**CS 273 Laboratory 7: Arrays II**

This laboratory will give you some practice with two-dimensional arrays. We will be implementing the logic for capturing pieces in the game of "Go".

**Preliminaries**

If you are not familiar with the game of "Go" read the following description:

* Go is played on a square board with a square grid.
* One player plays black stones; the other plays white. Players take turns placing a stone of their color on one of the grid intersections.
* A player may "capture" the pieces of the other player. A group of adjacent pieces of one color is captured and removed from the board when they are completely surrounded by pieces of the other color. Another way of thinking of this, for programming purposes, is that none of the pieces in the group is adjacent to an empty space. If any piece in the group has even one empty space next to it, then the group is not captured.
* The word "adjacent" for the purposes of this game means to be "immediately next to" in one of the **four** directions, up, down, left or right. The diagonal directions are not considered.

**Review**

Compile and run GoFrame.java in the lab7 project that you created last time.

At the present time, the project causes a frame to be displayed that contains four buttons and a Go board that by default is 19x19. The only buttons that are implemented is the “Spiral” button, and the “Clear” button that you implemented in the last lab. You should also be able to click near an intersection and see its contents toggle between white, black and blank, which you also implemented in the last lab.

Review the code you wrote.

The variable board is a two-dimensional, rectangular array that represents our board in the game of Go. An element of the array typically contains one of the values WHITE, BLACK or EMPTY, depending on whether or not the square is occupied. These names are defined as symbolic constants. The actual values are small integer constants such as 0, 1 and 2, but it is good programming practice to refer to them by their symbolic names: WHITE, BLACK and EMPTY. Whenever one of these squares changes in the array, it is reflected on the display the next time that the board is repainted. Your task is to modify and extend GoFrame.java so that it performs each of the tasks described below.

**Laboratory**

**Part 0: Bounds checking – Optional, but strongly encouraged**

You may wish to start this lab by creating a bounds checking method because you will often want to check if a specific location is actually within your board. To avoid writing the same code 40 times, a method could serve you well. Here is a possible method heading that you could implement:

//Returns true if and only if (bx, by) is a valid location on the board

boolean isValid(int bx, int by)

**Part 1: Fill the board with randomly-placed stones**

Modify the randomizeBoard code so that it sets each space "randomly" as follows: approximately 20% empty, 40% black, 40% white. (This code will be run whenever the Random button is pressed.)

HINT: You can use either the Math.random() method or the Random class. Here’s the concept: say you wanted to get white 50% of the time and black the other 50% of the time -- if you generate a random number between 0.0 and 1.0, you can make it white if it’s less than 0.5 and black otherwise. That will give you an equal chance of white and black. Now: how would you do that for 20% empty, 40% black, and 40% white like this checkpoint is asking for?

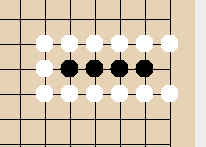
**checkpoint 1 (15 points): Demonstrate to your instructor or lab assistant that pressing the Random button causes stones to be placed on the board as specified above.**

**Overview of Parts 2-7: Capturing stones**

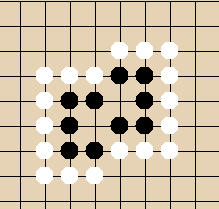
The rest of this lab involves the removeCapturedStones operation, whose purpose is to remove any stones that should be captured according to the rules of Go. This code is run whenever the Capture button is pressed.

The logic for capturing stones is conceptually simple: A stone may only be captured if it, and potentially any other stones of the same color that are adjacent to each other, are surrounded by stones of the opposite color with no empty spaces.

Examples:



These black stones may not be captured yet due to the empty space that one of the black stones in the group is adjacent to, but placing a white stone in that empty space will capture all the black stones.

In this situation, white can’t capture the black stones at all, because placing a white stone in one of the middle empty spaces would mean that that white stone is surrounded by black stones, so that white stone could be captured.

Another way to look at it: A stone should be captured and removed from the board EXCEPT when:

* It is adjacent to an empty space, or
* It is adjacent to a stone of the same color that is adjacent to an empty space, or
* It is adjacent to a stone of the same color that is adjacent to another stone of the same color that is … that is adjacent to an empty space.

We will therefore use two temporary "colors", BLACK\_IN\_PERIL and WHITE\_IN\_PERIL, which will help us figure out which stones to remove. BLACK\_IN\_PERIL stones will display as red; WHITE\_IN\_PERIL stones will display as pink. The idea of BLACK\_IN\_PERIL and WHITE\_IN\_PERIL is that these are stones that could be captured. However, we won't actually remove them until the “Capture” button is completely implemented and we know that they have no path to an empty space.

