove(move, board));
253 
254         move = new Move(3, 2, 1, 2);
255 
256         assertTrue(board[3][2].isValidMove(move, board));
257 
258         board[2][2] = new Pawn(Player.BLACK);
259 
260         board[3][6] = new Pawn(Player.BLACK);
261 
262         assertFalse(board[3][2].isValidMove(move, board));
263 
264         move = new Move(3, 2, 1, 3);
265 
266         assertFalse(board[3][2].isValidMove(move, board));
267 
268         assertSame("Rook", board[3][2].type());
269 
270     }
271 
272     // tests that Knight's move logic is working
273     @Test
274     public void testKnightIsValidMove() {
275         IChessPiece[][] board = new IChessPiece[8][8];
```

```
276
277         board[3][2] = new Knight(Player.WHITE);
278
279         assertSame(board[3][2].type(), "Knight");
280
281         Move move = new Move(3, 2, 5, 3);
282
283         assertTrue(board[3][2].isValidMove(move, board));
284
285         board[5][3] = new Pawn(Player.BLACK);
286
287         assertTrue(board[3][2].isValidMove(move, board));
288
289         move = new Move(3, 2, 5, 1);
290
291         assertTrue(board[3][2].isValidMove(move, board));
292
293         board[5][1] = new Pawn(Player.BLACK);
294
295         assertTrue(board[3][2].isValidMove(move, board));
296
297         move = new Move(3, 2, 1, 1);
298
299         assertTrue(board[3][2].isValidMove(move, board));
300
301         board[1][1] = new Pawn(Player.BLACK);
302
303         assertTrue(board[3][2].isValidMove(move, board));
304
305         move = new Move(3, 2, 1, 3);
306
307         assertTrue(board[3][2].isValidMove(move, board));
308
309         board[1][3] = new Pawn(Player.BLACK);
310
311         assertTrue(board[3][2].isValidMove(move, board));
312
313         move = new Move(3, 2, 4, 4);
314
315         assertTrue(board[3][2].isValidMove(move, board));
316
317         board[4][4] = new Pawn(Player.BLACK);
318
319         assertTrue(board[3][2].isValidMove(move, board));
320
321         move = new Move(3, 2, 4, 0);
322
323         assertTrue(board[3][2].isValidMove(move, board));
324
325         board[4][0] = new Pawn(Player.BLACK);
326
327         assertTrue(board[3][2].isValidMove(move, board));
328
329         move = new Move(3, 2, 2, 4);
330
```

```
331         assertTrue(board[3][2].isValidMove(move, board));
332
333         board[2][4] = new Pawn(Player.BLACK);
334
335         assertTrue(board[3][2].isValidMove(move, board));
336
337         move = new Move(3, 2, 2, 0);
338
339         assertTrue(board[3][2].isValidMove(move, board));
340
341         board[2][0] = new Pawn(Player.BLACK);
342
343         assertTrue(board[3][2].isValidMove(move, board));
344
345         move = new Move(3, 2, 1, 0);
346
347         assertFalse(board[3][2].isValidMove(move, board));
348
349         move = new Move(3, 2, 0, 0);
350
351         assertFalse(board[3][2].isValidMove(move, board));
352
353         move = new Move(3, 2, 1, 2);
354
355         assertFalse(board[3][2].isValidMove(move, board));
356
357         move = new Move(3, 2, 1, 4);
358
359         assertFalse(board[3][2].isValidMove(move, board));
360
361         move = new Move(3, 2, 4, 3);
362
363         assertFalse(board[3][2].isValidMove(move, board));
364
365         move = new Move(3, 2, 3, 3);
366
367         assertFalse(board[3][2].isValidMove(move, board));
368
369         move = new Move(3, 2, 2, 2);
370
371         assertFalse(board[3][2].isValidMove(move, board));
372
373         move = new Move(3, 2, 5, 0);
374
375         assertFalse(board[3][2].isValidMove(move, board));
376
377         move = new Move(3, 2, 5, 2);
378
379         assertFalse(board[3][2].isValidMove(move, board));
380
381         move = new Move(3, 2, 3, 0);
382
383         assertFalse(board[3][2].isValidMove(move, board));
384     }
385
```

```
386     // test simple model methods
387     @Test
388     public void testChessModelSimple() {
389         ChessModel model = new ChessModel();
390 
391         assertSame(8, model.numColumns());
392         assertSame(8, model.numRows());
393         assertSame(Player.WHITE, model.currentPlayer());
394 
395         model.setNextPlayer();
396         assertSame(Player.BLACK, model.currentPlayer());
397 
398         assertSame("Pawn", model.pieceAt(6,0).type());
399         assertSame(Player.BLACK, model.pieceAt(1,5).player());
400 
401         model.setPiece(4, 3, new Rook(Player.WHITE));
402         assertEquals("Rook", model.pieceAt(4, 3).type());
403         assertSame(Player.WHITE, model.pieceAt(4, 3).player());
404     }
405 
406     // test if moves are OOB
407     @Test(expected = IndexOutOfBoundsException.class)
408     public void testChessModelIsValidMoveOOB() {
409         ChessModel model = new ChessModel();
410 
411         Move move = new Move(5, 3, 10, 2);
412 
413         model.isValidMove(move);
414 
415     }
416 
417     // tests basic isValidMove method
418     @Test
419     public void testChessModelIsValidMove() {
420         ChessModel model = new ChessModel();
421 
422         Move move = new Move(5, 3, 4, 2);
423 
424         assertFalse(model.isValidMove(move));
425 
426         move = new Move(1, 0, 2, 0);
427 
428         assertFalse(model.isValidMove(move));
429 
430         move = new Move(6, 0, 4, 0);
431 
432         assertTrue(model.isValidMove(move));
433 
434         model.setPiece(5, 2, new Rook(Player.BLACK));
435 
436         model.setPiece(5, 3, new Pawn(Player.WHITE));
437 
438         model.setPiece(5, 4, new King(Player.WHITE));
439 
440         model.setPiece(7, 4, null);
```

