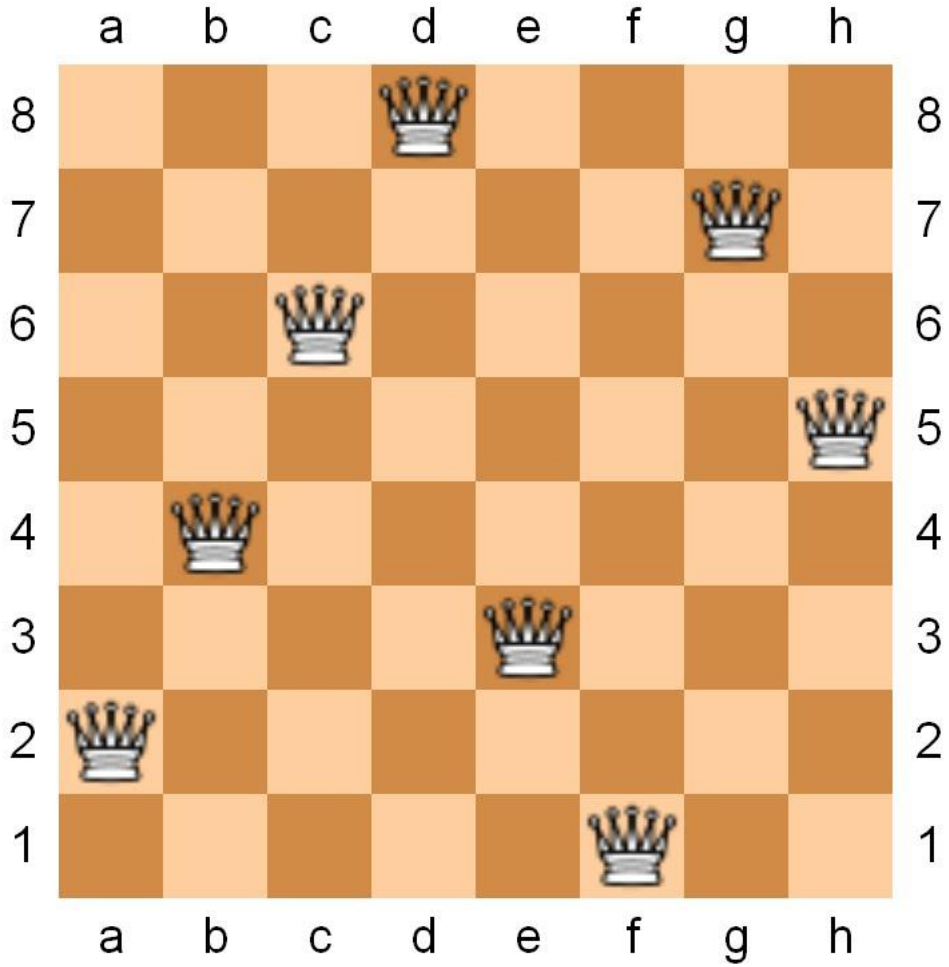
Recursive Backtracking

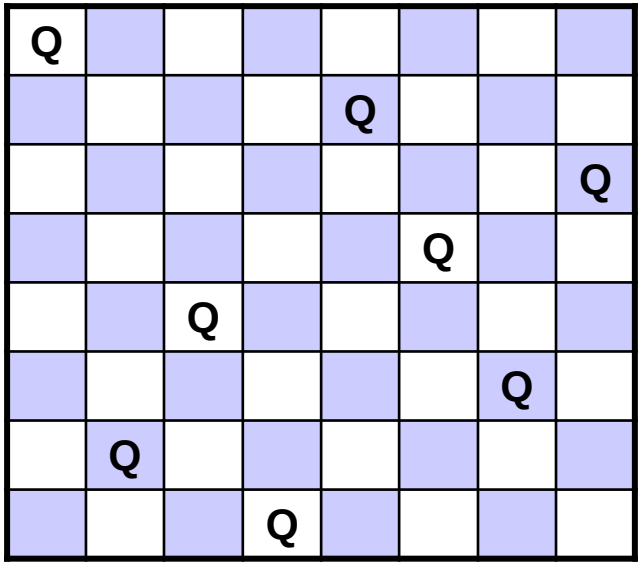


The n-Queens Problem

The n-Queens Problem

- **Goal:** to place n queens on an n x n chessboard so that no two queens occupy:
  - the same row
  - the same column
  - the same diagonal.

- Sample solution for n = 8:

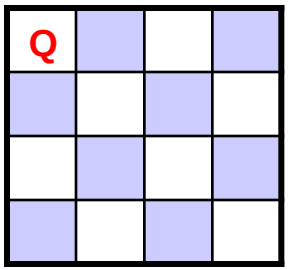


- This problem can be solved using a technique called *recursive backtracking*.

