ADPRO **IT University, E2015**

Miniproject on Java streams
Thursday 8 October 2015

Last update 2015-10-07

Goal of the miniproject

The goal of the miniproject is to make sure that you can use Java 8 (parallel) stream pipelines and parallel array operations through mostly-functional programming.

Do this first

You will need a Java 8 development kit and a development environment of your choice, such as Eclipse or Emacs. Java 7 will not work.

Get and unpack the example code in zip file adpro-java-streams.zip on the course LearnIT page.

Get and unpack some sample sudoku puzzles in zip file adpro-sudoku-puzzles.zip on the course LearnIT page.

Question 1: Stream-related exercises

Hand in your solutions to exercises 6.3, 6.4 and 6.5 from the 1 October 2015 lecture.

Question 2: Solving sudoku puzzles

The example source file AdproSudoku.java contains much of the infrastructure of a sudoku puzzle solver and sudoku puzzle generator, but some parts are missing. In this and the following questions you must write the missing parts, demonstrate that they work, and measure their efficiency.

1. Complete the `solutions` method in class State so that the program can solve sudoku puzzles, when called like this:

   ```
   java AdproSudoku solve < s0
   ```

   from the command line, to print and solve the sudoku puzzle in file `s0`. Check that the printed solution is indeed a solution: each row contains the numbers 1–9, each column contains the numbers 1–9, and each 3x3 block contains the numbers 1–9.

2. Apply the solver to the sudoku puzzles in files `s1`, `s2`, `s3` and `s4`. All should have exactly one solution as required of a sudoku puzzle.

3. Change a single cell in one of the sudoku puzzles and check that your solver finds no solution.

Question 3: Generating sudoku puzzles

In this question you must complete the generator of sudoku puzzles, by finishing methods `makeSkeleton`, `make`, `eraseEnough` and `eraseOne`. As outlined in the lecture, we generate sudoku puzzles by (1) generating full sudoku solutions, and (2) randomly erasing cells from the full solution to obtain a sudoku puzzle. Step (1) has two substeps: (1.1) generate a random skeleton, and (1.2) use the sudoku solver to turn that into a full solution, if possible. See also the lecture slides.

1. Implement substep (1.1) by completing the skeleton generator in method `makeSkeleton`. The fragment given fills in block 0 (upper left corner) with a random permutation of the numbers 1–9. Your task is to fill in the rest of row 0 with a random permutation of the numbers not already used in row 0; fill in the rest of column 0 with a random permutation of numbers not already used in column 0; fill in block 4 with numbers not already used in block 1 and 3 (that is, same column band and row band); fill in block 8 with numbers not already used in block 2 and 6 (that is, same column band and row band); and fill in block 5 with numbers not already used in block 2 and 3 and 4. See also the lecture slides.

   Generate and print some skeletons.

2. Use the solver to turn a stream of skeletons into a stream of full sudoku solutions, and print some of them. Check that they are indeed sudoku solutions.

3. Complete method `eraseEnough` in class State so that it produces a singleton stream consisting of the current state (representing a sudoku puzzle) if it has enough blanks already. Otherwise it calls method `eraseOne`, which produces a stream of puzzles with one more blank cell, and, for each of these, recursively calls `eraseEnough` to produce a stream of puzzles with enough blank cells and then takes at most one element from that stream. (Because we do not want multiple puzzles that have the exact same solution). See also the lecture slides.

4. Complete method `erase` in class State to that it produces a stream of sudoku puzzles with one more erased cell than the given one, while ensuring that every such erasure still has exactly one solution. You can do this by (a) obtaining an array list of all nonblank cells, (b) randomly permuting that list, (c) create a stream from the permuted list of cells, (d) map the given `erase` method over the stream of cells to obtain a stream of one-more-cell-erased sudoku puzzles, and (e) filter out those that have exactly one solution. In (c) it may be useful to limit the stream of cells to 2, 3 or 4 to reduce the amount of branching in the search for a good puzzle. In (e), use the solver to find and then count the solutions. See also the lecture slides.

