HW3 Sudoku/DB

Sudoku Part of Assignment by Nick Parlante

Homework 3 makes extensive use of the Java Collection classes while exercising OOP themes -- assembling a large solution out of modular classes, giving each class a good API for its clients, testing, and some GUI coding. You'll also get an opportunity to work with accessing a MySQL database. The whole thing is due at midnight of the evening of Wednesday February 5th.

Part A - Sudoku

For this part of the project, you will build code to solve Sudoku puzzles. Our approach will concentrate on OOP and API design, and give us a chance to start doing some GUI coding. You do not need to be good at Sudoku to build this code. I'm quite slow at them. In fact, this whole project is perhaps cheap revenge against the Sudoku puzzles I've struggled with.

Sudoku is a puzzle where you fill numbers into a grid. The history is that it originated in the Dell puzzle magazine in the 1970's, and later became very popular in Japan, possibly filling the niche that crossword puzzles play in English newspapers, as the Japanese language is not suited to crossword puzzles. Sometime around 2005 it because a worldwide sensation. (See the Wikipedia page for the full story.)

The Sudoku rules are: fill the empty squares in the 9x9 grid so that each square contains a number in the range 1..9. In each row and each column across the grid, the numbers 1..9 much appear just once ("Sudoku" translating roughly as "single"). Likewise, each of the nine 3x3 squares that make up the grid must contain the just numbers 1..9.

Here is an easy Sudoku puzzle. I can solve this one in about 5 minutes. Look at the topmost row. It is only missing 1 and 7. Looking down the columns, you can figure out where the 1 and 7 go in that row. Proceed in that way, looking at rows, columns, and 3x3 squares that are mostly filled in, gradually figuring out the empty squares. A common technique is to write the numbers that might go in a square in small letters at the top of the square, and write in a big number when it's really figured out. Solve this puzzle to get a feel for how the game works (the solution is shown on the next page).

	3	5	2	9		8	6	4
	8	2	4	1		7		3
7	6	4	3	8			9	
2	1	8	7	3	9		4	
			8		4	2	3	
	4	3		5	2	9	7	
4		6	5	7	1			9
3	5	9		2	8	4	1	7
8			9			5	2	6

1	3	5	2	9	7	8	6	4
9	8	2	4	1	6	7	5	3
7	6	4	3	8	5	1	9	2
2	1	8	7	3	9	6	4	5
5	9	7	8	6	4	2	3	1
6	4	3	1	5	2	9	7	8
4	2	6	5	7	1	3	8	9
3	5	9	6	2	8	4	1	7
8	7	1	9	4	3	5	2	6

Sudoku Strategy

There are **many** ways to solve Sudoku. We will use the following approach which is a sort of OOP interpretation of classic recursive backtracking search. Call each square in the puzzle a "spot". We want to do a recursive backtracking search for a solution, assigning numbers to spots to find a combination that works. (If you are rusty with recursion, see the practice recursion problems at javabat.com).

- When assigning a number to a spot, never assign a number that, at that moment, conflicts with the spot's row, column, or square. We are up-front careful about assigning legal numbers to a spot, rather than assigning any number 1..9 and finding the problem later in the recursion. Assume that the initial grid is all legal, and make only legal spot assignments thereafter.

- There are 81 spots in the game. You could try making assignments to the blank spots in any order. However, for our solution, first sort the spots into order by the size of their set of assignable numbers, with the smallest set (most constrained) spots first. Follow that order in the recursive search, assigning the most constrained spots first. Do not re-sort the spots during the search. It works well enough to just sort once at the start and then keep that ordering. The sorting is just a heuristic, but it's easy and effective.

- We will set a max number of solutions of 100 -- if the recursive search gets to a point where it has 100 or more solutions, it can stop looking and just return how many have been found so far.

Sudoku OOP Design

For this project, the starter code has some routine code and data, and the rest of the design is up to you. Your goal is to design classes and APIs so that the solve() method (below) is clean expression of the strategy described above, and the main() and GUI clients are clean.

We will take an OOP approach to the search by treating each spot as its own capable little object. Create a Sudoku class that encapsulates a Sudoku game and give it a "Spot" inner class that represents a single spot in the game. Constant factor efficiency is not a big concern -- we're going for correctness, clarity, and a reasonably smart strategy.

Concentrate on OOP design around the Spot class -- push complexity into the Spot, making things easy for clients of the Spot. For example, the Spot has its own access to the grid (remember, it's an inner class of Sudoku). Consider these two examples of client code:

```
// Bad
grid[spot.getRow()][spot.getCol()] = 6;

// Good
spot.set(6);
```

Your code may be designed however you like, within the following requirements...

- Sudoku(int[][] grid) -- constructor takes the initial grid state, and we assume that the client passes us a legal grid. Empty spots are represented by 0 in the grid. You may assume that the grid is 9x9. You do not need to use int[][] as the internal representation; it's just an input/output format.

