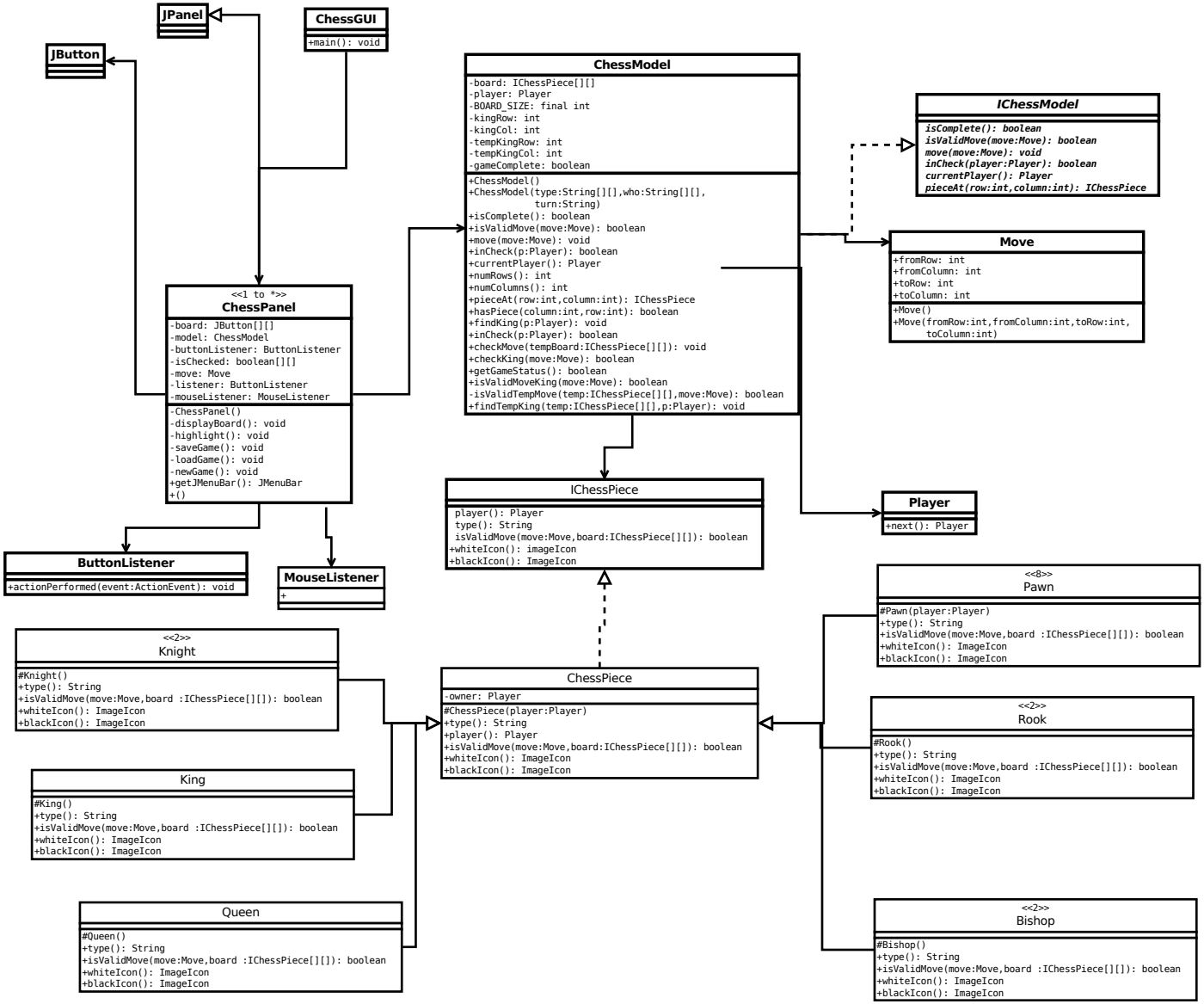
JPanel
ChessGUI
+main(): void
JButton
ChessModel
-board: IChessPiece[][]
-player: Player
-BOARD_SIZE: final int
-kingRow: int
-kingCol: int
-tempKingRow: int
-tempKingCol: int
-gameComplete: boolean
+ChessModel()
+ChessModel(type:String[][],who:String[][],
turn:String)
+isComplete(): boolean
+isValidMove(move:Move): boolean
+move(move:Move): void
+inCheck(p:Player): boolean
+currentPlayer(): Player
+numRows(): int
+numColumns(): int
+pieceAt(row:int,column:int): IChessPiece
+hasPiece(column:int,row:int): boolean
+findKing(p:Player): void
+inCheck(p:Player): boolean
+checkMove(tempBoard:IChessPiece[][]): void
+checkKing(move:Move): boolean
+getGameStatus(): boolean
+isValidMoveKing(move:Move): boolean
-isValidTempMove(temp:IChessPiece[][],move:Move): boolean
+findTempKing(temp:IChessPiece[][],p:Player): void
IChessModel
isComplete(): boolean
isValidMove(move:Move): boolean
move(move:Move): void
inCheck(player:Player): boolean
currentPlayer(): Player
pieceAt(row:int,column:int): IChessPiece
Move
+fromRow: int
+fromColumn: int
+toRow: int
+toColumn: int
+Move()
+Move(fromRow:int,fromColumn:int,toRow:int,
toColumn:int)
<<1 to *>>
ChessPanel
-board: JButton[][]
-model: ChessModel
-buttonListener: ButtonListener
-isChecked: boolean[][]
-move: Move
-listener: ButtonListener
-mouseListener: MouseListener
-ChessPanel()
-displayBoard(): void
-highlight(): void
-saveGame(): void
-loadGame(): void
-newGame(): void
+getJMenuBar(): JMenuBar
+()
IChessPiece
player(): Player
type(): String
isValidMove(move:Move,board:IChessPiece[][]): boolean
+whiteIcon(): imageIcon
+blackIcon(): ImageIcon
Player
+next(): Player
ButtonListener
+actionPerformed(event:ActionEvent): void
MouseListener
+
<<8>>
Pawn
#Pawn(player:Player)
+type(): String
+isValidMove(move:Move,board :IChessPiece[][]): boolean
+whiteIcon(): ImageIcon
+blackIcon(): ImageIcon
<<2>>
Knight
#Knight()
+type(): String
+isValidMove(move:Move,board :IChessPiece[][]): boolean
+whiteIcon(): ImageIcon
+blackIcon(): ImageIcon
ChessPiece
-owner: Player
#ChessPiece(player:Player)
+type(): String
+player(): Player
+isValidMove(move:Move,board:IChessPiece[][]): boolean
+whiteIcon(): ImageIcon
+blackIcon(): ImageIcon
<<2>>
Rook
#Rook()
+type(): String
+isValidMove(move:Move,board :IChessPiece[][]): boolean
+whiteIcon(): ImageIcon
+blackIcon(): ImageIcon
King
#King()
+type(): String
+isValidMove(move:Move,board :IChessPiece[][]): boolean
+whiteIcon(): ImageIcon
+blackIcon(): ImageIcon
Queen
#Queen()
+type(): String
+isValidMove(move:Move,board :IChessPiece[][]): boolean
+whiteIcon(): ImageIcon
+blackIcon(): ImageIcon
<<2>>
Bishop
#Bishop()
+type(): String
+isValidMove(move:Move,board :IChessPiece[][]): boolean
+whiteIcon(): ImageIcon
+blackIcon(): ImageIcon

```java
package chess;


/**************************************************************
 * ChessModel Class CIS 163-03 Project 3
 *
 * Chess game. According to Model View Controller software pattern, this
 * class controls the model (game). One exception being the chess pieces
 * contain image icons that violate this pattern, however this
 * implementation is a functional example of polymorphism.
 *
 * @author Michael Baldwin, Douglas Money, Nick Reitz
 * @version Winter 2014
 *************************************************************/
public class ChessModel implements IChessModel {

    /** Constant integer determining size of game board */
    private final int BOARD_SIZE = 8;

    /** Two dimensional array of IChessPiece objects for game board */
    private IChessPiece[][] board;

    /** Player object for determining the current player for game */
    private Player player;

    /** Integer storing row number King object is located at */
    private int kingRow;

    /** Integer storing column number King object is located at */
    private int kingCol;

    /** Integer storing row number temporary King object is found */
    private int tempKingRow;

    /** Integer storing column number temporary King object is found */
    private int tempKingCol;

    /** Boolean determining whether game is complete or not */
    private boolean gameComplete;
```

```java
/*****************************************************************
 * Default constructor for ChessModel game, creates two dimensional
 * array (board) of IChessPiece objects according to constant for
 * board size. Then instantiates Pawn, Rook, Knight, Bishop, King,
 * and Queen objects in proper locations on board. Sets current
 * player to White (all Chess games start with White player). Also
 * creates a new ArrayList of type IChessPiece objects to store
 * chess pieces captured during the game.
 *****************************************************************/
public ChessModel() {

    // creates new board of IChessPiece objects
    board = new IChessPiece[BOARD_SIZE][BOARD_SIZE];

    // loops through the board to create 16 Pawns in proper
    // locations
    int i = 0;
    while (i < board.length) {

        // creates a new Pawn object belonging to Black player and
        // a new Pawn object belonging to White player each loop
        board[1][i] = new Pawn(Player.BLACK);
        board[6][i] = new Pawn(Player.WHITE);
        i++;
    }

    // creates 2 new Rook objects belonging to Black player and
    // 2 new Rook objects belonging to White player
    board[0][0] = new Rook(Player.BLACK);
    board[0][7] = new Rook(Player.BLACK);
    board[7][0] = new Rook(Player.WHITE);
    board[7][7] = new Rook(Player.WHITE);

    // creates 2 new Knight objects belonging to Black player and
    // 2 new Knight objects belonging to White player
    board[0][1] = new Knight(Player.BLACK);
    board[0][6] = new Knight(Player.BLACK);
    board[7][1] = new Knight(Player.WHITE);
    board[7][6] = new Knight(Player.WHITE);
```

```
        // creates 2 new Bishop objects belonging to Black player and
        // 2 new Bishop objects belonging to White player
        board[0][2] = new Bishop(Player.BLACK);
        board[0][5] = new Bishop(Player.BLACK);
        board[7][2] = new Bishop(Player.WHITE);
        board[7][5] = new Bishop(Player.WHITE);

        // creates 2 new King objects belonging to Black player and
        // 2 new King objects belonging to White player
        board[0][4] = new King(Player.BLACK);
        board[7][4] = new King(Player.WHITE);

        // creates 2 new Queen objects belonging to Black player and
        // 2 new Queen objects belonging to White player
        board[0][3] = new Queen(Player.BLACK);
        board[7][3] = new Queen(Player.WHITE);

        // sets current player instance variable to white
        // Chess game rules dictate that white always goes first
        this.player = Player.WHITE;

    }

    /**************************************************************************
     * Constructor for ChessModel game that accepts a pair of
     * two-dimensional arrays of type String and a String variable as
     * parameters. Creates two dimensional array (board) of IChessPiece
     * objects according to constant for board size. Then instantiates
     * Pawn, Rook, Knight, Bishop, King, and Queen objects in proper
     * locations on board according to parameters passed which "load" a
     * board that was "saved." Sets current player to White (all Chess
     * games start with White player).
     *
     * @param type
     *            Two-dimensional array of String type determining what
     *            type of IChessPiece object.
     * @param who
     *            Two-dimensional array of String type determining what
     *            player owns each chess piece.
     * @param turn
```

```
     *            String determining which player's turn it is.
     ************************************************************************/
    public ChessModel(String[][] type, String[][] who, String turn) {

        // creates new board of chess pieces using constant for size
        board = new IChessPiece[BOARD_SIZE][BOARD_SIZE];

        // loop through the board
        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board.length; col++) {

                // if type is King and player is Black create new
                // King object belonging to Black player at that
                // location
                if (type[row][col].equals("King")
                        && who[row][col].equals("BLACK"))
                    board[row][col] = new King(Player.BLACK);

                // else if type King and player White, new White King
                else if (type[row][col].equals("King")
                        && who[row][col].equals("WHITE"))
                    board[row][col] = new King(Player.WHITE);

                // else if type Queen and player Black, new Black Queen
                else if (type[row][col].equals("Queen")
                        && who[row][col].equals("BLACK"))
                    board[row][col] = new Queen(Player.BLACK);

                // else if type Queen and player White, new White Queen
                else if (type[row][col].equals("Queen")
                        && who[row][col].equals("WHITE"))
                    board[row][col] = new Queen(Player.WHITE);

                // else if type Bishop and player Black, new Black
                // Bishop
                else if (type[row][col].equals("Bishop")
                        && who[row][col].equals("BLACK"))
                    board[row][col] = new Bishop(Player.BLACK);

                // else if type Bishop and player White, new White
```

```
// Bishop
else if (type[row][col].equals("Bishop")
        && who[row][col].equals("WHITE"))
    board[row][col] = new Bishop(Player.WHITE);

// else if type Knight and player Black, new Black
// Knight
else if (type[row][col].equals("Knight")
        && who[row][col].equals("BLACK"))
    board[row][col] = new Knight(Player.BLACK);

// else if type Knight and player White, new White
// Knight
else if (type[row][col].equals("Knight")
        && who[row][col].equals("WHITE"))
    board[row][col] = new Knight(Player.WHITE);

// else if type Rook and player Black, new Black Rook
else if (type[row][col].equals("Rook")
        && who[row][col].equals("BLACK"))
    board[row][col] = new Rook(Player.BLACK);

// else if type Rook and player White, new White Rook
else if (type[row][col].equals("Rook")
        && who[row][col].equals("WHITE"))
    board[row][col] = new Rook(Player.WHITE);

// else if type Pawn and player Black, new Black Pawn
else if (type[row][col].equals("Pawn")
        && who[row][col].equals("BLACK"))
    board[row][col] = new Pawn(Player.BLACK);

// else if type Pawn and player White, new White Pawn
else if (type[row][col].equals("Pawn")
        && who[row][col].equals("WHITE"))
    board[row][col] = new Pawn(Player.WHITE);

// else if type "*" and player "*" (no piece or player),
// set piece to null since no ChessPiece object there
else if (type[row][col].equals("*")
```

```
                    && who[row][col].equals("*")) {
                board[row][col] = null;
            }
        }
    }

    // if turn is white, sets current player to white
    // else if turn is black sets current player to black
    if (turn.equals("WHITE"))
        player = Player.WHITE;
    else if (turn.equals("BLACK"))
        player = Player.BLACK;
}

/**********************************************************************
 * Public method that returns the board object instance variable
 * (two-dimensional array of IChessPiece objects).
