TP C#11: Suji wa dokushin ni kagiru
数字は独身に限る

1 Guidelines

At the end of this tutorial, you have to submit an archive that respects the following architecture:

```
rendu-tp-sudok_u.zip
+-- sudok_u/
    |-- AUTHORS*
    |-- README
    +-- Sudoku
        |-- Sudoku.sln*
        +-- Sudoku*
            |-- Stuff.cs*
            |-- Program.cs*
            |-- Sudoku.cs*
            |-- IO.cs*
            +-- Everything except bin/ and obj/
```

Of course, you must replace "`sudok_u`" with your own login. All the files marked with an asterisk are *MANDATORY*.

Don't forget to verify the following before submitting your work:

- The `AUTHORS` file must be in the usual format (a *, a space, your login and a line feed).
- The `README` file (without extension) must contain all the information that you want to share with us. Any bonus unspecified will not be taken into account.
- No `bin` or `obj` directory in your project.
- **The code must compile, otherwise you will get a 0!**

Contents

2 Introduction

Read carefully the subject before rushing headlong.
All functions are prohibited except for otherwise indications given by the subject or your assistants. Don't forget to handle every error case, except if specified not to.

2.1 Suji wa doku...what ?

> « ***Sudoku*** *(数独 sudoku, Digit-single); originally called Number Place, is a logic-based, combinatorial number-placement puzzle. The puzzle was popularized in 1986 by the Japanese puzzle company Nikoli, under the name Sudoku, meaning single number. It became an international hit in 2005.* »
>
> — Wikipedia

The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "boxes", "blocks", "regions", or "sub-squares") contain all of the digits from 1 to 9. The four rules of Sudoku are the following:

- Only one digit a cell
- A digit can be present only once in a raw
- A digit can be present only once in a column
- A digit can be present only once in a region

Here is an example of a Sudoku grid that you can easily find in your favorite daily newspaper.

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

2.2 Goals

This week, the purpose of the practical is to help you tinker with Sudoku, from the creation to the solution of a grid (nice for your grand-mother, you will be able to prove her that IT and what you do is useful). We will use what you know about array handling, lists and files to accomplish this. But before rushing it, you will have to write a few small functions.

3 Before starting...

Firstly, here are some short exercises. You will have to put them in the *Stuff.cs* file.

3.1 TimeAfterTime

```
public static string TimeAfterTime(ref int days, ref int hours,
                                   ref int mins, ref int sec);
```

Goal

This function takes by reference parameters corresponding to a duration, and will reorganize them in order to be correct. For example if you give it 1664 seconds, it will send back 27 minutes and 44 seconds.

This function will return `true` if the conversion could be performed, otherwise `false`.

Do not forget...

You have to handle negative values. For example, with -10 minutes and 2403 seconds, you will have 30 minutes and 3 seconds. If the total result is negative, it should return `false`.

3.2 Transcription by dynamic repertoire

```
public static string Compression(string source);
public static string Decompression(string source);
```

The transcription by dynamic repertoire is a common approach of data compression that consists of replacing every word of a text by a number that stands for its position in the repertoire. Such a transcription is known as static if the repertoire is already known in advance. The main disadvantage of this method is the fact that the repertoire must be known to decompress the text.

On the other hand, a transcription by dynamic repertoire avoids this problem by deriving the content of the word repertoire from the text to be compressed. At the beginning of the process, the directory is empty. Considering the text from the beginning, when a word is in the directory it is replaced by the number of his position. If it is not present, it is added to the end and is left unchanged in the text.

Goal

First, you will have to apply this compression process to the string passed as parameter and then return it compressed.

The input parameter will only contain lowercase characters, spaces and line feeds. Spaces and line feeds will not be converted (so, they have to be kept).

The decompression function will perform the inverse process in order to recover the initial string.

Example

An example with this marvelous haiku:

```
string source = "the great blue dog howls
                 the sky howls loud with the dog
                 great blue much stormy";

string comp = Compression(source);

Console.WriteLine(comp);
/* prints "the great dog howls
           1 sky 4 loud with 1 3
           2 3 much stormy" */

Console.WriteLine(source == Decompression(comp));
/* prints True */
```

Hint

> How to get all words one after the other? Maybe you should visit MSDN for information on the string class.

