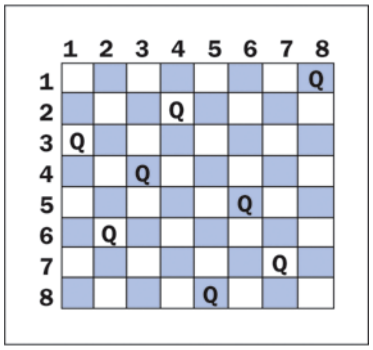
The N Queens problem
(a variation of the Eight Queens problem)

acknowledgements to Robert Glen Martin for this assignment

Perhaps you are familiar with the Eight Queens problem, which consists of trying to place eight queens on a chessboard (which is 8 by 8 squares) in such a manner that none of the queens can attack each other. According to the rules of chess, a queen can attack any other piece in the same row, column, or diagonal, so there can be at most one queen in each row, column, or diagonal of the board. It is not obvious that there is a solution, but the diagram at the right shows one of them.



In this assignment you will implement the N Queens problem, which seeks to place *n* queens on an *n* by *n* board in such a way that no queen can attack another queen. Theoretically we could solve for any number of queens on a chessboard of the same size.

To get started, click on the link and then download, extract, and save the following N Queens project files.

a. `NQueens` – the class that implements the solution to the N Queens problem (this is the only class you will modify).
b. `NQueensRunner` – contains the `main` method.
c. `Queen` – represents a queen on the board.
d. `NQueensWorld` – a special purpose world.

1. Compile and run the program as is. You should see a square board with no queens. Look over the code in the `NQueens` class. Pay particular attention to the instance variables and methods. Make sure you understand the method headings so you know what the methods need to do.

2. Many times recursive methods need a starter method to get the recursion started. In this case, the method that starts the recursion is the `solve` method. Complete the `solve` method by calling `placeQueen` to place a queen in the first row (which row is that?). Recall that the *q*th queen is placed in the row *q*, so what will be the simple one line command to place a queen in row 0?

   Then, to test your work so far, temporarily modify the `placeQueen` method to add a queen (using the `addQueen` method) to the first column of the correct row for that queen, and then remove the queen (using the `removeQueen` method). Note that `addQueen` and `removeQueen` require a `Location` object as a parameter. `Location` objects require a row number and a column number. Given row and column numbers as `(r, c)`, you might find it helpful to create a temporary `Location` object to work with as follows:

   ```
   Location loc = new Location(r, c);
   ```

   This way you will be able to easily call `addQueen(loc)` to add a queen and `removeQueen(loc)` to remove a queen from the location in question. (In this first test case, the location row and column numbers will both be zero.)

   Compile and Run the program and then click on the Run button at the bottom of the screen. You should see one queen appear and then disappear from the upper left corner of the board. It is likely that running the program will throw an exception at this stage, so don't be concerned if it does.

   Make sure this is working correctly before proceeding any further.

3. This next step is challenging, so let's start by reviewing the overall concept. The idea is to write a recursive method that continues to place all of the queens until all of them are placed. Every recursive method needs a **base case** (how to know when you're done) and a **recursive step** (where the method calls itself again). We are going to start with placing the first queen in the first (top) row, and then place the next queen in the next row and so on. In this case, the method is going to continue to recursively call itself to place the next queen (q + 1), until we run out of rows to place them in. In other words, if the current row is less than the total number of rows, then the recursive step takes place, otherwise we are done. Therefore, what will be the test for the base case? For an idea, think about how you might test to see if you are still in the middle of an array. Now apply that concept to check if a given row is within the board or not. It is helpful to know that you can ask the board how many rows it contains as follows: `board.getNumRows()`

   Now it's time to start writing the recursive method.

   - ✓ First, delete all of the code in the body of the `placeQueen` method.
   - ✓ In its place write an `if` statement that tests if we are *not* done yet. In other words, is `q` less than `board.getNumRows()`?
   - ✓ If that test is `true` (i.e. not done yet) create a temporary `Location` object and then make the one line recursive call as follows:

     ```
     Location loc = new Location(q, 0);
     if(placeQueen(q+1)) return true;
     ```

     The recursive call will attempt to place the next queen. This way if the recursive call successfully places a queen then it will signify it by returning `true`.
   - ✓ Place a `return false` statement at the very end of the `placeQueen` method (meaning we are all done placing queens). *Make sure your code compiles before proceeding any further.*

   Now, we want to use our recursive method to place a queen (using the `addQueen` method) in column 0 of each row, one-at-a-time. Again, recall that the *q*th queen is placed in row *q*. In other words, the call to `placeQueen(4)` places a queen in row 4. At this stage we are just going to place them all in the first column (column 0) of each row. Since we want them to be placed in order (starting at 0), do you want to make the call to `addQueen(loc)` immediately before or after the recursive step?