 *
 * @return IChessPiece[][] two-dimensional array of IChessPiece
 *         objects holding the pieces on the board.
 **********************************************************************/
public IChessPiece[][] getBoard() {

    // return board (two-dimensional array of IChessPiece objects)
    return board;
}

/**********************************************************************
 * Public method that returns the gameComplete instance variable
 * telling whether the game is complete or not (a player is in
 * checkmate or not).
 *
 * @return Boolean game status of whether game is complete or not
 **********************************************************************/
public boolean getGameStatus() {

    // return whether game is complete or not
    return gameComplete;
}
```

```java
/**********************************************************************
 * Public method that returns true if the game is complete or false
 * if the game is not complete. If a player is in checkmate (there
 * are no more valid moves they can make to get his or her king out
 * of check), then game is complete. If player can make one or more
 * valid moves to get out of check, then game is not complete.
 *
 * @return true if complete, false otherwise.
 **********************************************************************/
public boolean isComplete() {

    // game is complete unless a valid move is found in this method
    // that results in king not being in check, flag will be
    // returned
    boolean flag = true;

    // loop through the board
    for (int row = 0; row < numRows(); row++) {
        for (int col = 0; col < numColumns(); col++) {

            // if there is a piece at that location and that piece
            // belongs to the current player create temporary piece
            if (board[row][col] != null
                    && board[row][col].player() == this.player) {
                IChessPiece piece = board[row][col];

                // create a new move to each spot on board for that
                // piece
                for (int r = 0; r < numRows(); r++) {
                    for (int c = 0; c < numColumns(); c++) {
                        Move m = new Move(row, col, r, c);

                        // if that piece's move is valid on board
                        if (piece.isValidMove(m, board)) {

                            // if that potential move does not leave
                            // own player's king in check, set
                            // returned to false
                            if (!ownKingCheck(m, this.player))
                                flag = false;
```

```
                    }

                }
            }
        }
    }

    // if get to this point, return true
    return flag;
}

/*************************************************************
 * Public method that returns true if the Move object is able to be
 * performed. Otherwise, returns false if the move is not able to be
 * carried out according to the game and chess pieces rules.
 *
 * @param move
 *            a Move object describing the move to be made.
 * @return Boolean whether move is valid or not
 * @throws IndexOutOfBoundsException
 *             if either [move.fromRow, move.fromColumn] or
 *             [move.toRow, move.toColumn]} don't represent valid
 *             locations on the board.
 *************************************************************/
public boolean isValidMove(Move move) {

    // if either [move.fromRow][move.fromColumn] or
    // [move.toRow][move.toColumn] don't represent valid
    // locations on board, throw exception
    if ((move.fromRow > numRows() || move.fromRow < 0)
            || (move.fromColumn > numColumns()
                    || move.fromColumn < 0)
            || (move.toRow > numRows() || move.toRow < 0)
            || (move.toColumn > numColumns()
                    || move.toColumn < 0)) {

        // throw new IndexOutOfBoundsException
        throw new IndexOutOfBoundsException();
    }
```

```
        // if own move leaves own king in check, then can't make move
        if (!ownKingCheck(move, player)) {

            // only create temporary piece at location moving from if
            // a piece exists at that location
            if (board[move.fromRow][move.fromColumn] != null) {
                IChessPiece piece =
                        board[move.fromRow][move.fromColumn];

                // if temporary piece makes valid move, return true
                if (piece.isValidMove(move, board))
                    return true;
            }
        }

        // otherwise, if get to this point, return false
        return false;
    }

    /**********************************************************************
     * Moves the piece from location [move.fromRow, move.fromColumn] to
     * location [move.fromRow, move.fromColumn].
     *
     * @param move
     *            a Move object describing the move to be made.
     * @throws IndexOutOfBoundsException
     *             if either [move.fromRow, move.fromColumn] or
     *             [move.toRow, move.toColumn] don't represent valid
     *             locations on the board.
     **********************************************************************/
    public void move(Move move) {

        // if either [move.fromRow][move.fromColumn] or
        // [move.toRow][move.toColumn] don't represent valid
        // locations on board, throw exception
        if ((move.fromRow > numRows() || move.fromRow < 0)
                || (move.fromColumn > numColumns()
                || move.fromColumn < 0)
                || (move.toRow > numRows() || move.toRow < 0)
```

```
                || (move.toColumn > numColumns()
                        || move.toColumn < 0)) {

            // throw new IndexOutOfBoundsException
            throw new IndexOutOfBoundsException();
        }

        // create temporary piece from location moving from
        IChessPiece piece = board[move.fromRow][move.fromColumn];

        // if the Move object is valid and the player who owns the piece
        // is the current player, then move the piece
        if (isValidMove(move) && piece.player() == player) {

            // copy piece from location moving to captured piece
            // location
            board[move.toRow][move.toColumn] =
                    board[move.fromRow][move.fromColumn];

            // remove original piece moving
            board[move.fromRow][move.fromColumn] = null;

            // now switch player's turn
            this.player = player.next();

            // check to see if opponent of player who just moved is in
            // check. If true, ask if game is complete.
            if (inCheck(this.player)) {
                gameComplete = isComplete();
            }
        }

    }

    /**************************************************************************
     * Private helper method that determines whether it is a valid move
     * from location [move.fromRow, move.fromColumn] to location
     * [move.fromRow, move.fromColumn].
     *
     * @param move
```

```
     *            a Move object describing the move to be made.
     * @return true if valid move, false otherwise.
     ***********************************************************************/
    private boolean isValidMoveKing(Move move) {

        // only create temporary piece at from location if exists
        if (board[move.fromRow][move.fromColumn] != null) {

            IChessPiece piece = board[move.fromRow][move.fromColumn];

            // return true if piece moving is valid
            if (piece.isValidMove(move, board))
                return true;
        }

        // otherwise, return false
        return false;
    }

    /***********************************************************************
     * Private helper method that determines where a player's king is
     * located at based on which player is passed as parameter.
     *
     * @param p
     *            a Player object determining which king to find.
     ***********************************************************************/
    private void findKing(Player p) {

        // loop through board
        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board.length; col++) {

                // only create temporary piece if exists
                if (board[row][col] != null) {
                    IChessPiece piece = board[row][col];

                    // if that piece is a king and of current player
                    // then put location into instance variables
                    if (piece.type().equals("King")
                            && piece.player() == p) {
```

```
                    kingRow = row;
                    kingCol = col;
                }
            }
        }
    }
}

/**********************************************************************
 * Public method that determines whether the player passed in as
 * parameter is currently in check (player's king is in check).
 *
 * @param p
 *            a Player object determining which king to ask if
 *            inCheck
 * @return true if king is in check, false otherwise
 **********************************************************************/
public boolean inCheck(Player p) {

    // finds king of player passed in as parameter
    findKing(p);

    // loop through board
    for (int row = 0; row < numRows(); row++) {
        for (int col = 0; col < numColumns(); col++) {

            // only create temporary move if piece exists at
            // location
            if (pieceAt(row, col) != null) {
                Move tempMove =
                        new Move(row, col, kingRow, kingCol);

                // if valid move to take king with piece, return
                // true
                if (isValidMoveKing(tempMove))
                    return true;
            }
        }
    }
```

```
    // otherwise, return false
    return false;
}

/**********************************************************************
 * Private helper method that determines where a player's king is
 * located at based on which player is passed as parameter.
 *
 * @param temp
 *            Temporary board (IChess[][]) of chess pieces.
 * @param move
 *            a Move object describing the move to be made.
 * @return true if move is valid, otherwise false
 **********************************************************************/
private boolean isValidTempMove(IChessPiece[][] temp, Move move) {

    // only create temporary piece if exists at from location
    if (temp[move.fromRow][move.fromColumn] != null) {
        IChessPiece piece = temp[move.fromRow][move.fromColumn];

        // if move is valid on temporary board, return true
        if (piece.isValidMove(move, temp))
            return true;

    }

    // otherwise, return false
    return false;
}

/**********************************************************************
 * Private helper method that determines where a player's king is
 * located at on a temporary board based on which player is passed
 * as parameter.
 *
 * @param temp
 *            Temporary board (IChess[][]) of chess pieces.
 * @param p
 *            a Player object determining which temporary king to
 *            find
```

```
     ***************************************************************/
    private void findTempKing(IChessPiece[][] temp, Player p) {

        // loop through temporary board
        for (int row = 0; row < temp.length; row++) {
            for (int col = 0; col < temp.length; col++) {

                // only create temporary piece if exists at location
                if (temp[row][col] != null) {
                    IChessPiece piece = temp[row][col];

                    // if that piece is a king and of current player,
                    // put location into instance variables
                    if (piece.type().equals("King")
                            && piece.player() == p) {
                        tempKingRow = row;
                        tempKingCol = col;
                    }
                }
            }
        }
    }

    /*****************************************************************
     * Private helper method that determines whether a potential move
     * puts that player's own king in check based on results of move.
     * Creates a temporary board to assess results and returns boolean
     * based on what found.
     *
     * @param p
     *            a Player object determining which king to check
     * @param move
     *            a Move object describing the move to be made.
     * @return true if move puts own king into check, otherwise false
     ****************************************************************/
    private boolean ownKingCheck(Move move, Player p) {

        // set variable to return to false
        boolean flag = false;
```

```
        // create a temporary chess board and fill with pieces from
        // actual board of game
        IChessPiece[][] temp = new IChessPiece[BOARD_SIZE][BOARD_SIZE];
        for (int r = 0; r < numRows(); r++) {
            for (int c = 0; c < numColumns(); c++) {
                temp[r][c] = pieceAt(r, c);
            }
        }

        // make the temporary move, moving piece from into to place
        temp[move.toRow][move.toColumn] =
                temp[move.fromRow][move.fromColumn];
        temp[move.fromRow][move.fromColumn] = null;

        // find where the temporary king of player is on temporary board
        findTempKing(temp, p);

        // now check if any pieces can take that color's king by
        // looping through temporary board
        for (int row = 0; row < numRows(); row++) {
            for (int col = 0; col < numColumns(); col++) {

                // only create a move if location holds a chess piece
                if (temp[row][col] != null) {
                    Move attackMove = new Move(row, col, tempKingRow,
                            tempKingCol);

                    // if there are valid moves to attack king after
                    // moving temporary piece, return true
                    if (isValidTempMove(temp, attackMove))
                        flag = true;
                }
            }
        }

        // otherwise return false
        return flag;
    }

    /*************************************************************************
```

```
     * A method that returns the Player object instance variable, which
     * will return the current player in the game.
     *
     * @return the current player.
     ************************************************************************/
    public Player currentPlayer() {

        // returns current player in game
        return player;
    }

    /************************************************************************
     * A method that returns the number of rows in the game, which is
     * equal to the constant instance variable for board size.
     *
     * @return Integer representing number of rows on board
     ************************************************************************/
    public int numRows() {

        // returns the number of rows, which is equal to the constant
        // instance variable for board size
        return BOARD_SIZE;
    }

    /************************************************************************
     * A method that returns the number of columns in the game, which is
     * equal to the constant instance variable for board size.
     *
     * @return Integer representing number of columns on board
     ************************************************************************/
    public int numColumns() {

        // returns the number of columns, which is equal to the constant
        // instance variable for board size
        return BOARD_SIZE;
    }

    /************************************************************************
     * A method that gets the IChessPiece object at location on board
     * using the values passed as parameters to determine the indices of
```

```
     * the two dimensional array for the board.
     *
     * @param row
     *            The row numbered 0 through numRows -1
     * @param col
     *            The column numbered 0 through numColumns -1
     * @return the ChessPiece object at location [row, column].
     * @throws IndexOutOfBoundsException
     *             if [row, column] is not a valid location on the
     *             board.
     ***************************************************************/
    public IChessPiece pieceAt(int row, int column) {

        // if row is less than zero or greater than the number of rows,
        // or if col is less than zero or greater than the number of
        // columns, throw an IndexOutOfBoundsException, else return
        // IChessPiece object located at position on board
        if (row < 0 || column < 0 || row > numRows()
                || column > numColumns()) {
            throw new IndexOutOfBoundsException();
        } else
            return board[row][column];
    }

    /***************************************************************
     * A method that returns a boolean based on whether there is an
     * IChessPiece object at location on board using the values passed
     * as parameters to determine the indices of the two dimensional
     * array for the board. Invokes pieceAt(int row, int column) method
     *
     * @param row
     *            Index for row that piece is located at on board
     * @param col
     *            Index for column that piece is located at on board
     * @return Boolean whether there is a piece at location or not
     ***************************************************************/
    public boolean hasPiece(int row, int column) {

        // if there is a IChessPiece object at location on board,
        // return true, else return false
```

```
        if (pieceAt(row, column) != null)
            return true;
        else
            return false;

    }



}
```

```java
package chess;

import java.awt.BorderLayout;

/**************************************************************************
 * ChessPanel
 * Class CIS 163-03
 * Project 3
 *
 * Chess panel. According to the Model View Controller software pattern,
 * this class controls the model (game). One exception being the chess
 * piece classes contain image icons which are used in the panel class
 * to emphasize polymorphism.