Capturing will be performed as follows:

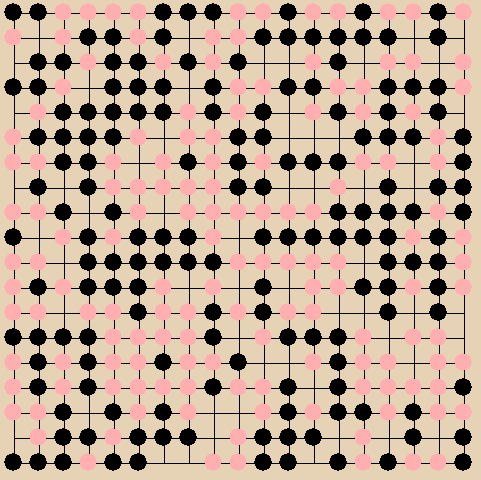
* Initially set all BLACK stones to BLACK\_IN\_PERIL and all WHITE stones to WHITE\_IN\_PERIL.
* Color each IN\_PERIL stone that is in a "safe group" back to its original color.
* Remove all remaining IN\_PERIL stones from the board.

Thus, after the entire capture has taken place, all spaces will be BLACK, WHITE or EMPTY. The user will never see the IN\_PERIL stones.

**Part 2: Capturing stones, step 1**

Implement the first part of the Capture method for white stones. First, change all the WHITE stones on the board to be WHITE\_IN\_PERIL. Again, WHITE\_IN\_PERIL means that it's a white stone, but we don't yet know if it is in a "safe" group or a "captured" group.

Do this much; then test your code. The board will display the WHITE\_IN\_PERIL stones as pink, so that you can see that you've done this much correctly. All white stones should turn pink, like so:



**checkpoint 2 (10 points): Demonstrate to your instructor or lab assistant that pressing the Capture button causes all white stones to turn pink.**

**Part 3: Capturing stones, step 2**

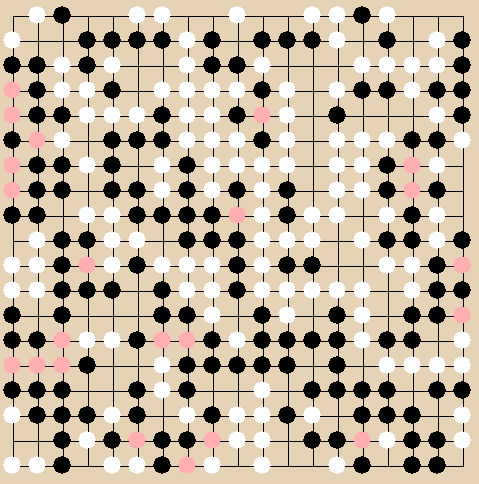
The next step is to identify the white stones that are part of safe groups, and to change their status back to WHITE. One thing to realize is that if any stone that is part of a group is safe, then the whole group is safe. This means that a WHITE\_IN\_PERIL that is next to a stone that is WHITE (and therefore already found to be safe) is also safe.

So now, do the following, looping through all locations on the board:

* If the stone is WHITE\_IN\_PERIL, then see whether any of the adjacent locations are EMPTY or contain a WHITE (safe) stone. If so, change the WHITE\_IN\_PERIL stone to WHITE.
* **Be careful**, do not perform any out-of-bounds array accesses when checking stones on the edges of the board. Otherwise you'll see that big black ‘X’. See Part 0 of these instructions for a recommendation of how to check for this using a helper method.
* Otherwise, do not change the stone’s color.

Stop and test your code to make sure that one pass through the board works correctly. This should cause all WHITE\_IN\_PERIL stones that are next to an empty space to change to WHITE, and may change other WHITE\_IN\_PERIL stones that are nearby. It will likely not change a WHITE\_IN\_PERIL stone that is "far away" from an empty space. That will be implemented in a later checkpoint.

There should be no pink stone next to an empty space. Most white stones will be next to an empty space. A string of white stones should start with a white stone next to an empty space at the top left of the string (assuming your algorithm iterates through the board from top left to bottom right).