```
441
442         move = new Move(5, 3, 4, 3);
443
444         assertFalse(model.isValidMove(move));
445     }
446
447     // tests move with OOB move data
448     @Test(expected = IndexOutOfBoundsException.class)
449     public void testChessModelMoveOOB() {
450         ChessModel model = new ChessModel();
451
452         Move move = new Move(5, 3, 10, 2);
453
454         model.move(move);
455
456     }
457
458     // tests basic move and undo functionality
459     @Test
460     public void testChessModelMoveAndUndo() {
461         ChessModel model = new ChessModel();
462
463         Move move = new Move(6, 2, 5, 2);
464
465         model.move(move);
466
467         assertNull(model.pieceAt(6,2));
468         assertSame("Pawn", model.pieceAt(5, 2).type());
469         assertSame(Player.WHITE, model.pieceAt(5, 2).player());
470         assertSame(Player.BLACK, model.currentPlayer());
471
472         model.undo();
473         assertNull(model.pieceAt(5,2));
474         assertSame("Pawn", model.pieceAt(6, 2).type());
475         assertSame(Player.WHITE, model.pieceAt(6, 2).player());
476         assertSame(Player.WHITE, model.currentPlayer());
477
478         model = new ChessModel();
479         model.setPiece(6,3, null);
480         move = new Move(7, 3, 4, 3);
481         model.move(move);
482         assertSame("Queen", model.pieceAt(4,3).type());
483         move = new Move(7, 2, 5, 4);
484         model.move(move);
485         assertSame("Bishop", model.pieceAt(5, 4).type());
486
487         assertEquals(move.toString(),("Move [fromRow=7, fromColumn=2, " +
488                 "toRow=5, toColumn=4]"));
489
490         Move defaultConstructor = new Move();
491         defaultConstructor = new Move(5, 4, 4,3);
492         model.move(defaultConstructor);
493         assertSame("Bishop", model.pieceAt(4, 3).type());
494     }
495
```

```
496     // tests inCheck method
497     @Test
498     public void testChessModelInCheck() {
499         ChessModel model = new ChessModel();
500 
501         model.setPiece(6, 4, null);
502 
503         model.setPiece(3, 4, new Queen(Player.BLACK));
504 
505         assertTrue(model.inCheck(Player.WHITE));
506         assertFalse(model.inCheck(Player.BLACK));
507     }
508 
509     // tests isComplete method
510     @Test
511     public void testChessModelIsCompleteAndNoValidMoves() {
512         ChessModel model = new ChessModel();
513 
514         assertFalse(model.isComplete());
515 
516         model.setPiece(6, 4, null);
517 
518         model.setPiece(5, 4, new Rook(Player.BLACK));
519 
520         assertFalse(model.isComplete());
521 
522         model.setPiece(4, 4, new Rook(Player.BLACK));
523 
524         model.setPiece(5, 4, null);
525 
526         model.setPiece(7, 3, new Pawn(Player.WHITE));
527 
528         model.setPiece(7, 6, null);
529 
530         model.setPiece(7, 5, new Pawn(Player.WHITE));
531 
532         assertTrue(model.noValidMoves());
533         assertTrue(model.isComplete());
534 
535         model = new ChessModel();
536         model.setPiece(0,0, null);
537         model.setPiece(0,1,null);
538         model.setPiece(0,2,null);
539         model.setPiece(0,3,null);
540         model.setPiece(0,5,null);
541         model.setPiece(0,6,null);
542         model.setPiece(0,7,null);
543         model.setPiece(1,0,null);
544         model.setPiece(1,1,null);
545         model.setPiece(1,2,null);
546         model.setPiece(1,3,null);
547         model.setPiece(1,4,null);
548         model.setPiece(1,5,null);
549         model.setPiece(1,6,null);
550         model.setPiece(1,7,null);
```

```
551         model.setPiece(2,3, new Rook(Player.WHITE));
552         model.setPiece(2,5,new Rook(Player.WHITE));
553         model.setPiece(1,0, new Rook(Player.WHITE));
554
555         model.setNextPlayer();
556
557         assertTrue(model.noValidMoves());
558         assertFalse(model.isComplete());
559
560
561     }
562
563     // tests logic in isEnPassant, isValidMove, move, and undo
564     @Test
565     public void testChessModelEnPassant() {
566         ChessModel model = new ChessModel();
567
568         model.setPiece(4,3, new Pawn(Player.BLACK));
569         Move move = new Move(6, 2, 4, 2);
570         model.move(move);
571         move = new Move(4, 3, 5, 2);
572         model.move(move);
573         assertNull(model.pieceAt(4, 2));
574         assertSame("Pawn", model.pieceAt(5, 2).type());
575         model.undo();
576         assertTrue(model.isValidMove(move));
577
578         model = new ChessModel();
579
580         model.setPiece(4,3, new Rook(Player.BLACK));
581         move = new Move(6, 2, 4, 2);
582         model.move(move);
583         move = new Move(4, 3, 5, 2);
584         assertFalse(model.isValidMove(move));
585
586         model = new ChessModel();
587
588         model.setPiece(4,3, new Pawn(Player.BLACK));
589         move = new Move(6, 4, 4, 4);
590         model.move(move);
591         move = new Move(4, 3, 5, 4);
592         model.move(move);
593         assertNull(model.pieceAt(4, 4));
594         assertSame("Pawn", model.pieceAt(5, 4).type());
595
596         model = new ChessModel();
597
598         model.setPiece(3,3, new Pawn(Player.WHITE));
599         model.setNextPlayer();
600         move = new Move(1, 4, 3, 4);
601         model.move(move);
602         move = new Move(3, 3, 2, 4);
603         model.move(move);
604         assertNull(model.pieceAt(3, 4));
605         assertSame("Pawn", model.pieceAt(2, 4).type());
```

