Overview

Connect 4 is a two-player game. Players drop tokens into columns, trying to connect 4 of their tokens either diagonally, horizontally, or vertically. In this project, you'll write logic for a virtual Connect 4 and create an AI opponent using whatever method you prefer.

The project is split into 4 files

- **Main.java**: Handles Swing setup.
- **Connect4.Java**: Yet more Swing, also game logic.
- **Player.java**: Dictates how your bot will "think" and move.

The only one you'll write code in is **Player.java**. Don't modify other classes.

More about Connect 4: https://en.wikipedia.org/wiki/Connect_Four

Steps

Step 0

Download and look over the starter code in **Player.java**.

Fill in your name in the `public String getName(){}` function. Important step if you want to enter into the tournament. E.g. set it to `return "John Doe";`

Step 1

(Player.java)

Choose some tokens to represent the board. You need 3: `MY_TOKEN`, `OPP_TOKEN`, and `EMPTY_TOKEN`.

`MY_TOKEN` is the char to represent the bot.
`OPP_TOKEN` is the char to represent the bot's opponent (in this case, you).
`EMPTY_TOKEN` is the char to represent an empty board space.

*Notice that these *look* like constants, but aren't marked with "final"? Yeah…that's to make tournament logic easier later on. Please don't add "final" to these. That'll break my tournament code.

Additionally, don't you dare use hard coded values after setting these variables. Use MY_TOKEN instead of '@', for example (if you've set MY_TOKEN = '@').

Step 2

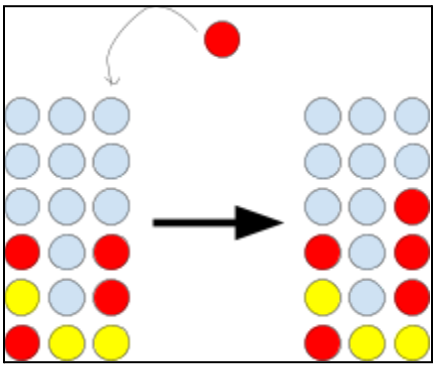
(Player.java)

If you try to run Main.java, you'll probably get an error. We haven't set how the bot moves yet – that's in `public int returnMove()`. This is supposed to return the column to drop your token in.

One simple way to make legal moves is

```
public int returnMove(...){
    return 0;
}
```

Dropping your token on the leftmost column every time. Not a winning strategy, but it's a start. Alternatively, you could make it choose a random column between 0 and COLS.



(What's meant by "dropping" a token)

Step 3
(Main.java)
Try running the program. You should get a playable bot. Not the best bot, but at least it makes moves.

Step 4 (NO CODING NEEDED)
(Player.java)
Right now our bot is kind of dumb. It loses pretty easily. We want to make it "smarter."

First of all, we could try to make our bot block obvious wins. Here's how that might work:

```
public int returnMove(){
    // Maybe we should drop in column i?
    // Simulate dropping in column i
    // Check if you won
    // Remove the token you just dropped
    // "I didn't win, maybe try another column" OR "I won, I should drop it there!"

    return ...column number…
}
```

This looks one move in advance and takes wins when it's available. It's a reasonable start.

Step 5
(Player.java)
To implement Step 4, first you'd need to implement a method that checks if you won. Remember, there are 3 ways to win: horizontally, vertically, or diagonally. E.g.

```
public boolean botWon(){
    for(…each column…){
        // Check horizontal wins
        // Check vertical wins
        // Check diagonal wins
    }

    return ...true if won / false if not…
}
```

Be careful of out-of-bounds errors. Use board[][] to access the current board state.

*In Player.java, the board[][] array represents the current grid. Use printBoard() to print this if you want to (useful for debugging). You might wonder why Player.java just instantiates an empty board. Well, the actual work of updating that board takes place in other classes, using the updateState() method. But don't worry about all that. Just know that board[][] will give you the current board state.

