Homework #2: Did you solve that sudoku correctly?

Due Thursday, September 13th at 11:59 p.m

Sudoku (originally called Number Place) is a logic-based, combinatorial number-placement puzzle. The player is given a 9×9 grid partially filled with digits 1 through 9. The objective is to fill in the rest of the grid so that each column, each row, and each of nine 3×3 subgrids that constitute the grid contains all of the digits 1 through 9.

A sample sudoku puzzle (left) and its solution (right) are[1]:

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

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

In this assignment, you will first implement a class representing a sudoku puzzle grid. For sudoku and several of its variations, you will then write a program that, given a sudoku puzzle and a proposed solution to it, verifies that the proposed solution is a correct solution.

Your solution to this homework should maximize code reuse and be extensible so that new puzzle verifiers could easily be added in the future. To achieve these design goals, you must appropriately define and implement Java interfaces. Additionally, you must write unit tests to check the correctness of your solution. Overall, your goals for this homework are to:

- Design software for extensibility and code reuse.
- Understand and apply the concepts of polymorphism, information hiding, and writing contracts, including an appropriate use of Java interfaces.
- Design and implement software based on your interpretation of informal specifications.
- Write unit tests and automate builds and tests using JUnit, Gradle and Travis-CI.

[1] From http://www.sudokuessentials.com/easy_sudoku.html.

- Write tests based on a functional specification, with good testing practices and style.
- Understand the benefits and limitations of code coverage metrics and interpret the results of coverage metrics.

Part 1: Representing a sudoku puzzle grid

Create a class representing a sudoku puzzle grid. To design this class, you must decide both its internal representation and its external API: its constructors and other common operations that client code would want from the grid as it represents a sudoku puzzle.

In `src/main/resources`, we have provided a set of valid sudoku puzzles. You should test your implementation with both valid and invalid puzzle instances. Your solution should define (and implement) an appropriate behavior if client code attempts to use the grid to represent an invalid puzzle. In our sample inputs, we use 0 to denote an empty cell.

Part 2: Verifying the correctness of a sudoku solution

Use your sudoku grid class from Part 1 to implement solution verifiers for sudoku and the three sudoku variations we describe below. Given a sudoku puzzle grid and a proposed solution to it, a solution verifier must check whether the proposed solution is indeed a correct solution.

Your design should achieve as much code reuse as possible among your four solution verifiers (one verifier per sudoko variant), and also be extensible for new puzzle types. You must demonstrate correct use of Java interfaces, defining and implementing interfaces as appropriate for this problem. When you are done, your solution should consist of up to four programs (one per variant) that read a sudoku problem from an input file, read a proposed solution from a file, and check whether the proposed solution is a correct solution. (A single program for all four variants is fine, but not necessary, to complete this assignment.)

Below are descriptions of three sudoku variations. You must write a solution verifier for sudoku and each of these variations.

Sudoku X

In addition to the constraints for the basic sudoku, sudoku X requires that both diagonals also contain the digits 1 through 9. A sample sudoku X puzzle (left) and its solution (right) are below.[2] We highlight the extra constraints with gray cells:

[2] From http://www.sudoku-space.com.

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

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

Hypersudoku

In addition to the constraints for the basic sudoku, hypersudoku (also known as windoku) requires that four additional 3×3 squares must also contain digits 1 through 9. A sample hypersudoku puzzle (left) and its solution (right) are below.[3] We highlight the four additional squares (the extra constraints) with gray cells:

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

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

Asterisk sudoku

Finally, in addition to the constraints for the basic sudoku, asterisk sudoku requires that nine specific cells that form an asterisk (`r2c5` (row 2, column 5), `r3c3`, `r3c7`, `r5c2`, `r5c5`, `r5c8`, `r7c3`, `r7c7`, and `r8c5`) must also contain the digits 1 through 9.

[3] From http://www.sudoku-space.com.

A sample asterisk sudoku puzzle (left) and its solution (right) are below.[4] We highlight the extra constraints with gray cells:

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

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

Testing your implementation

You must test your solution using JUnit tests. Notably, your testing should not be limited to the inputs we provide, and you are expected to write more tests to thoroughly test your solution. A good test suite usually consists of many independent tests per method being tested, with each test checking some specific behavior or properties of your solution. We recommend that you see the "common strategies" from our testing lecture to develop test cases for your solution. Although your goal is not just to achieve high test coverage, a good test suite will likely achieve nearly 100% line coverage, excluding the test code itself.

Evaluation

Overall this homework is worth 100 points. We will grade your work approximately as follows:

- Correctly applying the concepts of polymorphism and information hiding: 20 points
- Following best practices for Java and compatibility with our informal specification: 40 points
- Unit testing, including coverage and compliance with best practices: 30 points
- Documentation and style: 10 points

[4] From http://www.sudoku-space.com.

Hints

- Use the `Scanner` class to read our sample input files.
- We recommend that you start writing tests for your solution as you complete your solution. Do not delay writing unit tests until after your implementation is complete. It is far easier to test (and find any bugs in) the first part of your implementation before that part is used. If you discover bugs in your own implementation, it is good practice to write a bug report and fix the bug with a separate commit.
- We will assess your line coverage with the Jacoco reports that can be generated with `gradle jacocoTestReport`. The reports can be found in `build/reports/coverage`. You might want to use the IDE integration of coverage while you write your tests.