5. Finish the `make` method so that it uses the three methods completed above; the lecture slides show how.

6. Use the completed program to generate and print some sudoku puzzles with 50 blanks and 0 backtracking, along with their solutions, by running the program like this from the command line:

```
java AdproSudoku 50 0
```

7. Use the completed program to generate a LaTeX document `adpro-10.tex` containing 10 sudoku puzzles with 50 blanks and 0 backtracking, along with their solutions, by running the program like this from the command line:

```
java AdproSudoku 50 0 latex 10 adpro
```

8. Parallelize the sudoku puzzle generator by inserting `.parallel()` in a suitable place in the `make` method, and rerun the generation of sudoku puzzles:

```
java AdproSudoku 50 0
```

   Can you see any difference in generation speed?

9. Why does the `print` method in class State take a lock?

Question 4: Generating sudoku solutions by permutation

The purpose of this exercise is to replace step (1) in the sudoku generation process from Question 3 by a different one. The idea is to start with this trivial full solution:

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

and then apply random transformations to it, while preserving the property of being a sudoku solution. In particular, the following transformations can be applied to a sudoku solution, and the result will still be a sudoku solution:

- permutation of the rows within a row band, for all three row bands
- permutation of the columns within a column band, for all three column bands
- permutation of the three row bands
- permutation of the three column bands
- permutation of the nine numbers 1 2 3 4 5 6 7 8 9, eg to 2 4 5 7 3 8 6 1 9

Thus from a single solution one can produce a total of $(3!)^3 \cdot (3!)^3 \cdot 3! \cdot 3! \cdot 9!$ = 609,499,054,080 solutions. One could further consider rotations and horizontal as well as vertical mirror images, but these can already be obtained by combinations of the above operations, so add no new

1. Write five methods to implement each of the five transformations above. Each method should be an instance method in class State and produce as result a copy of the given State object (not just an updated State object) where the transformation has been randomly applied.

   Hint: You may want to implement a `transpose` method that returns a fresh State with rows and columns swapped, that is, where the 9-by-9 array is mirrored around the main diagonal. Then you only need to implement the two row-transforming methods (and the 1–9 permuting one) because the two random column-transforming methods can be expressed by a transpose followed by a random row transformation followed by a transpose.

2. Combine the five methods into a method `randomTransform` that takes as argument a State `s` and returns as result a randomly transformed copy of `s`. It may do that by selecting a transformation at random and then applying it to `s`.

3. Write a generator `makeSolutions` of random sudoku solutions that produces an infinite Stream<State> by starting with the trivial full solution shown above, and repeatedly applying the `randomTransform` method. Use a predefined method on class Stream<T>.

   Print some of the generated sudoku solutions. The first ones probably do not look very random. Change your program to print only the later, somewhat more random, ones.

4. You may notice that although many many different sudoku solutions can be generated from a single one by the above process, they still have some annoying similarities, such as the numbers 2 5 7 appearing together in every block in a block band. This (aesthetic) problem can be avoided by applying value-dependent transformations:

   - Within a row band, one can swap one block-internal 3-column with another block's block-internal 3-column provided they have the same set of elements. Obviously this preserves the set of numbers in each of the two blocks, in each of the two affected 9-columns, and in each of the row band's three affected 9-rows.
   - Similarly, within a column band, one can swap one block-internal 3-row with another block's block-internal 3-row provided they have the same set of elements.

   Write methods `swapWithinRowbands` and `swapWithinColbands` that determine whether any such transformations are possible, and if so applies them.

5. Add the two new methods to the set of transformations performed by `randomTransform`, so that it always performs such a swap after another transformation, if possible.

   Print some generated sudoku solutions; do they look sufficiently random?

6. Describe and implement other ways in which the random transformations you have implemented could be used to generate random-looking sudoku solutions.