```
606
607         model = new ChessModel();
608
609         model.setPiece(3,3, new Pawn(Player.WHITE));
610         model.setNextPlayer();
611         move = new Move(1, 2, 3, 2);
612         model.move(move);
613         move = new Move(3, 3, 2, 2);
614         model.move(move);
615         assertNull(model.pieceAt(3, 2));
616         assertSame("Pawn", model.pieceAt(2, 2).type());
617         model.undo();
618         assertTrue(model.isValidMove(move));
619
620     }
621
622     // tests logic in isCastling, isValidMove, move, and undo
623     @Test
624     public void testCastling() {
625         ChessModel model = new ChessModel();
626
627         model.setPiece(7,6,null);
628         model.setPiece(7,5, new Pawn(Player.BLACK));
629
630         Move move = new Move(7,4,7,6);
631
632         assertFalse(model.isValidMove(move));
633
634         model.setPiece(7,5, null);
635
636         assertTrue(model.isValidMove(move));
637
638         model.move(move);
639         assertSame("Rook", model.pieceAt(7,5).type());
640         assertSame("King", model.pieceAt(7,6).type());
641         assertNull(model.pieceAt(7,4));
642         assertNull(model.pieceAt(7,7));
643
644         model.undo();
645         assertSame("King", model.pieceAt(7,4).type());
646         assertSame("Rook", model.pieceAt(7,7).type());
647         assertNull(model.pieceAt(7,5));
648         assertNull(model.pieceAt(7,6));
649
650         model.setPiece(6,5,null);
651         model.setPiece(4,5, new Rook(Player.BLACK));
652         assertFalse(model.isValidMove(move));
653
654         model.setPiece(6,5,new Pawn(Player.WHITE));
655
656         model.setPiece(6,4, null);
657         model.setPiece(4,4, new Rook(Player.BLACK));
658         assertFalse(model.isValidMove(move));
659
660         model.setPiece(4,4, new Pawn(Player.WHITE));
```

```
661
662         move = new Move(7,4,7,5);
663         model.move(move);
664         move = new Move(7,5,7,4);
665         model.move(move);
666         move = new Move(7,4,7,6);
667         assertFalse(model.isValidMove(move));
668
669         model.undo();
670         model.undo();
671
672         move = new Move(7,7,7,5);
673         model.move(move);
674         move = new Move(7,5,7,7);
675         model.move(move);
676         move = new Move(7,4,7,6);
677         assertFalse(model.isValidMove(move));
678
679         model.undo();
680         model.undo();
681         assertTrue(model.isValidMove(move));
682
683         model = new ChessModel();
684
685         model.setPiece(7,3, null);
686         model.setPiece(7,2, null);
687         model.setPiece(7,1, new Pawn(Player.BLACK));
688
689         move = new Move(7,4,7,2);
690
691         assertFalse(model.isValidMove(move));
692
693         model.setPiece(7,1, null);
694
695         assertTrue(model.isValidMove(move));
696
697         model.move(move);
698         assertSame("Rook", model.pieceAt(7,3).type());
699         assertSame("King", model.pieceAt(7,2).type());
700         assertNull(model.pieceAt(7,4));
701         assertNull(model.pieceAt(7,0));
702
703         model.undo();
704         assertSame("King", model.pieceAt(7,4).type());
705         assertSame("Rook", model.pieceAt(7,0).type());
706         assertNull(model.pieceAt(7,3));
707         assertNull(model.pieceAt(7,2));
708
709         move = new Move(7,0,7,3);
710         model.move(move);
711         move = new Move(7,3,7,0);
712         model.move(move);
713         move = new Move(7,4,7,2);
714         assertFalse(model.isValidMove(move));
715
```

```
716         model.undo();
717         model.undo();
718         assertTrue(model.isValidMove(move));
719
720         model = new ChessModel();
721         model.setNextPlayer();
722
723         model.setPiece(0,6,null);
724         model.setPiece(0,5, new Pawn(Player.WHITE));
725
726         move = new Move(0,4,0,6);
727
728         assertFalse(model.isValidMove(move));
729
730         model.setPiece(0,5, null);
731
732         assertTrue(model.isValidMove(move));
733
734         model.move(move);
735         assertSame("Rook", model.pieceAt(0,5).type());
736         assertSame("King", model.pieceAt(0,6).type());
737         assertNull(model.pieceAt(0,4));
738         assertNull(model.pieceAt(0,7));
739
740         model.undo();
741         assertSame("King", model.pieceAt(0,4).type());
742         assertSame("Rook", model.pieceAt(0,7).type());
743         assertNull(model.pieceAt(0,5));
744         assertNull(model.pieceAt(0,6));
745
746         model.setPiece(1,5,null);
747         model.setPiece(4,5, new Rook(Player.WHITE));
748         assertFalse(model.isValidMove(move));
749
750         model.setPiece(1,5,new Pawn(Player.BLACK));
751
752         move = new Move(0,4,0,5);
753         model.move(move);
754         move = new Move(0,5,0,4);
755         model.move(move);
756         move = new Move(0,4,0,6);
757         assertFalse(model.isValidMove(move));
758
759         model.undo();
760         model.undo();
761         assertTrue(model.isValidMove(move));
762
763         move = new Move(0,7,0,5);
764         model.move(move);
765         move = new Move(0,5,0,7);
766         model.move(move);
767         move = new Move(0,4,0,6);
768         assertFalse(model.isValidMove(move));
769
770         model.undo();
```

