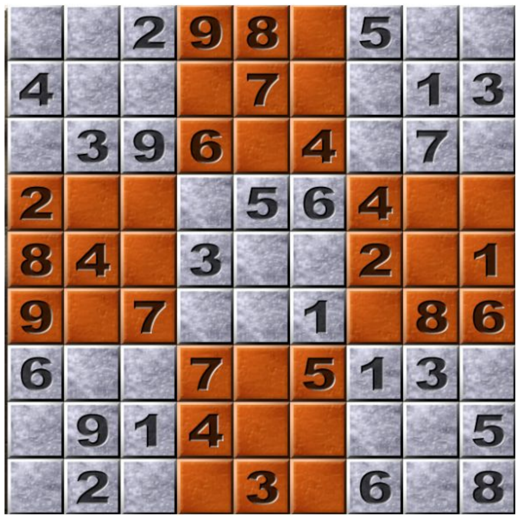
CS-0401 - Assignment #1
Sudoku Game Helper and Solver

1 Introduction

Sudoku games are board games that normally are played on a 9x9 configuration as shown in the figure below. In this game, the board comes with some numbers already placed on the board and the player must fill the rest of board by placing numbers in each empty cell using 3 simple rules (discussed later on in this assignment). The game is solved when the player correctly puts all the numbers on the entire board.
On this assignment, you will create a Java application that can either a) help the player to find the locations that a unique value can be placed on a given time, or b) solves the sudoku entirely by itself.



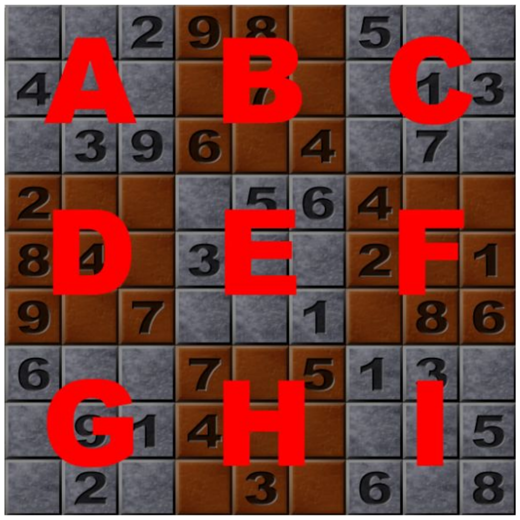
In this assignment you will practice many Java programming topics that we have already discussed in class, among them:

- Primitive variables
- 1D / 2D array manipulation
- Reading Text File
- Reading keyboard input
- Java Set Class
- IF statements
- Nested FOR loops
- Methods
- WHILE loops
- Java ArrayList Class

2 Sudoku Basic Rules

Even though two Sudoku games would look alike, they can be very different in terms of difficulty level. For beginner and intermediate levels, they can normally be solved by using the following three simple rules:

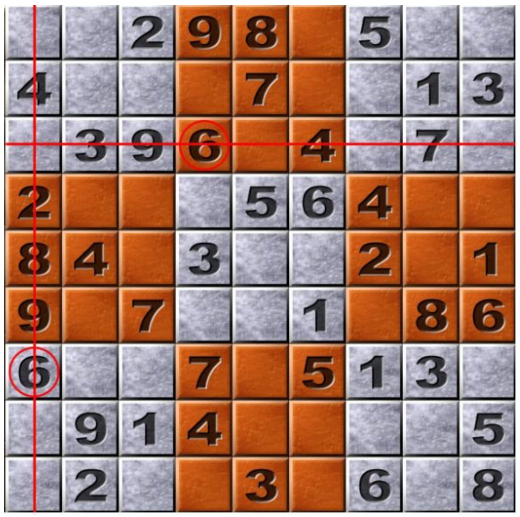
- Rule #1: a number cannot be repeated in a given row.
- Rule #2: a number cannot be repeated in a given column.
- Rule #3: a number cannot be repeated in a given group (3x3). The next figure shows these groups. Just for your convenience they are labelled as A, B, C… Based on this figure, a cell in the 6th row and 7th column belongs to group F (note: you do not need to implement your game using this letter convention, they are here to help on the rules explanation).



Example:

For the cell on the 3rd row and 1st column (see figure below), we cannot use the following numbers:

- Based on the first rule: 3, 9, 6, 4, 7 are not allowed
- Based on the second rule: 4, 2, 8, 9, 6 are not allowed
- Based on the third rule: 2, 3, 4, 9 are not allowed



So from the rules above, the only “survival (or candidates)” numbers that could be placed in that cell are 1 and 5. Since there are two numbers that could be used in that given cell at this point of the game, it is advised to not add any number there at this time.
By going over few empty cells we realize that the cell in the 1st row and 6th column has a unique number that can be placed in it:

- Based on the rule #1: 2, 9, 8, 5 are not allowed
- Based on the rule #4, 6, 1, 5 are not allowed
- Based on the rule #9, 8, 7, 6, 4 are not allowed

The only survivor number when applying the 3 rules in this case is number 3. Since it is the only number that can be placed in that cell, the player can place it without any worries.
After adding this number onto this cell, repeat the search for a cell that could host just one number again. After finding that cell, add the unique value and keep doing it until the game is over.

