Informatics 1: Object Oriented Programming

Tutorial 04

Week 5: 11/02 - 15/02

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

1 Introduction

In this tutorial you will apply your recently learned OO skills to create a class, `GameGrid`, for the Sudoku game grid. It will encapsulate most of the inner workings regarding grid representation, validity checking, printing, etc. and offer a clean API to the outside.

Refactoring Since a lot of the code for your Sudoku is already written, i.e. functions to check the validity of a value or a function to print the grid to the command line, you will have to apply a technique called refactoring to create the `GameGrid` class. Refactoring is a technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. There are specific rules to this technique which you will hear more about in a later lecture, for now it is mostly copy-paste and some modifications.

Error Handling Creating an API comes with a few challenges which you need to handle. Since you do not always have control over how your class will be used but you still want to provide a well functioning data type, you will have to add some error handling to the exposed public methods. This way you can make sure that the API is used the right way and the user is informed of potential mistakes.

Note 1: Make sure you compile and test new code regularly to catch errors early.

Note 2: For the purpose of using more natural variable names I have renamed **x** to **column** and **y** to **row**.

2 Exercises

The following exercises are broken down into multiple steps to make them more accessible. You will notice that there are more sub tasks than you are used to from previous tutorials. Each of them are, however, less effort to solve. The overall structure has five main sections: Creating a game grid class with basic functionality, adding already existing functionality to this class by refactoring existing code, adding new functionality, adding error handling and lastly integrating it into the `main` function.

2.1 The GameGrid Class

Task 1 - Class, Members and Constructor ◁ ~~Task~~

Create a new `public class GameGrid` in a new file with the same name. For now this class will internally hold the Sudoku data in an `int[][]` array like you are already using. For this purpose, create a `private final` member variable `grid` of the type `int[][]`. No need to initialise it at this point, you will do that in the constructor.

Create a constructor for the `GameGrid` class which receives a single argument `int[][] grid`. When calling the constructor later, this `grid` will be the same as the one you are currently using in your `main` function. In the constructor assign the argument to the member variable.

You can use this constructor for testing your intermediate implementation by creating an instance of the `GameGrid` class in the `main` function and passing the hard-coded integer array `grid` as argument to the constructor.

Lastly, to avoid hard coded values, add a few constants as `static` immutable members to the `GameGrid` class. You can use those throughout the class and since they are public also in other classes. For simplicity, you can copy those:

```
// Constants for coordinate boundaries and Sudoku numbers
public static final int GRID_DIM = 9;
public static final int SUBGRID_DIM = GRID_DIM / 3;
public static final int MAX_VAL = 9;
public static final int MIN_VAL = 1;
public static final int EMPTY_VAL = 0;
```

Task 2 - Field Getter and Setter ◁ ~~Task~~

In order to allow access to fields of the `private grid` member, you need to provide a getter and a setter.

Create a `public` method `getField` which receives two coordinate arguments `int column` and `int row` and returns the corresponding `int` value of the `grid` member.

Create a `public` method `setField` which receives two coordinate arguments `int column` and `int row`, a `int value` argument and returns nothing. Set the given `value` at the specified coordinates in the `grid` member.

2.2 Refactoring and Integration

In this section you will add the functionality for validity checking of numbers and printing a Sudoku game grid to the `GameGrid` class.

Task 3 - Refactoring `isValid` ◁ ~~Task~~

Copy and paste the functions `isValid`, `checkRow`, `checkColumn` and `checkSubGrid` from the main class to the `GameGrid` class.

All four methods do not need to be exposed to the outside, so you can make them all `private`. Also, they should be used as instance rather than class methods, so remove the `static` modifier from each function header. Lastly, you can remove the `int[][] grid` argument from each function signature. Since all four will work with the state members of their corresponding `GameGrid` class, this argument is no longer required.

The `isValid` method is integrated into your class by using it in the `setField` method. Change the header

of `setField` to return a `boolean` value. When a value is to be set with this method, you should check with `isValid` if the value is correct. If it is, set the value into the `grid` member and return `true`; otherwise, return `false`.

