Othello Section Takeaway Notes

Containment and Inheritance/Interfaces
There are many ways to design this project! We don’t have a “right” answer, but here are some suggestions:

- **Game class**
  - Like with previous projects, you may find it useful to have a Game class to manage some of the gameplay logic.

- **Referee class**
  - Can be useful to manage the players and turn-taking logic.
  - Not required, but helpful for some designs!

- **Minimax class vs. method in Computer class**
  - If Minimax is its own class, you could easily swap in different algorithms for the computer player.
  - Because you only need one AI for this project, this isn’t required.

- **BoardSquare class**
  - Could represent a space on board - empty, black piece, or white piece

- **Piece class**
  - Does each color piece need its own class, or could you use one class to represent both colors?

- **Move class**
  - You may find it useful to be able to track the row, column, and score of a move all together. How could a class help you accomplish this?

- **Control class**
  - Because there are so many buttons, it may be helpful to factor this into its own class in the PaneOrganizer.
  - Apply Settings should begin the game, so the Controls class will need a reference to the Game class.

- **Polymorphism**
  - What do human players and computer players have in common?
  - How could your game play without caring which type of player the user had selected?

- **Association**
  - This is a complicated game, so many classes will need to know about each other.
  - Still, try to minimize the number of associations. If you find that a method requires that its class knows about every other class, consider moving the method.

The Board

- **The board is most cleanly represented as a 2D array**
- **This can be 2 2D arrays, one with green squares and one with pieces, or one 2D array holding piece-square objects.**
- **One big advantage to the one 2D array design is that you have less to keep track of, and can’t accidentally introduce bugs indexing from one array to the other.**
● When will the board need to graphically (and logically) change?
   ○ When a player (human or computer) adds a piece.
   ○ When a sandwich is created and pieces change color.
   ○ When it is a human player’s turn, possible moves are highlighted.

● When will the board NOT need to graphically change?
   ○ As the AI makes recursive potential moves in the Minimax algorithm - the board should not show these graphically.
   ○ Instead, you’ll need an invisible copy of the board that exists logically, but not graphically.
   ○ How do we implement this?
     ■ Board should have copy constructor and visible flag. If board is not visible, squares and pieces should not be visually added to pane.
     ■ VirtualBoard class would be nearly identical to GraphicalBoard, so no need for 2 classes.

Sandwich checking
● This is looking at EMPTY spaces where you could potentially make a move. It’s asking “Can I put a piece here?” and returning a “Yes” or “No” in the form of a Boolean.
● A while true is a forever loop. We do this because we don’t know how long we want to loop for. However, this means that you have to be extra careful about what you put in the loop -- if you don’t make sure to break out of the code, it will go on infinitely.
● Note that actually flipping pieces will involve a lot of this same logic!
● How can you efficiently check all 8 directions? Consider that each direction is just a -1, 0, or 1 shift in both the column and the row. Can you loop over these possibilities somehow?
● Remember that this is high-level pseudocode! You cannot just copy and paste this into your project - you’ll need to consider edge cases and your own implementation.
● Pseudocode:

```java
boolean checkSandwich(row, col, player):
    sandwich = false
    for all 8 directions around board[row][col]:
        flag = false
        while true:
            if board[row][col] is empty OR out of bounds:
                move to next direction
            if board[row][col] is opponent’s piece:
                update row and col to move 1 square in direction
                flag = true
            else if board[row][col] is own piece:
                if flag is true:
                    sandwich = true
                    move to next direction
    return sandwich
```
Minimax

- Be sure to check out the Minimax handout for more information on this!
- Again, please remember that this is high-level pseudocode!
- Pseudocode:

  Move getBestMove(board, intelligence, current player):

  
  if game is over and current player wins:
    return high value move

  else if game is over and current player loses:
    return low value move

  else if game is over and it is a tie:
    return neutral move

  if there are no valid moves:

    if intelligence is 1:
      return low value move

    else:
      return -getBestMove(board, intelligence - 1, opp. player)

  for each valid move

    make move on copy board

    if intelligence is 1:
      value = evaluate copy board
    else:
      value = -getBestMove(copy board, intelligence - 1, opp. player)

  return best value move of valid moves