 *
 * @author Michael Baldwin , Douglas Money, Nick Reitz
 * @version Winter 2014
 *
 *************************************************************************/
public class ChessPanel extends JPanel {

    /** JButton 2-D array for game board */
    private JButton[][] board;

    /** Game Model */
    private ChessModel model;

    /** boolean array for highlighting */
    private static boolean[][] isChecked;

    /** JFrame panel */
    private JPanel center;

    /** JFrame panel */
    private JPanel right;

    /** JFrame panel */
    private JPanel top;

    /** JFrame panel */
    private JPanel bottom;
```

```java
/** Move passed into game engine */
private Move move;

/** Button Listener */
private ButtonListener listener;

/** Mouse Listener */
private MouseListen mouseListener;

/** JLabel for updating current players turn */
private JLabel turnLabel;

/** JMenuBar */
private JMenuBar menuBar;

/** JMenu */
private JMenu file;

/** JMenu */
private JMenu quickMates;

/** JMenuItem */
private JMenuItem save;

/** JMenuItem */
private JMenuItem load;

/** JMenuItem */
private JMenuItem newGame;

/** JMenuItem */
private JMenuItem exitGame;

/** JMenuItem */
private JMenuItem hippoCheck;

/** JMenuItem */
private JMenuItem legallsCheck;
```

```java
/**********************************************************************
 * Default constructor used for Chess Game GUI. Sets up properties
 * such as JMenuBars and JPanels and instantiates new game. Also
 * sets up initial game board
 **********************************************************************/
public ChessPanel() {

    // Set Panels
    right = new JPanel();
    top = new JPanel();
    center = new JPanel();
    bottom = new JPanel();

    // Sets Layouts
    this.setLayout(new BorderLayout());
    top.setLayout(new FlowLayout());
    bottom.setLayout(new FlowLayout());
    right.setLayout(new BorderLayout());

    // Add Panel
    add(center, BorderLayout.CENTER);
    add(top, BorderLayout.NORTH);
    add(bottom, BorderLayout.SOUTH);
    add(right, BorderLayout.EAST);

    // Setup JMenus
    menuBar = new JMenuBar();
    file = new JMenu("File");
    quickMates = new JMenu("Quick Mates");
    save = new JMenuItem("Save Game");
    load = new JMenuItem("Load Game");
    exitGame = new JMenuItem("Exit Game");
    newGame = new JMenuItem("New Game");
    hippoCheck = new JMenuItem("Hippo Check");
    legallsCheck = new JMenuItem("Legall's Check");

    // add(menuBar);
    menuBar.add(file);
    menuBar.add(quickMates);
```

```
file.add(newGame);
file.add(save);
file.add(load);
file.add(exitGame);

quickMates.add(hippoCheck);
quickMates.add(legallsCheck);

// board,model, and move instantiation

// blackGraveBoard = new JButton[][];
board = new JButton[8][8];
model = new ChessModel();
move = new Move();

turnLabel = new JLabel();
top.add(turnLabel);
turnLabel.setText(model.currentPlayer() + " Player's Turn");

// Listeners
listener = new ButtonListener();
mouseListener = new MouseListen();

legallsCheck.addActionListener(listener);
hippoCheck.addActionListener(listener);

save.addActionListener(listener);
load.addActionListener(listener);
newGame.addActionListener(listener);
exitGame.addActionListener(listener);

isChecked = new boolean[8][8];

// Initial Board Setup
for (int row = 0; row < board.length; row++)
    for (int col = 0; col < board.length; col++) {
        board[row][col] = new JButton("");
        board[row][col].addActionListener(listener);
        board[row][col].addMouseListener(mouseListener);
    }
```

```
    displayBoard();

}

/*****************************************************************
 * Visual representation of Chess that handles set up of Board.
 * Handles square colors, and piece Icon setup. This method will
 * update the piece ImageIcons via Polymorphism, but in theory will
 * violate MVC. For purposes of this project we feel it works
 *****************************************************************/
private void displayBoard() {

    center.removeAll();
    center.revalidate();
    center.repaint();

    for (int row = 0; row < board.length; row++)
        for (int col = 0; col < board.length; col++) {

            // resets and sets boarders
            board[row][col].setBorder(null);
            board[row][col].setBorder(BorderFactory
                    .createMatteBorder(1, 1, 1, 1, Color.BLACK));

            // handles square colors
            if ((row % 2 == 0 && col % 2 == 0)
                    || (col % 2 == 1 && row % 2 == 1)) {

                board[row][col].setBackground(Color.GRAY);
                board[row][col].setOpaque(true);

            } else {
                board[row][col].setBackground(Color.WHITE);
                board[row][col].setOpaque(true);
            }

            // handles pieceIcons
            if (model.hasPiece(row, col)
                    && model.pieceAt(row, col).player()
```

```
                        .equals(Player.WHITE)) {

                board[row][col].setIcon(model.pieceAt(row, col)
                        .whiteIcon());
                board[row][col].setText(null);
            }

            else if (model.hasPiece(row, col)
                    && model.pieceAt(row, col).player()
                            .equals(Player.BLACK)) {

                board[row][col].setIcon(model.pieceAt(row, col)
                        .blackIcon());
                board[row][col].setText(null);

            }

            else {
                board[row][col].setIcon(null);
                board[row][col].setText(null);

            }

            highLight();

            // re-add panel after reset and update current player
            // turn label
            center.add(board[row][col]);
            center.add(board[row][col]);
            turnLabel.setText(model.currentPlayer()
                    + " Player's Turn");
            center.setLayout(new GridLayout(8, 8));
            center.setBorder(new EmptyBorder(15, 70, 50, 50));
            top.setBorder(new EmptyBorder(15, 10, 10, 10));

        }

}

/*****************************************************************
```

```
     * Changes surrounding boarder of the "clicked" cell and also legal
     * moves that the specific piece can move to. Called in
     * displayBoard()
     ************************************************************/

    private void highLight() {

        // makes every valid move on the board and changes the boarder
        // color
        // if it is a valid move
        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board.length; col++) {

                // avoids null pointer
                if (board[row][col] != null
                        && model.pieceAt(row, col) != null) {

                    // boolean array is updated in mouse event
                    // controller
                    if (isChecked[row][col]
                            && model.pieceAt(row, col).player()
                                    .equals(model.currentPlayer())) {

                        board[row][col].setBorder(BorderFactory
                                .createMatteBorder(4, 4, 4, 4,
                                        Color.red));

                        for (int r = 0; r < board.length; r++)
                            for (int c = 0; c < board.length; c++) {

                                // sets up temporary move
                                Move tempMove = new Move(row, col, r, c);

                                // valid move results in highlighted
                                // squares
                                if (model.isValidMove(tempMove)) {

                                    board[r][c].setBorder(BorderFactory
                                            .createMatteBorder(4, 4, 4,
                                                    4, Color.red));
```

```
                        }

                    }

                }

            }
        }
    }
}

/**********************************************************************
 * Method that saves the current game state to a text document, used
 * in JMenu
 **********************************************************************/
private void saveGame() {

    PrintWriter out = null;

    try {

        out = new PrintWriter(new BufferedWriter(new FileWriter(

        "pieces.txt")));

    } catch (IOException e) {

        e.printStackTrace();

        System.out.println("sory");

    }

    // prints in the document
    for (int row = 0; row < board.length; row++)
        for (int col = 0; col < board.length; col++) {

            if (model.hasPiece(row, col)) {
```

```
                out.println(model.pieceAt(row, col).type());

                // System.out.println(model.pieceAt(row,
                // col).type());

            } else {

                out.println("*");

                // System.out.println("*");

            }
        }

    out.println(model.currentPlayer());

    // closes the document
    out.close();

    try {

        out = new PrintWriter(new BufferedWriter(new FileWriter(

        "player.txt")));

    } catch (IOException e) {

        e.printStackTrace();

        System.out.println("sory");

    }

    // prints in the document
    for (int row = 0; row < board.length; row++)
        for (int col = 0; col < board.length; col++) {

            if (model.hasPiece(row, col)) {

                out.println(model.pieceAt(row, col).player());
```

```
                    // System.out
                    // .println(model.pieceAt(row, col).player());

                } else {

                    out.println("*");

                    // System.out.println("*");

                }

            }

        // closes the document
        out.close();

        try {

            // writes or "saves" current players turn to .txt
            out = new PrintWriter(new BufferedWriter(new FileWriter(

            "turn.txt")));

        } catch (IOException e) {

            e.printStackTrace();

            System.out.println("sorry");

        }

        out.print(model.currentPlayer());
        out.close();

    }

    /**********************************************************************
     * Method that loads the saved game state from a text document, used
     * in JMenu
```

```
 ***************************************************************/
private void loadGame() {
    String[][] type = new String[8][8];
    String[][] who = new String[8][8];
    String turn = "";
    Scanner fileReader;
    try {
        fileReader = new Scanner(new File("pieces.txt"));
        for (int row = 0; row < board.length; row++)
            for (int col = 0; col < board.length; col++) {
                type[row][col] = fileReader.nextLine();
                // System.out.print(type[row][col]);
            }
    } catch (Exception e) {
        JOptionPane.showMessageDialog(null,
                "No Games Saved");
    }
    try {
        fileReader = new Scanner(new File("player.txt"));
        for (int row = 0; row < board.length; row++)
            for (int col = 0; col < board.length; col++) {
```

```
                who[row][col] = fileReader.nextLine();
                // System.out.print(type[row][col]);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        try {
            fileReader = new Scanner(new File("turn.txt"));
            turn = fileReader.nextLine();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        model = new ChessModel(type, who, turn);
        displayBoard();
    }

    /*****************************************************************
     * Method that loads pre-made game states to a text document, used
     * in JMenu
     *****************************************************************/

    private void loadHippoMate() {
        String[][] type = new String[8][8];
```

```
String[][] who = new String[8][8];
String turn = "";
Scanner fileReader;
try {
    fileReader = new Scanner(new File("hippoPieces.txt"));
    for (int row = 0; row < board.length; row++)
        for (int col = 0; col < board.length; col++) {
            type[row][col] = fileReader.nextLine();
            // System.out.print(type[row][col]);
        }
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
try {
    fileReader = new Scanner(new File("hippoPlayer.txt"));
    for (int row = 0; row < board.length; row++)
        for (int col = 0; col < board.length; col++) {
            who[row][col] = fileReader.nextLine();
            // System.out.print(type[row][col]);
        }
```

```
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        try {
            fileReader = new Scanner(new File("hippoTurn.txt"));
            turn = fileReader.nextLine();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        model = new ChessModel(type, who, turn);
        displayBoard();
    }

    /**************************************************************
     * Method that loads pre-made game states to a text document, used
     * in JMenu
     **************************************************************/

    private void loadLegallsMate() {
        String[][] type = new String[8][8];
        String[][] who = new String[8][8];
        String turn = "";
        Scanner fileReader;
        try {
```

```
        fileReader = new Scanner(new File("legallsPieces.txt"));
        for (int row = 0; row < board.length; row++)
            for (int col = 0; col < board.length; col++) {
                type[row][col] = fileReader.nextLine();
                // System.out.print(type[row][col]);
            }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    try {
        fileReader = new Scanner(new File("legallsPlayers.txt"));
        for (int row = 0; row < board.length; row++)
            for (int col = 0; col < board.length; col++) {
                who[row][col] = fileReader.nextLine();
                // System.out.print(type[row][col]);
            }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    try {
        fileReader = new Scanner(new File("legallsTurn.txt"));
```

```
            turn = fileReader.nextLine();

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        model = new ChessModel(type, who, turn);

        displayBoard();

    }

    /**********************************************************************
     * Method that creates new game of chess
     **********************************************************************/

    private void newGame() {

        model = new ChessModel();

        displayBoard();

    }

    /**********************************************************************
     * Getter Method called in ChessGUI.java for placing bar accordingly
     *
     * @return menuBar returns JMenuBar
     **********************************************************************/

    public JMenuBar getJMenuBar() {
        return menuBar;

    }

    /**********************************************************************
     * Action Listener class that controls JMenu options i.e saveGame,
     * loadGame, newGame
     **********************************************************************/
```

```
private class ButtonListener implements ActionListener {

    public void actionPerformed(ActionEvent event) {

        if (event.getSource() == save) {
            saveGame();
        }
        if (event.getSource() == load) {
            loadGame();
        }
        if (event.getSource() == newGame) {
            newGame();
        }

        if (event.getSource() == exitGame) {
            System.exit(0);
        }

        if (event.getSource() == hippoCheck) {
            loadHippoMate();
        }
        if (event.getSource() == legallsCheck) {
            loadLegallsMate();
        }

    }

}

/*****************************************************************
 * Mouse Listener class that controls setting up moves to be passed
 * into the Chess Game Engine. If a Player goes into check JDialog
 * will let you know, also if game is over JDiloag will let you know
 *****************************************************************/
private class MouseListen implements MouseListener {

    // "click counter" needed for creating piece moves
    boolean secondClick = true;
```

```
// move from row
int fromRow;

// move from column
int fromCol;

// move to row
int toRow;

// move to column
int toCol;

/*****************************************************************
 * Method not used
 * @param event not used
 *****************************************************************/

public void mouseClicked(MouseEvent event) {

}

/*****************************************************************
 * Method not used
 * @param event not used
 *****************************************************************/

public void mousePressed(MouseEvent event) {

}

/*****************************************************************
 * Sets up new move for use in game engine. Then passes move
 * into game.Updates isChecked[][] used for saving and loading
 * game states. Uses JDialog boxes to warn user of Check and
 * CheckMate situations.
