Lab 5: Sudoku[1]
Due: 8 March 2017

In this lab, you are going to write a program to solve Sudoku puzzles. A Sudoku puzzle is a 9x9 grid of spaces, some of which are empty and some of which already contain digits. The object of the puzzle is to finish filling in the grid, such that each row and each column includes all of the numbers 1 through 9, and each of nine 3x3 sub-grids also contain the numbers 1 through 9.

The Assignment

Your program should:

1. read in and display a Sudoku puzzle, and
2. solve the puzzle and display the solution.

You will need to write code to implement a stack to solve the puzzle using backtracking. I want you to focus on these aspects of the problem (writing the stack code and using backtracking to solve the puzzle, so I have provided code to do various things. I have provided a file with the **main** method (**TestSudoku.java**) that calls all the appropriate methods to create the puzzle, print it, solve it, and print the solution. The class **Sudoku.java** contains methods to read in the puzzle and its solution, make (and unmake) moves, and other functionality. The "only" thing you need to write in this class is the **solve** method. This is where the stack is created and used to solve the puzzle using backtracking. I have also provided a **Move class** that holds the data for a move (the position — row and column — of the move and the digit being put there). Finally, I have given you two Sudoku puzzles, **s1.txt** and **s2.txt**, with their solutions, to test your code on. The code I have given you will automatically read in the actual solution and check your computed solution against the actual. If you want to run your program on other puzzles that do not have solution files, just comment out the relevant code.

Use the parameterized **QueueType interface** and **Queue class** that I used in the Wolves and Sheep example as a guide to writing your **StackType interface** and **Stack class** (both of which are required). If you understand stacks and queues, there aren't many changes you need to make to the queue code to get the stack code. Once you have your stack code, the challenge is to use your stack to explore all possible ways of filling in the puzzle, backtracking when necessary, until you find a solution. We will assume there is a solution, as is the case (as far as I know) in all published Sudoku.

Essentially, you will maintain a stack of **Move**s. You will go in order, row by row, left to right, filling in empty spaces. You keep pushing moves onto the stack as long as there is a legal move for the next empty space on the board. When that is not the case, you will pop the top move off the stack (since this move led to a state in which no legal moves were possible). Now, being careful not to redo that move, see if there is another legal move you could try for that space. If you have tried all the possible moves for that space, pop the next move off the stack, etc.

For example, suppose you have filled in the first empty spaces in the top row with 4, 8, and 5 (pushing those moves onto the stack), but your program then finds that it is impossible to fill in

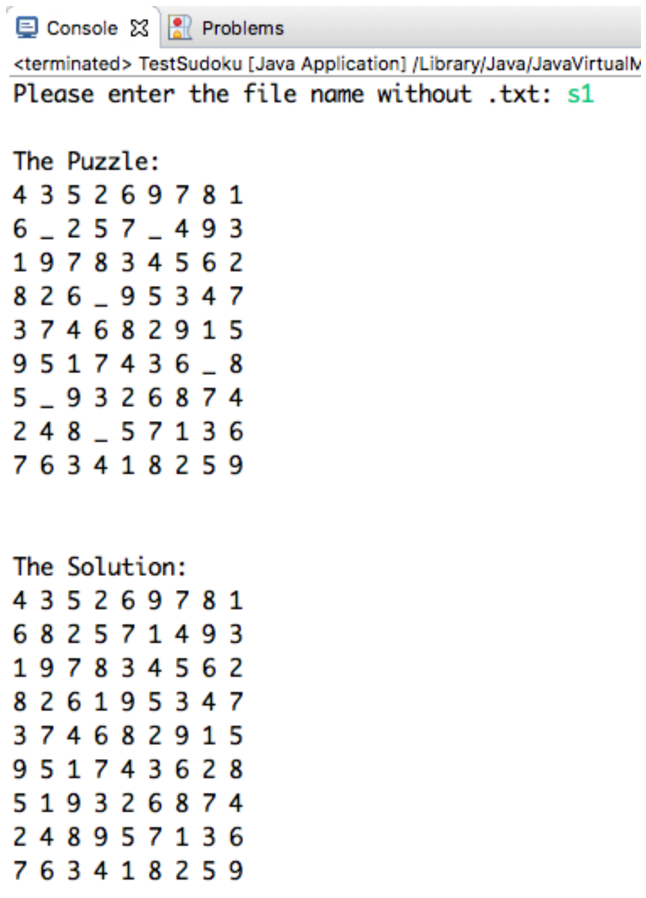
[1] Adapted from a Data Structures lab by Eric Chown.

the next blank space legally. It should “backtrack” by popping the 5 move off the stack. Then it will try digits greater than 5 in that space. If one of them is legal, it will push it on the stack and continue trying to solve the puzzle. If not, i.e. there is no digit that works in that space, then the next most recent move (the 8) needs to be popped off the stack, etc. As you can see, the Last In First Out (LIFO) behavior of a stack is perfectly suited to backtracking.

Do your initial testing with extremely easy puzzles, e.g. a puzzle where you have filled in all but a few of the empty spaces with the correct digits.

NOTE: Don’t get too excited when your program solves puzzle s1.txt. That puzzle can be solved with no backtracking, so solving it does not mean your backtracking code is working!

Sample Output



```
<terminated> TestSudoku [Java Application] /Library/Java/JavaVirtualM
Please enter the file name without .txt: s1

The Puzzle:
4 3 5 2 6 9 7 8 1
6 _ 2 5 7 _ 4 9 3
1 9 7 8 3 4 5 6 2
8 2 6 _ 9 5 3 4 7
3 7 4 6 8 2 9 1 5
9 5 1 7 4 3 6 _ 8
5 _ 9 3 2 6 8 7 4
2 4 8 _ 5 7 1 3 6
7 6 3 4 1 8 2 5 9


The Solution:
4 3 5 2 6 9 7 8 1
6 8 2 5 7 1 4 9 3
1 9 7 8 3 4 5 6 2
8 2 6 1 9 5 3 4 7
3 7 4 6 8 2 9 1 5
9 5 1 7 4 3 6 2 8
5 1 9 3 2 6 8 7 4
2 4 8 9 5 7 1 3 6
7 6 3 4 1 8 2 5 9
```

Submitting Your Program

Please submit your program in the usual way on BlackBoard and turn in a hardcopy.