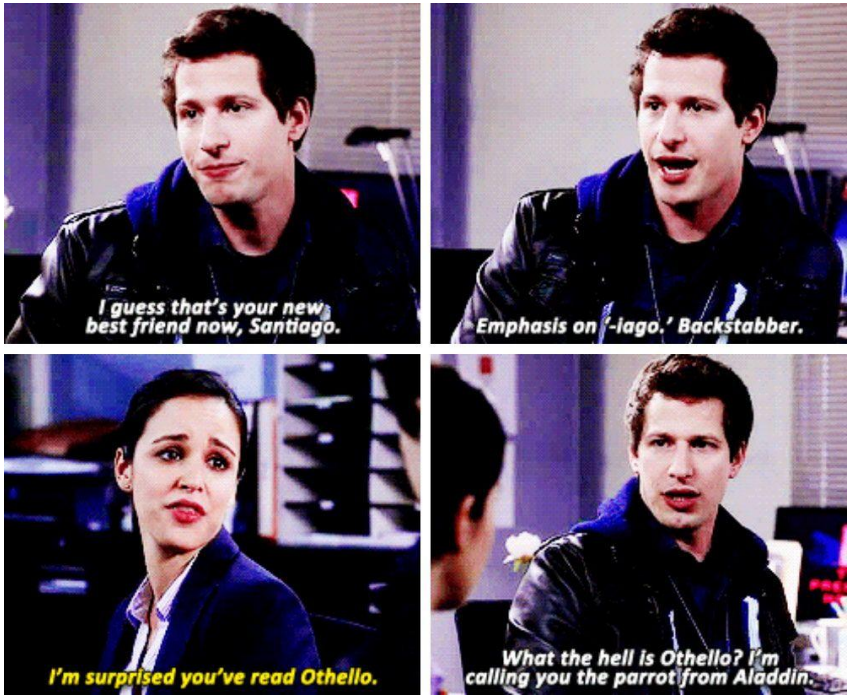
Othello Help Session

Othello TAs: Gil, Julia, Lisa, Lucy, Marina, Maura, Noah, Selena, Sierra, Tom, Tzion



The Office adjacent, right?!

A Quick Reminder

Please fill out the Final Project Declaration Form by Saturday if you haven't already!

- This form is ***binding!***
- Starting Sunday, you will have to email the HTAs to switch your FP.

Overview

- Writing the GUI
- Switching turns and the referee
- Checking for valid moves and flipping
- The MINIMAX(!!!) algorithm
- Copy constructor
- Project Design

Writing the GUI

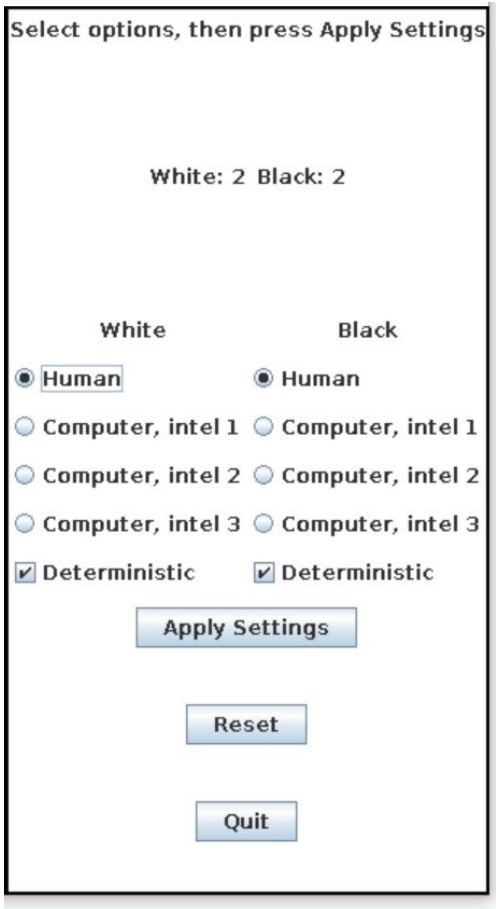
Writing the GUI (1/2)

Think Tetris! 2D Game Board

- Array of squares where each square represents a space on the board
- Could use two arrays – one for pieces and one for squares.
- Could use one array for square – each square can **contain** a piece.
  - Makes your squares smarter, easier to handle Game logic
- Board needs to react when you click on it… think of which components should be “smart”
- **Important question to answer early on:** How will you represent and store your Othello pieces?
  - more on this later!