 * @param event gets "clicked" location on board
 *****************************************************************/

public void mouseReleased(MouseEvent event) {
```

```
for (int r = 0; r < board.length; r++)
    for (int c = 0; c < board.length; c++) {

        // boolean array used for highlighting
        isChecked[r][c] = false;

        // sets up move and sends it to game
        if (board[r][c] == event.getSource()
                && !secondClick) {

            toRow = r;
            toCol = c;

            move = new Move(fromRow, fromCol, toRow, toCol);

            model.move(move);

            // is king in check
            if (model.inCheck(model.currentPlayer())) {
                JOptionPane.showMessageDialog(null,
                        model.currentPlayer()
                                + " king in check");
            }

            // is game over? it is? ok.. I hope you won
            // create new game
            if (model.getGameStatus()) {

                JOptionPane.showMessageDialog(null,
                        "Checkmate "
                                + model.currentPlayer()
                                        .next()
                                + " has won!! "
                                + "Please select New "
                                + "Game from file menu");
                // newGame();
            }

            secondClick = true;
```

```
                    // sets up move
                } else if (board[r][c] == event.getSource()
                        && model.hasPiece(r, c)) {

                    isChecked[r][c] = true;
                    fromRow = r;
                    fromCol = c;
                    secondClick = false;
                }

            }

        displayBoard();

    }

    /*************************************************************
     * Method not used
     * @param event not used
     *************************************************************/

    public void mouseEntered(MouseEvent arg0) {

    }

    /*************************************************************
     * Method not used
     * @param event  not used
     *************************************************************/

    public void mouseExited(MouseEvent arg0) {

    }

}

}
```

```java
package chess;

import java.awt.Dimension;

/**************************************************************************
 * ChessGUI Class CIS 163-03 Project 3
 *
 * ChessGUI initializer for Chess Game
 *
 * @author Douglas Money, Nick Reitz, Michael Baldwin
 * @version Winter 2014
 *************************************************************************/
public class ChessGUI {

    /**************************************************************************
     * Main Method for the Chess GUI, Method sets up ChessPanel and
     * brings in bars
     *
     * @param args
     *             main args
     *************************************************************************/
    public static void main(String[] args) {
        JFrame frame = new JFrame("Chess Game");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //initialize panel
        ChessPanel panel = new ChessPanel();
        frame.setJMenuBar(panel.getJMenuBar());
        frame.getContentPane().add(panel);
        frame.setSize(new Dimension(800, 800));

        //center frame on screen
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

}
```

```java
package chess;

import org.junit.Test;

/**********************************************************************
 * ChessTest Class CIS 163-03 Project 3
 *
 * Junit Tests to test the accuracy and validity of the overall chess
 * game model, including mainly the ChessModel class, but also other
 * classes involved in the game, such as Pawn class, Rook class, King
 * class, Queen class, and Knight class, and Bishop class.
 *
 * @author Douglas Money, Nick Reitz, Michael Baldwin
 * @version Winter 2014
 *********************************************************************/
public class ChessTest {

    /** Two dimensional array of IChessPiece objects for test board */
    private IChessPiece[][] board;

    /** Move object for use in test methods */
    private Move move;

    /** ChessModel object for use in test methods */
    private ChessModel model;

    /**********************************************************************
     * Default constructor for ChessTest class. Instantiates a new two
     * dimensional array board of IChessPiece objects and a new
     * ChessModel object.
     *********************************************************************/
    public ChessTest() {

        // creates two dimensional array board of IChessPiece objects
        board = new IChessPiece[8][8];

        // instantiates a new ChessModel object
        model = new ChessModel();
    }
```

```java
/**********************************************************************
 * Method that resets the board of pieces. Instantiates a new a two
 * dimensional array board of IChessPiece objects.
 **********************************************************************/
public void resetBoard() {

    // creates a new 2-D array (board) of IChessPiece objects
    board = new IChessPiece[8][8];
}

/**********************************************************************
 * Test method that tests the Pawn class. Tests the valid and
 * invalid movements of a Pawn object in the chess game.
 **********************************************************************/
@Test
public void testPawnMoves() {

    // reset board first
    resetBoard();

    // instantiate Pawn object belonging to white player
    Pawn p = new Pawn(Player.WHITE);

    // put White pawn into place to test valid and invalid moves
    board[6][4] = p;

    // create Move object, tell White Pawn to move onto self
    // (invalid move) because can't move from and to same location,
    // assert move is invalid / false
    move = new Move(6, 4, 6, 4);
    assertFalse(p.isValidMove(move, board));

    // drop another white pawn in front of White Pawn and tell white
    // pawn move two forward (invalid move - because can't jump
    // over piece), assert move is invalid / false
    Pawn blockingPawn = new Pawn(Player.WHITE);
    board[5][4] = blockingPawn;
    move = new Move(6, 4, 4, 4);
    assertFalse(p.isValidMove(move, board));
```

```
// first remove blocking pawn from board
// tell White pawn two forward (valid move)
// assert that move is valid / true
board[5][4] = null;
move = new Move(6, 4, 4, 4);
assertTrue(p.isValidMove(move, board));

// Remove White pawn from old location, move pawn to new
// location
// White pawn to move one backward (invalid move)
// assert that move is invalid / false
board[6][4] = null;
board[4][4] = p;
move = new Move(4, 4, 5, 4);
assertFalse(p.isValidMove(move, board));

// White pawn to move sideways to the left (invalid move)
// assert that move is invalid / false
move = new Move(4, 4, 4, 3);
assertFalse(p.isValidMove(move, board));

// White pawn to move sideways to the right (invalid move)
// assert that move is invalid / false
move = new Move(4, 4, 4, 5);
assertFalse(p.isValidMove(move, board));

// White pawn to move forwards two (invalid move - because
// pawn already moved once), assert that move is invalid / false
move = new Move(4, 4, 2, 4);
assertFalse(p.isValidMove(move, board));

// White pawn to move forwards one (valid move)
// assert that move is valid / true
move = new Move(4, 4, 3, 4);
assertTrue(p.isValidMove(move, board));

// drop another White pawn onto board diagonally of White pawn
// White pawn to take diagonally second white pawn (invalid move
// - because can't take own player's pieces
board[4][4] = null;
```

```
        board[3][4] = p;
        Pawn whitePawn = new Pawn(Player.WHITE);
        board[2][4] = whitePawn;
        move = new Move(3, 4, 2, 4);
        assertFalse(p.isValidMove(move, board));

        // drop Black pawn onto board in front of White Pawn
        // White pawn to move forwards one (invalid move - because
        // Black pawn blocking move), assert that move is invalid /
        // false
        Pawn blackPawn = new Pawn(Player.BLACK);
        board[2][4] = blackPawn;
        move = new Move(3, 4, 2, 4);
        assertFalse(p.isValidMove(move, board));

        // drop Black pawn onto board diagonally front left of White
        // Pawn
        // White pawn to move diagonally forwards to the left (valid
        // move), assert that move is valid / true
        board[2][3] = blackPawn;
        move = new Move(3, 4, 2, 3);
        assertTrue(p.isValidMove(move, board));

        // drop Black pawn onto board diagonally front right of White
        // White pawn to move diagonally forwards to the right (valid
        // move), assert that move is valid / true
        board[3][4] = null;
        board[2][3] = p;
        board[1][4] = blackPawn;
        move = new Move(2, 3, 1, 4);
        assertTrue(p.isValidMove(move, board));
    }

    /**********************************************************************
     * Test method that tests the Rook class. Tests the valid and
     * invalid movements of a Rook object in the chess game.
     **********************************************************************/
    @Test
    public void testRookMoves() {
```

```
// reset board first
resetBoard();

// instantiate Rook object belonging to white player
Rook r = new Rook(Player.WHITE);

// put White rook into place to test valid and invalid moves
board[7][0] = r;

// create Move object, tell White Rook to move onto self
// (invalid move - because can't move from and to same location,
// assert move is invalid / false
move = new Move(7, 0, 7, 0);
assertFalse(r.isValidMove(move, board));

// drop White Pawn onto board in front of White Rook, tell rook
// to move to pawn (invalid move - because can't take own
// color's
// pieces), assert move is invalid / false
Pawn whitePawn = new Pawn(Player.WHITE);
board[6][0] = whitePawn;
move = new Move(7, 0, 6, 0);
assertFalse(r.isValidMove(move, board));

// White Rook to move past / jump over white pawn in the way,
// (invalid move - because shouldn't be able to jump over),
// assert move is invalid / false
move = new Move(7, 0, 5, 0);
assertFalse(r.isValidMove(move, board));

// White Rook to move diagonally up to the right (invalid move -
// because rook can't move on diagonals), assert invalid / false
move = new Move(7, 0, 6, 1);
assertFalse(r.isValidMove(move, board));

// drop black Pawn onto board to the right of White Rook, tell
// rook to move to black pawn (valid move), assert valid / true
Pawn blackPawn = new Pawn(Player.BLACK);
board[7][1] = blackPawn;
move = new Move(7, 0, 7, 1);
```

```
assertTrue(r.isValidMove(move, board));

// White Rook to move six spaces right (valid move), assert
// move is valid / true
board[7][0] = null;
board[7][1] = r;
move = new Move(7, 1, 7, 7);
assertTrue(r.isValidMove(move, board));

// White Rook to move three spaces left (valid move), assert
// move
// is valid / true
board[7][1] = null;
board[7][7] = r;
move = new Move(7, 7, 7, 4);
assertTrue(r.isValidMove(move, board));

// White Rook to move two forward (valid move), assert true
board[7][7] = null;
board[7][4] = r;
move = new Move(7, 4, 5, 4);
assertTrue(r.isValidMove(move, board));

// White Rook to move one backwards (valid move), assert true
board[7][4] = null;
board[5][4] = r;
move = new Move(5, 4, 6, 4);
assertTrue(r.isValidMove(move, board));

// move white Rook diagonally down to the left (invalid move -
// because Rook can't move diagonally), assert invalid / false
board[5][4] = null;
board[6][4] = r;
move = new Move(6, 4, 7, 3);
assertFalse(r.isValidMove(move, board));

// drop white pawn two ahead of white Rook and drop black pawn
// two more ahead of white pawn, move white Rook to black pawn
// (invalid move - because can't jump pieces), assert false
board[4][4] = whitePawn;
```

```
        board[2][4] = blackPawn;
        move = new Move(6, 4, 2, 4);
        assertFalse(r.isValidMove(move, board));

        // remove white pawn blocking, move white Rook to black pawn
        // (valid move), assert move is valid / true
        board[4][4] = null;
        move = new Move(6, 4, 2, 4);
        assertTrue(r.isValidMove(move, board));
    }

    /**********************************************************************
     * Test method that tests the King class. Tests the valid and
     * invalid movements of a King object in the chess game.
     **********************************************************************/
    @Test
    public void testKingMoves() {

        // reset board first
        resetBoard();

        // instantiate King object belonging to white player
        King k = new King(Player.WHITE);

        // put White king into place to test valid and invalid moves
        board[7][4] = k;

        // create Move object, tell White King to move onto self
        // (invalid move - because can't move from and to same location,
        // assert move is invalid / false
        move = new Move(7, 4, 7, 4);
        assertFalse(k.isValidMove(move, board));

        // drop White Pawn onto board in front of White king, tell king
        // to move to pawn (invalid move - because can't take own
        // color's
        // pieces), assert move is invalid / false
        Pawn whitePawn = new Pawn(Player.WHITE);
        board[6][4] = whitePawn;
        move = new Move(7, 4, 6, 4);
```

```
assertFalse(k.isValidMove(move, board));

// remove white Pawn and move white king one forward (valid
// move)
// , assert move is valid / true
board[6][4] = null;
move = new Move(7, 4, 6, 4);
assertTrue(k.isValidMove(move, board));

// move white king one sideways to the right (valid move),
// assert
// move is valid / true
board[7][4] = null;
board[6][4] = k;
move = new Move(6, 4, 6, 5);
assertTrue(k.isValidMove(move, board));

// move white king one sideways to the left (valid move), assert
// move is valid / true
board[6][4] = null;
board[6][5] = k;
move = new Move(6, 5, 6, 4);
assertTrue(k.isValidMove(move, board));

// move white king one backwards (valid move), assert move is
// valid / true
board[6][5] = null;
board[6][4] = k;
move = new Move(6, 4, 7, 4);
assertTrue(k.isValidMove(move, board));

// move white king one upwards right diagonally (valid move),
// assert move is valid / true
board[6][4] = null;
board[7][4] = k;
move = new Move(7, 4, 6, 3);
assertTrue(k.isValidMove(move, board));

// move white king one backwards left diagonally (valid move),
// assert move is valid / true
```

```
        board[7][4] = null;
        board[6][3] = k;
        move = new Move(6, 3, 7, 2);
        assertTrue(k.isValidMove(move, board));

        // move white king two forwards (invalid move - because king can
        // only ever move one space any way), assert move invalid /
        // false
        move = new Move(6, 3, 4, 3);
        assertFalse(k.isValidMove(move, board));

        // move white king two sideways left (invalid move - because
        // king
        // can move one space only), assert move invalid / false
        move = new Move(6, 3, 6, 1);
        assertFalse(k.isValidMove(move, board));

        // move white king two sideways right (invalid move - because
        // king can move one space only), assert move invalid / false
        move = new Move(6, 3, 6, 5);
        assertFalse(k.isValidMove(move, board));

        // move white king two diagonal up-left (invalid move - because
        // king can move one space only), assert move invalid / false
        move = new Move(6, 3, 4, 1);
        assertFalse(k.isValidMove(move, board));

        // move white king two diagonal up-right (invalid move - because
        // king can move one space only), assert move invalid / false
        move = new Move(6, 3, 4, 5);
        assertFalse(k.isValidMove(move, board));

        // drop Black Pawn onto board in diagonal left-down behind White
        // king, move king to pawn (valid move), assert valid / true
        Pawn blackPawn = new Pawn(Player.BLACK);
        board[7][2] = blackPawn;
        move = new Move(6, 3, 7, 2);
        assertTrue(k.isValidMove(move, board));
    }
```

```java
/**********************************************************************
 * Test method that tests the Queen class. Tests the valid and
 * invalid movements of a Queen object in the chess game.