Recursive Strategy for n-Queens

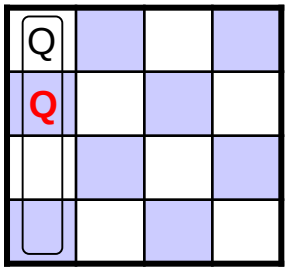
- Use a recursive method `findSolution(row)` that attempts to place a queen in the specified `row`:
  - consider one column at a time, looking for a "safe" one
  - if we find a safe column, place the queen there, and *make a recursive call* to move onto the next row
  - if we can't find one, *backtrack* by returning from the call, and try to find another safe column in the previous row.

- Example:
  - row 0:

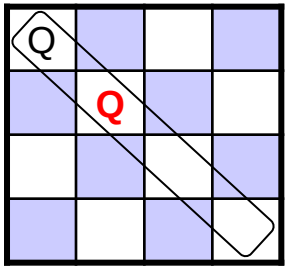


col 0: safe

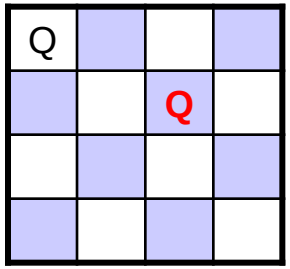
  - row 1:



col 0: same col



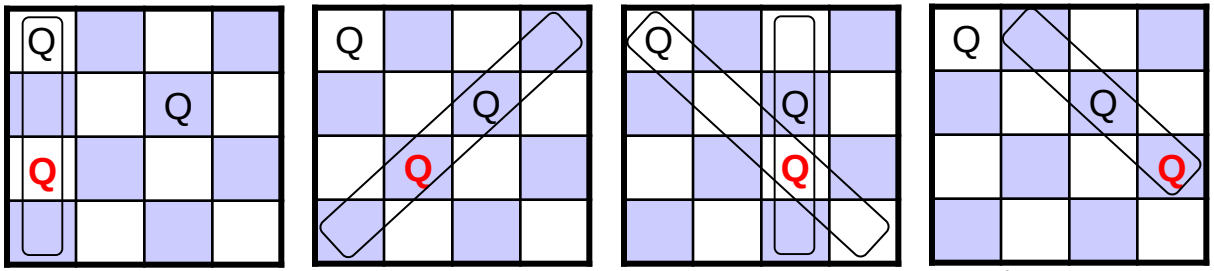
col 1: same diag



col 2: safe

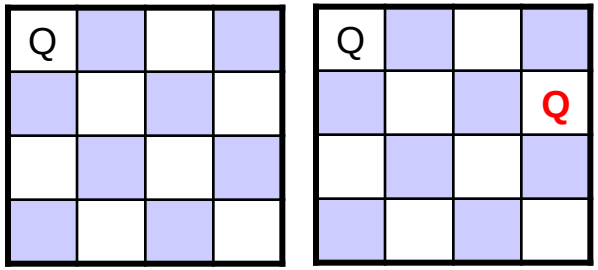
4-Queens Example (cont.)

- row 2:



col 0: same col *col 1: same diag* *col 2: same col/diag* *col 3: same diag*

- We've run out of columns in row 2!
- *Backtrack* to row 1 by returning from the recursive call.
  - pick up where we left off
  - we had already tried columns 0-2, so now we try column 3:

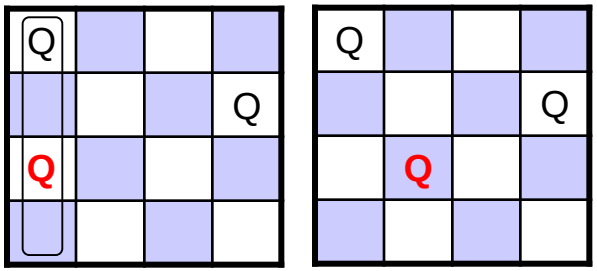


we left off in col 2 *try col 3: safe*

- Continue the recursion as before.

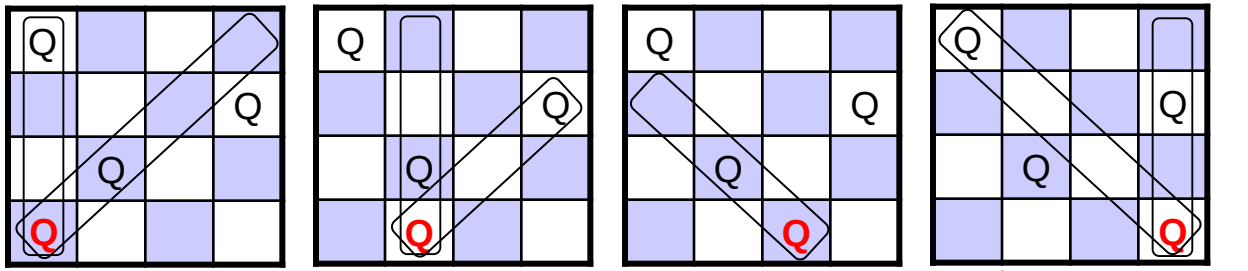
4-Queens Example (cont.)

