Maastricht University
Faculty of Humanities and Science
Department of Knowledge Engineering

Pre-existing Knowledge Assignment
Computer Science 1

Block 1.1:	Computer Science 1
Code:	KEN1120
Examiner:	Kurt Driessens
Date:	September 7th, 2016

Some hints/tips/requirements:

1. All programming should be done in Java.
2. If you notice during the reading of these assignments that you have overestimated your programming proficiency, it is probably better to simply hand in a blank solution than to try and fool me. I consider knowing what you know as one of the most important things you can learn.
3. Comments in your code will only help and will never do harm.
4. There are 3 pages and 3 assignments in this document.

Assignment 0:
In your email containing the commented solutions to the assignments below, describe to me where and how you learned to program, for example, whether you learned it in school or on your own at home. Also include a brief list of programming projects you've worked on (e.g. large assignments from school).

Assignment 1:
Write a program that simulates Conway's game of life.

(Taken from Wikipedia) The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The "game" is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

The universe of the Game of Life is an infinite two-dimensional grid of square *cells*, each of which is in one of two possible states, *live* or *dead*. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

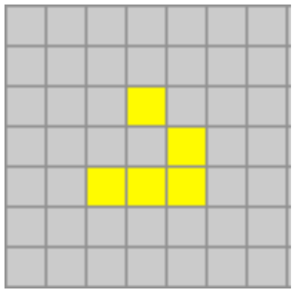
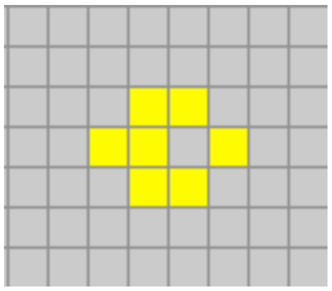
The initial pattern constitutes the *seed* of the system. Applying the above rules simultaneously to every cell in the seed creates the first generation. Births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a *tick* (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

You don't have to implement the Game of Life on an infinite grid. Use instead a size around 40-50 cells wide and high.

<u>**Important:**</u> Since many GoL implementations are available online **you are required to** make use of a GameOfLifeUI class (to be found in attachment to the email). This class offers the following methods:

- GameOfLife(int height, int width, int squaresize): this is the constructor for the class, where height and width specify the number of cells the window will display and squaresize determines the size of each cell. Good values are width and height around 40-50 cells and squaresize around 10-15 pixels. Constructing an object of the class will also cause the window to automatically appear, with all cells displayed in grey.
- setCellToYellow(int x, int y): this method makes sure that the next time the window is updated, the cell on row x and column y will be displayed in yellow. As standard in Java, cell-indexes range from 0 to {width,height}-1.
- setCellToGrey(int x, int y): this method makes sure that the next time the window is updated, the cell on row x and column y will be displayed in grey. As standard in Java, cell-indexes range from 0 to {width,height}-1.
- toggleCell(int x, int y): this method toggles a cell between being displayed as yellow and grey. As standard in Java, cell-indexes range from 0 to {width,height}-1.
- update(): redraws the window on screen. Only when update() is called will changes to the color values of cells be displayed.

Initialize your program with some interesting seeds such as a glider or a small exploder as shown below:

Glider:  Exploder: 

If updates of the GoL window happen too fast, you can use the following statement inside your execution loop to slow it down:

```
try {
        Thread.sleep(<some_delay>);
}
catch (Exception e) {}
```

Assignment 2:
It is known in mathematics that if you take any natural number *n* (i.e. 1, 2, 3 …) and apply the following rules:

- if n is even: divide n by 2
- if n is uneven: multiply by 3 and add 1

and continue to do this with the new number you obtain, you will at some point always reach the number 1.
Implement a Java Program that simulates this process, i.e. that reads in a integer number using the **Scanner class** (java.util.Scanner) and then uses the rules listed above to build a sequence of numbers until it reaches the value 1. Your program should print out all the numbers that make up the sequence together with, at the end, the number of steps it took to reach the value 1.
Important: (meaning you will not pass this test otherwise) make sure your program computes this series using a **recursive** method.

When finished, zip up your commented code and email it to kurt.driessens@maastrichtuniversity.nl *before* 20h today!!