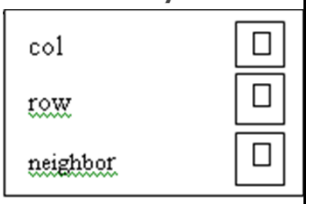
Non-attacking Queens problem

Object-oriented Solution
Cooperating **Queen** objects

Non-attacking chess queens problem

- In how many ways can *N* chess queens be placed on an *N*x*N* grid, so that none of the queens can attack any other queen?
  - I.e. no two queens on the same row, same column, or same diagonal.

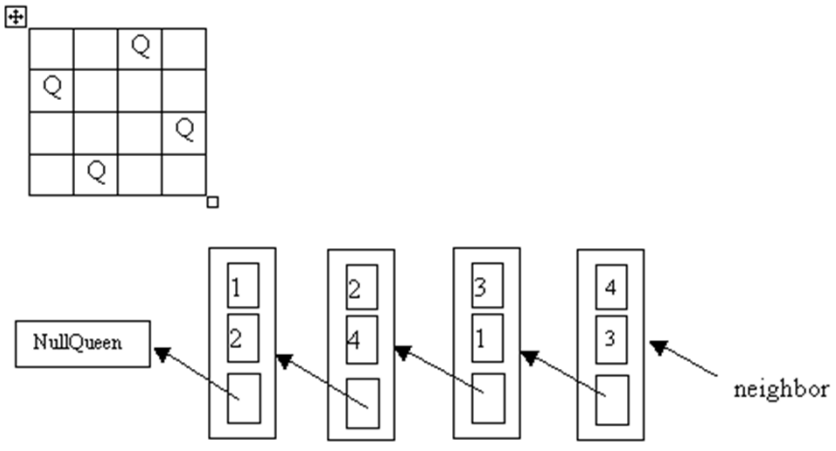
Object-oriented Solution by Timothy Budd



- The queen in each column is represented by a **RealQueen** object.
- Each **RealQueen** knows its column number (fixed), row number (varies), and the queen that is its neighbor to the left (fixed).
- The neighbor of the **RealQueen** in column 1 is a special **NullQueen** object
  - whose purpose is to simplify the code for the **RealQueen** methods
  - by eliminating the need for *if*s that check to see whether a Queen has a neighbor (every **RealQueen** does have a non-null neighbor).

The Linked List of Queen Objects

A board position is represented as a linked list of `Queen` objects:



Basic approach

- Each queen sends messages directly to its immediate neighbor to the left, and indirectly to *all* of its left neighbors.
- The return value that this queen receives after sending a message always provides information concerning *all* of the left neighbors.
  For example, when a queen executes
  `neighbor.canAttack(currentrow, col);`
  The 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?"
- Calls to `findFirst()` and `findNext()` have a similar protocol.

Algorithm outline 1/2

- Build the list of queens. Imagine that they have been assigned columns but are not yet on the board.
- Rightmost queen asks its neighbors (in the columns to its left) to find the first position in which none of them attack each other.
  - If they can find such a position, this queen tries to position itself so that it does not attack any of its neighbors.
  - If the rightmost queen (head of the linked list of queens) is successful at this, the first solution has been found, and the queens cooperate to record it.

Algorithm outline 2/2

- The Rightmost queen sees if there are other rows in which it does not attack any other queens.
  - If so, record them.
  - Otherwise, the queen asks its neighbors to find the next position in which they do not attack each other, and so on.
- When the queens get to the point where there is no next non-attacking position, all solutions have been found and the algorithm terminates.

Demonstrate (for N=4 case)

Main method

```
public static void main(String args[]) {
                          // set up the board
  if (args.length == 0)
    MAXROWS = 8;
  else
    MAXROWS = Integer.parseInt(args[0]);

  Queen neighbor = new NullQueen();

  for (int i=1; i<=MAXROWS; i++)      {
      Queen newQueen = new RealQueen(neighbor, i);
      neighbor = newQueen;
  }
                          // Now look for the solutions:
  if (neighbor.findFirst())      {
    System.out.println("SOLUTION:  " + neighbor);

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

Initialize by making a linked list of queens, with a NullQueen at the end.

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
```

