Assignment 4

Asterisk Sudoku

Startup

1. Create a new directory `sudoku` and open it in VSCode.
2. Download the file `SudokuSolver.java` and put it in this folder.
3. Open the file and fill in your names and student IDs.

When you're finished with the assignment, **submit `SudokuSolver.java` to Canvas.**

Problem description

Write a solver for *Asterisk Sudoku*. Asterisk Sudoku is a variation on the Sudoku puzzle. A Sudoku puzzle contains a grid of three by three boxes. These boxes contains three by three squares. Thus, the puzzle consists of nine by nine squares.

Initially, some squares are filled with a number. The challenge for the puzzler is to fill in all squares with the numbers one (1) to nine (9) such that each *row*, each *column*, and each *box* contains each number one through nine exactly once.

Asterisk Sudoku has an **extra constraint**: Nine marked squares that should also contain the numbers one through nine. In the example below, these *asterisk squares* are pointed to by > and < characters.

Terminology

- A *grid* is a matrix of nine by nine squares. Each square is either empty, or contains a number between one and nine.
- A *full grid* is a grid without empty squares.
- Each box contains one *asterisk square*:
  - In the center box, the asterisk square is in the center of the box.
  - In each corner box, the asterisk square is the square closest to the center box.
  - In each side box, the asterisk square is the square in the center of the box.
- We call a set of squares *conflict free* if every number between one and nine occurs at most once in that set.
- We call a grid *conflict free* if every row, column, box, and asterisk square is conflict free.
- A grid `h` is an *extension* of a grid `g` if each non-empty square in `g` contains the same number as in grid `h`.
- A *solution* of a grid `g` is a full grid that is conflict free and an extension of `g`.
- A *puzzle* is a grid that is conflict free and has, usually, exactly one solution.

Input

No user input.

Output

If the solver finds exactly one solution, your program prints that solution. Otherwise, your program prints the number of solutions found by the solver.

Example run

If you run your program on the given Sudoku, you should get the following output:

```
+-----------------+
|1 9 5|7 3 8|4 6 2|
|2 6 8|4>1<9|5 7 3|
|3 7>4|5 2 6|9<1 8|
+-----------------+
|7 5 1|3 8 2|6 4 9|
|4>3<9|6>5<1|2>8<7|
```

```
|8 2 6|9 4 7|3 5 1|
+-----------------+
|5 8>2|1 9 4|7<3 6|
|9 4 7|8>6<3|1 2 5|
|6 1 3|2 7 5|8 9 4|
+-----------------+
```

Approach

Use the given template in file `SudokuSolver.java`.

In the `SudokuSolver` class you find an instance variable `grid` of type `int[][]`. This variable represents the puzzle and all intermediate steps. Fill the grid with an initial puzzle. Use the value `0` to represent an empty square. We've already initialized it to a puzzle of medium difficulty for humans that we found on the Internet.

We recommend that you build your solution step-by-step, but you're not required to do so. However, you **must** use the template file as provided. Don't change the names or types of the given methods. You're free to add other methods and constants.

1. Implement the `print` method that prints a grid. For the given grid, the output looks like:

   ```
   +-----------------+
   |  9  |7 3  |4    |
   |     | > < |5    |
   |3  > |    6| <   |
   +-----------------+
   |     |    2|6 4  |
   | > < |6>5<1| > < |
   |    6|9   7|     |
   +-----------------+
   |5 8> |     | <   |
   |9    | > <3|  2 5|
   |6   3|     |8    |
   +-----------------+
   ```

   See also the example run above for an example of a printed solution.

   Test it.

2. In solving the puzzle, we will look for an empty square, try a number and see if it fits. In other words, we will check if that square has a conflict or not. A square has a *conflict* when the number we try to fill in occurs somewhere else in the same row, column, box, or is already part of the asterisk.

   Implement the method `boolean givesConflict(int r, int c, int d)` that determines whether filling in the number `d` in the square with position `r` (for row) and `c` (for column) will give a conflict or not. It is a good idea to separate further into four methods `rowConflict`, `colConflict`, `boxConflict`, and `asteriskConflict`.

3. Implement method `int[] findEmptySquare()` that tries to find the next empty square in reading order. That is, first left to right, then top to bottom. If this method succeeds, it returns the coordinates of the square as an array `{row, column}`, and otherwise as the array `{-1, -1}`. Instead of `{-1, -1}`, you can also return `null`.

   To improve performance, keep track of the last found empty square in instance variables `rEmpty` and `cEmpty`.

   Test this method.

4. Now implement method `void solve()` **and use recursion**. The `solve` method counts the number of solutions and stores the last solution that it found.

   Use a "brute force" strategy: Look for an empty square, fill it in with a number and see if it fits. If it doesn't fit, try the next number. Etc. When we find a number that fits, we fill it in, and continue by tackling the smaller problem of finding the solutions of the grid with the extra filled-in number.

   Keep only one grid in the instance variable `grid`. This instance variable `grid` is shared by all recursive calls.

   When all tries for a certain cell fail, you have to track back and reconsider a previous choice. Recursion will take care of this reconsidering, but you have to re-establish the grid to the state before the call. In other words, clean up the mess behind you!

   Hint When you test this method, adding a statement to print the current grid may be helpful. To avoid an output flood, however, start with a grid that is almost solved, or print the grid only now and then. Remove this superfluous print statement once you're convinced your method is working.

5. Finally, implement the method `solveIt` using the methods you implemented in the previous steps. Look for a solution. If the solver finds exactly one solution, your program prints that solution. Otherwise, your program prints the number of solutions found by the solver.