4 Sudoku

4.1 Sudoku.cs

We will start slowly with some easy array manipulation methods.

Objective

You must create a class called `Sudoku` and give it attributes. If you read carefully, a sudoku is simply composed of a grid of 9x9 cells, which all contain digits from 0 to 9 (0 stands for an empty cell). We advise you to have a `Random` attribute, that can help you for a method... It's up to you to give them the appropriate accessibility levels.

Before writing your constructor, you may start by the following methods :

Init

```
public void Init(int init);
```

Fill the grid with the given integer `init`.

Print

```
public void Print();
```

Print on the console the grid as the following:

```
+-----------------------+
| 4 9 7 | 8 3 6 | 2 1 5 |
| 1 3 5 | 4 2 7 | 6 9 8 |
| 2 8 6 | 9 5 1 | 3 7 4 |
+-------+-------+-------+
| 9 6 8 | 5 4 2 | 7 3 1 |
| 3 7 4 | 6 1 9 | 5 8 2 |
| 5 2 1 | 3 7 8 | 4 6 9 |
+-------+-------+-------+
| 8 5 3 | 1 6 4 | 9 2 0 |
| 7 4 9 | 2 8 3 | 1 5 6 |
| 6 1 2 | 7 9 5 | 8 4 3 |
+-----------------------+
```

RandomlyFill

```
private void RandomlyFill(int nb);
```

This method will fill the grid with `nb` more random numbers. They must respect the sudoku rules. Your code will be a loop executed `nb` times. At each iteration, for a cell:

- Verify that it is not already filled.
- Verify that its column does not already contain it.
- Verify that its line does not already contain it.
- Verify that its region does not already contain it.
- If all those conditions are verified, you can put the number inside and decrement `nb`. Otherwise, we do another loop without decrementing.

You can add any additional functions that you want to have. Do not forget to comment them!

Hint

> Only use this function with a small `nb`, that way the grid will fill correctly ($nb < 60$).

Constructor

The constructor will now simply initialize attributes and fill randomly the grid with an integer asked to the user.

Bonus strongly recommended

As you may have seen, the algorithm is not the best one. As a bonus you can try a better one... Don't forget to explain it in your *README*.

Hint

> Did you read all the subject?

4.2 IO.cs

`IO` will be used for file interactions, it will allow us to get their contents.

Goal

Now that you can display nice grids, let's continue with the `IO` class. It will help you get the content of a file specified by the user in order to build a sudoku grid from it, and the possibility to save a grid in a file.

```
1 private static bool FileToTab(string filename, int[,] tab)
```

This function opens the file `filename` and puts in the `tab` array all the numbers contained in it. It removes spaces, tabulations and line feeds. It also manages all errors. The function returns `true` if the array has been filled correctly, `false` otherwise.

```
1 public static void LoadFile(int[,] tab)
```

This function calls the function `FileToTab` with the file name asked to the user while the array is not correctly filed.

```
1 public static bool SaveFile(int[,] tab)
```

This function asks the user a file name and tries to open it. If the file already exists, it asks if the file can be rewritten over. For all other errors with the file, return `false`. Otherwise, if the file is correctly opened, write each line of the sudoku in the file.

4.3 Back to Sudoku.cs

Last step, the longest, solving the Sudoku grid! In addition to using your knowledge about arrays, you will be able to impress us with some lists. Need a reminder? MSDN is your friend, cherish it.
Firstly, a little lesson about named and optional arguments.