**checkpoint 3 (35 points): Demonstrate to your instructor or lab assistant that pressing the Capture button causes only white stones that are “far away” from an empty space to turn pink.**

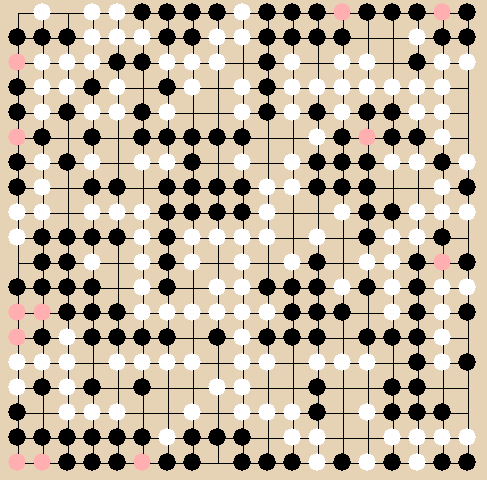
**Part 4: Capturing stones, step 3**

If any stones were changed from WHITE\_IN\_PERIL to WHITE, then we need to check the entire board again to see if stones next to the newly-WHITE stone need to be changed. So "wrap" your WHITE\_IN\_PERIL-to-WHITE nested loop in an additional loop that repeats the process three times.

Stop and test your code.

When you've looped through the entire board three times, then most safe stones that had been WHITE\_IN\_PERIL should have been changed back to WHITE. The only ones that are not WHITE are those whose only path from a blank square contains more than three up/left length moves (assuming your algorithm iterates through the board from top left to bottom right).

The only pink stones left should be surrounded by black stones, like the two pink stones in the top row below. Otherwise, a stone may be pink if it’s “far away” from a space, like the first pink stone in the first column or the pink stone in the right-middle of the board in the example below.



**checkpoint 4 (10 points): Demonstrate to your instructor or lab assistant that pressing the Capture button causes all of the capture-able white stones (and some of the non-capture-able ones) to turn pink.**

**Part 5: Capturing stones, step 4**

Replace the for-loop in the above step with a while-loop that keeps iterating until an entire pass has been made through the board without any stone being changed. To do this:

* Declare a boolean variable before your outermost loop that you will use to indicate whether a change has been seen in the previous pass through the board.
* The while-loop condition should cause the loop to continue if a change has been seen on the previous pass (i.e., if the boolean variable’s value is true).
* Initialize the boolean variable to true at the time it is declared so that the loop body is executed the first time through.
* As soon as you enter the while-loop, set the value of the boolean variable to false. This will indicate that on the current pass over the board, no change has been seen. (This is trivially true because no square on the board has yet been examined--the pass has not yet started.)
* In the body of the loop, any time a stone is set from WHITE\_IN\_PERIL to WHITE, set the boolean variable's value to be true.

Stop and test your code.

When you've looped through the entire board until no changes have been detected, then all safe stones that had been PINK should have been changed back to WHITE. The only remaining pink stones should be surrounded by black stones.

**checkpoint 5 (20 points): Demonstrate to your instructor or lab assistant that pressing the Capture button causes only the capture-able white stones to turn pink.**

If you have completed all the above checkpoints, you now have a C (75) for this lab.

**Part 6: Capturing stones, step 5**

After the loop for the previous checkpoint, loop through the entire board again, removing any remaining WHITE\_IN\_PERIL stones (i.e., change them to EMPTY).

**checkpoint 6 (5 points): Demonstrate to your instructor or lab assistant that pressing the Capture button causes (only) the capture-able white stones to be removed from the board.**

If you have completed all the above checkpoints, you now have a B (85) for this lab.

**Part 7: Capturing stones, step 6**

Implement steps 1 through 5 (above) for black stones. In other words, copy the code and change the places that deal with WHITE and WHITE\_IN\_PERIL stones to do the same for BLACK and BLACK\_IN\_PERIL stones.

**checkpoint 7 (5 points): Demonstrate to your instructor or lab assistant that pressing the Capture button causes (only) the capture-able black and white stones to be removed from the board.**

If you have completed all the above checkpoints, you now have an A (100) for this lab.

**Extra Credit #1 (up to 10 points)**

Implement a way to change WHITE\_IN\_PERIL to WHITE that checks fewer squares. E.g., pretend that each call to board[someRow][someColumn] takes five minutes, so you want to do that as seldom as possible! There are several ways to do this – any could get you some of these extra credit points.

Similarly, you could also get these extra credit points by implementing a way to change WHITE\_IN\_PERIL to WHITE without needing to loop through the board multiple times via a while loop like you did in checkpoint 5.

**Extra Credit #2 (up to 15 points)**

Implement a game (or at least part of a game) using the Go board. Some ideas:

* Snake.
* Checkers.
* Connect Four.