```
771         model.undo();
772         assertTrue(model.isValidMove(move));
773
774         model = new ChessModel();
775         model.setNextPlayer();
776
777         model.setPiece(0,3, null);
778         model.setPiece(0,2, null);
779         model.setPiece(0,1, new Pawn(Player.BLACK));
780
781         move = new Move(0,4,0,2);
782
783         assertFalse(model.isValidMove(move));
784
785         model.setPiece(0,1, null);
786
787         assertTrue(model.isValidMove(move));
788
789         model.move(move);
790         assertSame("Rook", model.pieceAt(0,3).type());
791         assertSame("King", model.pieceAt(0,2).type());
792         assertNull(model.pieceAt(0,4));
793         assertNull(model.pieceAt(0,0));
794
795         model.undo();
796         assertSame("King", model.pieceAt(0,4).type());
797         assertSame("Rook", model.pieceAt(0,0).type());
798         assertNull(model.pieceAt(0,3));
799         assertNull(model.pieceAt(0,2));
800
801         move = new Move(0,0,0,3);
802         model.move(move);
803         move = new Move(0,3,0,0);
804         model.move(move);
805         move = new Move(0,4,0,2);
806         assertFalse(model.isValidMove(move));
807
808         model.undo();
809         model.undo();
810         assertTrue(model.isValidMove(move));
811
812     }
813
814     // test if pawns can promote
815     @Test
816     public void testChessModelCanPromote() {
817         ChessModel model = new ChessModel();
818
819         assertFalse(model.canPromote());
820
821         model.setPiece(0,5,null);
822         model.setPiece(1,5,new Pawn(Player.WHITE));
823
824         Move move = new Move(1,5,0,5);
825
```

```
826         model.move(move);
827
828         assertTrue(model.canPromote());
829
830         model.promote("Queen");
831
832         assertSame("Queen", model.pieceAt(0,5).type());
833
834         model.undo();
835
836         assertNull(model.pieceAt(0,5));
837         assertSame("Pawn", model.pieceAt(1,5).type());
838
839         model.move(move);
840
841         model.promote("Knight");
842
843         assertSame("Knight", model.pieceAt(0,5).type());
844
845         model.setPiece(7,5,null);
846         model.setPiece(6,5,new Pawn(Player.BLACK));
847
848         move = new Move(6,5,7,5);
849
850         model.move(move);
851
852         assertTrue(model.canPromote());
853         model.promote("Bishop");
854
855         assertSame("Bishop", model.pieceAt(7,5).type());
856
857         model.undo();
858
859         assertNull(model.pieceAt(7, 5));
860         assertSame("Pawn", model.pieceAt(6,5).type());
861
862         model.move(move);
863
864         model.promote("Rook");
865
866         assertSame("Rook", model.pieceAt(7,5).type());
867
868
869     }
870
871     //Tests movements for king
872     @Test
873     public void testKingIsValidMove(){
874         //Creates Chess board
875         IChessPiece[][] board = new IChessPiece[8][8];
876
877         //Creates new King to move up and Checks it has been created properly
878         board[7][2] = new King(Player.WHITE);
879         assertTrue(board[7][2].player() == Player.WHITE);
880         assertTrue(board[7][2].type().equals("King"));
```

```
881
882         //Checks that moving up 1 is a valid move
883         Move kingUp1 = new Move(7, 2, 6, 2);
884         assertTrue(board[7][2].isValidMove(kingUp1, board));
885         //Checks that king can't make move if player has piece there
886         board[6][2] = new Pawn(Player.WHITE);
887         assertFalse(board[7][2].isValidMove(kingUp1, board));
888         //checks that King can still make move with enemy piece there
889         board[6][2] = new Pawn(Player.BLACK);
890         assertTrue(board[7][2].isValidMove(kingUp1, board));
891         //Checks that king can't move more than 1 row up
892         Move kingUp2 = new Move(7, 2, 5, 2);
893         assertFalse(board[7][2].isValidMove(kingUp2, board));
894         board[6][2] = null;
895
896         //Checks that moving left 1 is a valid move
897         Move kingLeft1 = new Move(7,2,7,1);
898         assertTrue(board[7][2].isValidMove(kingLeft1, board));
899         //Checks that King can't make move if player has piece there
900         board[7][1] = new Pawn(Player.WHITE);
901         assertFalse(board[7][2].isValidMove(kingLeft1, board));
902         //Checks that King can still make move with enemy piece there
903         board[7][1] = new Pawn(Player.BLACK);
904         assertTrue(board[7][2].isValidMove(kingLeft1, board));
905         //Checks that King can't move more than 1 column left
906         Move kingLeft2 = new Move(7, 2, 7, 0);
907         assertFalse(board[7][2].isValidMove(kingLeft2, board));
908         board[7][1] = null;
909
910         //Checks that moving right 1 is a valid move
911         Move kingRight1 = new Move(7, 2, 7, 3);
912         assertTrue(board[7][2].isValidMove(kingRight1, board));
913         //Checks that King can't make move if player has piece there
914         board[7][3] = new Pawn(Player.WHITE);
915         assertFalse(board[7][2].isValidMove(kingRight1, board));
916         //Check that King can still make move with enemy piece there
917         board[7][3] = new Pawn(Player.BLACK);
918         assertTrue(board[7][2].isValidMove(kingRight1, board));
919         //Checks that King can't move more than 1 column right
920         Move kingRight2 = new Move(7, 2, 7, 4);
921         assertFalse(board[7][2].isValidMove(kingRight2, board));
922         board[7][3] = null;
923
924         //Checks that moving up left 1 is a valid move
925         Move kingUpLeft1 = new Move(7, 2, 6, 1);
926         assertTrue(board[7][2].isValidMove(kingUpLeft1, board));
927         //Checks that King can't make move is player has piece there
928         board[6][1] = new Pawn(Player.WHITE);
929         assertFalse(board[7][2].isValidMove(kingUpLeft1, board));
930         //Checks that King can still make move with enemy piece there
931         board[6][1] = new Pawn(Player.BLACK);
932         assertTrue(board[7][2].isValidMove(kingUpLeft1, board));
933         //Checks that king can't move more than 1 up left
934         Move kingUpLeft2 = new Move(7, 2, 5, 0);
935         assertFalse(board[7][2].isValidMove(kingUpLeft2, board));
```