 **********************************************************************/
@Test
public void testQueenMoves() {

    // reset board first
    resetBoard();

    // instantiate King object belonging to white player
    Queen q = new Queen(Player.WHITE);

    // put White king into place to test valid and invalid moves
    board[7][3] = q;

    // create Move object, tell White Queen to move onto self
    // (invalid move - because can't move from and to same location,
    // assert move is invalid / false
    move = new Move(7, 3, 7, 3);
    assertFalse(q.isValidMove(move, board));

    // drop White Pawn onto board in front of White Queen, queen
    // to move to pawn (invalid move - because can't take own
    // color's
    // pieces), assert move is invalid / false
    Pawn whitePawn = new Pawn(Player.WHITE);
    board[6][3] = whitePawn;
    move = new Move(7, 3, 6, 3);
    assertFalse(q.isValidMove(move, board));

    // remove white Pawn and move Queen one space forward (valid
    // move), assert move is valid / true
    board[6][3] = null;
    move = new Move(7, 3, 6, 3);
    assertTrue(q.isValidMove(move, board));

    // move white Queen three spaces forward (valid move), assert
    // move is valid / true
    board[7][3] = null;
```

```
board[6][3] = q;
move = new Move(6, 3, 3, 3);
assertTrue(q.isValidMove(move, board));

// move white Queen two spaces backwards (valid move), assert
// move is valid / true
board[6][3] = null;
board[3][3] = q;
move = new Move(3, 3, 5, 3);
assertTrue(q.isValidMove(move, board));

// move white Queen four spaces right (valid move), assert
// move is valid / true
board[3][3] = null;
board[5][3] = q;
move = new Move(5, 3, 5, 7);
assertTrue(q.isValidMove(move, board));

// move white Queen five spaces left (valid move), assert
// move is valid / true
board[5][3] = null;
board[5][7] = q;
move = new Move(5, 7, 5, 2);
assertTrue(q.isValidMove(move, board));

// move white Queen two spaces diagonally up-right (valid move),
// assert move is valid / true
board[5][7] = null;
board[5][2] = q;
move = new Move(5, 2, 3, 4);
assertTrue(q.isValidMove(move, board));

// move white Queen three spaces diagonally down-left (valid
// move), assert move is valid / true
board[5][2] = null;
board[3][4] = q;
move = new Move(3, 4, 6, 1);
assertTrue(q.isValidMove(move, board));

// move white Queen two spaces forward, one space right (invalid
```

```
        // move - because like a knight), assert move is invalid / false
        board[3][4] = null;
        board[6][1] = q;
        move = new Move(6, 1, 4, 2);
        assertFalse(q.isValidMove(move, board));

        // move white Queen one space forward, two spaces right (invalid
        // move - because like a knight), assert move is invalid / false
        move = new Move(6, 1, 5, 3);
        assertFalse(q.isValidMove(move, board));

        // drop black Pawn onto board four spaces diagonally up-right,
        // and drop white Pawn onto board blocking move, move Queen to
        // black pawn (invalid move - because can't jump pieces), assert
        // move is invalid / false
        Pawn blackPawn = new Pawn(Player.BLACK);
        board[2][5] = blackPawn;
        board[4][3] = whitePawn;
        move = new Move(6, 1, 2, 5);
        assertFalse(q.isValidMove(move, board));

        // remove white Pawn from board, move Queen to black pawn (valid
        // move), assert move is valid / true
        board[4][3] = null;
        move = new Move(6, 1, 2, 5);
        assertTrue(q.isValidMove(move, board));
    }

    /**************************************************************
     * Test method that tests the Knight class. Tests the valid and
     * invalid movements of a Knight object in the chess game.
     **************************************************************/
    @Test
    public void testKnightMoves() {

        // reset board first
        resetBoard();

        // instantiate Knight object belonging to white player
        Knight k = new Knight(Player.WHITE);
```

```
// put White knight into place to test valid and invalid moves
board[7][1] = k;

// create Move object, tell White Knight to move onto self
// (invalid move - because can't move from and to same location,
// assert move is invalid / false
move = new Move(7, 1, 7, 1);
assertFalse(k.isValidMove(move, board));

// drop White Pawn onto board, knight to move to white pawn
// (invalid move - because can't take own color's pieces),
// assert
// move is invalid / false
Pawn whitePawn = new Pawn(Player.WHITE);
board[5][2] = whitePawn;
move = new Move(7, 1, 5, 2);
assertFalse(k.isValidMove(move, board));

// drop Black pawn onto board, White knight to move to black
// pawn
// (valid move - because can move two forward, one sideways - in
// an L-shaped pattern), assert move is valid / true
Pawn blackPawn = new Pawn(Player.BLACK);
board[5][0] = blackPawn;
move = new Move(7, 1, 5, 0);
assertTrue(k.isValidMove(move, board));

// White knight to move to open space over white pawn (valid
// move
// - because knight can jump over pieces), assert move is true
board[7][1] = null;
board[5][0] = k;
move = new Move(5, 0, 4, 2);
assertTrue(k.isValidMove(move, board));

// White knight to move too far in an L-shaped pattern, move two
// forward, two sideways (invalid move), assert invalid / false
board[5][0] = null;
board[5][2] = null;
```

```
move = new Move(4, 2, 2, 4);
assertFalse(k.isValidMove(move, board));

// White knight to move up one, left two (valid move), assert
// move is valid / true
board[4][2] = k;
move = new Move(4, 2, 3, 0);
assertTrue(k.isValidMove(move, board));

// White knight to move down one, left two (valid move), assert
// move is valid / true
move = new Move(4, 2, 5, 0);
assertTrue(k.isValidMove(move, board));

// White knight to move up two, left one (valid move), assert
// move is valid / true
move = new Move(4, 2, 2, 1);
assertTrue(k.isValidMove(move, board));

// White knight to move down two, left one (valid move), assert
// move is valid / true
move = new Move(4, 2, 6, 1);
assertTrue(k.isValidMove(move, board));

// White knight to move down two, right one (valid move), assert
// move is valid / true
move = new Move(4, 2, 6, 3);
assertTrue(k.isValidMove(move, board));

// White knight to move down one, right two (valid move), assert
// move is valid / true
move = new Move(4, 2, 5, 4);
assertTrue(k.isValidMove(move, board));

// White knight to move up one, right two (valid move), assert
// move is valid / true
move = new Move(4, 2, 3, 4);
assertTrue(k.isValidMove(move, board));

// White knight to move up two, right one (valid move), assert
```

```
        // move is valid / true
        move = new Move(4, 2, 2, 3);
        assertTrue(k.isValidMove(move, board));

    }

    /**********************************************************************
     * Test method that tests the Bishop class. Tests the valid and
     * invalid movements of a Bishop object in the chess game.
     **********************************************************************/
    @Test
    public void testBishopMoves() {

        // reset board first
        resetBoard();

        // instantiate Bishop object belonging to white player
        Bishop b = new Bishop(Player.WHITE);

        // put White Bishop into place to test valid and invalid moves
        board[7][2] = b;

        // create Move object, tell White Bishop to move onto self
        // (invalid move - because can't move from and to same location,
        // assert move is invalid / false
        move = new Move(7, 2, 7, 2);
        assertFalse(b.isValidMove(move, board));

        // drop White Pawn onto board in front of White Bishop, bishop
        // to move to pawn (invalid move - because can't take own
        // color's
        // pieces), assert move is invalid / false
        Pawn whitePawn = new Pawn(Player.WHITE);
        board[6][3] = whitePawn;
        move = new Move(7, 2, 6, 3);
        assertFalse(b.isValidMove(move, board));

        // drop Black Pawn onto board in place of White pawn, White
        // Bishop to black pawn (valid move), assert valid / true
        Pawn blackPawn = new Pawn(Player.BLACK);
```

```
board[6][3] = blackPawn;
move = new Move(7, 2, 6, 3);
assertTrue(b.isValidMove(move, board));

// White Bishop to move sideways left (invalid move - bishop can
// only move diagonally), assert move is invalid / false
board[7][2] = null;
board[6][3] = b;
move = new Move(6, 3, 6, 2);
assertFalse(b.isValidMove(move, board));

// White Bishop to move sideways right (invalid move - bishop
// can
// only move diagonally), assert move is invalid / false
move = new Move(6, 3, 6, 4);
assertFalse(b.isValidMove(move, board));

// White Bishop to move up straight (invalid move - bishop can
// only move diagonally), assert move is invalid / false
move = new Move(6, 3, 7, 3);
assertFalse(b.isValidMove(move, board));

// White Bishop to move down straight (invalid move - bishop can
// only move diagonally), assert move is invalid / false
move = new Move(6, 3, 5, 3);
assertFalse(b.isValidMove(move, board));

// White Bishop to move down-right diagonally (valid move),
// assert move is valid / true
move = new Move(6, 3, 7, 4);
assertTrue(b.isValidMove(move, board));

// White Bishop to move up-left diagonally (valid move),
// assert move is valid / true
move = new Move(6, 3, 3, 0);
assertTrue(b.isValidMove(move, board));

// White Bishop to move up-right diagonally (valid move),
// assert move is valid / true
move = new Move(6, 3, 2, 7);
```

```
        assertTrue(b.isValidMove(move, board));

        // White Bishop to move down-left diagonally (valid move),
        // assert move is valid / true
        move = new Move(6, 3, 4, 5);
        assertTrue(b.isValidMove(move, board));

        // drop white pawn blocking White bishop from taking black pawn,
        // white bishop to black pawn (invalid move), assert false
        board[4][5] = whitePawn;
        board[2][7] = blackPawn;
        move = new Move(6, 3, 2, 7);
        assertFalse(b.isValidMove(move, board));

        // remove white pawn and move bishop to black pawn (valid move),
        // assert move is valid / true
        board[4][5] = null;
        move = new Move(6, 3, 2, 7);
        assertTrue(b.isValidMove(move, board));
    }

    /**********************************************************************
     * Test method that tests the inCheck method of the Model class.
     * Tests to see if once a piece is threatening an opponent's king if
     * the inCheck method works properly.
     **********************************************************************/
    @Test
    public void testInCheck() {

        // reset the model and set board according to the model class
        model = new ChessModel();
        board = model.getBoard();

        // move white pawn forward two
        move = new Move(6, 3, 4, 3);
        model.move(move);

        // move black pawn forward one
        move = new Move(1, 2, 2, 2);
        model.move(move);
```

```
        // assert that white king is in not in check yet
        assertFalse(model.inCheck(model.currentPlayer()));

        // move white pawn forward one
        move = new Move(6, 7, 5, 7);
        model.move(move);

        // move black queen diagonally, putting white king in check
        move = new Move(0, 3, 3, 0);
        model.move(move);

        // assert that white king is in check
        assertTrue(model.inCheck(model.currentPlayer()));

        // move bishop up to block black queen putting king in check
        move = new Move(7, 2, 6, 3);
        model.move(move);

        // assert that white king is no longer in check
        assertFalse(model.inCheck(model.currentPlayer()));

        // move black queen to take white bishop, putting king in check
        move = new Move(3, 0, 6, 3);
        model.move(move);

        // assert that white king is back in check
        assertTrue(model.inCheck(model.currentPlayer()));

        // move white queen to take black queen checking king
        move = new Move(7, 3, 6, 3);
        model.move(move);

        // assert that white king is no longer in check again
        assertFalse(model.inCheck(model.currentPlayer()));
    }

    /**********************************************************************
     * Test method that tests that once a player's king is put in check
     * whether that player can move certain pieces to certain locations.
```

```
     * While king is in check, player should only be able to move pieces
     * that will result in king not being in check, unless player is in
     * checkmate, which results in game being over.
     ************************************************************/
    @Test
    public void testMoveWhileInCheck() {

        // reset the model and set board according to the model class
        model = new ChessModel();
        board = model.getBoard();

        // move white pawn forward two
        move = new Move(6, 3, 4, 3);
        model.move(move);

        // move black pawn forward one
        move = new Move(1, 2, 2, 2);
        model.move(move);

        // move white pawn forward one
        move = new Move(6, 7, 5, 7);
        model.move(move);

        // move black queen diagonally, putting white king in check
        move = new Move(0, 3, 3, 0);
        model.move(move);

        // assert that white king is in check
        assertTrue(model.inCheck(model.currentPlayer()));

        // assert that moving pawn up one (not blocking check) is an
        // invalid move (can't move if king remains in check)
        move = new Move(6, 5, 5, 5);
        assertFalse(model.isValidMove(move));

        // assert that moving bishop up to block black queen that is
        // putting king in check is a valid move (king does not remain
        // in check)
        move = new Move(7, 2, 6, 3);
        assertTrue(model.isValidMove(move));
```

```
        // make move and assert that king is no longer in check
        model.move(move);
        assertFalse(model.inCheck(model.currentPlayer()));
    }

    /**********************************************************************
     * Test method that tests the isComplete method of the Model class.
     * If player is in checkmate, then game complete. Otherwise, if
     * player can make a move that results in his or her king not being
     * in check after move is made, then game is not complete (player is
     * not in checkmate).
     **********************************************************************/
    @Test
    public void testIsComplete() {

        // reset the model and set board according to the model class
        model = new ChessModel();
        board = model.getBoard();

        // move white pawn forward one
        move = new Move(6, 5, 5, 5);
        model.move(move);

        // move black pawn forward two
        move = new Move(1, 4, 3, 4);
        model.move(move);

        // assert that game is not complete
        assertFalse(model.getGameStatus());

        // move white pawn two forward
        move = new Move(6, 6, 4, 6);
        model.move(move);

        // move black queen to put white king in checkmate
        move = new Move(0, 3, 4, 7);
        model.move(move);

        // assert that game is complete
```

```
        assertTrue(model.getGameStatus());

        // reset the model and set board according to the model class
        model = new ChessModel();
        board = model.getBoard();

        // move white pawn forward two
        move = new Move(6, 4, 4, 4);
        model.move(move);

        // move black pawn forward two
        move = new Move(1, 6, 3, 6);
        model.move(move);

        // move white knight out
        move = new Move(7, 1, 5, 2);
        model.move(move);

        // assert that game is not complete
        assertFalse(model.getGameStatus());

        // move black pawn forward two
        move = new Move(1, 5, 3, 5);
        model.move(move);

        // move white queen out to check black king (checkmate)
        move = new Move(7, 3, 3, 7);
        model.move(move);

        // assert that game is now complete
        assertTrue(model.getGameStatus());
    }

}
```

```java
package chess;

import javax.swing.ImageIcon;

/**************************************************************************
 * ChessPiece Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * Chess Piece. According to the Model View Controller software pattern,
 * this class controls the Chess Pieces. One exception being the chess
 * piece classes contain image icons which are used in the panel class
 * to emphasize polymorphism.