- Sudoku(String text) -- takes in puzzle in text form -- 81 numbers. (Starter file provides some parsing code.)

- String toString() -- override toString() to return a String made of 9 lines that shows the rows of the grid, with each number preceded by a space (use the StringBuilder class which replaces the old StringBuffer). (Essentially, the reverse of the text constructor.) For example, here is the toString() of the "medium 5 3" puzzle...

```
 5 3 0 0 7 0 0 0 0
 6 0 0 1 9 5 0 0 0
 0 9 8 0 0 0 0 6 0
 8 0 0 0 6 0 0 0 3
 4 0 0 8 0 3 0 0 1
 7 0 0 0 2 0 0 0 6
 0 6 0 0 0 0 2 8 0
 0 0 0 4 1 9 0 0 5
 0 0 0 0 8 0 0 7 9
```

- int solve() -- tries to solve the puzzle using the strategy described above. Returns the number of solutions and sets the state for getSolutionText() and getElapsed() (below). The original grid of the sudoku should not be changed by the solution (i.e. toString() is still the original problem). The included puzzles have 1 solution each.

- String getSolutionText() -- after a solve, if there was one or more solutions, this is the text form of the first one found (otherwise the empty string). Which solution is found first may vary, depending on quirks of your solve() implementation.

- long getElapsed() -- after a solve, returns the elapsed time spent in the solve measured in milliseconds. See System.currentTimeMillis(). In particular, it's interesting to get visibility into the timing effects of some of your code changes.

- You do not need to write unit tests, although you may want to anyway. (You may make Spot public for the purpose of writing unit tests for it.)

It's fine to use Integer or int or whatever to track the grid state -- whatever you find most convenient. It's good to leverage Set<Integer>/HashSet<Integer> and their built in methods contains(), addAll(), removeAll() to help solve the problem.

Do not pre-compute and store the possible numbers for a spot. Pre-computation worked very well in tetris, but it does not work well here. Each spot assignment changes the possible numbers for 20 other spots. However, in the search, you only care about the possible numbers for the one spot you look at next.

Therefore, doing the computation for all 20 spots ahead of time is a bad strategy -- better to compute the possible numbers for a spot at the moment you need them, based on the grid state at that moment.

Deliverable main()

Your Sudoku main() should be as below, using your code to print the problem and solution for the "hard 3 7" puzzle (contained in Sudoku.java file). As usual, comment out your other extraneous printing before turning in, so we can run your code to see just your clean output.

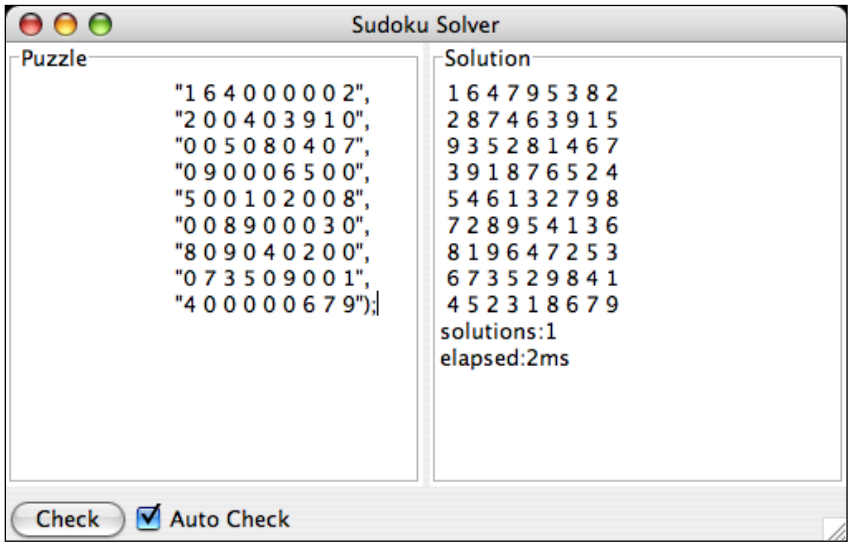
```
public static void main(String[] args) {
    Sudoku sudoku;
    sudoku = new Sudoku(hardGrid);
    System.out.println(sudoku); // print the raw problem

    int count = sudoku.solve();
    System.out.println("solutions:" + count);
    System.out.println("elapsed:" + sudoku.getElapsed() + "ms");
    System.out.println(sudoku.getSolutionText());
}
```

Deliverable GUI

Finally, with the core logic of the Sudoku done, it's time to nest it inside a SudokuFrame to make your hard algorithmic work available to a grateful public. The idea is that someone building a Sudoku puzzle could use this to play around with a puzzle they are building. The included puzzles all have a single solution. However, as you start adding 0's, they get more solutions. For example, changing the 7 of the hard 3 7 puzzle should yield 6 solutions. (This is easiest to play with when you have the GUI working.)