Step 6
(Player.java)
Continuing our implementation of Step 4, we also need a method to simulate & undo moves. Here's some pseudocode to get you started:

```
public void makeMove(int column){
    // find lowest you can go in a single column
    // set board[][] of that lowest-cell in that column to MY_TOKEN
}

public void undoMove(int column){
    // find first instance of non-empty board[][], going top -> bottom
    // set board[][] of that lowest-cell in that column to EMPTY_TOKEN
}
```

This step might require some debugging.

Step 7
(Player.java)
Putting Steps 4-6 all together, your returnMove() might look like

```
public int returnMove(){
    for(…each column i…){
        makeMove(i)
        if(botWon()){
            // Yes, let's drop in this column
        }
        undoMove(i)
    }
    return …chosen column…
}
```

Congratulations, if everything's working right, your bot will be a fair bit harder to play against.

*Potential bug: what if the column is full? Then you can't use makeMove() on it…

Step 8
(Player.java)
Let's take this a bit further. Currently, your bot can recognize obvious wins, but it doesn't really defend against imminent wins by *the opponent*. Fortunately, this is a relatively easy fix. First we test for obvious wins (this is already done in Step 7). If no wins are possible, we test for possible wins by the opponent — and when we find one, drop our own token in that column pre-emptively, blocking the opponent from winning on the next turn.

First, modify botWon() to determine *which side* won (if you haven't already). A potential method is to return the character of the winning sequence, later using that to determine if the bot or the opponent won. But there are other ways to go about it.

At this point, you might want to rename botWon() to something like someoneWon(), etc.

Next, modify makeMove() with an additional parameter: the color of the token to drop, either MY_TOKEN or OPP_TOKEN.

```
public void makeMove(int column, char side){
    // stuff here
}
```

undoMove() needs no modification, as long as you're removing the topmost token in a column.

Step 9
(Player.java)
Now we add all the logic in returnMove():

```
public int returnMove(){
    // check for wins
    for(…each column i…){
        makeMove(...column i, MY_TOKEN...)
        if(…bot won…){
            // Yes, let's drop in this column
        }
        undoMove(...column i...)
    }

    if(...column chosen...) return …chosen column…

    // else, check for imminent wins by the opponent
    for(...each column i...){
        makeMove(...column i, OPP_TOKEN...)
        if(…opponent won…){
            // Yikes, our opponent could win by dropping here
            // So let's be smart and drop here pre-emptively
        }
        undoMove(...column i...)
    }

    return …chosen column…
}
```

Our bot is now even smarter: it not only takes obvious wins, but prevents obvious losses. In fact, it's probably almost as strong as your average human player.

Remember: be careful about calling makeMove() on full columns.

That's pretty much the basic requirements of this assignment. Some practice in 2D array methods (checking wins) and fundamental game logic. If you'd like to make your bot even smarter, though, you can continue to do the optional steps.

Winners of the upcoming Connect-4 tournament will get bragging rights. Looking for near-optimal bots, fast runtimes, etc. Board size may be larger, so beware.

Step 10 [optional]
(Player.java)
If you want, you can try to make the bot look multiple moves ahead. The easiest way to do this is with minimax (https://en.wikipedia.org/wiki/Minimax#Pseudocode). Basically, it brute-forces every possible move, looks even further ahead to see which one is better, and plays the best move given this depth.

The pseudocode in Wikipedia can be repurposed for Player.java. Remember, to find the value of the board, the procedure is the same:

1. Simulate dropping a token
2. Check win/loss/draw
3. Remove token you simulated dropping.

A draw is when the entire board is full (no “empty” chars anywhere).

Warning: This step will likely involve lots of testing and debugging.

Step 11 [optional]
(Player.java)
Some extensions you could possibly take:

1. *Alpha-beta pruning*: currently, the program might take too long if you want a large minimax depth (10, maybe). Alpha-beta pruning can improve this. As always, the Wikipedia article is worth a read.
   https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

2. *Heuristics*: Any patterns you notice in general? E.g. it tends to be better to play in the center rather than the sides. Combine this with the previous alpha-beta pruning method to find good moves quickly.

3. *Randomness*: Never a good idea for a program to be too predictable, especially in games. Maybe add an element of randomness, where you choose between several equally good moves.