- row 2:



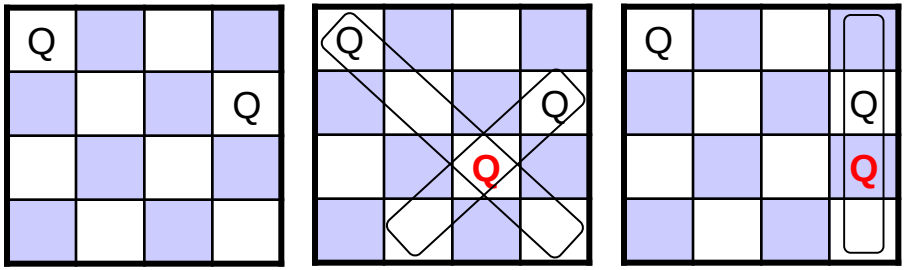
col 0: same col *col 1: safe*

- row 3:



col 0: same col/diag *col 1: same col/diag* *col 2: same diag* *col 3: same col/diag*

- Backtrack to row 2:



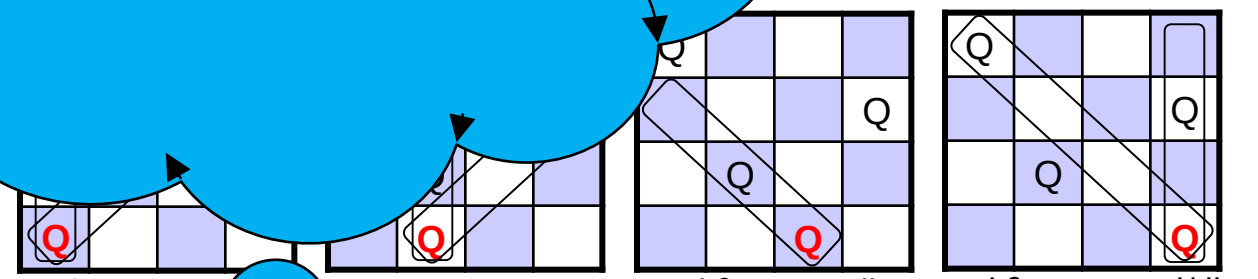
we left off in col 1 *col 2: same diag* *col 3: same col*

- Backtrack to row 1.

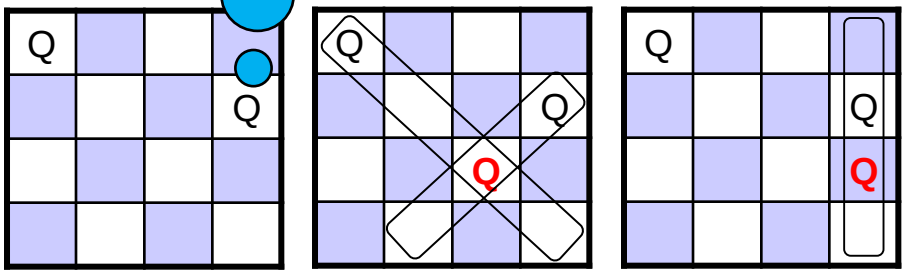
4-Queens Example (cont.)

- row 2:
- r
- Backtrack to row 2:
- Backtrack to row 1.

But we have placed our queen on the last column of row 1 so, we ...



