Informatics 1: Object Oriented Programming

Tutorial 05

Week 6: 25/02 - 01/03

Volker Seeker (volker.seeker@ed.ac.uk)
Naums Mogers (naums.mogers@ed.ac.uk)

1 Introduction

In this tutorial you will implement a solver for your Sudoku game. The solver will make use of a *backtracking* algorithm. This algorithm is a brute force search algorithm which goes through each empty field in the Sudoku trying possible values until it has found a valid one according to Sudoku rules. It then moves on to the next empty field to repeat this process. Should it reach the end of the grid, the Sudoku is solved. Should it reach an empty field where none of the available numbers fit, it moves backwards trying different combinations in earlier fields.

During this process, only empty fields should be filled while the **clue** fields which have a number from the beginning of the game should remain unchanged. In our current implementation of the Sudoku, fields are not clearly marked as **clue**, i.e. initial field. Hence, the first step of this tutorial exercise is to encapsulate the field information into a new class `Field`. You can then add a member for initial clue fields to this class.

2 Exercises

The tasks of this exercise are broken down into two parts. The first refactors the current `GameGrid` class so that it uses a new `Field` class for its inner data management. The second part implements the actual solver.

2.1 The Field Class

The `Field` class encapsulates information about an integer game field. It is not necessary to make this specific to a Sudoku game, hence also negative integers could be allowed. The client class `GameGrid` will ensure that only valid Sudoku data is used in the field for its purposes.

Task 1 - Class Implementation

◁ **Task**

Create a new `public class Field` with the two member variables `value` of type `int` and `initial` of type `boolean`. The member `initial` should be immutable for this class, hence it should get a `final` modifier.

Create two constructors: The first is a no-argument constructor which creates a `Field` instance with the default values zero and `false` for its members. The second is a constructor which gets arguments with initial values for both members and sets them accordingly.

Provide a getter and a setter method for the `value` member. Since `initial` is immutable, provide only a getter for this one.

Lastly, provide a `toString` method which returns a `String` representation of the `Field`'s value. You can indicate with an additional apostrophe or similarly whether the value is initial or not.

Task 2 - Refactoring GameGrid

◁ **Task**

In this task you will make use of the new `Field` class in your `GameGrid` class. In `GameGrid`, change the type of the `grid` member from `int[][]` to `Field[][]`. This will bring up a couple of errors which you now have to fix.

Firstly, you need to change the access to the `grid` member for each method using it in `GameGrid`. Every `grid` access needs to be changed into a getter call and every assignment of a value to the `grid` array needs to be changed into a setter call.

Secondly, you need to change the initialisation of the `grid` member in all constructors. Currently the constructors simply assign an `int[][]` array to the `grid` member as initial value. This could be passed in directly or read from file using the `IOUtils` class. Now, the data type has changed and you need to create a `Field` wrapper for each value of the passed in grid argument. You should implement a new `private` function `initialiseGrid` to do this, so you can use it for both constructors.

Task 3 - Integrating `isInitial`

◁ **Task**

With the basic functionality refactored, you can now integrate functionality for the new feature indicating if a field was set as clue, i.e. initial to the start of the game. To do this, write a new `public` method `isInitial` as part of the `GameGrid` API which receives a row and a column coordinate as argument and checks the internal `grid` member if the specified value is initial to the game. The corresponding `boolean` result should be returned. Make sure you provide the necessary error handling for this new method.

You should now also use this new method in `setField` as additional constraint when checking validity. Only set a grid value if the new value is valid and if it is not applied to a clue field.

2.2 The Solver Class

In this second part you will implement the actual solver. Let's use a test driven design approach to do this, i.e. we start with the client and `Solver` API and then fill in the necessary implementation.

Task 4 - Setting up the Client

◁ **Task**

Extend the game menu with a new entry for finding a solution for the running Sudoku game. The corresponding handler code in your `main` function will contain the solver's client code.

In the handler, call a class method `solve` on a `Solver` class (which does not exist yet). The `solve` method should get an instance of the currently running `GameGrid` as argument and return `true` if a solution was found and `false` otherwise. Based on the result, print the solution to the console or a corresponding message if none was found.

The solution should be calculated directly in the `GameGrid` instance passed into the `solve` method, i.e. as a *side effect* of this method. In order to maintain the original state of the game, you should create a **deep** copy of the `GameGrid` instance you use for playing before you pass it to the solver's method.

There are different ways of copying objects and their states in Java. One uses a `clone` method which is called on the object to be copied. This method has a few disadvantages, hence we will follow a different way by using a copy constructor. A copy constructor is a constructor, that gets an argument of the same type it is supposed to create. It has full access to all members of the argument (even private ones) and can make the necessary copies to initialise the new object.

Create a copy constructor in the `GameGrid` class with the following header: `public GameGrid(GameGrid grid)`. This constructor should initialise the `grid` member by copying all values from the passed in `grid` argument. Make sure that this is a **deep** rather than a **shallow** copy of all `Field` instances.

With the copy constructor in place, make sure you pass a copy of your `GameGrid` instance to the solver's method.

Task 5 - The Backtracking Algorithm

◁ **Task**

In this last task, you will finally implement the actual solver. This solver should use backtracking as mentioned in the introduction. First create a new class `Solver` according to the requirement of your client code and add the corresponding class method `solve`.

The backtracking algorithm should start in the top left corner of the grid. It then iterates through all values until it finds an empty field. In this field it will try possible Sudoku values starting with 1. If a valid value is found, it moves to the next empty field and repeats this procedure. If it reaches the bottom right corner of the grid, the Sudoku is solved.

If it reaches an empty field where none of the numbers from 1 to 9 are allowed, it has made a mistake earlier and needs to track back. It will now move back to the previously set field (ignoring clue fields) to increase the number it has put in there earlier. Should a different number work, it continues moving forward again.

If it reaches the beginning of the grid again, it has tried all possible combinations but has run into a dead end for all of them. That means the given Sudoku is unsolvable. With this exercise you are given a few unsolvable Sudokus to test this.

Have a look at this Wikipedia page to see an animated example:

https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

The solution should be determined using the copy of the `GameGrid` argument the `solve` method receives. Hence, make use of the `setValue`, `clearField` and `isInitial` methods as required.

3 Optional Extra Tasks

If you have finished all of the above tasks and look to do more, please consider the following suggestions or come up with your own ideas:

- **Recursive Solver** Write a recursive solution of the backtracking solver by adding a `public static boolean solveRecursive` function to your `Solver` class and implementing required functionality.
- **List of Lists** Refactor the Sudoku data representation in your `GameGrid` class by using a list of lists, i.e. `ArrayList<ArrayList<Field>>`, instead of a two-dimensional array, i.e. `int[][]`, for the `grid` member. Modify the remainder of the class accordingly.