 *
 * @author Michael Baldwin , Douglas Money, Nick Reitz
 * @version Winter 2014
 *
 *************************************************************************/

public abstract class ChessPiece implements IChessPiece {

    /** Player "black" or "white" */
    protected Player owner;

    /**************************************************************
     * Constructor for ChessPiece that updates piece owner
     *
     * @param player
     *             sets this piece to color "BLACK" or "WHITE"
     **************************************************************/

    protected ChessPiece(Player player) {
        this.owner = player;
    }

    /**************************************************************
     * Abstract method for piece type i.e rook, bishop, pawn , etc
     **************************************************************/
```

```
public abstract String type();

/*************************************************************************
 * Essentially a getter method that returns this pieces type
 *
 * @return owner return "Player" as in "Black" or "White" piece
 *************************************************************************/

public Player player() {

    return owner;

}

/*************************************************************************
 * Abstract method for white players Image Icons
 *************************************************************************/

public abstract ImageIcon whiteIcon();

/*************************************************************************
 * Abstract method for black players Image Icons
 *************************************************************************/

public abstract ImageIcon blackIcon();

/*************************************************************************
 * Verify that the starting and ending locations are different.
 *
 * Verify that this piece is located at [move.fromRow,
 * move.fromColumn] on the board.
 *
 * Verify that[move.toRow,move.toColumn] does not contain a piece
 * belonging to the same player.
 *
 * @return boolean true or false based on cases stated above
 *************************************************************************/

public boolean isValidMove(Move move, IChessPiece[][] board) {
```

```
        IChessPiece x = board[move.fromRow][move.fromColumn];
        IChessPiece y = board[move.toRow][move.toColumn];

        // not the same location
        if (((move.fromRow == move.toRow)
                && (move.fromColumn == move.toColumn)))
            return false;

        // is this piece located at fromRow fromCol on board
        if (!(this.equals(x))) {

            return false;
        }

        // are players different?
        if (y != null && this.owner.equals(y.player()))
            return false;

        else
            return true;

    }
}
```

```java
package chess;

import javax.swing.ImageIcon;

/**************************************************************************
 * Pawn Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * A Pawn piece that extends ChessPiece. It holds the rules to how the
 * pawn will be able to interact with the board it its laws.
 *
 * @author Douglas Money, Nick Reitz, Michael Baldwin
 * @version Winter 2014
 *************************************************************************/
public class Pawn extends ChessPiece {

    /**********************************************************************
     * Default constructor for the Pawn class that creates a pawn piece.
     *
     * @param player
     *             Who's piece it is
     *********************************************************************/
    protected Pawn(Player player) {
        super(player);
    }

    /**********************************************************************
     * Method for the Pawn class that returns that it is a pawn in the
     * form of a string.
     *
     * @return String type of piece
     *********************************************************************/
    public String type() {
        return "Pawn";
    }

    /**********************************************************************
```

```
     * Method for the Pawn class that assigns a white pawn image to a
     * imageIcon and returns it.
     *
     * @return Icon image icon for chess piece
     ************************************************************/
    public ImageIcon whiteIcon() {
        ImageIcon Icon = new ImageIcon("src/icons/white_pawn.png");
        return Icon;
    }

    /************************************************************
     * Method for the Pawn class that assigns a black pawn image to an
     * imageIcon and returns it.
     *
     * @return Icon image icon for chess piece
     ************************************************************/
    public ImageIcon blackIcon() {
        ImageIcon Icon = new ImageIcon("src/icons/black_pawn.png");
        return Icon;
    }

    /************************************************************
     * Method for the Pawn class that Checks to make sure that the move
     * a player has made is a valid move.
     *
     * @param move
     *            accepts a game Move
     *
     * @param board
     *            accepts a board of chess pieces
     *
     * @return boolean true if the move is legal
     ************************************************************/
    public boolean isValidMove(Move move, IChessPiece[][] board) {

        // check WHITE pieces
        if ((super.player() == Player.WHITE)) {

            int i = 0;
```

```
// loops through the board
while (i < 8) {

    IChessPiece temp = board[5][i];

    // checks to make sure there is not a piece in front of
    // it
    if (super.isValidMove(move, board)
        && ((move.fromRow == 6) && (move.fromColumn == i))
        && (((move.toRow == 4) && (move.toColumn == i))
        || ((move.toRow == 5) && (move.toColumn == i)))
        && temp != null)

        // returns false if there is
        return false;

    // make sure there is an black piece in a corner spot to
    // take it
    if (super.isValidMove(move, board)
            && ((move.fromRow <= 6)
                    && (move.fromColumn == i))
            && (move.toRow == move.fromRow - 1)
            && ((move.toColumn == move.fromColumn + 1)
            || (move.toColumn == move.fromColumn - 1))
            && board[move.toRow][move.toColumn] != null
            && (board[move.toRow][move.toColumn].player()
                    .equals(Player.BLACK))) {

        // returns true if there is
        return true;
    }

    // make sure first move - two spaces or one is what the
    // user has
    // picked
    else if (super.isValidMove(move, board)
        && ((move.fromRow == 6) && (move.fromColumn == i))
        && (((move.toRow == 4) && (move.toColumn == i))
        || ((move.toRow == 5) && (move.toColumn == i)))
        && board[move.toRow][move.toColumn] == null)
```

```
                // returns true if it is
                return true;

            // move one space with no capture and not first move
            else if (super.isValidMove(move, board)
                && ((move.fromRow <= 5) && (move.fromColumn == i))
                && (move.toRow == move.fromRow - 1)
                && (move.fromColumn == move.toColumn)
                && board[move.toRow][move.toColumn] == null)

                // returns true if it is
                return true;

            i++;
        }

    }

    // check BLACK pieces
    else if ((super.player() == Player.BLACK)) {

        int i = 0;

        // loops through the board
        while (i < 8) {

            IChessPiece temp = board[2][i];

            // checks to make sure there is not a piece in front of
            // it
            if (super.isValidMove(move, board)
                && ((move.fromRow == 1) && (move.fromColumn == i))
                && (((move.toRow == 3) && (move.toColumn == i))
                || ((move.toRow == 2) && (move.toColumn == i)))
                && temp != null)

                // returns false if there is
                return false;
```

```
            // make sure there is an white piece in a corner spot to
            // take it
            if (super.isValidMove(move, board)
                && ((move.fromRow >= 1) && (move.fromColumn == i))
                    && (move.toRow == move.fromRow + 1)
                    && ((move.toColumn == move.fromColumn + 1)
                    || (move.toColumn == move.fromColumn - 1))
                    && board[move.toRow][move.toColumn] != null
                    && (board[move.toRow][move.toColumn].player()
                            .equals(Player.WHITE))) {

                // returns true if there is
                return true;
            }

            // make sure first move - two spaces or one is what the
            // user has
            // picked
            else if (super.isValidMove(move, board)
                && ((move.fromRow == 1) && (move.fromColumn == i))
                && (((move.toRow == 3) && (move.toColumn == i))
                || ((move.toRow == 2) && (move.toColumn == i)))
                && board[move.toRow][move.toColumn] == null)

                // returns true if there is
                return true;

            // move one space with no capture and not first move
            else if (super.isValidMove(move, board)
                && ((move.fromRow >= 2) && (move.fromColumn == i))
                && (move.toRow == move.fromRow + 1)
                && (move.fromColumn == move.toColumn)
                && board[move.toRow][move.toColumn] == null)

                // returns true if there is
                return true;

            i++;

        }
```

```
        }

        // returns false if nothing is true
        return false;
    }

    /**********************************************************************
     * Method for the Pawn class that returns what color and type of
     * piece it is.
     *
     * @return String toString override for current piece type
     **********************************************************************/
    public String toString() {

        // if it is a black piece
        if (super.player() == Player.BLACK)

            // returns string of black and pawn
            return "Black Pawn";

        // if it is a white piece
        if (super.player() == Player.WHITE)

            // returns string of white and pawn
            return "White Pawn";

        // if it is neither
        else

            // returns a blank string
            return "";

    }

}
```

```java
package chess;

import javax.swing.ImageIcon;

/**************************************************************************
 * Rook Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * A Rook piece that extends ChessPiece. It holds the rules to how the
 * rook will be able to interact with the board it its laws.
 *
 * @author Douglas Money, Nick Reitz, Michael Baldwin
 * @version Winter 2014
 *************************************************************************/
public class Rook extends ChessPiece {

    /**********************************************************************
     * Default constructor for the Rook class that creates a rook piece.
     *
     * @param player
     *             Who's piece it is
     *********************************************************************/

    protected Rook(Player player) {
        super(player);
    }

    /**********************************************************************
     * Method for the Rook class that returns that it is a rook in the
     * form of a string.
     *
     * @return String type of piece
     *********************************************************************/

    public String type() {
        return "Rook";
    }
```

```java
/**********************************************************************
 * Method for the Rook class that assigns a white rook image to an
 * imageIcon and returns it. Violates MVC but great example of poly
 *
 * @return Icon image icon for chess piece
 **********************************************************************/

public ImageIcon whiteIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/white_rook.png");
    return Icon;
}

/**********************************************************************
 * Method for the Rook class that assigns a black rook image to an
 * imageIcon and returns it. Violates MVC but great example of poly
 *
 * @return Icon image icon for chess piece
 **********************************************************************/

public ImageIcon blackIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/black_rook.png");
    return Icon;
}

/**********************************************************************
 * Method for the Rook class that Checks to make sure that the move
 * a player has made is a valid move.
 *
 * @param move
 *            accepts a game Move
 *
 * @param board
 *            accepts a board of chess pieces
 *
 * @return boolean true if the move is legal
 *
 **********************************************************************/

public boolean isValidMove(Move move, IChessPiece[][] board) {
```

```
int countPiece = -1;
// int steps = Math.abs(move.toRow - move.fromRow);

// up and down
if (move.fromColumn == move.toColumn
        && super.isValidMove(move, board)) {

    // south to north movement
    for (int i = move.fromRow; i >= move.toRow; i--) {

        IChessPiece x = board[i][move.toColumn];

        if (x != null) {
            countPiece++;
        }

    }

    // north to south movement
    for (int i = move.fromRow; i <= move.toRow; i++) {

        IChessPiece x = board[i][move.toColumn];

        if (x != null)
            countPiece++;

    }
}

// side to side
if (move.fromRow == move.toRow
        && super.isValidMove(move, board)) {

    // west to east movement
    for (int i = move.fromColumn; i <= move.toColumn; i++) {

        IChessPiece x = board[move.toRow][i];

        if (x != null)
```

```
                countPiece++;
            }

            // east to west movement
            for (int i = move.fromColumn; i >= move.toColumn; i--) {

                IChessPiece x = board[move.toRow][i];

                if (x != null)
                    countPiece++;

            }

        }

        if (countPiece == 1)
            // try take piece
            if ((move.fromRow == move.toRow
            || move.fromColumn == move.toColumn)
                    && board[move.toRow][move.toColumn] != null
                    && board[move.toRow][move.toColumn].player()
                            .equals(player().next()))
                return true;

        // if path is clear
        if (countPiece == 0)
            return true;

        else
            return false;
    }

    /**********************************************************************
     * Method for the Rook class that returns what color and type of
     * piece it is.
     *
     * @return String toString override for current piece type
     **********************************************************************/

    public String toString() {
```

```
        // if it is a black piece
        if (super.player() == Player.BLACK)

            // returns string of black and rook
            return "Black Rook";

        // if it is a white piece
        if (super.player() == Player.WHITE)

            // returns string of white and rook
            return "White Rook";

        // if it is neither
        else

            // returns blink string
            return "";

    }

}
```

```java
package chess;

import javax.swing.ImageIcon;

/**************************************************************************
 * King Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * A King piece that extends ChessPiece. Essentially the rules to how
 * the king will be able to interact with the board it its laws.
 *
 * @author Douglas Money, Nick Reitz, Michael Baldwin
 * @version Winter 2014
 *************************************************************************/

public class King extends ChessPiece {

    /**********************************************************************
     * Default constructor for the King class that creates a king piece.
     *
     * @param player
     *             Who's piece it is
     *********************************************************************/

    protected King(Player player) {
        super(player);
    }

    /**********************************************************************
     * Method for the King class that returns that it is a king in the
     * form of a string.
     *
     * @return String type of piece
     *********************************************************************/

    public String type() {
        return "King";
```

```
}

/*****************************************************************
 * Method for the King class that assigns a white king image to a
 * imageIcon and returns it.
 *
 * @return Icon image icon for chess piece
 *****************************************************************/

public ImageIcon whiteIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/white_king.png");
    return Icon;
}

/*****************************************************************
 * Method for the King class that assigns a black king image to a
 * imageIcon and returns it.
 *
 * @return Icon image icon for chess piece
 *****************************************************************/

public ImageIcon blackIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/black_king.png");
    return Icon;
}

/*****************************************************************
 * Method for the King class that Checks to make sure that the move
 * a player has made is a valid move.