col 0: same co *col 1: same col/diag* *col 2: same diag* *col 3: same col/diag*

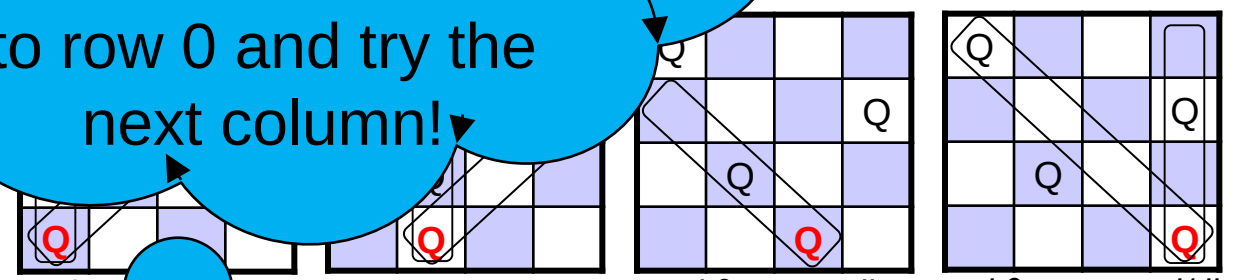


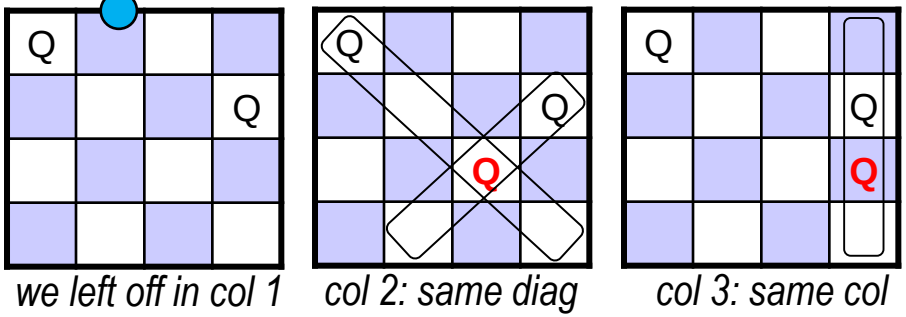
we left off in col 1 *col 2: same diag* *col 3: same col*

4-Queens Example (cont.)

- row 2:
- r
- Backtrack to row 2:
- Backtrack to row 1.

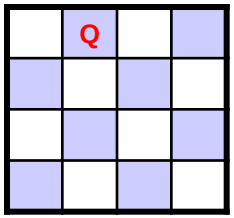
But we have placed our queen on the last column of row 1 so, we have to *backtrack* to row 0 and try the next column!



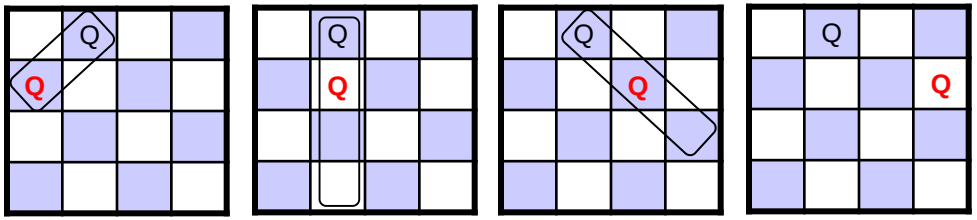


4-Queens Example (cont.)

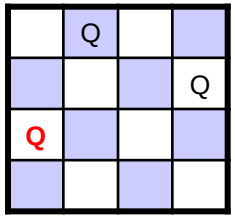
- row 0:



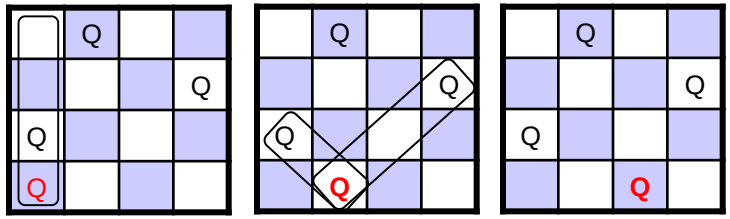
- row 1:



- row 2:



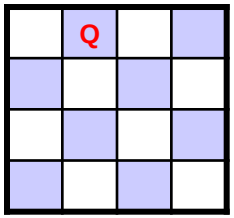
- row 3:



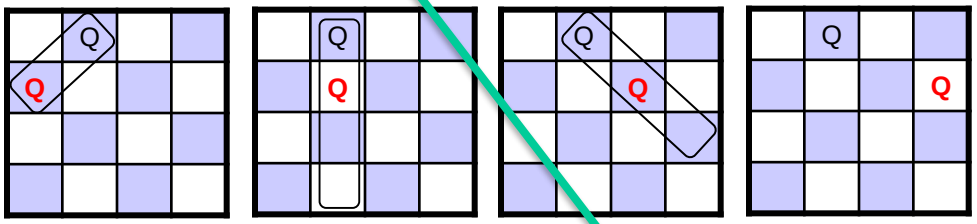
A solution!

4-Queens Example (cont.)

