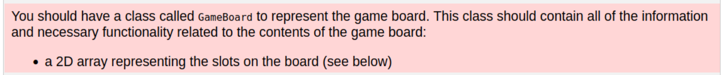
Connect Four

discuss in office hours!

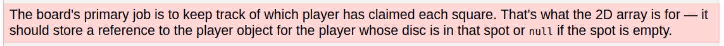
- creating classes – how do you know when to add instance variables, getters and setters, etc?
  - most of the class design is given in the handout – include those elements
  - add additional elements if you need them
    - parameters allow information to be provided to a method – add if the method needs info it doesn't have
      - a constructor needs parameters when there isn't a default way to initialize instance variables, or different values are desired in different circumstances
    - variables store stuff – add instance variables if there are additional properties (but don't make something that should be a local variable an instance variable instead)
    - methods do stuff / provide access to stuff – add methods if something outside the class needs additional access (but don't provide more access than necessary)
  - private helper methods let you pull distinct or repeated tasks out of a larger method to simplify it
    - use for the same reason you'd create functions or subroutines in non-object-oriented programming

Connect Four

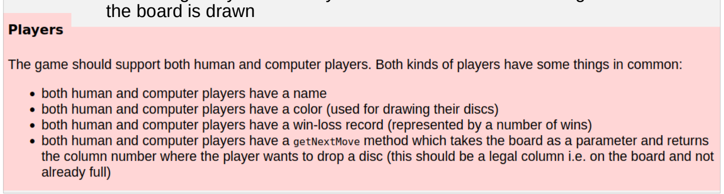
- how do you keep track of whose pieces are where?


You should have a class called `GameBoard` to represent the game board. This class should contain all of the information and necessary functionality related to the contents of the game board:

- a 2D array representing the slots on the board (see below)


The board's primary job is to keep track of which player has claimed each square. That's what the 2D array is for — it should store a reference to the player object for the player whose disc is in that spot or `null` if the spot is empty.

  - the 2D array will store `Player` objects – when a player's piece goes into a particular slot in the game, the corresponding `Player` object is stored in the corresponding slot in the array
    - this then gives you the info you need to draw discs in the right color when the board is drawn

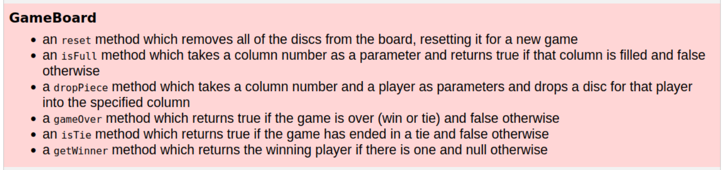

Players

The game should support both human and computer players. Both kinds of players have some things in common:

- both human and computer players have a name
- both human and computer players have a color (used for drawing their discs)
- both human and computer players have a win-loss record (represented by a number of wins)
- both human and computer players have a `getNextMove` method which takes the board as a parameter and returns the column number where the player wants to drop a disc (this should be a legal column i.e. on the board and not already full)

Connect Four

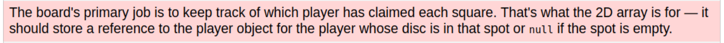
How do you –
- figure out where there's an open space for a game piece?
- make the computer player recognize where it can place a piece?
- reset the board?
- check for a win?

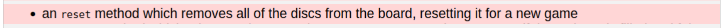

GameBoard

- an `reset` method which removes all of the discs from the board, resetting it for a new game
- an `isFull` method which takes a column number as a parameter and returns true if that column is filled and false otherwise
- a `dropPiece` method which takes a column number and a player as parameters and drops a disc for that player into the specified column
- a `gameOver` method which returns true if the game is over (win or tie) and false otherwise
- an `isTie` method which returns true if the game has ended in a tie and false otherwise
- a `getWinner` method which returns the winning player if there is one and null otherwise

  - most of these are questions about how to implement `GameBoard` operations

Connect Four


The board's primary job is to keep track of which player has claimed each square. That's what the 2D array is for — it should store a reference to the player object for the player whose disc is in that spot or `null` if the spot is empty.


- an `reset` method which removes all of the discs from the board, resetting it for a new game

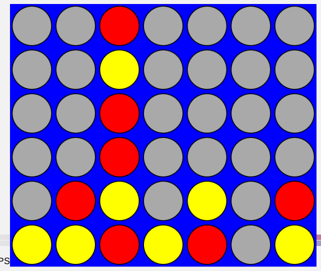
  - `reset` makes all the spots empty – set each slot of the array to `null`

Connect Four

The board's primary job is to keep track of which player has claimed each square. That's what the 2D array is for — it should store a reference to the player object for the player whose disc is in that spot or `null` if the spot is empty.

- an `isFull` method which takes a column number as a parameter and returns true if that column is filled and false otherwise
- a `dropPiece` method which takes a column number and a player as parameters and drops a disc for that player into the specified column

  – these involve working out something in a (2D) array – remember the example-and-picture strategy for figuring out remove from a 1D array



how can you tell if a column is full?

where does a piece land when it is dropped?

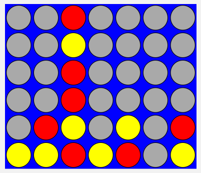
in both cases, think about which spots in the array you need to check and what you are looking for in those spots to be able to answer the question

Connect Four

The board's primary job is to keep track of which player has claimed each square. That's what the 2D array is for — it should store a reference to the player object for the player whose disc is in that spot or `null` if the spot is empty.

- an `isTie` method which returns true if the game has ended in a tie and false otherwise

- a tie occurs when the board is full without a winner
  – how can you tell if the board is full?

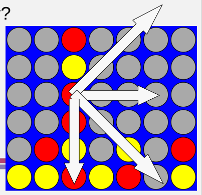


Connect Four

The board's primary job is to keep track of which player has claimed each square. That's what the 2D array is for — it should store a reference to the player object for the player whose disc is in that spot or `null` if the spot is empty.

- a `getWinner` method which returns the winning player if there is one and null otherwise

- checking for four in a row takes some care not to miss a case
  – start with something you are confident is a correct strategy and which you can implement correctly
    - check whether a particular player won rather than whether anyone has won
    - for each spot on the board, is it the start of four in a row?
      – (why do you only need to check the four directions shown?)
      – (this is a good situation for private helper methods)
    - how do you identify four of the same player in a row?
      – check for the same contents i.e. `board[r1][c1] == board[r2][c2]`
  – consider a more efficient strategy when you have something working
    - can you scan each row, column, and diagonal only once?



Connect Four

- how do you do different things for computer and human player turns?

Players

The game should support both human and computer players. Both kinds of players have some things in common:

- both human and computer players have a name
- both human and computer players have a color (used for drawing their discs)
- both human and computer players have a win-loss record (represented by a number of wins)
- both human and computer players have a `getNextMove` method which takes the board as a parameter and returns the column number where the player wants to drop a disc (this should be a legal column i.e. on the board and not already full)

You should make use of inheritance: create a class `Player` for the things common to both kinds of players and then create subclasses `ComputerPlayer` and `HumanPlayer` to do the things specific to one type of player. All three classes should have a constructor which takes the player's name and color as parameters. Make appropriate decisions about where instance variables and methods are declared, and what (if anything) is abstract.

  – write different bodies for `getNextMove` in `ComputerPlayer` and `HumanPlayer`
  – utilize polymorphism – in the main program, create an array to hold the two players and just create the right kind of object for each player when the array is initialized
    - think `AccountDemo3`