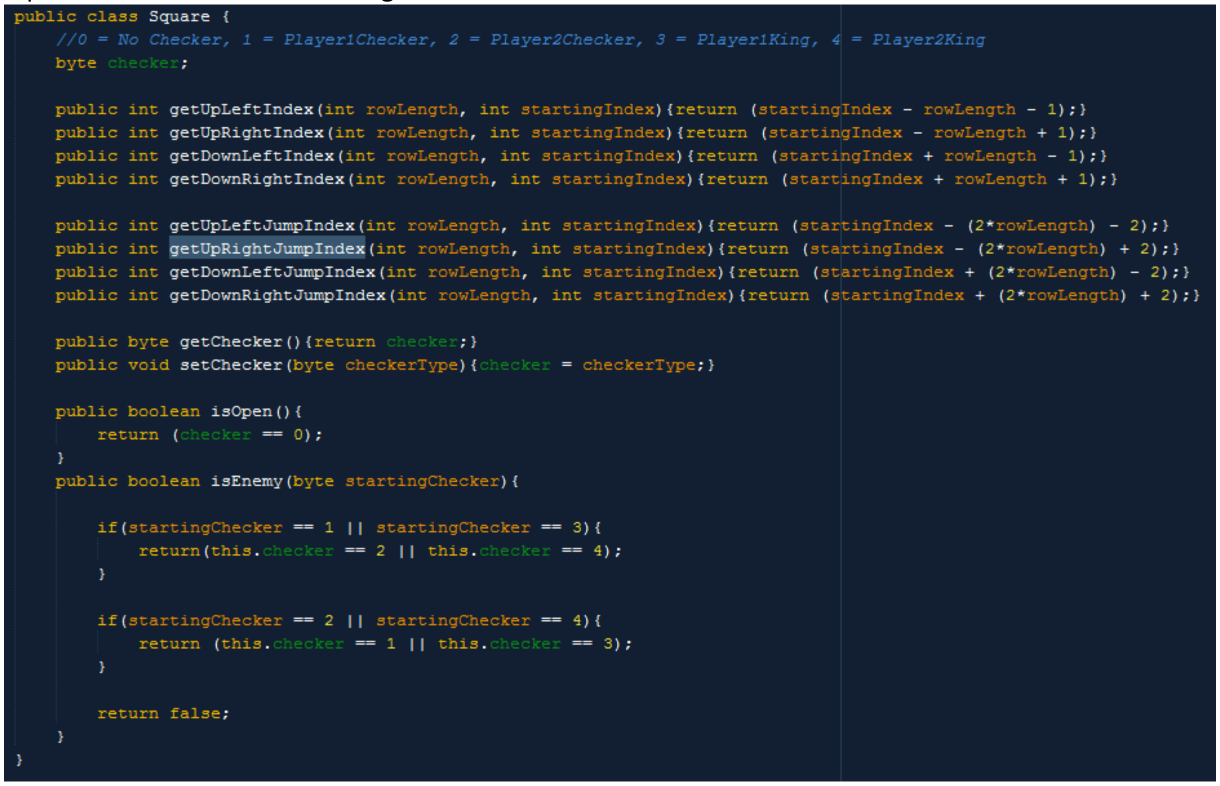
Michael Rallo - msr5zb - 12358133
Checkers Data Model

The board in which our checkers game is taking place will be represented as a single array of size: “n x n.” This array will consists of Square objects, ranging with indices of 0 to “n x n – 1.”

Square[] board = new Square[rows*columns];

The kind of checker within a square can be represented as a byte.

