Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden
Christoph Daniel Schulze, Steven Smyth

Institut für Informatik

7. Übung zur Vorlesung „Programmierung“

Assignmented Sevenfold: Sounding the Seventh Submission
Abgabe am Sonntag, 18. Dezember 2016 - 23:55

This week, we will continue our endeavours to write classes that do things. This time, we start overriding other methods than just `run()`, and we start working with arrays and lists. Also, note that we're starting to put classes into packages. In each assignment, we tell you which package to put your classes in (or the package is clear from the source code you download).

Also note that we of course expect you to write Javadoc comments for every class entry, even if this is not explicitly mentioned in an assignment!

Aufgabe 1 - A Mandatory Logic-Based Combinatorial Number-Placement Puzzle — 49 Punkte

While the title of this assignment sounds impressively important and complicated, the actual assignment is to implement a class that models a Sudoku board. Wikipedia explains Sudoku like this:

> 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") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution.

We have written a class called `NumberBoard` that you can download here. Do so, and import it into an Eclipse project. You will find that the class models a board that consists of rows and columns, thus forming cells. Each cell can either be empty or be filled with an arbitrary integer ≥ 0. Read the Javadoc comments for all of the class's entries to familiarize yourself with the functionality the class provides. During this assignment, leave this class unchanged.

Create a subclass of `NumberBoard` called `Sudoku` in the same package as the `NumberBoard` class. The idea here is that you learn how to sensibly subclass an existing class that already implements a bit of functionality. Implement the following features:

- `NumberBoard` has a constructor that can construct number boards of arbitrary sizes. However, Sudoku boards always have nine rows and nine columns. Add a constructor to `Sudoku` that doesn't take any parameters and invokes the superclass constructor in a way to construct a board with nine rows and nine columns.
- The implementation of the `setValueAt(int row, int col, int value)` method in `NumberBoard` checks if `value` is valid before writing it into the specified cell. As far as `NumberBoard` is concerned, `value` is valid if it is either `NumberBoard.EMPTY` or ≥ 0. However, in a Sudoku board, `value` can either be empty or it must hold that $1 \leq value \leq 9$. Overwrite `setValueAt(...)` such that the superclass implementation is only called if `value` is valid in the Sudoku sense. Note that you shouldn't check if the Sudoku board overall is valid in this method. That's what the next method does.
- Speaking of the next method: implement a method `isValid()` that returns `true` if the Sudoku board's current state is valid or not. The state is valid if there is no row, no column, and no 3×3 sub-grid that contains a number more than once. Note that there can be arbitrarily many empty fields. You may want to write a private method each to check the rows, columns, and sub-grids for validity, but that is up to you.

- Override the `toString()` method such that it returns a `String` that represents the current state of the board. Each row should be represented by a line in the string. Each pair of cells in a row should be divided by a space character. Empty cells should be represented by a space character. You can start a new line in the string by adding the newline character, `"\n"`. Printing the result to the console should look like this:

```
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
```

Aufgabe 2 - Mandatory Optimus Prime Numbers 49 Punkte

A number $n \in \mathbb{N}$ is a *prime number* if it can only be divided without remainder by 1 and itself. Now, before we continue: wars have been fought about whether 1 itself is a prime number or not. While this point may seem worth to be discussed about over a nice pint of beer, let's simply go with the popular opinion that 1 is not a prime number.

The *Sieve of Eratosthenes* is an algorithm that yields all prime numbers between 2 and a given upper limit N. To apply the algorithm, you start by writing down a list of the integers between 2 and N. For example:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

You then mark the first number in the list, indicating that you have found a prime. Whenever you mark a number as a prime, you go through the rest of the list and cross off every multiple of that number, since none of those multiples can itself be a prime. Thus, after executing the first cyle of the algorithm, your list will look like this:

2, 3, _, 5, _, 7, _, 9, _, 11, _, 13, _, 15, _, 17, _, 19, _

Complete the process by finding the next number that has neither been marked, nor crossed off and repeating the algorithm until each number in your list is either marked as a prime or crossed off.

Write a subclass of `ConsoleProgram` called `SieveOfEratosthenes` in a package called `programming.set7.optimus`. When run, your program should ask the user for an integer $N \geq 2$ and print all prime numbers between 2 and N to the console, each on its own line. If the number entered by the user is < 2, throw an `IllegalStateException` with an appropriate error message. Solve this problem **with an array**, not with a list.

Finally, in addition to the usual comments, add comments to your program that explain and relate the following terms to your code if examples appear there:

- Array
- List
- Exception
- `throw`

Aufgabe 3 - The Average Optional Assignment 1 Punkt

It's almost Christmas, so let's learn everything about statistics! Here we go. Consider the following list of numbers:

```
5, 92, 42, 17, 53, 23, 13, 1, 6
```

The sum of the numbers is 252, the minimum is 1, the maximum is 92, the average is 28, and the standard deviation is about $27,9$.

Instead of calculating all this yourself, write a class that can keep track of a list of numbers and calculate statistical things for us. Follow the following followable guidelines:

- Name your class `StatisticsCollector` and put it in a package called `programming.set7.statistics`.
- Write the following methods:
  - `addItem` takes a single parameter of type `double` and adds that to the list of numbers managed by your class. In our example above, this method would be called once for every item in the list.
  - `getCount()` returns the number of items added thus far as an integer.
  - `getSum()` returns the sum of all items added thus far.
  - `getMinimum()` and returns the minimum item added thus far. If there are no items yet, the method should return `Double.POSITIVE_INFINITY`.
  - `getMaximum()` and returns the maximum item added thus far. If there are no items yet, the method should return `Double.NEGATIVE_INFINITY`.
  - `getAverage()` returns the average of all items added thus far. If there are no items yet, the average is 0.
  - `getStandardDeviation()` returns the standard deviation of all items added thus far. The standard deviation is the square root of the variance. The variance, in turn, is the average of the squared differences between each item and the average of all items. If there are no items yet, the standard deviation is 0.

Hint: Use a list to save all the items passed to your collector. Think about why this is a better idea than using an array!

Zusatzaufgabe 4 - Systematically Solving Sudokus for Serious Scientific Success — 1 Punkt

We don't solve Sudokus by hand. That's for other people. We're computer scientists. So in this assignment, you will write a Sudoku solver. While we will give you the skeleton of what your solver class should look like, the exact algorithm you'll be using is up to you. The old "Hey, we have computers, so lets simply throw computing power at the problem and try every possibility!" is as valid as "Let's think about the problem and develop a smart solution."

Write a class `SudokuSolver` in the same package as the classes in Assignment 7.1. The class should provide the following features:

- A constructor that takes the `Sudoku` instance to be solved. The constructor should make a copy of that instance and put the copy into an instance variable.
- A method `solve()` that doesn't return anything, but simply executes the solver.
- A method `isSolved()` that returns a `boolean` value indicating whether the Sudoku is solved or not. A Sudoku is solved if it is valid and if none of its cells are empty.
- A method `getSolution()` that returns the solved Sudoku, or `null` if the Sudoku wasn't solved. (Note that if an unsolved Sudoku is passed to the solver, but the `solve()` method wasn't called yet, the Sudoku is not solved yet and this method should thus return `null`.)

Note that you're free to implement more methods. However, keep them private to avoid confusion as to how to use your class.