```
936         board[6][1] = null;
937
938         //Checks that moving up right 1 is a valid move
939         Move kingUpRight1 = new Move(7, 2, 6, 3);
940         assertTrue(board[7][2].isValidMove(kingUpRight1, board));
941         //Checks that King can't make move if player has piece there
942         board[6][3] = new Pawn (Player.WHITE);
943         assertFalse(board[7][2].isValidMove(kingUpRight1, board));
944         //Checks that King can still make move with enemy piece there
945         board[6][3] = new Pawn(Player.BLACK);
946         assertTrue(board[7][2].isValidMove(kingUpRight1, board));
947         //Checks that King can't move more than 1 up right
948         Move kingUpRight2 = new Move(7, 2, 5, 4);
949         assertFalse(board[7][2].isValidMove(kingUpRight2, board));
950         board[6][3] = null;
951
952         //Removes old King
953         board[7][2] = null;
954         //Creates new King to move up and Checks it has been created properly
955         board[5][2] = new King(Player.WHITE);
956         assertTrue(board[5][2].player() == Player.WHITE);
957         assertTrue(board[5][2].type().equals("King"));
958
959         //Checks that moving down 1 is valid move
960         Move kingDown1 = new Move(5, 2, 6, 2);
961         assertTrue(board[5][2].isValidMove(kingDown1, board));
962         //Checks that king can't make move if player has piece there
963         board[6][2] = new Pawn(Player.WHITE);
964         assertFalse(board[5][2].isValidMove(kingDown1, board));
965         //Checks that king can still make move with enemy piece there
966         board[6][2] = new Pawn(Player.BLACK);
967         assertTrue(board[5][2].isValidMove(kingDown1, board));
968         //Checks that king can't move more than 1 row down
969         Move kingDown2 = new Move(5, 2, 7, 2);
970         assertFalse(board[5][2].isValidMove(kingDown2, board));
971
972         //Checks that moving down left is valid move
973         Move kingDownLeft1 = new Move(5, 2, 6, 1);
974         assertTrue(board[5][2].isValidMove(kingDownLeft1, board));
975         //Checks that King can't make move if player has a piece there
976         board[6][1] = new Pawn(Player.WHITE);
977         assertFalse(board[5][2].isValidMove(kingDownLeft1, board));
978         //Checks the King can make move with enemy piece there
979         board[6][1] = new Pawn(Player.BLACK);
980         assertTrue(board[5][2].isValidMove(kingDownLeft1, board));
981         //Checks that King can't move more than 1 down left
982         Move kingDownLeft2 = new Move(5, 2, 7, 0);
983         assertFalse(board[5][2].isValidMove(kingDownLeft2, board));
984
985         //Checks that moving down right is valid move
986         Move kingDownRight1 = new Move(5, 2, 6, 3);
987         assertTrue(board[5][2].isValidMove(kingDownRight1, board));
988         //Checks that King can't make move if player has a piece there
989         board[6][3] = new Pawn(Player.WHITE);
990         assertFalse(board[5][2].isValidMove(kingDownRight1, board));
```