 *
 * @param move
 *            accepts a game Move
 *
 * @param board
 *            accepts a board of chess pieces
 *
 * @return boolean true if the move is legal
 *****************************************************************/

public boolean isValidMove(Move move, IChessPiece[][] board) {
```

```
// Determines if the move is valid acording to ChessPiece class
if (super.isValidMove(move, board)) {

    // if spot is one to the spot is one up the board.
    if ((move.toRow == move.fromRow - 1)
            && (move.fromColumn == move.toColumn)) {
        return true;
    }

    // if the spot is one down the board
    if ((move.toRow == move.fromRow + 1)
            && (move.fromColumn == move.toColumn)) {
        return true;
    }

    // if the spot is one to the right
    if ((move.toColumn == move.fromColumn + 1)
            && (move.fromRow == move.toRow)) {
        return true;
    }

    // if the spot is one to the left
    if ((move.toColumn == move.fromColumn - 1)
            && (move.fromRow == move.toRow)) {
        return true;
    }

    // if the spot is in the upper right corner
    if ((move.toColumn == move.fromColumn + 1)
            && (move.toRow == move.fromRow - 1)) {
        return true;
    }

    // if the spot is in the upper left corner
    if ((move.toColumn == move.fromColumn - 1)
            && (move.toRow == move.fromRow - 1)) {
        return true;
    }
```

```
            // if the spot is in the lower left corner
            if ((move.toColumn == move.fromColumn - 1)
                    && (move.toRow == move.fromRow + 1)) {
                return true;
            }

            // if the spot is in the lower right corner
            if ((move.toColumn == move.fromColumn + 1)
                    && (move.toRow == move.fromRow + 1)) {
                return true;
            }
        }

        // returns false if the spot is none of thies options
        return false;
    }


}
```

```java
package chess;

import javax.swing.ImageIcon;

/**************************************************************************
 * Queen Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * A Queen piece that extends ChessPiece. It holds the rules to how the
 * queen will be able to interact with the board it its laws.
 *
 * @author Douglas Money, Nick Reitz, Michael Baldwin
 * @version Winter 2014
 *************************************************************************/
public class Queen extends ChessPiece {

    /** Top to Bottom in a positive direction(TBPos) */
    private int countPieceTBPos;

    /** Top to Bottom in a negative direction(TBNeg) */
    private int countPieceTBNeg;

    /** Bottom to Top in a positive direction(BTPos) */
    private int countPieceBTPos;

    /** Bottom to Top in a negative direction(BTNeg) */
    private int countPieceBTNeg;

    /**********************************************************************
     * Default constructor for the Queen class that creates a queen
     * piece.
     *
     * @param player
     *             Who's piece it is
     *********************************************************************/
    protected Queen(Player player) {
        super(player);
```

```
}

/**********************************************************************
 * Method for the Queen class that returns that it is a queen in the
 * form of a string.
 *
 * @return String type of piece
 **********************************************************************/
public String type() {
    return "Queen";
}

/**********************************************************************
 * Method for the Queen class that assigns a white queen image to a
 * imageIcon and returns it.
 *
 * @return Icon image icon for chess piece
 **********************************************************************/
public ImageIcon whiteIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/white_queen.png");
    return Icon;
}

/**********************************************************************
 * Method for the Queen class that assigns a black queen image to an
 * imageIcon and returns it.
 *
 * @return Icon image icon for chess piece
 **********************************************************************/
public ImageIcon blackIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/black_queen.png");
    return Icon;
}

/**********************************************************************
 * Method for the Queen class that Checks to make sure that the move
 * a player has made is a valid move.
 *
 * @param move
 *            accepts a game Move
```

```
     *
     * @param board
     *            accepts a board of chess pieces
     *
     * @return boolean true if the move is legal
     ***********************************************************************/
    public boolean isValidMove(Move move, IChessPiece[][] board) {

        // if the move is vertical or horizontal
        if ((move.fromColumn == move.toColumn)
                || (move.fromRow == move.toRow)) {

            // sends it to the rook movement
            return checkRookMovement(move, board);

            // if not vertical or horizontal
        } else

            // sends it to the bishop movement
            return checkBishopMovement(move, board);

    }

    /***********************************************************************
     * Method for the Queen class that Checks to make sure that the
     * queen is able to move like a rook.
     *
     * @param move
     *            accepts a game Move
     *
     * @param board
     *            accepts a board of chess pieces
     *
     * @return boolean true if the move is legal
     ***********************************************************************/
    public boolean checkRookMovement(Move move, IChessPiece[][] board) {

        int countPiece = -1;

        // make sure it is a valid move according to chessPiece
```

```
if (super.isValidMove(move, board)) {

    // make sure move is in the same column
    if (move.fromColumn == move.toColumn) {

        //
        for (int i = move.fromRow; i >= move.toRow; i--) {
            IChessPiece x = board[i][move.toColumn];

            if (x != null)
                countPiece++;

        }

        // north to south movement
        for (int i = move.fromRow; i <= move.toRow; i++) {
            IChessPiece x = board[i][move.toColumn];

            if (x != null)
                countPiece++;

        }
    }

    // side to side
    if (move.fromRow == move.toRow) {
        // west to east movement
        for (int i = move.fromColumn; i <= move.toColumn; i++) {
            IChessPiece x = board[move.toRow][i];

            if (x != null)
                countPiece++;
        }
        // east to west movement
        for (int i = move.fromColumn; i >= move.toColumn; i--) {
            IChessPiece x = board[move.toRow][i];

            if (x != null)
                countPiece++;
        }
```

```
        }

    }

    if (countPiece == 1)
        // try take piece
        if ((move.fromRow == move.toRow
                    || move.fromColumn == move.toColumn)
                && board[move.toRow][move.toColumn] != null
                && board[move.toRow][move.toColumn].player()
                        .equals(player().next()))
            return true;

    // if path is clear
    if (countPiece == 0)
        return true;

    else
        return false;

}

/*************************************************************************
 * Method for the Queen class that Checks to make sure that the
 * queen is able to move like a bishop.
 *
 * @param move
 *            accepts a game Move
 *
 * @param board
 *            accepts a board of chess pieces
 *
 * @return boolean true if the move is legal
 ************************************************************************/
public boolean checkBishopMovement(Move move,IChessPiece[][] board){

    checkForClearPath(move, board);
```

```
for (int i = 0; i < 8; i++) {

    // Check for valid move

    if ((move.fromColumn + i == move.toColumn)
            && (move.fromRow + i == move.toRow)
            && countPieceTBPos == 0
            && super.isValidMove(move, board)) {

        return true;
    }

    else if ((move.fromColumn - i == move.toColumn)
            && (move.fromRow + i == move.toRow)
            && countPieceTBNeg == 0
            && super.isValidMove(move, board)) {

        return true;
    }

    else if ((move.fromColumn + i == move.toColumn)
            && (move.fromRow - i == move.toRow)
            && countPieceBTPos == 0
            && super.isValidMove(move, board)) {
        return true;
    }

    else if ((move.fromColumn - i == move.toColumn)
            && (move.fromRow - i == move.toRow)
            && countPieceBTNeg == 0
            && super.isValidMove(move, board)) {
        return true;
    }

    // Check Capture

    // Top to Bottom
    // Left to Right

    if ((move.fromColumn + i == move.toColumn)
```

```
            && (move.fromRow + i == move.toRow)
            && countPieceTBPos == 1
            && super.isValidMove(move, board)) {

        if (board[move.toRow][move.toColumn] != null
                && board[move.toRow][move.toColumn].player()
                        .equals(player().next())) {
            return true;
        }
    }

    // Check Capture

    // Top to Bottom
    // Right to Left

    if (((move.fromColumn - i == move.toColumn)
            && (move.fromRow + i == move.toRow)
            && countPieceTBNeg == 1 && super.isValidMove(move,
            board))) {

        if (board[move.toRow][move.toColumn] != null
                && board[move.toRow][move.toColumn].player()
                        .equals(player().next())) {
            return true;
        }
    }

    // Check Capture

    // Bottom to Top
    // Left to Right

    if ((move.fromColumn + i == move.toColumn)
            && (move.fromRow - i == move.toRow)
            && countPieceBTPos == 1
            && super.isValidMove(move, board)) {

        if (board[move.toRow][move.toColumn] != null
                && board[move.toRow][move.toColumn].player()
```

```
                                .equals(player().next())) {
                    return true;
                }
            }

            // Check Capture

            // Bottom to Top
            // Right to Left

            if ((move.fromColumn - i == move.toColumn)
                    && (move.fromRow - i == move.toRow)
                    && countPieceBTNeg == 1
                    && super.isValidMove(move, board)) {

                if (board[move.toRow][move.toColumn] != null
                        && board[move.toRow][move.toColumn].player()
                                .equals(player().next())) {
                    return true;
                }
            }

        }

        return false;

    }

    /**********************************************************************
     * Method for the Queen class that Checks to make sure that the move
     * a player has made is a clear path to the move location. Primarily
     * the bishop like movement
     *
     * @param move
     *            accepts a game Move
     *
     * @param board
     *            accepts a board of chess pieces
     *
     * @return boolean true if the move is legal
```

```
 ***********************************************************************/

public void checkForClearPath(Move move, IChessPiece[][] board) {

    // Top to Bottom in a positive direction(TBPos)
    countPieceTBPos = 0;
    countPieceTBNeg = 0;
    countPieceBTPos = 0;
    countPieceBTNeg = 0;


    // Top to Bottom
    // Left to Right
    if (super.isValidMove(move, board)) {

        int tempRow = move.fromRow;
        int tempCol = move.fromColumn;

        for (int i = move.fromRow; i < move.toRow; i++) {
            tempRow += 1;
            tempCol += 1;
            if ((tempRow < 8 && tempCol < 8)
                    && (tempRow >= 0 && tempCol >= 0)) {

                IChessPiece x = board[tempRow][tempCol];

                if (x != null)
                    countPieceTBPos++;
            }
        }
    }

    // Top to Bottom
    // Right to Left
    if (super.isValidMove(move, board)) {
        int tempRow = move.fromRow;
        int tempCol = move.fromColumn;

        for (int i = move.fromRow; i < move.toRow; i++) {
            tempRow += 1;
```

```
            tempCol -= 1;
            if ((tempRow < 8 && tempCol < 8)
                    && (tempRow >= 0 && tempCol >= 0)) {

                IChessPiece x = board[tempRow][tempCol];

                if (x != null)
                    countPieceTBNeg++;

            }
        }
    }

    // Bottom to Top
    // Left to Right

    if (super.isValidMove(move, board)) {
        int tempRow = move.fromRow;
        int tempCol = move.fromColumn;

        for (int i = move.fromRow; i > move.toRow; i--) {
            tempRow -= 1;
            tempCol += 1;

            if ((tempRow < 8 && tempCol < 8)
                    && (tempRow >= 0 && tempCol >= 0)) {

                IChessPiece x = board[tempRow][tempCol];

                if (x != null)
                    countPieceBTPos++;

            }
        }
    }

    // Bottom to Top
    // Right to Left
    if (super.isValidMove(move, board)) {
        int tempRow = move.fromRow;
```

```
            int tempCol = move.fromColumn;

            for (int i = move.fromRow; i > move.toRow; i--) {
                tempRow -= 1;
                tempCol -= 1;

                if ((tempRow < 8 && tempCol < 8)
                        && (tempRow >= 0 && tempCol >= 0)) {

                    IChessPiece x = board[tempRow][tempCol];

                    if (x != null)
                        countPieceBTNeg++;

                }
            }
        }

}

/**************************************************************************
 * Method for the Queen class that returns what color and type of
 * piece it is.
 *
 * @return String toString overide for current piece type
 *************************************************************************/

public String toString() {

    // if it is a black piece
    if (super.player() == Player.BLACK)

        // returns string of black and queen
        return "Black Queen";

    // if it is a white piece
    if (super.player() == Player.WHITE)

        // returns string of white and queen
        return "White Queen";
```

```
        // if it is neither
        else

            // returns blink string
            return "";

    }

}
```

```java
package chess;

import javax.swing.ImageIcon;

/**************************************************************************
 * Knight Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * A Knight piece that extends ChessPiece. It holds the rules to how the
 * knight will be able to interact with the board it its laws.
 *
 * @author Douglas Money, Nick Reitz, Michael Baldwin
 * @version Winter 2014
 *************************************************************************/
public class Knight extends ChessPiece {

    /**********************************************************************
     * Default constructor for the Knight class that creates a knight
     * piece.
     *
     * @param player
     *             Who's piece it is
     *********************************************************************/

    protected Knight(Player player) {
        super(player);
    }

    /**********************************************************************
     * Method for the Knight class that returns that it is a knight in
     * the form of a string.
     *
     * @return String type of piece
     *********************************************************************/

    public String type() {
        return "Knight";
```

```
}

/**********************************************************************
 * Method for the Knight class that assigns a white knight image to
 * a imageIcon and returns it.
 *
 * @return Icon image icon for chess piece
 **********************************************************************/

public ImageIcon whiteIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/white_knight.png");
    return Icon;
}

/**********************************************************************
 * Method for the Knight class that assigns a black knight image to
 * an imageIcon and returns it.
 *
 * @return Icon image icon for chess piece
 **********************************************************************/

public ImageIcon blackIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/black_knight.png");
    return Icon;
}

/**********************************************************************
 * Method for the Knight class that Checks to make sure that the
 * move a player has made is a valid move.
 *
 *
 * @param move
 *            accepts a game Move
 *
 * @param board
 *            accepts a board of chess pieces
 *
 * @return boolean true if the move is legal
 **********************************************************************/
```

```
public boolean isValidMove(Move move, IChessPiece[][] board) {

    // checks to make sure the move is a valid move
    if (super.isValidMove(move, board)) {

        // if the spot is two down and one to the left
        if ((move.fromColumn + 2 == move.toColumn)
                && (move.fromRow - 1 == move.toRow)) {

            // returns true if it is
            return true;
        }

        // if the spot is two up and one to the left
        if ((move.fromColumn - 2 == move.toColumn)
                && (move.fromRow - 1 == move.toRow)) {
            return true;
        }

        // if the spot it two down and one to the right
        if ((move.fromColumn + 2 == move.toColumn)
                && (move.fromRow + 1 == move.toRow)) {
            return true;
        }

        // if the spot is two up and one to the right
        if ((move.fromColumn - 2 == move.toColumn)
                && (move.fromRow + 1 == move.toRow)) {
            return true;
        }

        // if the spot is one down and two to the left
        if ((move.fromColumn + 1 == move.toColumn)
                && (move.fromRow - 2 == move.toRow)) {
            return true;
        }

        // if the spot is one up and two to the left
        if ((move.fromColumn - 1 == move.toColumn)
                && (move.fromRow - 2 == move.toRow)) {
```

```
            return true;
        }

        // if the spot is one down and two to the right
        if ((move.fromColumn + 1 == move.toColumn)
                && (move.fromRow + 2 == move.toRow)) {
            return true;
        }

        // if the spot is one up and two to the right
        if ((move.fromColumn - 1 == move.toColumn)
                && (move.fromRow + 2 == move.toRow)) {
            return true;
        }
    }

    // returns false if none of them are the spot the user has
    // chosen
    return false;
}

/**********************************************************************
 * Method for the Knight class that returns what color and type of
 * piece it is.
