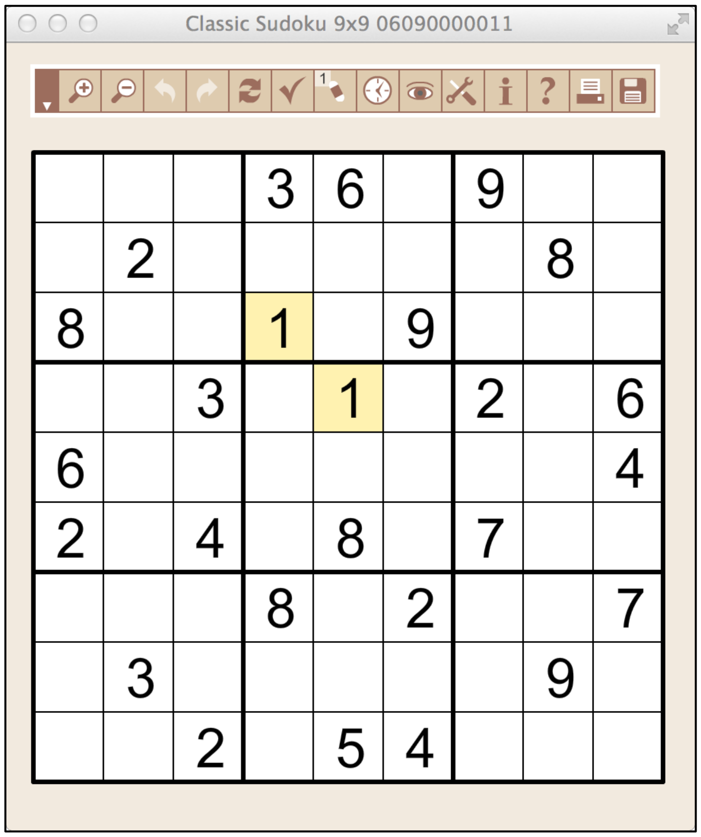
CS 46B
Fall 2017
Homework 6: Sudoku



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

Due: 11:59PM, November 6.

INTRODUCTION: Sudoku is just a puzzle, but the backtracking technique for solving it is used in many important application domains. To solve a Suduku without backtracking, you could generate every possible solution, then evaluate all of them and collect the legal ones. But there are 9^81 ways to fill in a Sudoku grid. 9^81 is ~= 2*10^77. If you could evaluate a billion grids per second, it would take 2*10^68 seconds. The universe is less than 10^18 seconds old. For this homework, you will learn the power of backtracking by writing a Sudoku solver that finds answers in just a few seconds.

SUDOKU Rules: A puzzle looks like the one on the previous page: a 9x9 grid with some numbers filled in. The challenge is to fill in each empty space with a digit 1 through 9, so that no digit is repeated in any row, column or "block". "Block" is my name for the nine 3x3 blocks outlined in thicker black lines in the picture.

Your Assignment: In the Eclipse workspace of your choice, create a new Java project containing package "sudoku". Import the 4 starter files that you downloaded with this assignment: Evaluation.java, Grid.java, Solver.java, and TestGridSupplier.java. You won't need to change Evaluation.java or TestGridSupplier.java. Your assignment is to finish Grid.java and Solver.java.

Grid: This class models a Sudoku puzzle that is unsolved, partially solved, or completely solved. The class has a 9x9 array of ints, called "values". If a Sudoku square is empty, the corresponding cell in "values" is zero; otherwise the cell in "values" contains the number in the Sudoku square. The starter class has a ctor and a toString() method that you should not change. It also has 4 empty methods that you need to write: next9Grids(), isLegal(), isFull(), and equals(). Do not change their names, arg lists, or return types. The comments on the unfinished methods tell you all you need to know about what they should do. It's ok to add more methods to this class.

You don't need to provide this class with hashCode() or compareTo() methods. The equals() method is just so that you can compare your puzzle solutions to solutions in TestGridSupplier. (That is, you won't be collecting Grid instances into a hash set a tree set.)

Solver: Most of this class has already been written. Complete the solveRecurse() method using the backtracking technique you saw in lecture and lab. Also complete the evaluate() method. The main() method is for you to use while testing your code with the puzzles and solutions in TestGridSupplier.

Evaluation: You saw this enum in lecture and lab. It contains 3 values that represent the 3 possible outcomes of the evaluate() method of the Grid class. Look at the source code. Sometimes enums can be complicated, but this one is not. You might need to write a simple enum for the next midterm.

TestGridSupplier: This class contains static methods that return Grid instances. Some of them are puzzles, some are solutions, and some are for testing your code.

STRATEGY: Plan your work before you start. The Solver class needs the Grid class to be working properly, so start with Grid. First write the simple methods isFull(), and equals(). Add a main method that tests these methods with instances from TestGridSupplier. If you get unexpected results, use the debugger or println statements to step through your code and see exactly where the problem is. Then write isLegal(), which is complicated. You'll have to check 9 rows, 9 columns, and 9 blocks. You might write a method called containsNonZeroRepeat(), whose input is an array of 9 ints. For each row, column, and block, build and array of 9 ints, containing the values in that row/col/block; then let containsNonZeroRepeat() figure out if your grid is legal or illegal.

One place where students had problems in past semesters was the next9Grids() method. Figure out which empty location in "this" grid will be filled in next. Then, *without altering "this" grid*, construct 9 new instances, put the new instances in an array list, and return the array list. With such a large method, there is a high probability that you'll have some bugs. Test next9Grids() but writing a main() method that creates a simple grid (maybe using TestGridSupplier), generates the next grids, and prints them out. Think about what you should see. If you don't see exactly that, single-step through your code using the debugger (or use println statements) until the output is perfect. *Only then*, move on.

Use a similar strategy for Solver. Solver is much simpler than Grid, but it contains the recursive backtracking method. Be sure you undertand this method completely. If you don't get correct answers, or if your program crashes or runs forever, use the debugger or println statements.

As always, understand every line of code that you wrote. *If you understand it, fixing bugs is easy. If you don't understand it, fixing bugs is impossible.*

SUBMISSION: As usual, export your sources from Eclipse. Be sure your jar contains all your sources. You will receive zero points if your jar doesn't contain sources. No work will be accepted after the deadline except for documented emergencies.