Square will contain the following members:



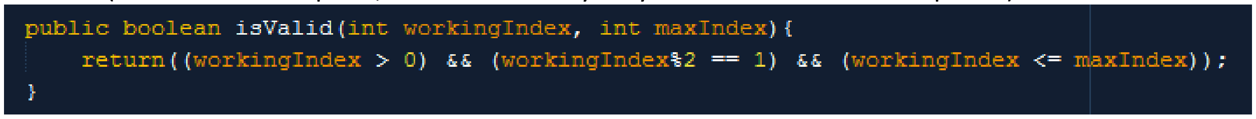
```
public class Square {
    //0 = No Checker, 1 = Player1Checker, 2 = Player2Checker, 3 = Player1King, 4 = Player2King
    byte checker;

    public int getUpLeftIndex(int rowLength, int startingIndex){return (startingIndex - rowLength - 1);}
    public int getUpRightIndex(int rowLength, int startingIndex){return (startingIndex - rowLength + 1);}
    public int getDownLeftIndex(int rowLength, int startingIndex){return (startingIndex + rowLength - 1);}
    public int getDownRightIndex(int rowLength, int startingIndex){return (startingIndex + rowLength + 1);}

    public int getUpLeftJumpIndex(int rowLength, int startingIndex){return (startingIndex - (2*rowLength) - 2);}
    public int getUpRightJumpIndex(int rowLength, int startingIndex){return (startingIndex - (2*rowLength) + 2);}
    public int getDownLeftJumpIndex(int rowLength, int startingIndex){return (startingIndex + (2*rowLength) - 2);}
    public int getDownRightJumpIndex(int rowLength, int startingIndex){return (startingIndex + (2*rowLength) + 2);}

    public byte getChecker(){return checker;}
    public void setChecker(byte checkerType){checker = checkerType;}

    public boolean isOpen(){
        return (checker == 0);
    }
    public boolean isEnemy(byte startingChecker){

        if(startingChecker == 1 || startingChecker == 3){
            return(this.checker == 2 || this.checker == 4);
        }

        if(startingChecker == 2 || startingChecker == 4){
            return (this.checker == 1 || this.checker == 3);
        }

        return false;
    }
}
```

Possible/Allowable moves are:

- Move a checker forward (diagonally) one square onto a darkColored square if no checker is in the desired square.
- Jump an enemy checker provided an empty space lays ahead.
- Have a checker make it to the end of a board and be kinged.

Legal Moves:

- Forwards: A Checker may move forward into an empty square.
- Backwards: If a checker is kinged, the checker may move backwards.
- Jump: A checker may jump over a checker of a different color if the enemy checker is 1 row forward (or backward if kinged) and the space beyond the enemy checker is opened. Note a checker *must* jump multiple times as long as the rules are not broken.
- NOTE, A CHECKER MUST MAKE ALL POSSIBLE JUMPS IT CAN!
- A valid square for a checker to move into must be between of index 0 to “n x n - 1” and be odd indexed (A dark colored square, for checkers may only exist on dark colored squares).



```
public boolean isValid(int workingIndex, int maxIndex){
    return((workingIndex > 0) && (workingIndex%2 == 1) && (workingIndex <= maxIndex));
}
```

Illegal Moves

- Blocked: A Checker cannot move into a space that already has a checker.
- OffBoard: A Checker cannot move off of the board or into red squares.
- FriendlyFire: A Checker cannot jump a checker of the same color.
- Direction: A checker cannot move backwards if it is not kinged.
- Square Placement: A checker cannot be placed on even indexed Squares (lightColored squares).
- Turn: A player may only move a checker during their turn.

Winner, Loser, Draw

- Winner: is determined by removing all enemy checkers from the board.
- Loser: is determined by a player having no checkers left on the board that belongs to them.
- Draw: will be determined if either player cannot claim a single opponent’s checker or king a checker within 50 turn cycles OR if a player offers a draw and the other player accepts.

AI Algorithm

The AI will implement a Min-Max algorithm to determine its move.
The AI points for this algorithm will be determined by, in order of points granted:

- Removing all of the opponent’s checkers (Winning State)
- Kinging a checker
- Removing an opponent’s king checker
- Checker Count
- Opponent checker count
- Removing all own checkers (Losing State)

Look ahead of this algorithm will start at 4, AI may start looking further ahead depending on performance.

UI representation of the game and game play.