We'll make a simple layout like this: Use a BorderLayout(4,4) -- the "4" is a little spacer between the areas. Create 15x20 JTextArea in the center to hold the "source" puzzle text. Create a second 15x20 JTextArea in the east to hold the results. Create a horizontal box in the south to hold the controls. The code -- textarea.setBorder(new TitledBorder("title")) -- puts the little titled border around any component.



When the Check button is clicked, construct a Sudoku for the text in the left text area and try to solve it:

- If the text is mal-formed in any way so the construction of the Sudoku throws an exception, just write "Parsing problem" in the results text area. (Use a try/catch.)

- Otherwise, after the solve(), if there is at least one solution, write its text into the results text area.

- If there is a solution, write the "solutions:xxx\n" "elapsed:xxx\n" at the end of the results text area.

- Finally, the "Auto" checkbox should make it so that every keystroke add/delete in the text area automatically does a "Check". To implement this, get the "document" object from the text area. The document supports a DocumentListener object which gets notifications whenever the text changes. The Auto feature should work only if the auto checkbox is checked, which it should be by default.

Part B – Database

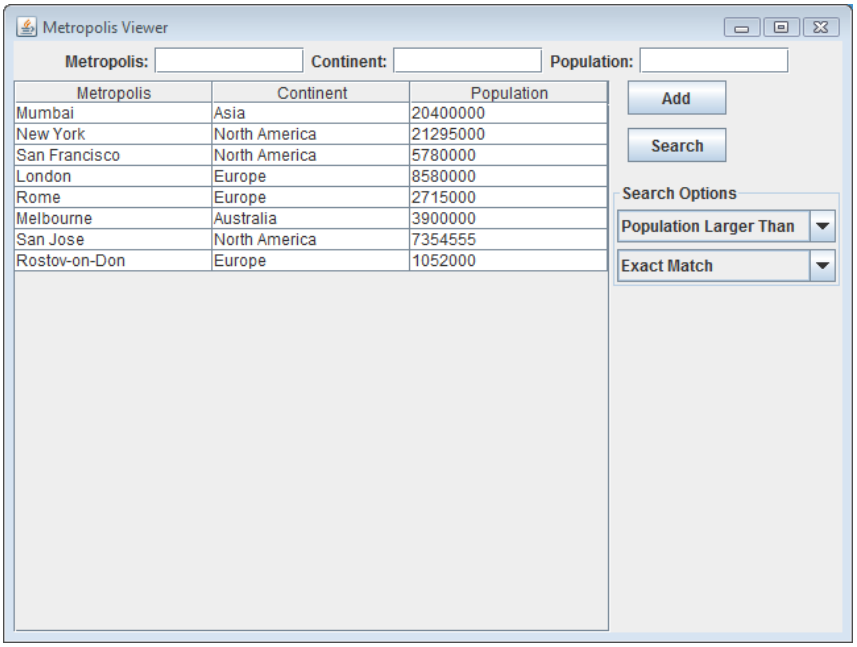
In this part of the assignment you will create a GUI-based (Graphical User Interface) application which will allow a user to access the example metropolises database from the Database handout (which will be out next week). Before working on this assignment, go ahead and **run the MySQL monitor** and load the metropolises database using the metropolises.sql file (this file is included in this assignment's zip file of starter code). Remember if you ever want to reset your database, simply reload the metropolises.sql file using MySQL's SOURCE command.

Warning: Before loading the metropolises.sql file make sure you replace the name of my database in the USE statement on the first line. You should use your own assigned database not my database, otherwise you will get an access error.

The only starter file we are providing for the database part of the assignment is the MyDBInfo.java file, which you need to modify so that it has your MySQL account information in it. **Important:** When you connect to the database, get the account information from the MyDBInfo.java file. When the TAs grade your assignment, they will replace your MyDBInfo.java file with a different MyDBInfo.java file containing their own database info. If you do not get the information from this file and instead hardcode the information directly into your MySQL getConnection calls, the TAs will not be able to properly connect to the grading database and you will lose points.

There are no other starter files associated with this part of the assignment. You'll need to create everything from scratch. You'll discover fewer starter files as we get further in the class. This is to get you ready for professional programming—whether in industry or a research institution there are no starter files unless you are expanding on pre-existing work.

Here is what your application should look like:



The graphical layout of your application does not have to be an exact match with this screenshot, but it should contain the same information shown here and should look reasonably attractive and organized.

The application will provide options allowing the user to either search for metropolises matching particular criteria or to add a new metropolis to the database. The application should include a JTable which displays information gathered from the metropolises database. This table should start out empty and only show data in response to Search or Add requests.

