# 2IP67.5 – Instruction 3

## Preliminaries

Remember:

- Start every file you write with a line of comment with your name(s) and the date.
- Work in pairs, if possible.
- Keep your code neat from the beginning on, with proper indentation, etc.
- Have each exercise checked by one of your instructors.

Furthermore:

- Use names that start in lowercase for variables.
- Use all uppercase for names of variables that do not change after initialisation (constants).
- Use assertions where appropriate.

## 3.1 Square root

You are going to write a program that computes the *integer square root* of an integer number.

1. Write a program that reads, after a prompt, a non-negative integer from input, stores it in $N$ and then computes the integer $x$ that is the square root of $N$, rounded down to the nearest integer. In other words:

   $x^2 \leq N < (x+1)^2$

   Do this by starting from 0 and increasing $x$ until you find the right value. Don't use the built-in square root function of Java. Use $x * x$ to compute $x^2$. At the end, print the values of $N$, $x$, and $x^2$.

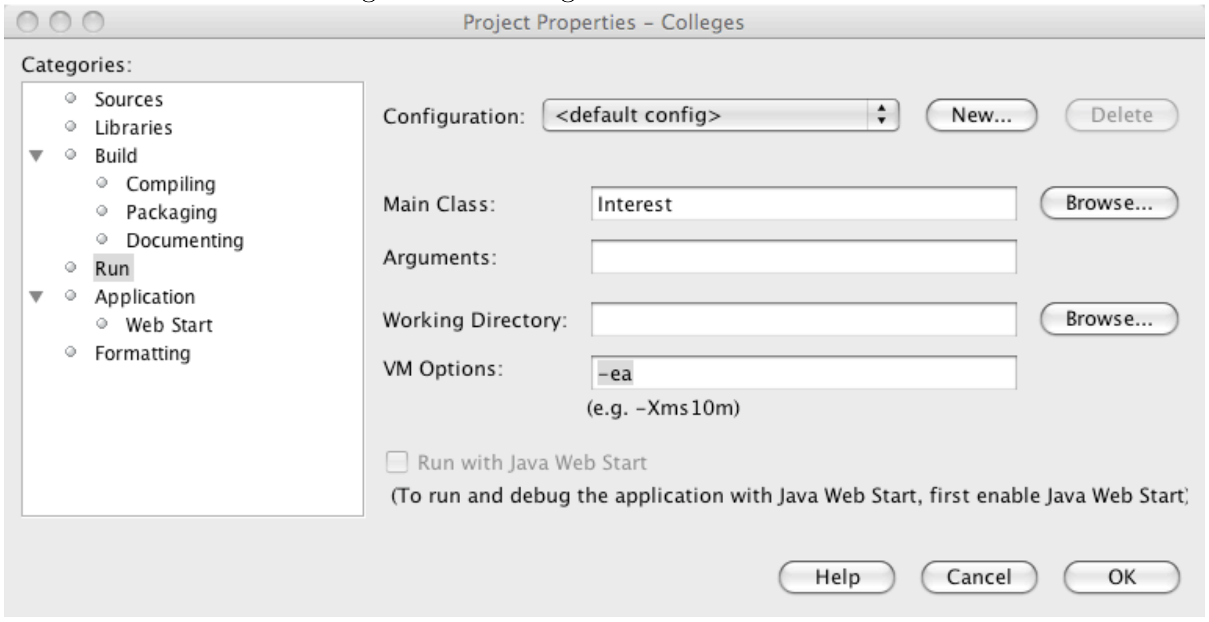
   Include `assert` statements between all your statements. Don't use commented asserts.

2. By default, Java doesn't execute assert statements. It only compiles them. You can change this behaviour by changing the project properties in Netbeans. Open the Project Properties (via the menu or right clicking your project) and add the string "-ea" to the VM Options in the Run category. (see figure 1). Insert an assertion that is false to see that it works.

3. Is your annotation (all assertions) enough to prove that your program indeed calculates the correct value for $x$?

4. * The program above is not as efficient as possible. It tries all values from 0 to the square root. We call this a *linear search* approach. Try to find a better solution, one that tries fewer values.