```
 991         //Checks that king can make move with enemy piece there
 992         board[6][3] = new Pawn(Player.BLACK);
 993         assertTrue(board[5][2].isValidMove(kingDownRight1, board));
 994         //Checks that King can't move more than 1 down right
 995         Move kingDownRight2 = new Move(5,2, 7, 4);
 996         assertFalse(board[5][2].isValidMove(kingDownRight2, board));
 997     }
 998     @Test
 999     //Tests queens movement
1000     public void testQueenIsValidMove(){
1001         //Creates Chess board
1002         IChessPiece[][] board = new IChessPiece[8][8];
1003
1004         //Creates new Queen to move forward and Checks it has been
1005         // created properly
1006         board[4][4] = new Queen(Player.WHITE);
1007         assertTrue(board[4][4].player() == Player.WHITE);
1008         assertTrue(board[4][4].type().equals("Queen"));
1009
1010         //Checks that moving down 1 is valid move
1011         Move queenDown1 = new Move(4, 4, 5, 4);
1012         assertTrue(board[4][4].isValidMove(queenDown1, board));
1013         //Checks that moving down 3 is valid move
1014         Move queenDown3 = new Move(4, 4, 7, 4);
1015         assertTrue(board[4][4].isValidMove(queenDown3, board));
1016         //Checks that queen can move to spot with opponent piece
1017         board[5][4] = new Pawn(Player.BLACK);
1018         assertTrue(board[4][4].isValidMove(queenDown1, board));
1019         //Checks that queen cannot move to spot with player piece
1020         board[5][4] = null;
1021         board[5][4] = new Pawn(Player.WHITE);
1022         assertFalse(board[4][4].isValidMove(queenDown1, board));
1023         //Checks that queen cannot move through player piece
1024         assertFalse(board[4][4].isValidMove(queenDown3, board));
1025         //Checks that queen cannot move through opponent piece
1026         board[5][4] = null;
1027         board[5][4] = new Pawn(Player.BLACK);
1028         assertFalse(board[4][4].isValidMove(queenDown3, board));
1029         board[5][4] = null;
1030
1031         //Checks that moving down left 1 is valid move
1032         Move queenDownLeft1 = new Move(4, 4, 5, 3);
1033         assertTrue(board[4][4].isValidMove(queenDownLeft1, board));
1034         //Checks that moving down left 3 is valid move
1035         Move queenDownLeft3 = new Move(4, 4, 7, 1);
1036         assertTrue(board[4][4].isValidMove(queenDownLeft3, board));
1037         //Checks that queen can move to spot with opponent piece
1038         board[5][3] = new Pawn(Player.BLACK);
1039         assertTrue(board[4][4].isValidMove(queenDownLeft1, board));
1040         //Checks that queen cannot move to spot with player piece
1041         board[5][3] = null;
1042         board[5][3] = new Pawn(Player.WHITE);
1043         assertFalse(board[4][4].isValidMove(queenDownLeft1, board));
1044         //Checks that queen cannot move through player piece
1045         assertFalse(board[4][4].isValidMove(queenDownLeft3, board));
```

```
1046          //Checks that queen cannot move through opponent piece
1047          board[5][3] = null;
1048          board[5][3] = new Pawn(Player.BLACK);
1049          assertFalse(board[4][4].isValidMove(queenDownLeft3, board));
1050          board[5][3] = null;
1051
1052          //Checks that moving down right 1 is valid move
1053          Move queenDownRight1 = new Move(4, 4, 5, 5);
1054          assertTrue(board[4][4].isValidMove(queenDownRight1, board));
1055          //Checks that moving down right 3 is a valid move
1056          Move queenDownRight3 = new Move(4, 4, 7, 7);
1057          assertTrue(board[4][4].isValidMove(queenDownRight3, board));
1058          //Checks that queen can move to spot with opponent piece
1059          board[5][5] = new Pawn(Player.BLACK);
1060          assertTrue(board[4][4].isValidMove(queenDownRight1, board));
1061          //Checks that queen cannot move to spot with player piece
1062          board[5][5] = null;
1063          board[5][5] = new Pawn(Player.WHITE);
1064          assertFalse(board[4][4].isValidMove(queenDownRight1, board));
1065          //Checks that queen cannot move through player piece
1066          assertFalse(board[4][4].isValidMove(queenDownRight3, board));
1067          //Checks that queen cannot move through opponent piece
1068          board[5][5] = null;
1069          board[5][5] = new Pawn(Player.BLACK);
1070          assertFalse(board[4][4].isValidMove(queenDownRight3, board));
1071          board[5][5] = null;
1072
1073          //Checks that moving left 1 is valid move
1074          Move queenLeft1 = new Move(4, 4, 4, 3);
1075          assertTrue(board[4][4].isValidMove(queenLeft1, board));
1076          //Checks that moving left 3 is valid move
1077          Move queenLeft3 = new Move(4, 4, 4, 1);
1078          assertTrue(board[4][4].isValidMove(queenLeft3, board));
1079          //Checks that queen can move to spot with opponent piece
1080          board[4][3] = new Pawn(Player.BLACK);
1081          assertTrue(board[4][4].isValidMove(queenLeft1, board));
1082          //Checks that queen cannot move to spot with player piece
1083          board[4][3] = null;
1084          board[4][3] = new Pawn(Player.WHITE);
1085          assertFalse(board[4][4].isValidMove(queenLeft1, board));
1086          //Checks that queen cannot move through player piece
1087          assertFalse(board[4][4].isValidMove(queenLeft3, board));
1088          //Checks that queen cannot move through opponent piece
1089          board[4][3] = null;
1090          board[4][3] = new Pawn(Player.BLACK);
1091          assertFalse(board[4][4].isValidMove(queenLeft3, board));
1092          board[4][3] = null;
1093
1094          //Checks that moving right 1 is valid move
1095          Move queenRight1 = new Move(4, 4, 4, 5);
1096          assertTrue(board[4][4].isValidMove(queenRight1, board));
1097          //Checks that moving left 3 is valid move
1098          Move queenRight3 = new Move(4, 4, 4, 7);
1099          assertTrue(board[4][4].isValidMove(queenRight3, board));
1100          //Checks that queen can move to spot with opponent piece
```