Your app should either:

1) solve the game by itself, i.e., it finds the cell that has the unique value, place it by itself, and loop again to find another cell that has unique value, etc… until the game is over.

OR

2) help the player to find the location of the cell that has unique value and ask the player to provide the number that he/she thinks it should be placed in that cell.

Both options are highlighted in the code shown in Section 3 (using the same colors as above). You must use ONLY one way in your code and then later try the other one, by commenting out the first one you have chosen.

3 Sudoku App General Methods and Execution Flow

The code below is the “bare bones” of the program that you must implement. However, you can start from scratch if you prefer doing so.
Each method shown here must be implemented by you, as discussed in the next sections of this assignment.

```
public class SudokuGameTerminalWindow {

    private int[][] gameBoard;
    private Scanner keyboard;

    public static void main(String[] args) {
        new SudokuGameTerminalWindow();
    }

    public SudokuGameTerminalWindow() {
        // initialize game
        initializeGame();
        printGameBoard();
        while (!hasGameBeenSolved()) {
            int[] cellLocation = getNextCell();
            if (cellLocation != null) {
                System.out.print("Enter number in (" + cellLocation[0] + ","
                       + cellLocation[1] + "): ");
                gameBoard[cellLocation[0]][cellLocation[1]] = keyboard.nextInt();
                gameBoard[cellLocation[0]][cellLocation[1]] = cellLocation[2];
                printGameBoard();
            } else {
                System.out.println("Difficult Level Sudoku Game. \n"
                  + "You need to implement more rules to solve it...");
                break;
            }
        }
    }
```

4 Description of the Methods

Method signature: private void initializeGame()
Description: Opens a text file containing a 9x9 space-separated integer numbers representing a single sudoku game. After reading it, the method places those numbers in the global gameBoard 2D variable. For testing your code use the same numbers as in the first figure of this assignment.

Method signature: private void printGameBoard()
Description: Prints the game board in the terminal window. Later in this course we will implement it in a GUI app.

Method signature: private boolean hasGameBeenSolved()
Description: Returns true if the game has been completely solved; false otherwise.Hint, loop over all the cells and if you find one cell that has 0 returns false immediately. If after looping over all the cells you could not find any cell with 0, then returns true (game has been solved since there is no empty cells left).

Method signature: private int[ ] getNextCell()
Description: Returns a 1D integer array containing the following information: row, col, and unique value. Based on the yellow highlighted example shown two pages ago, this method would return a 1D array with the following data [0, 5, 3].
This method should:

- Loop over the whole gameboard searching for the next cell that could have just one possible number (see below for things that should be done inside this loop).
- If the cell has a number already, just skip that cell.
- If the cell has a 0 value:
  - Check which numbers are already defined in a given row, as part of the 1st rule. Create a set (similar to an array) containing those numbers.
  - Check which numbers are already defined in a given column, as part of the 2nd rule. Create a set containing those numbers.
  - Check which numbers are already defined in the group (A, B, C, etc.), as part of the 3rd rule. Create a set containing those numbers.
  - Consolidate these (non-allowed) numbers into a single set of numbers.
  - Based on this consolidated set of “non-allowed” numbers, get the set of allowed numbers. Example, if the set of non-allowed numbers contains 1, 2, 4, 6, 7, 9, then the set of allowed numbers should contain 3, 5, 8.
  - If the set of allowed numbers contains just one number, then you found a cell that can have only one number in it. In this case, create the returning array for this method with the row, column, and value in it.
  - If you cannot find any cell that has only one number, then this puzzle is more advanced and cannot be solved using only those 3 rules. Return null in this case.

Example of Expected Code:

```
  private int[] getNextCell() {
    for (int row = 0; row < 9; row++) {
      for (int col = 0; col < 9; col++) {
        if (gameBoard[row][col] == 0) {
          Set<Integer> firstRuleElimination = getFirstRuleElimination(row);
          Set<Integer> secondRuleElimination = getSecondRuleElimination(col);
          Set<Integer> thirdRuleElimination = getThirdRuleElimination(row, col);
          Set<Integer> survivors = getSurvivors(firstRuleElimination,
                                  secondRuleElimination, thirdRuleElimination);
          if (survivors.size() == 1) {
            List<Integer> list = new ArrayList(survivors);
            int uniqueValue = list.get(0);
            return new int[]{row, col, uniqueValue};
          }
        }
      }
    }
    return null;
  }
```

Note: The Set class can be used to hold an array of unique values, such as an array of unique integer values. A value will not be placed in a set if it is already defined in that set. This keeps away duplicate entries, which in this case is a good feature. To create a set use the following command:

```
Set<Integer> set = new HashSet();
```

And to add numbers into it:

```
set.add(number);
```

Note: due to some odd situation, Sets don't have a get(int index) method, so if you need to get a value from a set, let's say the first value, you need first to convert the Set into an ArrayList and then use its get(int index) from it. See code above with the same red highlight.