Writing the GUI (2/2)

- Player Menus
  - Clicking a radio button sets a player as human or computer
  - Should be able to switch between **human** and **computer players**, and two computers of different intelligence levels, **at any point during a game**
    - Play with the demo to see how this works!
  - Have a default player configuration for when the game begins (otherwise you may have `NullPointerException`s!)



Switching Turns and the Referee

Switching Turns and the Referee 1/3

- Game would be pretty boring if it only had one player
- Need some way of figuring out when one player's turn is over to switch to next player
- We can use a Referee class!
  - keeps a reference to both players, keep track of which one is the current player
  - tells a player to move when it is their turn

Switching Turns and the Referee 2/3

- Referee handles checking for illegal moves
  - keeps players from moving out of turn
  - keeps players from moving when the game has ended
  - makes sure that a human doesn’t try to move during computer player’s turn (and vice versa)
  - etc…

Switching Turns and the Referee 3/3

- Referee should contain a `Timeline`! But why?
  - Computer player's moves should update on screen with each timer tick
  - So how do we do that? The first line of `handle()`should call `stop()` on the `Timeline`. Where then should we be calling `play()`?
  - Should the referee tell itself that the player has finished moving or should the referee be told that a move, either a human player's or a computer players, has ended?

Move Validity & Flipping

Checking for Valid Moves 1/2

- How fun would a game be if there was no such thing as a “valid” move? (**hint:** not at all!)
- Checking for a valid move:
  - first, make sure that the space is empty
  - then, make sure that a sandwich can be made by placing a piece at that location

Checking for Valid Moves 2/2

- How to check for sandwiches?
  - you have two potential tools – **recursion** and/or **iteration**
  - start with a square, check to see if the adjacent square is of the opposite color (i.e. the beginning of a sandwich)
  - then, check the next square, and the next square, and the next square, etc…. **UNTIL**:
    - You reach a square with a piece of your own color … move is valid!
    - Empty square or board edge is reached … move is invalid!
  - must check all 8 adjacent squares – if a sandwich is found in at least one of those directions, the move is valid.
    - How can you check 8 directions in a code-minimizing way?

Flipping Pieces 1/2

- Now that we've figured out how to check for valid moves, what should we do when we one is made?
- Flip pieces! This boils down to:
  - check in each direction to see which pieces have been sandwiched
  - for each direction, flip all of the pieces in the middle of the sandwich, until you reach a piece of your own color
  - this could be done **recursively** or **iteratively**
  - very similar algorithm to checking for a sandwich

Flipping Pieces 2/2

- What about when you get to the edges of the board?
  - something has gone wrong, you shouldn't reach one of these when you are flipping!
- Remember that a sandwich could be created in more than one direction, so we must flip pieces in all directions that have a sandwich
- After you get flipping working, you're almost done with the Human Player!

… But we're all about computers in CS15 …

Minimax Algorithm

Don’t worry, your computer won’t be *that* smart :)

Minimax Overview 1/4

- This is **one** algorithm that allows your computer to play at **three** levels of intelligence, where each level corresponds to a level of recursion
  - so, a level three computer means three method calls (the initial invocation + 2 recursive invocations per move)
  - decides player's best move by looking ~ into the future ~
  - **please see the AI handout for more in-depth information.**

Minimax Overview 2/4

- **Level 1**: For each possible move
  - make the move and see what the score would be
  - keep track of best move (one with highest score)
  - after all moves have been considered, return the best move (i.e., the one with the highest `boardEval()`).

Minimax Overview 3/4

- **Level 2**: For each of your possible moves
  - make the move, and find all of the opponent's valid moves
  - now, switch the perspective to the opponent and recursively call `miniMax()` on each of the opponent's possible moves.
  - assume that the opponent would make the best possible move, given your move, with intelligence 1 (`intelligence - 1`).
  - the call to minimax should return the opponent's best move, given your first move
  - then negate their move, and use that information to keep track of the best move that *you* can take. Negate the move because you should be trying to minimize the opponent's score.