```
1101          board[4][5] = new Pawn(Player.BLACK);
1102          assertTrue(board[4][4].isValidMove(queenRight1, board));
1103          //Checks that queen cannot move to spot with player piece
1104          board[4][5] = null;
1105          board[4][5] = new Pawn(Player.WHITE);
1106          assertFalse(board[4][4].isValidMove(queenRight1, board));
1107          //Checks that queen cannot move through player piece
1108          assertFalse(board[4][4].isValidMove(queenRight3, board));
1109          //Checks that queen cannot move through opponent piece
1110          board[4][5] = null;
1111          board[4][5] = new Pawn(Player.BLACK);
1112          assertFalse(board[4][4].isValidMove(queenRight3, board));
1113          board[4][5] = null;
1114
1115          //Checks that moving up left 1 is valid move
1116          Move queenUpLeft1 = new Move(4, 4, 3, 3);
1117          assertTrue(board[4][4].isValidMove(queenUpLeft1, board));
1118          //Checks that moving down left 3 is valid move
1119          Move queenUpLeft3 = new Move(4, 4, 1, 1);
1120          assertTrue(board[4][4].isValidMove(queenUpLeft3, board));
1121          //Checks that queen can move to spot with opponent piece
1122          board[3][3] = new Pawn(Player.BLACK);
1123          assertTrue(board[4][4].isValidMove(queenUpLeft1, board));
1124          //Checks that queen cannot move to spot with player piece
1125          board[3][3] = null;
1126          board[3][3] = new Pawn(Player.WHITE);
1127          assertFalse(board[4][4].isValidMove(queenUpLeft1, board));
1128          //Checks that queen cannot move through player piece
1129          assertFalse(board[4][4].isValidMove(queenUpLeft3, board));
1130          //Checks that queen cannot move through opponent piece
1131          board[3][3] = null;
1132          board[3][3] = new Pawn(Player.BLACK);
1133          assertFalse(board[4][4].isValidMove(queenUpLeft3, board));
1134          board[3][3] = null;
1135
1136          //Checks that moving up right 1 is valid move
1137          Move queenUpRight1 = new Move(4, 4, 3, 5);
1138          assertTrue(board[4][4].isValidMove(queenUpRight1, board));
1139          //Checks that moving down right 3 is a valid move
1140          Move queenUpRight3 = new Move(4, 4, 1, 7);
1141          assertTrue(board[4][4].isValidMove(queenUpRight3, board));
1142          //Checks that queen can move to spot with opponent piece
1143          board[3][5] = new Pawn(Player.BLACK);
1144          assertTrue(board[4][4].isValidMove(queenUpRight1, board));
1145          //Checks that queen cannot move to spot with player piece
1146          board[3][5] = null;
1147          board[3][5] = new Pawn(Player.WHITE);
1148          assertFalse(board[4][4].isValidMove(queenUpRight1, board));
1149          //Checks that queen cannot move through player piece
1150          assertFalse(board[4][4].isValidMove(queenUpRight3, board));
1151          //Checks that queen cannot move through opponent piece
1152          board[3][5] = null;
1153          board[3][5] = new Pawn(Player.BLACK);
1154          assertFalse(board[4][4].isValidMove(queenUpRight3, board));
1155          board[3][5] = null;
```

```
1156
1157        //Checking obvious invalid moves
1158        Move invalid1 = new Move(4, 4, 3, 7);
1159        assertFalse(board[4][4].isValidMove(invalid1, board));
1160        Move invalid2 = new Move(4, 4, 7, 3);
1161        assertFalse(board[4][4].isValidMove(invalid2, board));
1162    }
1163
1164    @Test
1165    //Tests Bishops movement
1166    public void testBishopIsValid(){
1167        //Creates chess board
1168        IChessPiece[][] board = new IChessPiece[8][8];
1169
1170        //Creates new Bishop and checks it has been created properly
1171        board[4][4] = new Bishop(Player.WHITE);
1172        assertEquals(board[4][4].player(),  Player.WHITE);
1173        assertEquals(board[4][4].type(), ("Bishop"));
1174
1175        //Checks that moving up left 1 is valid move
1176        Move bishopDiagForwardLeft1 = new Move(4, 4, 3, 3);
1177        assertTrue(board[4][4].isValidMove(bishopDiagForwardLeft1, board));
1178        //Checks that moving up left 3 is a valid move
1179        Move bishopDiagForwardLeft3 = new Move(4, 4, 1, 1);
1180        assertTrue(board[4][4].isValidMove(bishopDiagForwardLeft3, board));
1181        //Checks that moving to a spot with an enemy piece is valid
1182        board[3][3] = new Pawn(Player.BLACK);
1183        assertTrue(board[4][4].isValidMove(bishopDiagForwardLeft1, board));
1184        //Checks that move is not valid if player already has piece there
1185        board[3][3] = null;
1186        board[1][1] = new Pawn(Player.WHITE);
1187        assertFalse(board[4][4].isValidMove(bishopDiagForwardLeft3, board));
1188        //Checks that move is not valid if bishop passes through opponent piece
1189        board[1][1] = null;
1190        board[3][3] = new Pawn(Player.BLACK);
1191        assertFalse(board[4][4].isValidMove(bishopDiagForwardLeft3, board));
1192        //Checks that move is not valid if bishop passes through players piece
1193        board[3][3] = null;
1194        board[3][3] = new Pawn(Player.WHITE);
1195        assertFalse(board[4][4].isValidMove(bishopDiagForwardLeft3, board));
1196
1197        //Checks that moving up right 1 is valid move
1198        Move bishopDiagForwardRight1 = new Move(4, 4, 3, 5);
1199        assertTrue(board[4][4].isValidMove(bishopDiagForwardRight1, board));
1200        //Checks that moving up right 3 is valid move
1201        Move bishopDiagForwardRight3 = new Move(4, 4, 1, 7);
1202        assertTrue(board[4][4].isValidMove(bishopDiagForwardRight3, board));
1203        //Checks that moving to a spot with an enemy piece is valid
1204        board[3][5] = new Pawn(Player.BLACK);
1205        assertTrue(board[4][4].isValidMove(bishopDiagForwardRight1, board));
1206        //Checks that move is not valid if player already has a piece there
1207        board[3][5] = null;
1208        board[1][7] = new Pawn(Player.WHITE);
1209        assertFalse(board[4][4].isValidMove(bishopDiagForwardRight3, board));
1210        //Checks that move is not valid if bishop passes through opponent piece
```

