# Exhaustive Search and Backtracking

Continued from yesterday

Program output:

```
>java RealQueen 5
SOLUTION:  1 3 5 2 4
SOLUTION:  1 4 2 5 3
SOLUTION:  2 4 1 3 5
SOLUTION:  2 5 3 1 4
SOLUTION:  3 1 4 2 5
SOLUTION:  3 5 2 4 1
SOLUTION:  4 1 3 5 2
SOLUTION:  4 2 5 3 1
SOLUTION:  5 2 4 1 3
SOLUTION:  5 3 1 4 2
```

Check out Queens from SVN

## Top-level solution code (I removed the solutionCount code to make this slide simpler)

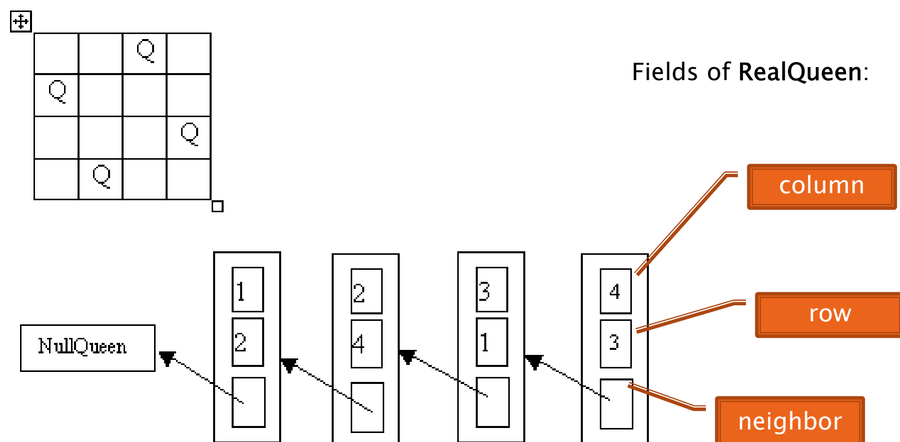
```
static int findSolutions(int size) {
    // Set up the board: a linked list of queens, ends with a NullQueen
    Queen previousQueen = new NullQueen();
    for (int i = 1; i <= size; i++) {
        Queen newQueen = new RealQueen(previousQueen, i, size);
        previousQueen = newQueen;
    }

    // Look for the solutions:
    final Queen lastQueen = previousQueen;
    if (lastQueen.findFirst()) {
        System.out.println("SOLUTION: " + lastQueen);

        while (lastQueen.findNext()) {
            System.out.println("SOLUTION: " + lastQueen);
        }
    }
}
```

## The Linked List of Queen Objects

- Board configuration represented by a linked list of **Queen** objects




Designed by Timothy Budd
http://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/slides/chap06/java.htm

## Queen Interface Methods

- **`findFirst()`**
- **`findNext()`**
- **`canAttack(int row, int col)`**

- `Already Implemented by NullQueen`
  `(does this implementation make sense to you?)`

Your job (15 points extra credit if you do it before the end of class):

Understand the job of each of these methods.

Javadoc from the Queen interface can help

Fill in the (recursive) details in the RealQueen class

Debug

Submit to dropbox on Moodle by the end of your class period.

More details on next slides

## In-class implementation exercise:

- For 15 extra-credit HW points, submit a solution by the end of your class period today.
- Submit a ZIP file that contains all of the Java source files.
- If you work with a partner, one of you should submit it; include both of your usernames in the name of your ZIP file.
- See the next slides for algorithm details.

## Outline of the algorithm

- Each queen sends messages directly to its immediate neighbor to the left (and recursively to all of its left neighbors)
- Return value provides information concerning **all** of the left neighbors:
- Example: `neighbor.canAttack(currentRow, col)`
  - Message goes to the immediate neighbor, but the real question to be answered by this call is
  - "Hey, neighbors, can any of you attack me if I place myself on this square of the board?"

## More algorithm outline

1. Queen asks its neighbors to find the first position in which none of them attack each other
   - Found? Then queen tries to position itself so that it cannot be attacked.
2. If the rightmost queen is successful, then a solution has been found! The queens cooperate in recording it.
3. Otherwise, the queen asks its neighbors to find the next position in which they do not attack each other
4. When the queens get to the point where there is no next non-attacking position, all solutions have been found and the algorithm terminates

And recursion does its magic!