Minimax Overview 4/4

- **Level 3**: For each possible move
  - make each possible first move, find all of the opponent's moves
  - call `miniMax()` recursively from opponent's perspective, at one less intelligence level (`intelligence - 1`)
  - from there, the algorithm should make each possible second move
  - it should then call `miniMax()` recursively from the opponent's opponent's perspective – the first player – at (`intelligence - 1`)
  - now, we are at the base case. Do what we would do in the level 1 case, remembering to return the best move
  - Then, at the second level, the opponent should negate its opponent's best move, and use that information to choose and return its best move to the first player
  - Then, the first player should negate the second player's best move and use that information to decide what its own best move is.

Minimax Continued 1/3

- How do we make a whole bunch of moves?
  - we want to make sure that we are able to “try out” possible best moves on our board without updating the actual board visually
  - solution: Make a *dummy board* that is a copy of the actual game board
    - One way we can do this is a copy constructor (more on this later)

Minimax Continued 2/3

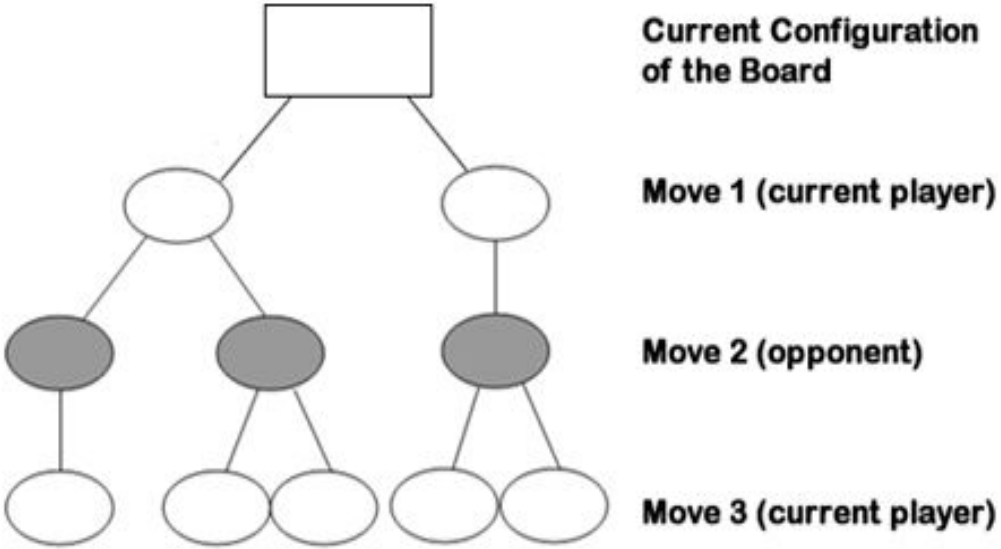
- What does it mean for a move to be the “best”?
  - each space on the board has a particular value, or weight, that is high for “good” spaces (i.e. corners) and low for relatively bad ones
  - we’ve provided a table of suggested values in the AI handout
  - you are welcome to implement your own!
  - you’ll want to write a `boardEval()` method that decides the “value” of a player’s move
  - the move with the highest `boardEval()` is the best move

Minimax Continued 3/3

- How do we return a “move”?
  - need to return a row, column pair representing the move, as well as the value of that move
  - think of how you can return multiple pieces of information from a single method …
- Let’s walk through an example of Minimax in action…