The checkerboard UI will be built upon the board array. The board will generate squares with the same “n x n” dimensions and the square array (single array). This way, we will be able to determine where to place a checker simply by looking at the index and checking its checker data value.

StateSpace

StateSpace will consist of playerTurn and the Board Array.

Initial StateSpace

The initial StateSpace board will first create the board array. After the array is created and initialized with its square objects, then the checkers will fill as desired. Dark colored squares (odd indexed squared) 3 rows off the top will have player2 checkers placed in them. Dark colored squares (odd indexed squared) 3 rows from the bottom will have player1 checkers placed in them. Note, player1 will always prioritize the bottom of the board, for a human should always get the bottom POV when facing an AI.

Future, Legal StateSpaces

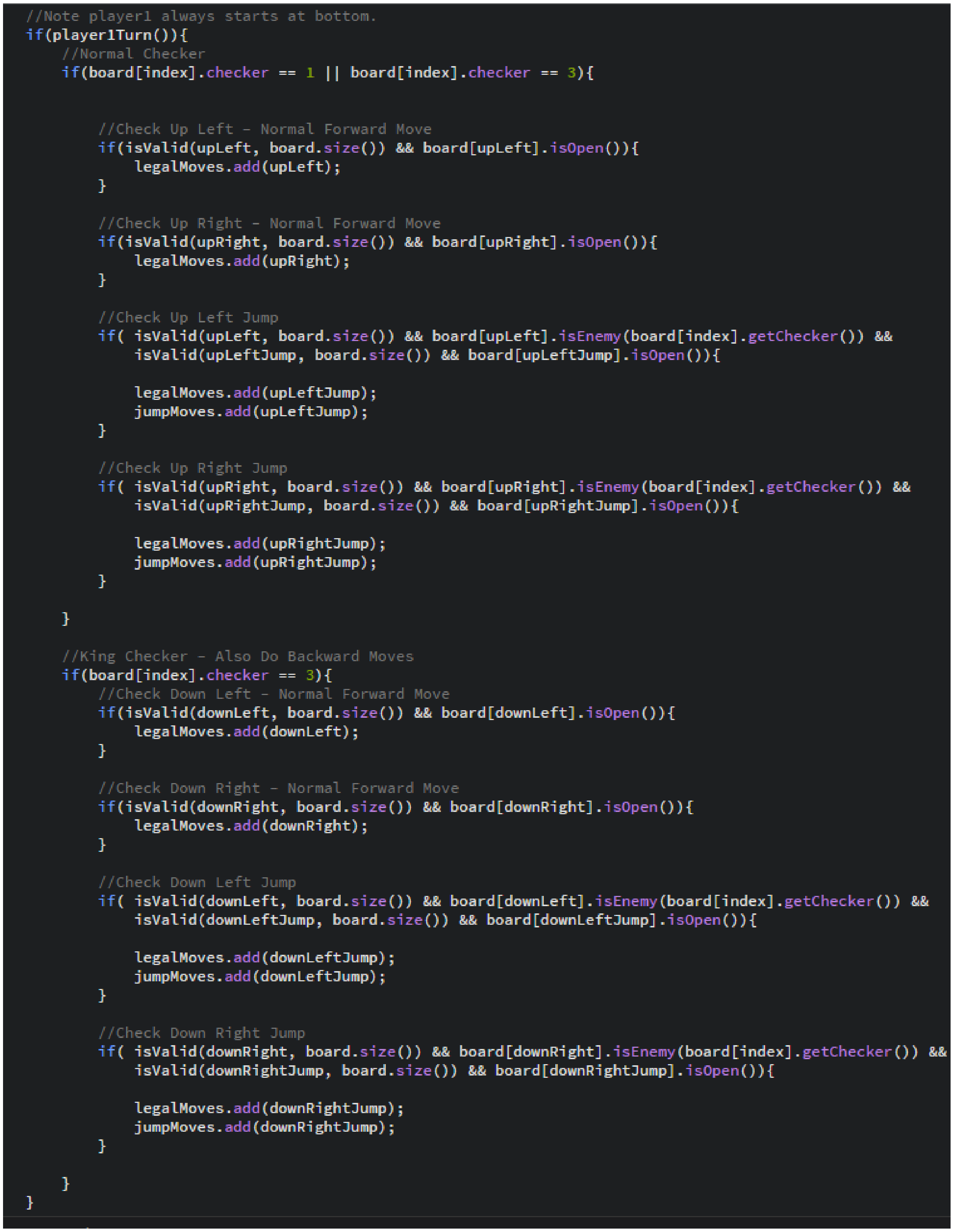
The StateSpace will be altered as the player make his/her moves. (Refer to possible, legal, and illegal moves above). The AI may temporarily store predicted StateSpaces in order to see which would be the best move to make (in accordance with the Min-Max algorithm) – these StateSpaces must adhere by the legal guidelines as well.