Figure 1: Enabling assertions in NetBeans


Project Properties – Colleges

Categories:
- Sources
- Libraries
- Build
  - Compiling
  - Packaging
  - Documenting
- Run
- Application
  - Web Start
- Formatting

Configuration: <default config>

Main Class: Interest

Arguments:

Working Directory:

VM Options: -ea

(e.g. -Xms10m)

Run with Java Web Start

(To run and debug the application with Java Web Start, first enable Java Web Start)

## 3.2 Linear Cellular Automata

A *cellular automaton* consists of a grid of cells. Each cell contains a symbol, from a finite set of symbols. The content of the whole grid, called a generation, changes in one step, according to some, usually simple, rules.

The rules for computing a new generation satisfy four requirements.

(a) The rules use only the current generation to compute the next one.

(b) The rules define one value per cell.

(c) The rules are the same for each cell.

(d) The rules define the new value based on the values in a set of cells in fixed positions with respect to the cell the value is computed for; such a set is often referred to as the *neighborhood* of the cell.

This exercise concerns *linear* cellular automata, i.e., automata with a one dimensional array of cells as a grid. The neighbourhood of a cell are the immediate neighbours in the array (two or one). Furthermore, just two different symbols are used, referred to as "occupied" and "empty" (or not occupied). In the book *A new kind of science*, Stephen Wolfram, the creator of the Mathematica tool, shows that such simple automata can display very interesting behavior.

The exercises show the effect of some rules.

Write a program, a class, along the following general lines.

(a) Represent the grid as an array of booleans, called a *line*. This is a natural choice in the case of two values, allowing simple coding of the rules: the values in the cells can be directly used to form boolean expressions that can be used in guards.

(b) To avoid having to deal with head and tail borderline cases for the neighborhood definition, add an extra cell to the array at the beginning and at the end. These cells are not displayed and are always empty (*false*, in our case).

(c) Unless otherwise stated, in the initial configuration all cells are empty, except for one occupied one cell in the middle (or close to the middle).

(d) To get a good visualization of the various generations, display the relevant part of the array (i.e., ignoring the end-cells) as an output line: a space is output for a cell that contains `false`, a * for a cell that contains `true`. You may not like this visualisation, so to be able to easily modify the visualization, use variables `OCC` (occupied) and `EMPTY` to store the space and the *, respectively.

(e) Note, that the new value for each cell should be computed using the *current* values in `line`. Therefore, the newly computed values should be stored in a separate array, `newLine`, till they are all computed. Only then the values in `newLine` can be copied to `line`, overwriting the old ones. (Other approaches are possible here.)

5. Write a first version that sets the length to, say 121, initializes (middle cell is occupied, rest is empty), and draws the array and then computes and draws, say, 60 generations, where the next generation is simply a copy of the previous one.

6. We define automaton $A$ by the following rules:

   (a) if any of the immediate neighbours of an empty cell is occupied, the cell becomes occupied;

   (b) if a cell is occupied, it remains occupied.

   Think first what you would expect as output. Then build a program that behaves as this automaton. Does it match your expectations?

7. Build automaton $B$ (you can best copy your program, use a new class name), defined by the rules:

   (a) if a cell is occupied, it remains occupied only if both of its neighbours are not occupied;

   (b) if the cell is not occupied, it becomes occupied only if exactly one neighbour is occupied.

8. For automaton $C$ the rules are:

   (a) if a cell is occupied, it remains occupied only if both of its neighbours are not occupied or if only its right neighbour is occupied;

   (b) if the cell is not occupied, it becomes occupied only if exactly one neighbour is occupied.

   Note that this automaton is asymmetric. Had you been able to predict this output?

## 3.3 More fun with automata

1. Try some other rules. When do you get complex behaviour, when boring behaviour?

2. Let the program stop when a stable generation is reached, i.e., a generation that is identical to the previous one.

3. Let the program stop when a repeating generation within 5 previous generations has been reach.

4. Change the representation so as to not just indicate that a cell is occupied, but a number stating how long this cell was occupied uninterruptedly (if the number gets too large to put in the cell, use a different symbol, like #, to indicate that it is older than, e.g., 9).

5. In a two-party version, occupied cells exist in a two symbols variety. A player wins when all symbols of the opponent's kind are eliminated. When an empty cell becomes occupied, its symbol is determined by the dominating symbols of its neighbor occupied cells. Start with a random or pre-chosen starting pattern with half the occupied cells containing symbols of each kind. After one iteration, the first player may add one symbol of his own and remove one symbol of his opponent. After the next iteration the other player can do the same, and so forth.

---

Kees Huizing – c.huizing@tue.nl – 15 Sep, 2009