Some Queens Numbers

```
addiator 7:46am > java RealQueen
6:  4 solutions found, 1 milliseconds.     1708 canAttack calls
7:  40 solutions found, 3 milliseconds.     8055 canAttack calls
8:  92 solutions found, 10 milliseconds.     40282 canAttack calls
9:  352 solutions found, 4 milliseconds.     206451 canAttack calls
10:  724 solutions found, 17 milliseconds.     1091856 canAttack calls
11:  2680 solutions found, 96 milliseconds.     6180871 canAttack calls
12:  14200 solutions found, 590 milliseconds.     37512342 canAttack calls
13:  73712 solutions found, 3461 milliseconds.     239507629 canAttack calls
14:  365596 solutions found, 22610 milliseconds.     1623486774 canAttack calls
15:  2279184 solutions found, 175544 milliseconds.     11621556251 canAttack calls
```

Check It Out!

- Work with your SlidingBlocks partner
- Check out the **Queens** project from your SVN repository
- Look at the code together and try to figure out:
  - main()
  - Queen interface
  - NullQueen class
  - RealQueen class
    - What should findNext() do?

Queen Interface

```
public interface Queen
{

  // in the descriptions of these methods, "its neighbors" means
  // all queens "to the left" of this.  "neighbor' means the
  // immediate neighbor (if any).

  public boolean findFirst();
  // finds the first position for this queen and its neighbors
  // such that none of them attack each other.  Returns true if
  // it finds such a position, false otherwise.

  public boolean findNext();
  // moves this queen to its next legal position (in which it doesn't
  // attack any neighbors).  If no such position is found,
  // it asks its first neighbor to move, and then starts over at row 1.
  // If neighbors have no untried positions, returns false.  Otherwise
  // returns true.

  public boolean canAttack(int row, int col);
  // returns true if this queen (or its neighbors) can attack
  // the given row and column, false otherwise.

  public String toString();
  // A string representing the rows in which this queen and its neighbors.
  // are placed.

}
```

NullQueen Class

```
public class NullQueen implements Queen
{
  // The NullQueen represents the end-of-the-line, off-the-board,
  // no-real-queen-so-nothing-to attack, only-one-choice.

  public boolean findFirst() {
    // There is no queen to position.
      return true;
  }

  public boolean findNext() {
  // There is no alternate position.  If the null queen is
  // asked to move, we have searched all board configurations.
    return false;
  }

  public boolean canAttack(int row, int col) {
  // A null queen doesn't attack anything.
      return false;
  }

  public String toString() {
    return "";
  }
}
```

RealQueen Class

- Some methods are on the next slide.
- You will write some other methods.

```
private  Queen neighbor;     // next queen to the left.
private int currentRow;      // where am I now?
private static int MAXROWS; // How big is the board?
private int column;          // What's my (permanent) column?

RealQueen(Queen neighbor, int col)
// Constructor function.  These characteristics, once initialized,
//    never change.
{
  this.neighbor = neighbor;
  this.column = col;
}

public boolean findFirst()
//Find the first row in which to place myself legally. If none
// exists, ask my neighbor to move.
{
  currentRow = 1;
  if (neighbor.findFirst())
    return testOrAdvance();
  else
    return false;
}

private boolean testOrAdvance()
// If this is a legal row for me, say so. If not, try the next row.
{
  if (neighbor.canAttack(currentRow, column))
    return  findNext();
  return true;
}
```

What should `findNext` do?

Exercise (with a partner)

- Add your names at the top of the RealQueens.java file.
- Write the remaining three methods (stubs are provided). You should not have to change any of the instance methods that are already complete.
- Test for a small value of MAXROWS to make sure that your code works.
- Add a "solution counter" to main().
- After finding all solutions, print the count.
- Once you are sure the program is working, you may want to add two if statements in main() so as to only print each individual solution if MAXROWS <= 6. Thus, you simply print the solution count for large values of MAXROWS.
- Test your code for various values of MAXROWS. How high a value of MAXROWS can your program do in a reasonable time? Can you use System.currentTimeInMilliseconds() to estimate how long it takes to find the solutions for each value of MAXROWS, and try to get a big-Oh estimate for the running time?
- When you are done:
  - Commit to your repository (just one of the partners needs to commit it);