Adding Data

When in add mode, the application will take the Metropolis, Continent, and Population as entered in the text boxes and enter them into the metropolises database. When an add operation is performed, reset the central table to display the newly added entry only. Do not worry about the possibility of duplicate entries.

Searching for Data

The user can enter search criteria in any or all of the Metropolis, Continent, and Population text fields. If no text is entered in a particular text field, we will assume that the user does not care about that particular criterion. If all fields are left blank, display all data in the database.

In addition to the text fields, two pull down menus are provided.

Population Pulldown – This JComboBox allows the user to determine if we are looking for metropolitan areas with populations "larger than" or "smaller than or equal to" the number entered in the Population Text Field. It is ignored if the Population field is left empty.

Match Type Pulldown – This JComboBox allows the user to determine if the Metropolis and Continent text fields contain substrings we are searching for or exact matches. For example if this is set to "Exact Match" and the Continent text field contains "North" no matches will be found, whereas if the pulldown is set to "Partial Match" metropolitan areas in North America will be listed. Note that a "Partial Match" can occur anywhere in the string so a "Partial Match" for "ON" would retrieve "London" and "Rostov-on-Don". MySQL searches are case-insensitive by default, this is fine for our purposes so go ahead and leave this behavior unchanged.

You may assume that the user does not enter any special MySQL regular expression characters into any of the search fields. In other words, assume the user will not enter '%', '_', or any other characters which are used in MySQL regular expressions. We don't care what your program's behavior is if these characters are entered.

Strategy

As previously noted other than the simple MyDBInfo.java file which allows the TAs to easily swap databases for grading, the database part of this assignment comes with no starter files, and you are to build it from scratch. This is good practice, as in industry there are, of course, no starter files and usually no directions provided on how to go about solving a problem. But here is a brief hint to get you started.

The JTable which is at the heart of the application should be constructed using a custom table model, which you can base on AbstractTableModel. Your table model will retrieve its data from the MySQL database. When the user changes the search criteria, the table model will retrieve new data from MySQL and will use the fireTableDataChanged to inform the JTable that it needs to redraw itself.

Your table model will need to provide a means of allowing clients to pass in search criteria. It will also support the standard methods that are usually overridden in a subclass of AbstractTableModel (see the official documentation for AbstractTableModel). In addition you may want to use the table model to add data to the database.

Note: Do not simply load the entire contents of the database when the program first starts, store the data locally, and then simply search through the data that you downloaded in response to user requests. This defeats the use of a database and will not work if the database were to change while your program is running (something that very commonly occurs when working with real databases). You need to re-retrieve data from the actual database each time the user requests a search.

Documentation

In industry documentation is very important. To give you a bit of practice, create documentation for the class that you've developed based on AbstractTableModel (this is the only class you need to create documentation for, for this assignment). Please write documentation comments for each method, and include descriptions of parameters, and (if appropriate) return values. For methods which are overrides to AbstractTableModel methods, you may copy the descriptions from the official Java Swing documentation.

Enter comments in your Java source code, and then export it to HTML using the JavaDoc tool. Comments preceded by /** instead of /* will be picked up by the JavaDoc tool. For descriptions on individual parameters, use the @ followed by the name of the parameter and then your description. For return values use @return followed by a description. Consider for example the following comments before the textToGrid method:

```
/**
 * Given a single string containing 81 numbers, returns a 9x9 grid.
 * Skips all the non-numbers in the text.
 * (provided utility)
 * @param text string of 81 numbers
 * @return grid
 */
public static int[][] textToGrid(String text) {
      …
}
```

This will generate the following documentation:

textToGrid

```
public static int[][] textToGrid(java.lang.String text)
```

Given a single string containing 81 numbers, returns a 9x9 grid. Skips all the non-numbers in the text. (provided utility)

Parameters:

`text` - string of 81 numbers

Returns:

grid

You can find additional examples in Sudoku.java.

Eclipse can run JavaDoc by selecting a class and then right-mouse clicking and choosing Export. JavaDoc will show up under the "Java" category of the Export Dialog box. The first time you run JavaDoc from Eclipse, you will need to show Eclipse where the JavaDoc executable is. It should be in your Java folder in the JDK binary directory. On my computer it is in:

```
C:\Program Files\Java\jdk1.6.0_05\bin
```

If you can't find a JDK directory you may need to download the JDK from[1]

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Database Administration

Remove your password from the source code when turning in your files—just leave an empty string "" where the password should go. We will be using a different database account for testing purposes, and will replace your account name, password, and database name with our own.

If you've correctly used the MyDBInfo.java file, you can actually just not submit this file, instead, but having the file present may help the TAs debug for any students who have not properly connected to the database.

[1] The JDK is the Java Development Kit. It includes a variety of tools to help Java developers.