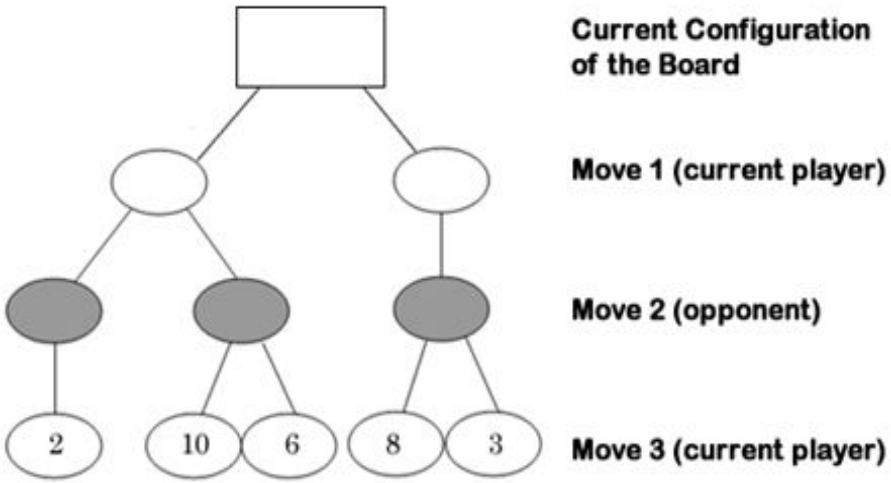
Minimax Example (1/5)

- Example in the handout with an AI of level 3 intelligence
- Each possible choice for a move is simulated on a “dummy” board
- Since the intelligence level is 3, this is done for three successive moves (current player – opponent – current player)



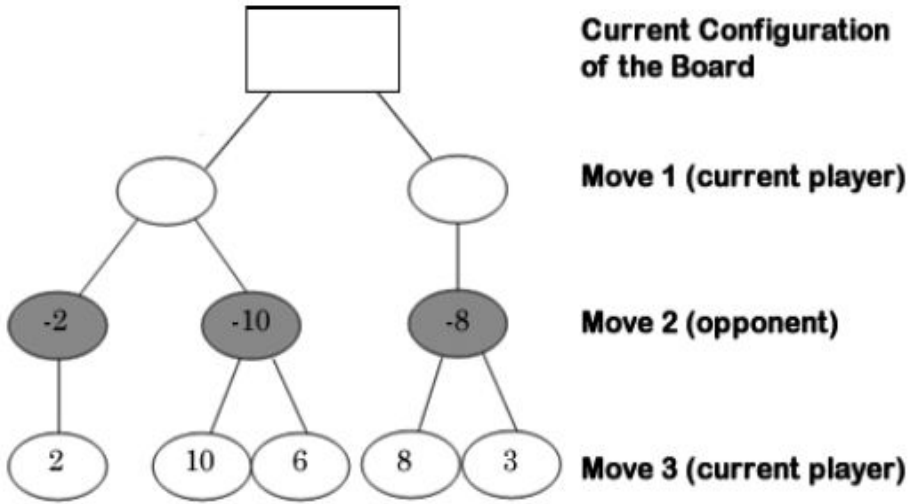
Minimax Example (2/5)

- At the base case, the board advantage of the current player is determined using the `boardEval()` method you will write, which takes advantage of the default board weights
- The board evaluation is simply summing over all of the values of the spaces that the current player "owns", and subtracting that from the sum of the values of the spaces that the opponent "owns"
- Since this is the base case, the values are directly calculated using `boardEval()` for the current player



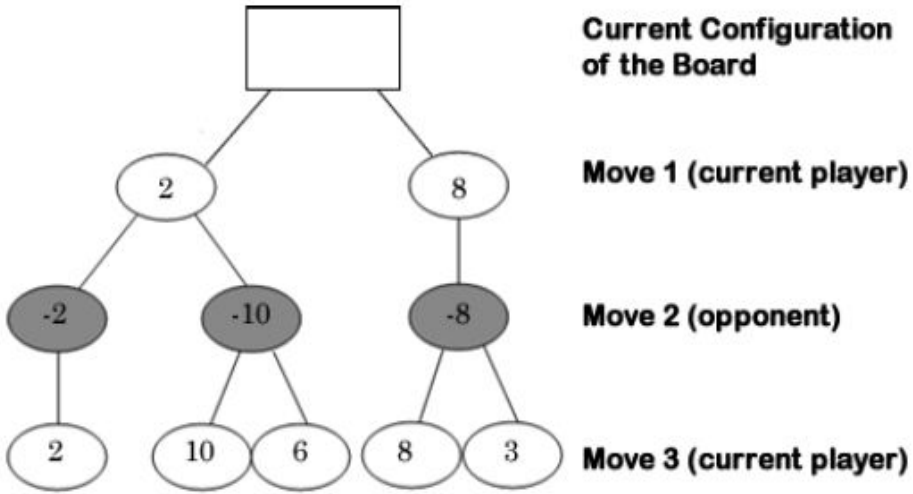
Minimax Example (3/5)

- At the 2nd level, board values are again evaluated for each of the possible configurations
- However, we use the greatest value from the board values returned from the previous level
- Then, this value ***must get*** negated; we do this because we are now considering the board from opponent's perspective – a good configuration for my opponent is a bad one for me!