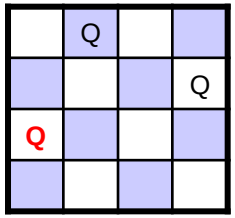
- row 0:



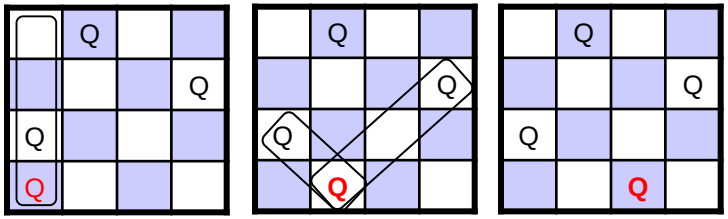
- row 1:



- row 2:



- row 3:

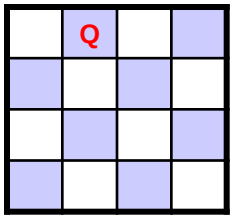


A solution!

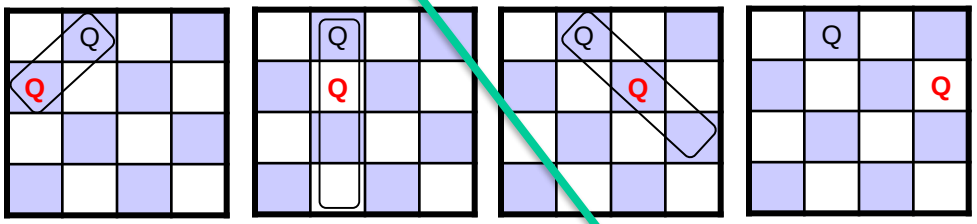
Note that to find the solution we were forced to go back and try a new different option in row 0.

4-Queens Example (cont.)

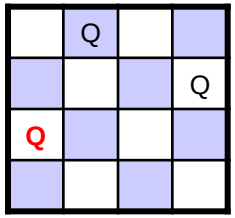
- row 0:



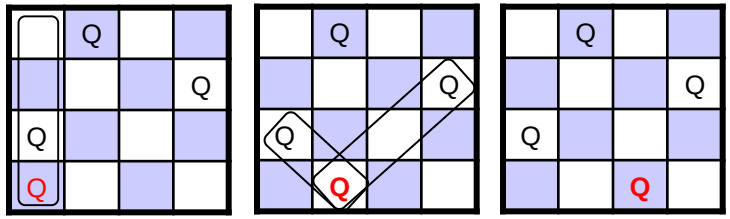
- row 1:



- row 2:



- row 3:



A solution!

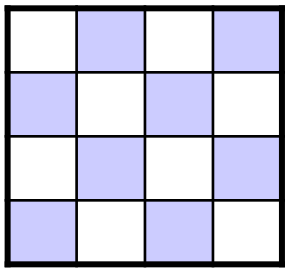
Power of the Stack!

A Blueprint Class for an N-Queens Solver

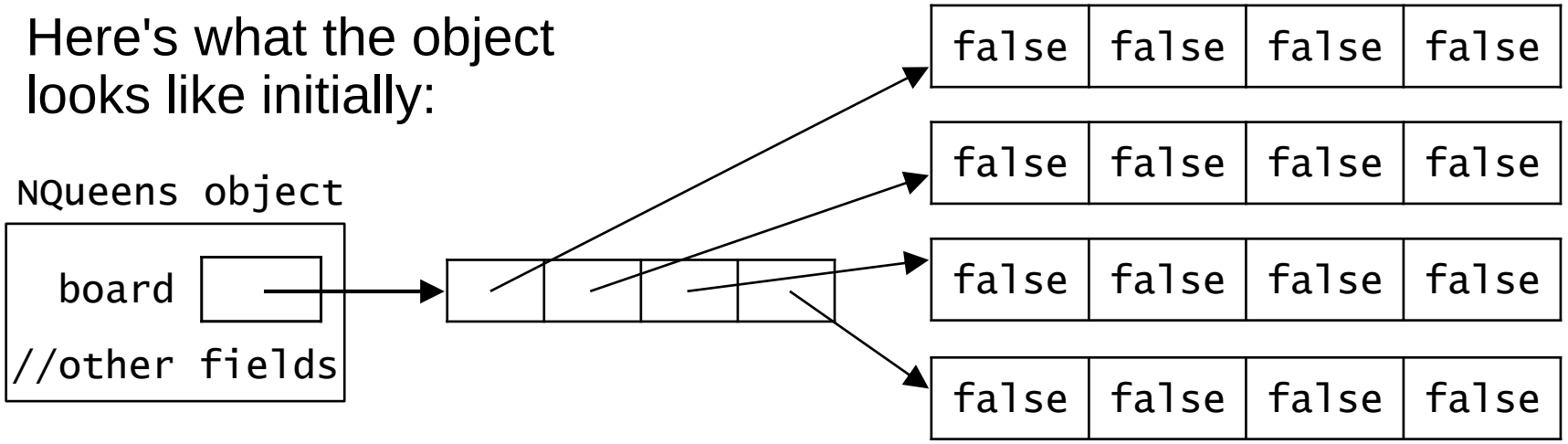
```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    ...
```