   Once you have determined where it should go, then add the code to `placeQueen` that will add a queen in the appropriate location. Compile and run your code to see that the queens appear properly (starting at the top left corner, and going down in a straight line all the way to the bottom of the board). *Make sure this is working correctly before proceeding any further.*

4. In a similar fashion, after all of the queens have been placed, we want to now remove them, one-at-a-time, except in reverse order (last one first). Think about where in the `placeQueen` method you will want to make the call to `removeQueen(loc)`. Will it be immediately before or after the recursive step? When you have determined where it should go, then add the code to `placeQueen` that will remove a queen from the appropriate location.

   Note: For some reason it is common for Java to throw a `NullPointerException` during the execution of this program, and one solution is to slow things down by having the CPU sleep for a few ticks. Insert the following line in the `placeQueens` method immediately after the call to `addQueen`.

   ```
   try {Thread.sleep(200);} catch (Exception e){}
   ```

   Compile and run your code to see that all of the queens appear and then disappear properly. *Make sure this is working correctly before proceeding any further.*

5. We need to have `placeQueen` check the `locationIsOK` method before going ahead and adding a queen to a given location. Immediately after the creation of the temporary `Location` variable `loc`, write an if statement that checks if the `locationIsOK` before doing anything else (use `loc` as the parameter). The current version of `locationIsOK` should not have any effect on the current placement of queens, so that means a queen will still be added to the first column of each row. *Make sure this is working correctly before proceeding any further.*

6. Now we need to deal with placing the queen in a column other than the first one. Immediately after checking if `q` is within `board.getNumRows()`, write a for loop that iterates through each column number (using `board.getNumCols()`). Then modify the temporary `Location` object `loc` to reference row `q` and column `c` (or whatever variable you used in your for loop).

   Next should come the call to see if the `locationIsOK`. The queen should be added to the first location (column) that returns `true`. If for some reason `locationIsOK` should return false for every column in a given row, then `addQueen`, `removeQueen` and the recursive step would not get called for that row, and `placeQueen` should return `false` (because a queen did not get placed). In other words, a `return false` statement should be just outside the end of the for loop (in addition to the final `return false` statement at the end of the method).

   Be sure that your calls to `addQueen` and `removeQueen` now reference the location of the first column where `locationIsOK`, and not hard-coded to just the first column (as they were before). You should now notice that now after all of the queens are placed in the first column of each row, the last queen is removed, and then added to the next column to the right, and so on. If left alone, the program will now display each of the N! unique ways to display N queens on a board. You probably don't have time to watch them all, but you should see that it appears to be working correctly for the bottom rows before stopping the program. *Make sure this is working correctly before proceeding any further.*

7. The final steps involve modifying the `locationIsOK` method to return `false` if any other queens are in a position to attack the location where we are considering putting another queen. Begin by checking to see if any queens have already been placed in the same column as the location parameter. In other words, loop through each prior row to see if there is a queen in the same column. What method will you use to get the object at a given Location? You just want to make sure that indeed nothing is there (what value is nothing in this case?). If there is something there, then the method of course will return `false`. When you have this working you should see the queens initially fill in the diagonal from location (0, 0) to location (n-1, n-1). *Make sure this is working correctly before proceeding any further.*

8. The last thing we need `locationIsOK` to check for is any queens that may be on a diagonal from the given location. The routine for checking up and to the right has already been written for you. Notice that it uses a complex `for` loop that uses two counter variables. Using this as an example, write the routine that will check the other diagonal (locations up and to the left of the given location). Note that we only have to check in an upwards direction as there are no queens (yet) below the given location.

   If working properly, `locationIsOK` should now only return `true` if and only if there are no other queens in the same column, or on either 45 degree diagonal line from the given location.

9. Finally, in order to properly stop the recursion after we have found a solution, you will need to change the base case return statement (i.e the final return statement in `placeQueen`) from `false` to `true`.

10. If everything is working correctly, your program should now continue to run until it places all of the queens properly. It may take several minutes for the program to find a solution, but it should eventually. Verify that it does so, and then demonstrate your work for me to observe.