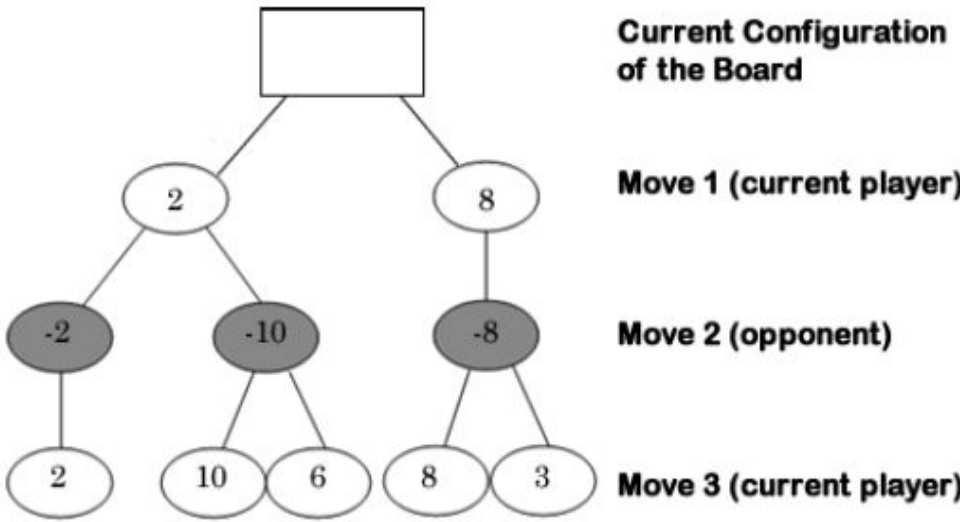
Minimax Example (4/5)

- At the 1st level, board values are again evaluated for each of the possible configurations after the 1st move
- However, we use the greatest value from the board values returned from the previous (2nd) level
- Then, this value ***must get*** negated because we are now considering the board from current player's perspective – *A bad configuration for my opponent is a good one for me*
- See any commonalities? (**hint:** yes!)



Minimax Example (5/5)

- In this example, the current player should do the move that will earn them 8 points since it's the best move from using our algorithm
- Our Minimax algorithm allows us pick ***the*** best move, based on the assumption that the opponent will try to make ***their*** best move, based on the assumption that we would make ***our*** best move.
- In general, you can use this algorithm to look ahead as many moves as you'd like