- Here's what the object looks like initially:

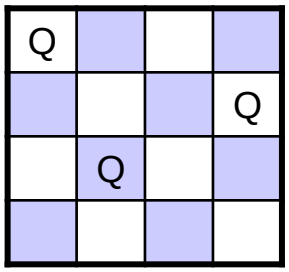


A Blueprint Class for an N-Queens Solver

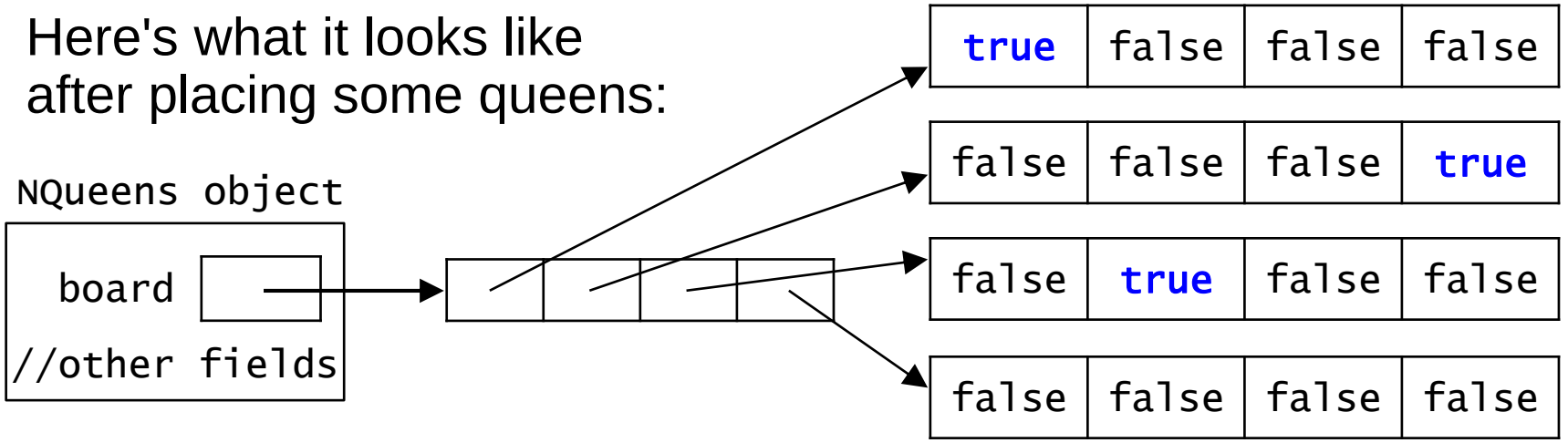
```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
        // modify other fields here...
    }
```



- Here's what it looks like after placing some queens:



A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
        // modify other fields here...
    }

    private void removeQueen(int row, int col){
        this.board[row][col] = false;
        // modify other fields here...
    }

    private boolean isSafe(int row, int col) {
        // returns true if [row][col] is "safe", else false
    }

    public boolean findSolution(int row) {
        // see next slide!
    ...
```

private helper methods that will only be called by code within the class.

A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    public boolean findSolution(int row) {
        // see next slide!
    ...
```

private helper methods that will only be called by code within the class.

Making them private means they are only accessible from methods of the class and hidden from outside the class.

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {     // base case
        this.displayBoard();
        return true;
    }
}
```

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {     // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;     // backtrack!
}
```

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {     // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;     // backtrack!
}
```

Note we are kicking off the next recursive call here!

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {     // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;     // backtrack!
}
```

And we continue until the base case is reached or…

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {      // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;      // backtrack!
}
```

we cannot find a safe placement in a subsequent row and we need to try another column in **this** row!

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {      // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;      // backtrack!
}
```

we cannot find a safe placement in a subsequent row and we need to try another column in **this** row!

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {      // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;      // backtrack!
}
```

If we are here we got stuck and backtracked from a subsequent row!

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {      // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;      // backtrack!
}
```

remove our current placement of the queen and try the next column of this row.

