Project: Connect4

In this project, you will finish a program that plays the game Connect Four. You will need to understand the framework code we provide, which is rather complex; fix some small "bugs" in the code; write some smarts for the computer player, and implement a bomb piece.

Background

The game Connect Four involves dropping pieces into a board that is seven spaces wide by six spaces high (making 7 columns by 6 rows). Two players alternate turns and the first to place four pieces in a row (either horizontally, vertically, or diagonally) wins. The board is held vertically, and a player only chooses which column to place a piece; the piece will fall to the lowest unoccupied row.

Connect 4 is a very old game. The name "Connect Four" comes from a board game released by Milton Bradley and is copyrighted, so we will use "Connect 4" instead. There are many on-line versions of Connect Four available, and it will be especially valuable for you to familiarize yourself with it through playing games. As it turns out, the game has recently been "solved", and a computer can be programmed to play perfectly (you are not being asked to program a perfect solution, however!). Search the internet with the term "Connect Four", and you should find several playable versions.

While the standard version of Connect Four involves two players on a 7x6 board, the program you write will be able to handle any number of players and any (reasonably) sized board.

Framework code

There are four main areas of the code, represented by single or groups of classes:

- The `Connect4` class sets up the graphics, handles the main loop of the game, and handles mouse clicks.
  The most important area, from your standpoint, is in the `play()` and `takeTurn()` methods, which call each other recursively. This is called "mutual recursion", and is used in this case to let control switch between listeners, used by human players, and `getMove` method that need to be called, as performed by computer players.
- The `Board` class maintains the two dimensional array that maintains the state of the board—that is, which pieces are where. There are three important methods that you will write for this class, detailed below.
- The `Player` class and its subclasses `ComputerPlayer` (which is abstract) and `StupidComputerPlayer`, which basically guesses randomly. There is a `Player` instance for each player in the game, which keeps track of how many bombs the player has left, and so forth. The `ComputerPlayer` subclasses include a method `getMove`, which analyses a board and returns the best move for that player. You'll get to make that method smarter!

- The `Move` class and its subclasses `QuitMove` and `BombMove`. A `Move` instance is simply way of communicating moves and possible moves between other classes. At its simplest, a `Move` instance represents the number of the column to drop a piece in.

Take some time to understand the framework code. The version that it provides is not working fully, although you should play it and see what happens.

Requirements

There are five requirements in total. Some of these requirements are decidedly easier than others.

1. Implement the `Board` methods:
    - `possibleMove`, which returns `true` if a given move can take place with a given board state. About the only way a move isn't possible is if a column is filled already (although this won't stop a bomb move).
    - `addPiece`, which updates the board for a given move.
    - `winner`, which returns `true` if a player has four of their pieces in a row.
2. Fix a bug: clicks continue to have an effect after the game has been won. Do you *need* to add an instance variable to the `Connect4` class for this?
3. Implement "quit". The computer should quit when there are no possible moves to make, and the human should have a way to quit if your brilliantly coded computer player out foxes him.
4. Implement `SmarterComputerPlayer`, a subclass of `ComputerPlayer` that is smarter than the provided `StupidComputerPlayer`. This is the task that *could* be endless, but a few things are required:
    - Your player should always block if the other player has three in a row.
    - Your player shouldn't be fooled when the other player has two pieces in a row with two open spaces on either end.
    - Your player should have a preference for the center position when all else is equal. (The center is the strongest column in the seven column wide board).
    - Your player should do one other “smart” thing; you’ll explain what you did in your write-up.

    That said, there are other things you can do for your smart player:

    - Choose things smartly when it doesn't have to block to save the game. A common technique in game programming is "looking ahead", which is easy to do when your code has a powerful design. The basic idea is that you engage in a series of "what ifs" by making many hypothetical boards, where those boards are made by systematically proposing "what if I moved here, and the other player responded by moving here, ...". To do this successfully, you need a way of ranking which of the hypothetical boards are best so that you can choose a move that leads you to them.
        There is quite a bit of information on the web about MINMAX, which is an algorithm useful in two-player games that works on just these principles. Based on your rankings of the hypothetical boards, your player should choose the move with

the maximum score. With enough "look ahead", you'll need to simulate your opponents choices by assuming they will choose a move that offers the minimum score (for your player). Implementing this would certainly be challenging!

5. Implement bombs. Each player has a limited number of bomb pieces, and playing a bomb in a certain column removes the piece it lands on and any of the pieces in the 9 neighboring cells. If this removes a piece upon which another piece was resting, that piece should fall into a new position!

   You should also include the making of bomb moves in your `SmarterComputerPlayer` class, although this need not be too complex: play one when you are about to lose.

Furthermore, you need to implement these requirements such that they work for more than two players and for boards of any (reasonable) size. Reasonable means at least 4 wide and high and not so big that you can't see the board.

When you are done with these requirements, you can easily run a game where the computer plays itself, or three computer players play themselves. You just need to set up `ArrayList` of `Players` in the `Connect4` class with the right kind of `Player` instances.

Even before you are done with all of task #1 above, you can play human versus human games. Play with your friends or classmates, as this is a good way to get a feel for how the code works.

Having Trouble?

Talk to your class mates. And, talk to your instructor! You'll have several lab sessions to work on this where you instructor would *love* to see and talk to you.

Extras

There are many, many ways you can add extras to this program. For instance, consider:

- …animating piece drops, and bombs.
- …adding randomness to your computer player in smart ways.
- …connect 5?
- …changing the nature of the bomb piece in interesting ways (removing a whole row, or column; switching colors; etc).
- …different kinds of computer players, with different algorithms for picking a move. You might simulate games in which computer players with different algorithms play eachother.

Extra work will be taken into account when grading!

Write-up and submission

Your Java code needs to be well commented. You need to have testing code that exercises *all* of the additions you've made to the framework that you were give, with clearly labeled ways to run those tests. It is suggested that you use JUnit, but you can write regular Java classes to do your testing as well (add `Test` to the end of the testing classes names).

In addition to your Java project, you'll need to include a document that includes:

- A description of the additional "smart" action that your computer player performs: what it does, and when it occurs, where it is in your code, and how you wrote it.
- Descriptions of anything extra you did that wasn't covered in this document.

A text file in your Java project is a reasonable way to submit the write-up.