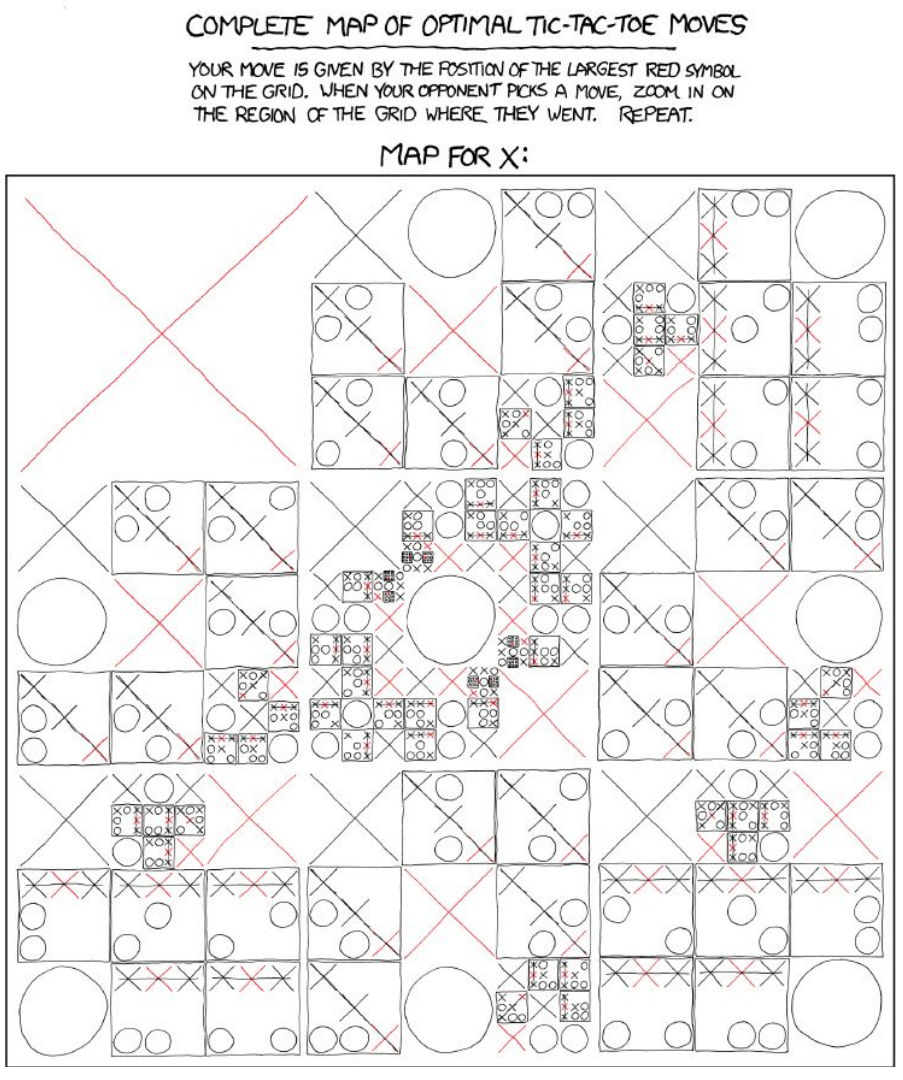


Minimax Pseudocode

```
public <return type> getBestMove(<parameters>)
    for each move that the current player can make:
        make a dummy board
        make the move on that dummy board

        if we are at the base case (i.e., intelligence = 1):
            move's value is the boardEval for the current player

        otherwise:
            move's value is that of getBestMove(…)
            from the opponent's perspective (remember to
            negate the value of what gets returned)

    return the move with the highest value
```

Minimax: Tic Tac Toe Edition

If you want to take a closer look at this, here is the link!
https://xkcd.com/832/



Special Cases 1/2

- Sometimes, a player may not be able to move if there are no legal moves during their turn. In this scenario, the opponent gets to go again, i.e., twice in a row.
  - this means that you should first check to see if there are any moves before trying to make all of the possible moves!

Special Cases For 2/2

- Sometimes, neither player can make any valid moves, even if there are open spaces left on the board
  - this means that the game is over – you should check for this special case as well!
- Sometimes, the person you're playing against isn't very good
  - very dangerous special case – you should approach this scenario by leaving **no mercy** for your opponent

Copy Constructor

Background: Method Overloading (1/3)

- In Java, we can do something called **method overloading**, in which we can create multiple methods of the same name *only if* they have different argument lists.
  - This allows us to write methods that have a similar end goal, but the steps are different depending on the inputs

Background: Method Overloading (2/3)

- **Example:** Say I want to create a `sum` method, and I want to be able to add in the cases where my inputs are (1) two `int`s, (2) three `int`s, (3) two `double`s

```
public class Sum{
  // constructor

  public int sum(int a, int b){
    return (a + b);
  }
}
```

```
public class Sum{
  // constructor

  public int sum(int a, int b,
   int c){
    return (a + b + c);
  }
}
```

```
public class Sum{
  // constructor

  public int sum(double a,
   double b){
    return (int) (a + b);
  }
}
```

- The compiler will call whichever method matches the arguments!

Background: Method Overloading (3/3)

JavaFX is full of examples!

Constructors

Constructor and Description
Rectangle() Creates an empty instance of Rectangle.
Rectangle(double width, double height) Creates a new instance of Rectangle with the given size.
Rectangle(double x, double y, double width, double height) Creates a new instance of Rectangle with the given position and size.
Rectangle(double width, double height, **Paint** fill) Creates a new instance of Rectangle with the given size and fill.