Game StateSpace Change

When a player selects a valid piece during their turn (player turn/color is determined randomly, darkcolor always goes first). A Boolean player1Turn will be used to lock/unlock player actions. Once selected, possible moves are generated (shown below). When a user chooses a valid square that fits the legalMoves list, the move will commence. After the move, we check to see if it was a jump, see below move algorithm why. After move is completed fully, we end the players turn and enable the other player to be allowed to make their move. This is the start of the new StateSpace.

Move Algorithm

Moves will be determined based on players turn, checkertype, and squares clicked. Note player1 moves are mirrored player2 moves. Example, forward is ‘up’ on the board for player1 and ‘down’ on the board for player2. (Note, player1 will always prioritize the bottom of the board, for a human should always get the bottom POV when facing an AI.) Below is a sample algorithm of how to calculate legal moves. After the calculation, we compare move the user decides to take and see if it was a jump move of not. If it was a jump move, we MUST check to see if they can jump again, for the rules state a piece must make possible jumps after a single jump.



```
//Note player1 always starts at bottom.
if(player1Turn()){
    //Normal Checker
    if(board[index].checker == 1 || board[index].checker == 3){


        //Check Up Left - Normal Forward Move
        if(isValid(upLeft, board.size()) && board[upLeft].isOpen()){
            legalMoves.add(upLeft);
        }

        //Check Up Right - Normal Forward Move
        if(isValid(upRight, board.size()) && board[upRight].isOpen()){
            legalMoves.add(upRight);
        }

        //Check Up Left Jump
        if( isValid(upLeft, board.size()) && board[upLeft].isEnemy(board[index].getChecker()) &&
            isValid(upLeftJump, board.size()) && board[upLeftJump].isOpen()){

            legalMoves.add(upLeftJump);
            jumpMoves.add(upLeftJump);
        }

        //Check Up Right Jump
        if( isValid(upRight, board.size()) && board[upRight].isEnemy(board[index].getChecker()) &&
            isValid(upRightJump, board.size()) && board[upRightJump].isOpen()){

            legalMoves.add(upRightJump);
            jumpMoves.add(upRightJump);
        }

    }

    //King Checker - Also Do Backward Moves
    if(board[index].checker == 3){
        //Check Down Left - Normal Forward Move
        if(isValid(downLeft, board.size()) && board[downLeft].isOpen()){
            legalMoves.add(downLeft);
        }

        //Check Down Right - Normal Forward Move
        if(isValid(downRight, board.size()) && board[downRight].isOpen()){
            legalMoves.add(downRight);
        }

        //Check Down Left Jump
        if( isValid(downLeft, board.size()) && board[downLeft].isEnemy(board[index].getChecker()) &&
            isValid(downLeftJump, board.size()) && board[downLeftJump].isOpen()){

            legalMoves.add(downLeftJump);
            jumpMoves.add(downLeftJump);
        }

        //Check Down Right Jump
        if( isValid(downRight, board.size()) && board[downRight].isEnemy(board[index].getChecker()) &&
            isValid(downRightJump, board.size()) && board[downRightJump].isOpen()){

            legalMoves.add(downRightJump);
            jumpMoves.add(downRightJump);
        }

    }
}
```