```
1211          board[1][7] = null;
1212          board[3][5] = new Pawn(Player.BLACK);
1213          assertFalse(board[4][4].isValidMove(bishopDiagForwardRight3, board));
1214          //Checks that move is not valid if bishop passes through players piece
1215          board[3][5] = null;
1216          board[3][5] = new Pawn(Player.WHITE);
1217          assertFalse(board[4][4].isValidMove(bishopDiagForwardRight3, board));
1218
1219          //Checks that moving down left 1 is valid move
1220          Move bishopDiagBackLeft1 = new Move(4, 4, 5, 3);
1221          assertTrue(board[4][4].isValidMove(bishopDiagBackLeft1, board));
1222          //Checks that moving down left 3 is valid move
1223          Move bishopDiagBackLeft3 = new Move(4, 4, 7, 1);
1224          assertTrue(board[4][4].isValidMove(bishopDiagBackLeft3, board));
1225          //Checks that moving to a spot with an enemy piece is valid
1226          board[5][3] = new Pawn(Player.BLACK);
1227          assertTrue(board[4][4].isValidMove(bishopDiagBackLeft1, board));
1228          //Checks that move is not valid if player already has a piece there
1229          board[5][3] = null;
1230          board[7][1] = new Pawn(Player.WHITE);
1231          assertFalse(board[4][4].isValidMove(bishopDiagBackLeft3, board));
1232          //Checks that bishop cannot pass through opponent piece
1233          board[7][1] = null;
1234          board[5][3] = new Pawn(Player.BLACK);
1235          assertFalse(board[4][4].isValidMove(bishopDiagBackLeft3, board));
1236          //Checks that bishop cannot pass through players piece
1237          board[5][3] = null;
1238          board[5][3] = new Pawn(Player.WHITE);
1239          assertFalse(board[4][4].isValidMove(bishopDiagBackLeft3, board));
1240
1241          //Checks that moving down right 1 is valid move
1242          Move bishopDiagBackRight1 = new Move(4, 4, 5, 5);
1243          assertTrue(board[4][4].isValidMove(bishopDiagBackRight1, board));
1244          //Checks that moving up right 3 is valid move
1245          Move bishopDiagBackRight3 = new Move(4, 4, 7, 7);
1246          assertTrue(board[4][4].isValidMove(bishopDiagBackRight3, board));
1247          //Checks that moving to a spot with enemy piece is valid
1248          board[5][5] = new Pawn(Player.BLACK);
1249          assertTrue(board[4][4].isValidMove(bishopDiagBackRight1, board));
1250          //Checks that move is not valid if player already has a piece there
1251          board[5][5] = null;
1252          board[7][7] = new Pawn(Player.WHITE);
1253          assertFalse(board[4][4].isValidMove(bishopDiagBackRight3, board));
1254          //Checks that bishop cannot pass through opponent piece
1255          board[7][7] = null;
1256          board[5][5] = new Pawn(Player.BLACK);
1257          assertFalse(board[4][4].isValidMove(bishopDiagBackRight3, board));
1258          //Checks that bishop cannot pass through players piece
1259          board[5][5] = null;
1260          board[5][5] = new Pawn(Player.WHITE);
1261          assertFalse(board[4][4].isValidMove(bishopDiagBackRight3, board));
1262
1263          //Checks that the bishop cannot move straight up at all
1264          Move bishopForward1 = new Move(4, 4, 3, 4);
1265          assertFalse(board[4][4].isValidMove(bishopForward1, board));
```

```
1266          Move bishopForward3 = new Move(4, 4, 1, 4);
1267          assertFalse(board[4][4].isValidMove(bishopForward3, board));
1268          //Checks that the bishop cannot move straight down at all
1269          Move bishopBack1 = new Move(4, 4, 5, 4);
1270          assertFalse(board[4][4].isValidMove(bishopBack1, board));
1271          Move bishopBack3 = new Move(4, 4, 7, 4);
1272          assertFalse(board[4][4].isValidMove(bishopBack3, board));
1273          //Checks that the bishop cannot move straight left at all
1274          Move bishopLeft1 = new Move(4, 4, 4, 3);
1275          assertFalse(board[4][4].isValidMove(bishopLeft1, board));
1276          Move bishopLeft3 = new Move(4, 4, 4, 1);
1277          assertFalse(board[4][4].isValidMove(bishopLeft3, board));
1278          //Checks that the bishop cannot move straight right at all
1279          Move bishopRight1 = new Move(4, 4, 4, 5);
1280          assertFalse(board[4][4].isValidMove(bishopRight1, board));
1281          Move bishopRight3 = new Move(4, 4, 4, 7);
1282          assertFalse(board[4][4].isValidMove(bishopRight3, board));
1283      }
1284
1285  }
```