Copy Constructor (1/3)

- Minimax requires a lot of “testing out” possible moves – how do we do this without modifying our main game board?

- Answer: Use a copy constructor!
- Why use a copy constructor?
  - let’s you replicate the current state of the game
  - gives you an internal board that is a mirror image of, but logically and graphically distinct from, your visual game board
  - since higher intelligence requires looking at two or three moves in the future, we must pass the copied boards, on which we made previous moves, to successive levels of recursion

Copy Constructor (2/3)

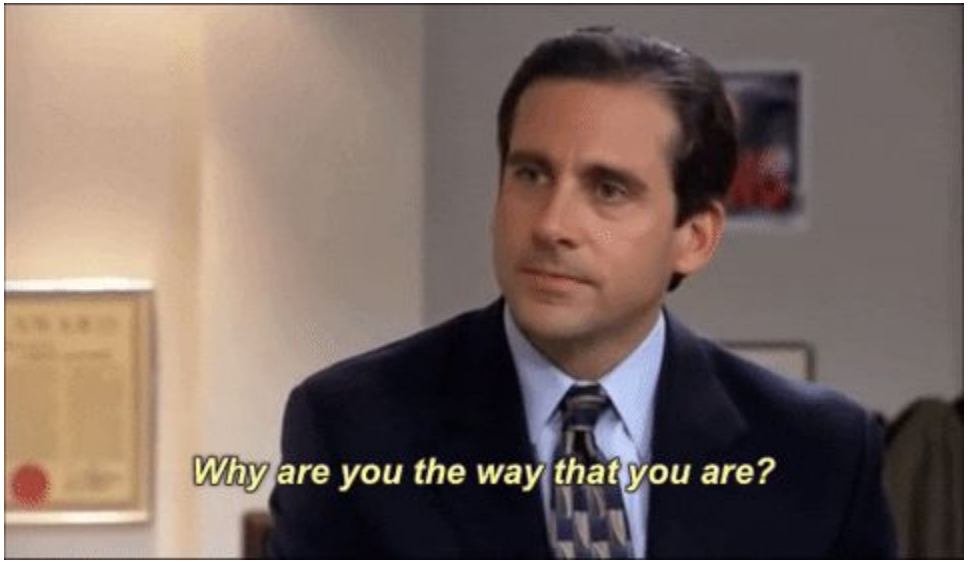
- Similar to method overloading, a copy constructor will create an **overloaded constructor** for a class
- Takes in exactly one parameter – a reference to some instance of the same class
  - For example: `public Car(Car car)`
- Copy constructor then uses accessors to retrieve important information from the instance and assigns the *exact same values* to itself
  - **Note:** you should never copy objects from one class to another, only primitive types like `int`s or `boolean`s!

Copy Constructor (3/3)

```
public  class Student{
  // old school constructor
  public Student(){ … }

  // new school, copy constructor
  public Student(Student stud){
    _age = stud.getAge();
    _name = stud.getName();
    _grade = stud.getGrade();
  }
}
```

- This example would make an identical, copy student of whatever student is passed in!
- Watch out for assigning arrays like this, though:
  - `_boardArray = gameBoard.getBoard()`
  - It will make your copied array point to the same memory location as your visual board
- A board is just a 2D array of objects, so all we need to do is iterate through each location in the old array and make a copy of its contents for each corresponding location in the new array

Program Design



Program Design

- Since Othello is a large project, there are many different designs that can work!
  - However, make sure that you abide by OOP standards; although there are many designs that will function, there are still better designs than others
- We'll go over some classes you might consider using
  - This list is not extensive, and not all are required! These are just pointers towards one valid way of thinking about the project
- As always, justify all design choices in the README!