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {      // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;      // backtrack!
}
```

remove our current placement of the queen and try the next column of this row.

findSolution() Method

```
public boolean findSolution(int row) {
    if (row == this.board.length) {      // base case
        this.displayBoard();
        return true;
    }

    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);

            // Move onto the next row.
            if (this.findSolution(row + 1)) {
                return true;
            }

            // If we get here, we've backtracked.
            this.removeQueen(row, col);
        }
    }

    return false;    // backtrack!
}
```

We have tried all columns in this row and cannot find a solution, so we need to backtrack to the **prior** row and try all columns again!

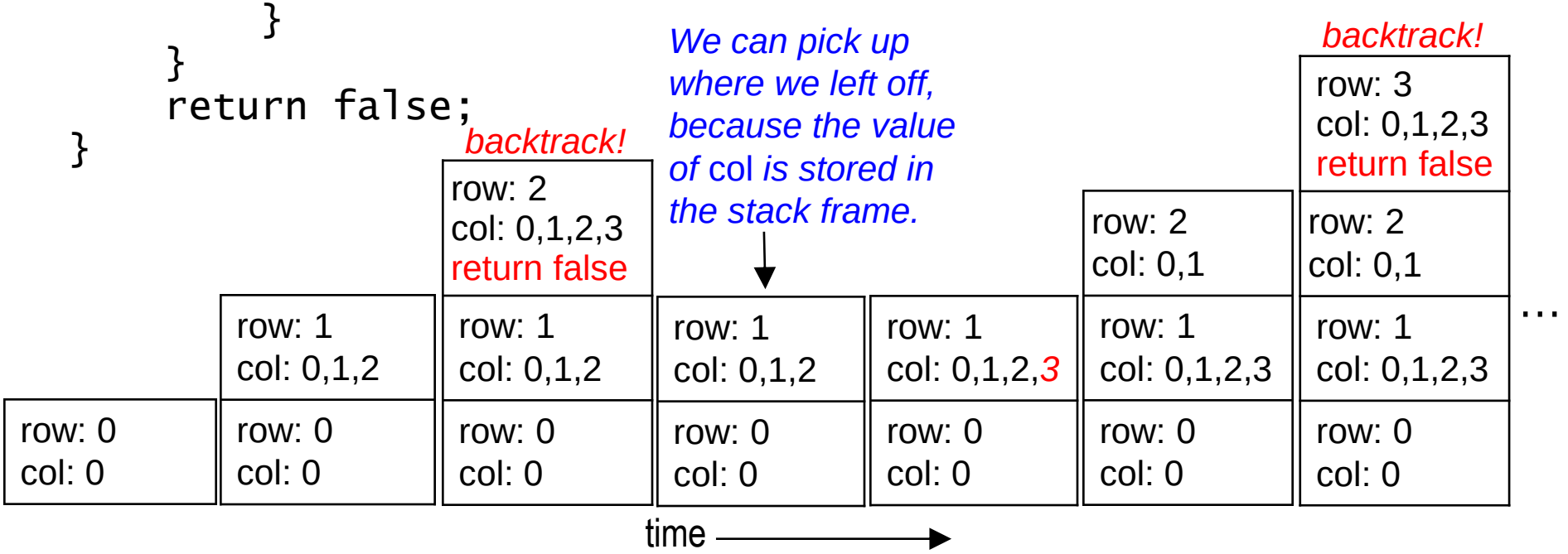
Tracing findSolution()

```
public boolean findSolution(int row) {
    if (row == this.board.length) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```

time ⟶

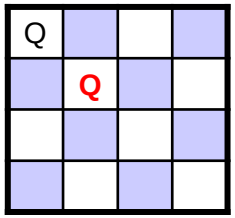
Tracing findSolution()

```
public boolean findSolution(int row) {
    if (row == this.board.length) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```



Recursive Backtracking in General

- Useful for *constraint satisfaction problems*
  - involve assigning values to variables according to a set of constraints
  - n-Queens: variables = Queen's position in each row
    constraints = no two queens in same row/col/diag
  - many others: factory scheduling, room scheduling, etc.

- Backtracking greatly reduces the number of possible value assignments that we consider.
  - ex: 

    this doesn't work, so we don't even consider any of the 16 possible solutions that begin with queens in these two positions!

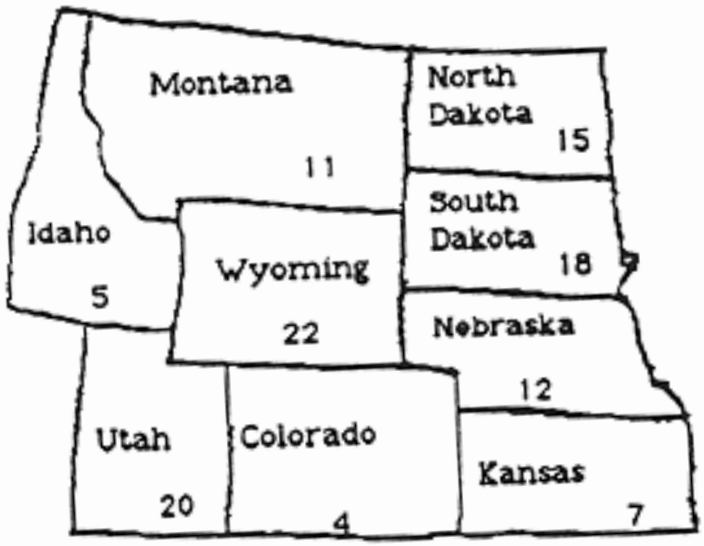
- Using recursion allows us to easily handle an arbitrary number of variables.
  - stores the state of each variable in a separate stack frame

Template for Recursive Backtracking