Course

Functions in C# can use arguments said **optionals**, it means that you can call this function without providing this argument, a default one is already given. Those optional parameters must be put at the end. An example to be clear :

```
1 public static void HelloToSomeone(string someone = "World", int nb = 1)
  {
3     for (int i = 0; i < nb; i++)
          Console.WriteLine("Hello " + someone + "!");
5 }

7 /* In your Main */
  HelloToSomeone("Alex", 2);  // prints "Hello Alex!" 2 times
9 HelloToSomeone();           // prints "Hello World!"
  /* HelloToSomeone(42); */   // ERROR
```

To avoid this last error, it is possible to **name** explicitly an argument. It can also be used for non-optional argument.

```
HelloToSomeone(nb : 42);    // prints "Hello World!" 42 times
```

New constructor

Your constructor will now take into account the fact that the user prefers to use his own grid. This constructor will have a single boolean argument `getFromUser`, that will be optional. Its default value is `false`.

Example

Both cases must work.

```
Sudoku defaultSudoku = new Sudoku();
/* constructs a new sudoku ramdomly filled */

Sudoku fromUser = new Sudoku(true);
/* constructs a new sudoku after asking the user a file to load */
```

Hint

| You just made some useful functions in *IO.cs*...

Save

```
public bool Save();
```

This method will ask the user if he wants to save its solved grid. While the Save method you made in IO is false, ask again the user if he wants to save the grid.

Quick reminder about lists

When initializing a list, it is possible to directly add elements (as for an array) thanks to curly brackets.

```
List<string> namae = new List<string> { "Arekkusu", "Rec9", "Poni" };
/* the list namae contains now "Arekkusu", "Rec9" and "Poni" */
```

GetPossibleNumbers

Here are three methods that you have to implement for the `Sudoku` class:

```
void GetLinePossibleNumbers(ref List<int> numList, int line);
void GetColumnPossibleNumbers(ref List<int> numList, int column);
void GetRegionPossibleNumbers(ref List<int> numList, int line,
                                                     int column);
```

Each of these functions will find all possible numbers for the empty cells of the sudoku grid among the number contained in the `numList`, respectively by line, column and region. So, you should perform the deletion of the numbers present in the list `numList` given as parameter.

Be aware for `GetRegionPossibleNumbers`, the parameters `line` and `column` match with a cell position, so you should search among the cells that share the same region.

GetMinList

```
void GetMinList(ref List<int> minList, ref int xIndex, ref int yIndex);
```

This function will find the empty cell which has the least possible numbers that can fill it. Then, it returns by reference the indices x and y of the cell from the grid, and finally the list of possibilities.

Hint

> Some clues:
>
> - Creation and fitted initialization of the list
> - Check each cell in order to find the smallest list
>
> Previously coded functions might like to be used.

IsFinished

A little bit easier?

```
1 public bool IsFinished();
```

This function will browse the entire grid and check if the array is correctly filled, respecting the sudoku rules. It obviously returns the result.

Hint

> Remember that previous functions can be very helpful.
> Think about the conditions for which the grid is correctly set up.

Solve

And now, what you have all been waiting for...the Solver!

```
1 public bool Solve();
```

Well, you probably understand that this function will solve the sudoku grid if it is correct. As we are really nice, here is a little algorithm:

- Return true if the grid is finished
- Otherwise, save the current state of the grid
- Until cells have only one option, fill them with it.
- For all possible numbers of the cell that has the least amount of possible numbers: fill this cell with one of them, use recursion, return the result if true, otherwise continue with the next number and continue.
- Test again if the grid is finished in case the previous state is not reached. Return `true` if it is...true, otherwise, restore the saved grid and return `false`.

Try to use this function on an empty grid to generate a generic grid. Another thing to test: this function on a grid filled by the function `RandomlyFill` (with between 0 and 20 as parameter): you will have your lovely random grid! Be aware that the grid generated by `RandomlyFill` is correct, but has not necessarily a solution.

If you need some help on the functions of this exercise, do not hesitate to ask your ACDC.

4.4 Program.cs

Last effort! You have a class `Sudoku` that works very well (yeah, I'm sure of it), you now need to fill your `Main` program; so that the user can easily use your solver.

Goal

At least, you should: Fill your `Main` with what is needed for the user to choose to load a grid from a file or generate a random one, solve the grid and save the solved grid.

Bonus

Impress us! Pretty colors, game interface for the sudoku...

The bullshit is strong with this one...