Task 4 - Refactoring `printGrid`

◁ ~~Task~~

Next, you will refactor printing the game grid. For that purpose we make use of the `toString` method which classes in Java offer to print a human readable representation of their current state. Create a `public` method `toString` which has no arguments and returns a `String`.

Following the implementation of your `printGrid` function in the main class, implement the same functionality in the new `toString` method with the **following significant difference**: `toString` methods do not print to the command line directly. Instead, they create a `String` which can then be printed by the calling code. Hence, when building the `String` representation of your game grid, make sure you do this in a `String` variable which you will then return in the end.

2.3 New Functionality

For convenience, you will add two new functions to the `GameGrid` class.

Task 5 - Clearing a Field

◁ ~~Task~~

Create a `public` method `clearField` which receives `int column` and `int row` coordinates and returns nothing. This method should set the corresponding entry in the `grid` member to zero.

Task 6 - Loading a Sudoku from File

◁ **Task**

Overload the constructor by creating a second one which gets a `String` argument. This `String` should contain the path to a file with initial Sudoku grid data. In the constructor, make use of the `static loadFromFile` function you can find in the `IOUtils` class provided for this tutorial. This function loads a game grid from the specified file and returns it as a two dimensional integer array. Assign this array to your `grid` member.

2.4 Error Handling

To avoid misuse of your API, you will now add some error handling for invalid arguments to `public` methods of your class.

Task 7 - Preventing Null Pointer Exceptions

◁ **Task**

As you have learned, uninitialised objects reference `null` and can lead to `NullPointerExceptions` in Java when used. Protect each method which gets an object argument with a call to requireNonNull in the very first line. This method, which you can find in the class `java.util.Objects`, checks if a given object points to `null` and throws an exception if so.

Task 8 - Handling Incorrect Coordinates and Values

◁ **Task**

To avoid invalid arguments for coordinates and Sudoku number values passed to your `public` methods, you should add error handling if-clauses to the beginning of each of them. Should a coordinate be out of bounds, e.g. smaller than 0 or larger than 8 you should throw an `IllegalArgumentException` with a corresponding message. This is an unchecked exception and does not need to be indicated with a `throws` modifier at the method header. Do the same should a Sudoku number value be invalid.

2.5 Integration

Task 9 - Integration into the Main Program

◁ **Task**

Now it is time to use the new `GameGrid` class. In the main `Sudoku` class, remove the functions you have copied earlier: `isValid`, `checkRow`, `checkColumn`, `checkSubGrid` and `printGrid`. Also remove the hard-coded two dimensional integer array `grid`.

Your `main` function should now expect a single command line argument which is the path to a Sudoku data file on your system. With this exercise you have been given a few sample files you can use. Make sure they are available under the specified path. If you are using Eclipse, you can import them into your project and place them outside of your src folder. The corresponding path is then simply the name of the file.

Add some error handling in the first line to check if the given `args` array contains at least one value. If not, you can exit the program or throw an `IllegalArgumentException`.

Next, create an instance of the type `GameGrid` called `game` by passing the first argument of your `args` array to it. This will call the constructor which makes use of the `readFromFile` function.

Replace each call to the old `printGrid` with a simple `System.out.println(game)`, remove the call to the old `isValid` and replace it with a call to `game.setField` and lastly replace the direct manipulation of the hard coded `grid` to set a zero value with a call to `game.clearField`.

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:

- **StringBuilder** Since `String`s are immutable, the current `toString` implementation creates a lot of objects just to throw them away again. The `java.lang.StringBuilder` class can do this much more efficiently. Refactor your `toString` code so it uses `StringBuilder` instead of a `String` variable.
- **Saving to file** Upgrade the `IOUtils` class with a class method `saveToFile` which gets a `GameGrid` argument and a `String` path argument and saves the corresponding Sudoku data to the file specified in the `String`. You can make use of the `java.io.PrintWriter` class to accomplish this.