Method signature:	private Set<Integer> getFirstRuleElimination(int row)
Description:	Loops over all the columns of the given row (method input) and adds all the numbers found in these cells into a Set object. Returns this set object.

Method signature:	private Set<Integer> getSecondRuleElimination(int column)
Description:	Loops over all the rows of the given column (method input) and adds all the numbers found in these cells into a Set object. Returns this set object.

Method signature: private Set<Integer> getThirdRuleElimination(int row, int column)
Description: Each cell defined by a (row, column) belongs to a group (A, B, C, … as shown in the second figure of this assignment). This method shall initially call two methods (see description about the methods below): one returning an array with min and max rows and the other with min and max columns for the current group that this cell defined by a row-col belongs to. And then, loop over the 9 cells defined in this group (2 Nested FOR loops) and collect all the numbers found in the gameboard into a Set object and return that object.

Method signature: private int[] getRowRangeForGroup(int row)
Description: Based on the given row, it defines the minimum and maximum row for that group. Example for row = 1 then min and max row for this group is 0 and 2 respectively, while for row = 4 then min and max rows are 3 and 5. This method returns an integer array with those two numbers.

Method signature: private int[] getColRangeForGroup(int column)
Description: Based on the given column, it defines the minimum and maximum column for that group. Example for column = 7 then min and max column for this group is 6 and 8 respectively.

Method signature: private Set<Integer> getSurvivors(Set<Integer> firstRuleElimination, Set<Integer> secondRuleElimination, Set<Integer> thirdRuleElimination)
Description: This method does two tasks: first it creates a temporary Set and add all the sets given in the method input into it (a consolidated set with all non-allowed numbers found when applying the 3 rules). Then it creates another Set that is the opposite of this consolidated set, i.e., contains only the allowed numbers for the given current cell.

Example: For cell in the 1st row and 1st column:
firstRuleElimination Set: 2, 5, 8, 9
secondRuleElimination Set: 2, 4, 6, 8, 9
thirdRuleElimination Set: 2, 3, 4, 9
Consolidated unique non-allowed value Set: 2, 3, 4, 5, 6, 8, 9
Allowed values Set: 1, 7 (method returns this set)

5 Suggestions on implementing this app

a) Don't try to attach all the methods at the same time. Create one method and check if it is doing what is expected. After being sure that it does, then go ahead and create another required method and so on.
b) If you find anything wrong in my writeup, please let me know as soon as possible, so I can tell the other students about it.
c) After successfully creating a program that solves the puzzle shown in figure 1, try your program with other sudoku values. For a list of text sudoku games, go to http://lipas.uwasa.fi/~timan/sudoku/. Click on the links and copy and paste the numbers into the text file used by your application.
d) Got stuck? Ask the lab TA and me.

6 Examples of running the code

6.1 Application solving puzzle all by itself

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

6.2 Application Helping user by identifying next cell to be filled in

0 0 2 9 8 0 5 0 0 4 0 0 0 7 0 0 1 3 0 3 9 6 0 4 0 7 0 2 0 0 0 5 6 4 0 0 8 4 0 3 0 0 2 0 1 9 0 7 0 0 1 0 8 6 6 0 0 7 0 5 1 3 0 0 9 1 4 0 0 0 0 5 0 2 0 0 3 0 6 0 8 ****************** Enter number in cell at (0,5): 3	0 0 2 9 8 3 5 0 0 4 0 0 0 7 0 0 1 3 0 3 9 6 0 4 0 7 0 2 0 0 0 5 6 4 0 0 8 4 0 3 0 0 2 0 1 9 0 7 0 0 1 0 8 6 6 0 0 7 0 5 1 3 0 0 9 1 4 0 0 0 0 5 0 2 0 0 3 0 6 0 8 ****************** Enter number in cell at (0,8): 4	0 0 2 9 8 3 5 0 4 4 0 0 0 7 0 0 1 3 0 3 9 6 0 4 0 7 0 2 0 0 0 5 6 4 0 0 8 4 0 3 0 0 2 0 1 9 0 7 0 0 1 0 8 6 6 0 0 7 0 5 1 3 0 0 9 1 4 0 0 0 0 5 0 2 0 0 3 0 6 0 8 ****************** Enter number in cell at (0,7): 6	0 0 2 9 8 3 5 6 4 4 0 0 0 7 0 0 1 3 0 3 9 6 0 4 0 7 0 2 0 0 0 5 6 4 0 0 8 4 0 3 0 0 2 0 1 9 0 7 0 0 1 0 8 6 6 0 0 7 0 5 1 3 0 0 9 1 4 0 0 0 0 5 0 2 0 0 3 0 6 0 8 ****************** Enter number in (1,5): 2	0 0 2 9 8 3 5 6 4 4 0 0 0 7 2 0 1 3 0 3 9 6 0 4 0 7 0 2 0 0 0 5 6 4 0 0 8 4 0 3 0 0 2 0 1 9 0 7 0 0 1 0 8 6 6 0 0 7 0 5 1 3 0 0 9 1 4 0 0 0 0 5 0 2 0 0 3 0 6 0 8 ****************** Enter number in (1,3):