```
// n is the number of the variable that the current
// call of the method is responsible for
boolean findSolution(int n, possibly other params) {
    if (found a solution) {
        this.displaySolution();
        return true;
    }
    // loop over possible values for the nth variable
    for (val = first to last) {
        if (this.isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolution(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }
    return false;
}
```

Recursive Backtracking II: Map Coloring

- Using just four colors (e.g., red, orange, green, and blue), we want color a map so that no two bordering states or countries have the same color.
- Sample map (numbers show alphabetical order in full list of state names):



- This is another example of a problem that can be solved using recursive backtracking.

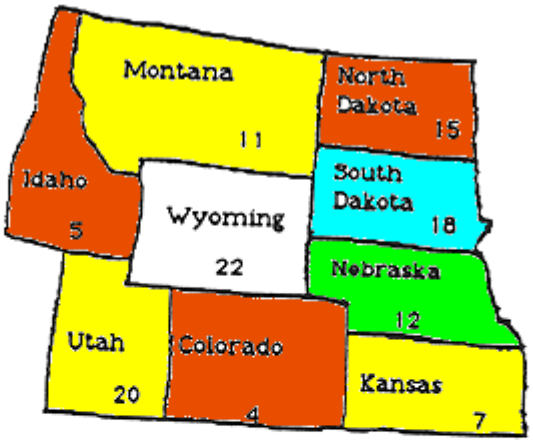
Applying the Template to Map Coloring

```
boolean findSolution(n, perhaps other params) {
    if (found a solution) {
        this.displaySolution();
        return true;
    }
    for (val = first to last) {
        if (this.isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolution(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }
    return false;
}
```

template element	***meaning in map coloring***
n	state number
found a solution	state number > num of last state
val	color (iterates over the four colors)
isValid(val, n)	no bordering states have the color
applyValue(val, n)	apply the color to the state
removeValue(val, n)	**remove the color from the state**

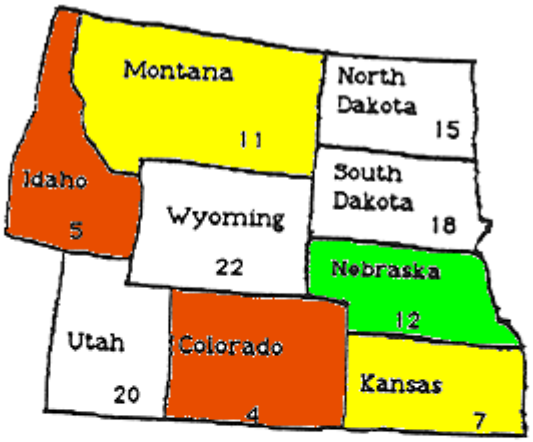
Map Coloring Example

consider the states in alphabetical order. colors = { red, yellow, green, blue }.



We color Colorado through Utah without a problem.

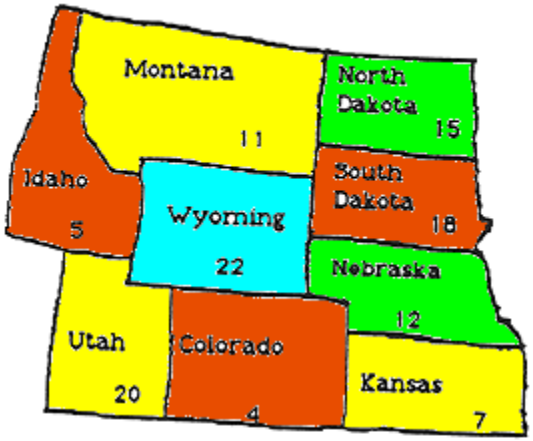
- Colorado: red
- Idaho: red
- Kansas: yellow
- Montana: yellow
- Nebraska: green
- North Dakota: red
- South Dakota: blue
- Utah: yellow



No color works for Wyoming, so we backtrack…

- Color Utah green.
- No color works for Wyoming.
- Backtrack to Utah.
- Color Utah blue.
- No color works for Wyoming.
- Backtrack to Utah.
- No colors left to try for Utah.
- Backtrack to South Dakota.
- No colors left to try for SD.
- Backtrack to North Dakota.

Map Coloring Example (cont.)



Now we can complete the coloring:

North Dakota: green
South Dakota: red
Utah: yellow
Wyoming: blue
done!