Some Potential Classes

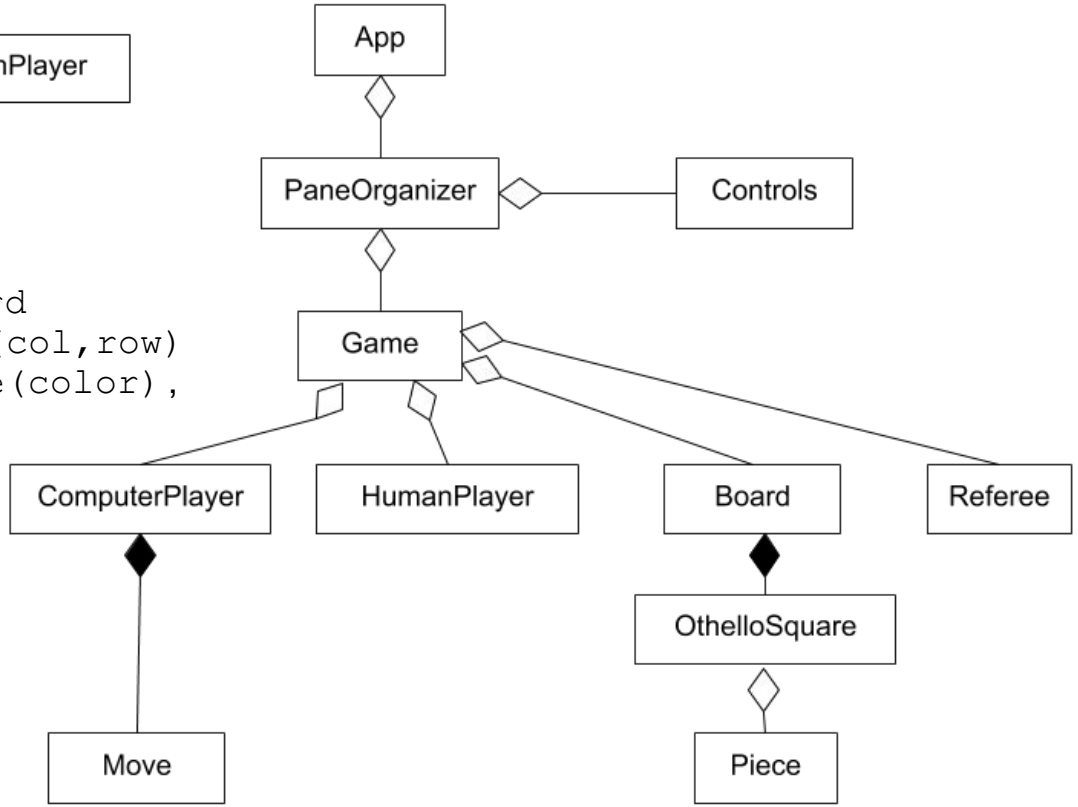
- **Game Class:** You may find it useful to have a Game class to manage some of the gameplay logic
- **Referee Class:** Can be used to manage the players and turn-taking
  - Not required, but helpful for some designs!
- **BoardSquare Class:** Could represent a space on board - empty, black piece, or white piece
- **Piece Class:** Consider if you need two classes for black vs. white pieces, or could create one class for them both
- **Move Class:** Used to track the row, column, and score of a move
- **Control Class:** Can factor out from PO since there are lots of buttons

Design Questions to Consider

- What are the benefits of having Minimax as a separate class versus a method in the ComputerPlayer Class?
  - For this project, you do not have to have a separate class
- **Polymorphism:**
  - What do the Human and Computer Players have in common?
  - How could you play without caring which player was playing?
- **Association:**
  - What classes *absolutely* need to know about each other?
  - Try to minimize the number of associations -- this is a great way to figure out what the best class is for a given method!

Our Recommended Design

- OthelloSquare can have methods like `getRow()` and `getCol()` so that `Board` doesn't need to convert from `(y,x)` to `(col,row)`
- It can also have methods like `addPiece(color)`, `flipPiece()`, etc.
- This suggestion is a "smart square"!



Good Luck!

- Start **NOW** (seriously, there's no reason to wait)
- Make sure you get human vs. human working *early* so you leave enough time to write the computer player algorithm
- Don't underestimate Minimax! It is a challenging algorithm to grasp conceptually. Come to hours early to talk about it!
- Hand simulate, use print lines, think critically about your problems.
    - Write a method for debugging that prints the current board state
- Did we mention start now?
- Good luck – Othello's tough, but (in our opinion) the most gratifying final project to do. **You got this.**

Questions?