 *
 * @return String toString override for current piece type
 **********************************************************************/
public String toString() {

    // if it is a black piece
    if (super.player() == Player.BLACK)

        // returns string of black and knight
        return "Black Knight";

    // if it is a white piece
    if (super.player() == Player.WHITE)

        // returns string of white and knight
        return "White Knight";
```

```
        // if it is neither
        else

            // returns a blink piece
            return "";

    }

}
```

```java
package chess;

import javax.swing.ImageIcon;

/**************************************************************************
 * Bishop Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * A Bishop piece that extends ChessPiece. It holds the rules to how the
 * bishop will be able to interact with the board it its laws.
 *
 * @author Douglas Money, Nick Reitz, Michael Baldwin
 * @version Winter 2014
 *************************************************************************/
public class Bishop extends ChessPiece {

    /** Top to Bottom in a positive direction(TBPos) */
    private int countPieceTBPos;

    /** Top to Bottom in a negative direction(TBNeg) */
    private int countPieceTBNeg;

    /** Bottom to Top in a positive direction(BTPos) */
    private int countPieceBTPos;

    /** Bottom to Top in a negative direction(BTNeg) */
    private int countPieceBTNeg;

    /**********************************************************************
     * Default constructor for the Bishop class that creates a bishop
     * piece.
     *
     * @param player
     *             Who's piece it is
     *********************************************************************/

    protected Bishop(Player player) {
```

```
    super(player);
}

/*****************************************************************
 * Method for the Bishop class that returns that it is a bishop in
 * the form of a string.
 *
 * @return String type of piece
 *****************************************************************/

public String type() {
    return "Bishop";
}

/*****************************************************************
 * Method for the Bishop class that assigns a white bishop image to
 * a imageIcon and returns it.
 *
 * @return Icon image icon for chess piece
 *****************************************************************/

public ImageIcon whiteIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/white_bishop.png");
    return Icon;
}

/*****************************************************************
 * Method for the Bishop class that assigns a black bishop image to
 * a imageIcon and returns it.
 *
 * @return Icon image icon for chess piece
 *****************************************************************/

public ImageIcon blackIcon() {
    ImageIcon Icon = new ImageIcon("src/icons/black_bishop.png");
    return Icon;
}

/*****************************************************************
 * Method for the Bishop class that Checks to make sure that the
```

```
     * move a player has made is a valid move.
     *
     * @param move
     *            accepts a game Move
     *
     * @param board
     *            accepts a board of chess pieces
     *
     * @return boolean true if the move is legal
     ***********************************************************************/

    public boolean isValidMove(Move move, IChessPiece[][] board) {

        checkForClearPath(move, board);

        for (int i = 0; i < 8; i++) {

            // Check for valid move

            if ((move.fromColumn + i == move.toColumn)
                    && (move.fromRow + i == move.toRow)
                    && countPieceTBPos == 0
                    && super.isValidMove(move, board)) {

                return true;
            }

            else if ((move.fromColumn - i == move.toColumn)
                    && (move.fromRow + i == move.toRow)
                    && countPieceTBNeg == 0
                    && super.isValidMove(move, board)) {

                return true;
            }

            else if ((move.fromColumn + i == move.toColumn)
                    && (move.fromRow - i == move.toRow)
                    && countPieceBTPos == 0
                    && super.isValidMove(move, board)) {
                return true;
```

```
}

else if ((move.fromColumn - i == move.toColumn)
        && (move.fromRow - i == move.toRow)
        && countPieceBTNeg == 0
        && super.isValidMove(move, board)) {
    return true;
}

// Check Capture

// Top to Bottom
// Left to Right

if ((move.fromColumn + i == move.toColumn)
        && (move.fromRow + i == move.toRow)
        && countPieceTBPos == 1
        && super.isValidMove(move, board)) {

    if (board[move.toRow][move.toColumn] != null
            && board[move.toRow][move.toColumn].player()
                    .equals(player().next())) {
        return true;
    }
}

// Check Capture

// Top to Bottom
// Right to Left

if (((move.fromColumn - i == move.toColumn)
        && (move.fromRow + i == move.toRow)
        && countPieceTBNeg == 1 && super.isValidMove(move,
        board))) {

    if (board[move.toRow][move.toColumn] != null
            && board[move.toRow][move.toColumn].player()
                    .equals(player().next())) {
        return true;
```

```
            }
        }

        // Check Capture

        // Bottom to Top
        // Left to Right

        if ((move.fromColumn + i == move.toColumn)
                && (move.fromRow - i == move.toRow)
                && countPieceBTPos == 1
                && super.isValidMove(move, board)) {

            if (board[move.toRow][move.toColumn] != null
                    && board[move.toRow][move.toColumn].player()
                            .equals(player().next())) {
                return true;
            }
        }

        // Check Capture

        // Bottom to Top
        // Right to Left

        if ((move.fromColumn - i == move.toColumn)
                && (move.fromRow - i == move.toRow)
                && countPieceBTNeg == 1
                && super.isValidMove(move, board)) {

            if (board[move.toRow][move.toColumn] != null
                    && board[move.toRow][move.toColumn].player()
                            .equals(player().next())) {
                return true;
            }
        }

    }

    return false;
```

```
}

/*****************************************************************
 * Method for the Bishop class that Checks to make sure that the
 * move a player has made is a clear path to the move location
 *
 * @param move
 *            accepts a game Move
 *
 * @param board
 *            accepts a board of chess pieces
 *
 * @return boolean true if the move is legal
 *****************************************************************/

public void checkForClearPath(Move move, IChessPiece[][] board) {

    // Top to Bottom in a positive direction(TBPos)
    countPieceTBPos = 0;
    countPieceTBNeg = 0;
    countPieceBTPos = 0;
    countPieceBTNeg = 0;


    // Top to Bottom
    // Left to Right
    if (super.isValidMove(move, board)) {

        int tempRow = move.fromRow;
        int tempCol = move.fromColumn;

        for (int i = move.fromRow; i < move.toRow; i++) {
            tempRow += 1;
            tempCol += 1;
            if ((tempRow < 8 && tempCol < 8)
                    && (tempRow >= 0 && tempCol >= 0)) {

                IChessPiece x = board[tempRow][tempCol];
```

```
            if (x != null)
                countPieceTBPos++;
        }
    }
}

// Top to Bottom
// Right to Left
if (super.isValidMove(move, board)) {
    int tempRow = move.fromRow;
    int tempCol = move.fromColumn;

    for (int i = move.fromRow; i < move.toRow; i++) {
        tempRow += 1;
        tempCol -= 1;
        if ((tempRow < 8 && tempCol < 8)
                && (tempRow >= 0 && tempCol >= 0)) {

            IChessPiece x = board[tempRow][tempCol];

            if (x != null)
                countPieceTBNeg++;

        }
    }
}

// Bottom to Top
// Left to Right

if (super.isValidMove(move, board)) {
    int tempRow = move.fromRow;
    int tempCol = move.fromColumn;

    for (int i = move.fromRow; i > move.toRow; i--) {
        tempRow -= 1;
        tempCol += 1;

        if ((tempRow < 8 && tempCol < 8)
                && (tempRow >= 0 && tempCol >= 0)) {
```

```
                IChessPiece x = board[tempRow][tempCol];

                if (x != null)
                    countPieceBTPos++;

            }
        }
    }

    // Bottom to Top
    // Right to Left
    if (super.isValidMove(move, board)) {
        int tempRow = move.fromRow;
        int tempCol = move.fromColumn;

        for (int i = move.fromRow; i > move.toRow; i--) {
            tempRow -= 1;
            tempCol -= 1;

            if ((tempRow < 8 && tempCol < 8)
                    && (tempRow >= 0 && tempCol >= 0)) {

                IChessPiece x = board[tempRow][tempCol];

                if (x != null)
                    countPieceBTNeg++;

            }
        }
    }

}

/*************************************************************************
 * Method for the Bishop class that returns what color and type of
 * piece it is.
 *
 * @return String toString overide for current piece type
 *************************************************************************/
```

```
    public String toString() {

        if (super.player() == Player.BLACK)
            return "Black Bishop";
        if (super.player() == Player.WHITE)
            return "White Bishop";
        else
            return "";

    }

}
```

```java
package chess;

import javax.swing.ImageIcon;

/**********************************************************************
 * IChessPiece Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * Interface Chess Piece. According to the Model View Controller
 * software pattern, this class controls the Chess Pieces. One exception
 * being the chess piece classes contain image icons which are used in
 * the panel class to emphasize polymorphism.
 *
 * @author Michael Baldwin , Douglas Money, Nick Reitz
 * @version Winter 2014
 *
 *********************************************************************/

public interface IChessPiece {

    /**
     * Return the player that owns this piece.
     *
     * @return the player that owns this piece.
     */
    Player player();

    ImageIcon whiteIcon();

    ImageIcon blackIcon();

    /**
     * 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
```

```
     */
    String type();

    /**
     * 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 W13project3.Move} object describing the move
     *            to be made.
     * @param board
     *            the {@link W13project3.IChessBoard} in which this
     *            piece resides.
     * @return {@code true} if the proposed move is valid, {@code false}
     *         otherwise.
     * @throws IndexOutOfBoundsException
     *             if either {@code [move.fromRow, move.fromColumn]} or
     *             {@code [move.toRow,
     *                                  move.toColumn]} don't represent
     *             valid locations on the board.
     * @throws IllegalArgumentException
     *             if {@code this} object isn't the piece at location
     *             {@code [move.fromRow, move.fromColumn]}.
     */
    boolean isValidMove(Move move, IChessPiece[][] board);

}
```

```java
package chess;

/**
 * Objects implementing this interface represent the state of a chess
 * game. Notice that this interface is designed to maintain the game
 * state only, it does not provide any methods to control the flow of
 * the game.
 *
 * @author
 */

public interface IChessModel {

    /**
     * Returns whether the game is complete.
     *
     * @return {@code true} if complete, {@code false} otherwise.
     */
    boolean isComplete();

    /**
     * Returns whether the piece at location
     * {@code [move.fromRow, move.fromColumn]} is allowed to move to
     * location {@code [move.fromRow, move.fromColumn]}.
     *
     * @param move
     *            a {@link Move} object describing the move to be made.
     * @return {@code true} if the proposed move is valid, {@code false}
     *         otherwise.
     * @throws IndexOutOfBoundsException
     *             if either {@code [move.fromRow, move.fromColumn]} or
     *             {@code [move.toRow,
     *                                 move.toColumn]} don't represent
     *             valid locations on the board.
     */
    boolean isValidMove(Move move);

    /**
     * Moves the piece from location
     * {@code [move.fromRow, move.fromColumn]} to location
```

```
     * {@code [move.fromRow,
     * move.fromColumn]}.
     *
     * @param move
     *             a {@link Move} object describing the move to be made.
     * @throws IndexOutOfBoundsException
     *              if either {@code [move.fromRow, move.fromColumn]} or
     *              {@code [move.toRow,
     *                                  move.toColumn]} don't represent
     *              valid locations on the board.
     */
    void move(Move move);

    /**
     * Report whether the current player is in check.
     *
     * @return {@code true} if the current player is in check,
     *         {@code false} otherwise.
     */
    boolean inCheck(Player player);

    /**
     * Return the current player.
     *
     * @return the current player
     */
    Player currentPlayer();

    /**
     * Return the {@code ChessPiece} object at location
     * {@code [row, column]}.
     *
     * @param row
     *             the row (numbered {@code 0} through {@code numRows -1}
     * @param column
     *             the row (numbered {@code 0} through
     *             {@code numColumns -1}
     * @return the {@code ChessPiece} object at location
     *         {@code [row, column]}.
     * @throws IndexOutOfBoundsException
```

```
     *              if {@code [row, column]} is not a valid location on
     *              the board.
     */
    IChessPiece pieceAt(int row, int column);

}
```

```java
package chess;

/**********************************************************************
 * Move Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * Packages the four components of a move into a single object.
 * (Instance variables are public because this object is a simple
 * container.)
 *
 * @author Michael Baldwin , Douglas Money, Nick Reitz
 * @version Winter 2014
 *********************************************************************/

public class Move {

    /** move from row */
    public int fromRow;

    /** move from column */
    public int fromColumn;

    /** move to row */
    public int toRow;

    /** move from column */
    public int toColumn;

    /**********************************************************************
     * Default constructor for Move class.
     *********************************************************************/
    public Move() {
    }

    /**********************************************************************
     * Constructor for Move class. Sets variables for object container
     *
```

```
     * @param fromRow
     *            accepts move from row
     *
     * @param fromColumn
     *            accepts move from column
     *
     * @param toRow
     *            accepts move to row
     *
     * @param toColumn
     *            accepts move to column
     *
     ************************************************************/

    public Move(int fromRow, int fromColumn, int toRow, int toColumn) {
        this.fromRow = fromRow;
        this.fromColumn = fromColumn;
        this.toRow = toRow;
        this.toColumn = toColumn;
    }
}
```

```java
package chess;

/**********************************************************************
 * Player Class
 *
 * CIS 163-03
 *
 * Project 3
 *
 * Enumerated Type Color Representation for Chess Pieces
 *
 * @author Michael Baldwin , Douglas Money, Nick Reitz
 *
 * @version Winter 2014
 *********************************************************************/
public enum Player {
    BLACK, WHITE;

    /**********************************************************************
     * Return the {@code Player} whose turn is next.
     *
     * @return the {@code Player} whose turn is next
     *********************************************************************/
    public Player next() {
        return this == BLACK ? WHITE : BLACK;
    }
}
```