AP(IT) Assessed Exercise 2017

Dr. Simon Rogers, January 2017

Administration

This exercise is worth 30% of your final grade. Note that this means that if you do not submit something you cannot get credit for the course (regardless of your exam results).

Submission will be via Moodle. The deadline is 4pm on Monday 20th February 2017. Submissions after this time will incur the standard penalty (two bands per day, or part of day).

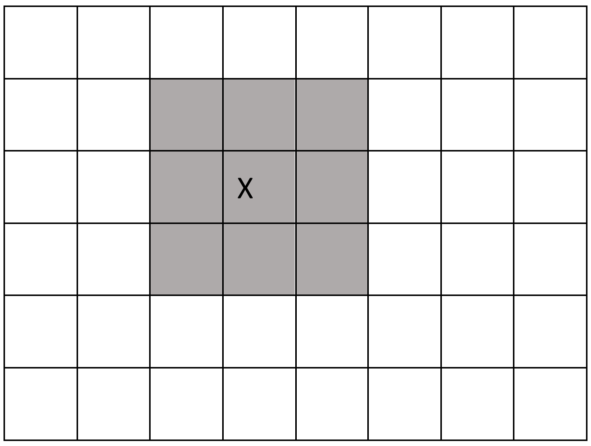
I am happy to answer questions about the exercise in lectures and on the Moodle and Piazza forums. I will not be willing to answer questions about it in the lab (I think it's fair that questions are only answered in a space where **everyone** has access to the answers).

Background

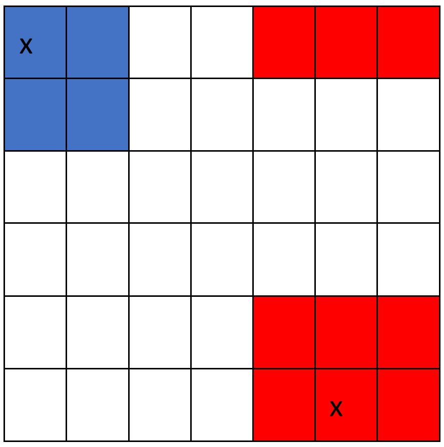
Cellular automata is the name given to a family of approaches that model agents on a grid. Your task is to design and build a cellular automata system in Java, using threads.

Imagine you have two species of creature that live in a one dimensional grid world with R rows and C columns.

- Each space in the grid can either be unoccupied or occupied by one creature from species 1 or one creature from species 2.
- Each creature when born is given a lifespan which should be generated randomly between 0 seconds and a maximum time dependent on the species type.
- Each creature when born is given a fitness value between 0 and 1 which is, again, species dependent.
- Maximum lifespan and fitness are fixed for a species at the start of the simulation and do not change thereafter. Suggested values:
  - species 1: max lifespan = 10 seconds, fitness = 0.8
  - species 2: max lifespan = 5 seconds, fitness = 0.4
- At the end of its lifespan, a creature reproduces and then dies.
- When reproducing, the creature can potentially place a child in any of the 9 grid squares in its direct neighbourhood (e.g. in any of the shaded squares for the 'x' square below:



- For each neighbouring square, the reproducing behaviour depends on whether it is already occupied. If unoccupied, a child is placed with probability equal to the fitness of the parent. If occupied, a child is placed with probability equal to the fitness of the parent minus the fitness of the occupier (when searching neighbours the parent's square (i.e. 'x') should be considered empty). *See hints below for dealing with probabilities*.
- If a parent places a child in an occupied square, the previous occupier should be murdered. If a creature is murdered, it **doesn't** reproduce.
- Each of the creatures in your simulation should run on **its own thread**.
- Edge cases: your system should handle two cases – where the edges of the grid are the edges of the world (blue example below) and where the grid wraps around (red example below).



Tasks

1. Design (worth 40% of the assessment)

Design the class hierarchies that will be used in your system. You can submit this design as either a textual description or a UML diagram. Note that in either case, your description should be clear enough that another developer could use your design to understand your implementation and understand **why** you have chosen to design things in this way. If using UML feel free to add notes / annotations to your diagram.

A good design should be modular and allow for substantial code re-use, and minimise code duplication. i.e. I'm looking for how well you can abstract the entities in the problem into a class hierarchy.

2. Implementation (worth 60% of the assessment)

Implement your system in Java.

- Your code should be clear and well documented (JavaDoc level of documentation is not required but comments should allow me to understand what's going on).
- Your implementation should include a class with a main method called TestWorld.java that I can run and watch the simulation happen.
- Your implementation should output the world (and the total population) every 0.5 seconds (text output in the console is fine, please don't write a Swing GUI)…perhaps using one line for each row in the grid and the characters '1', '2', and '-' to represent whether a grid square is occupied by a creature of species 1, species 2, or it is empty.

- In your TestWorld main method, you can either have the world with edges, or the wrap-around world enabled. Please also show how you would use the other but comment it out on submission.
- Your implementation should take necessary precautions to avoid multiple threads simultaneously changing a shared object.

What should you submit IMPORTANT

Submit a single .zip file. This should be one zipped directory. In the directory should be all of the .java files required for me to compile the project as well as a **.pdf** documenting your design.

When I am marking I will go into the directory and type:

```
>javac *.java
```

```
>java TestWorld
```

It would be sensible for you to check that these commands actually work within the directory you zip and submit.

Please **DO NOT:**

- Submit code in a complex directory structure as is produced by Eclipse.
- Submit code that has 'package' lines at the top of the .java files. If these are produced by Eclipse please chop them off in your submitted code.
- Compress your code with some other compression tool (7-zip, rar, tar, etc).
- Submit a word document. If you make your design in word, convert to .pdf before submission.

Mark breakdown

To help you prepare your submission, here is the mark breakdown I will be using to do the assessment:

Design (40% of total)

- Completeness [10%]
- Clarity [10%]
- Good use of object orientation (e.g. Abstraction / Hierarchies etc) [20%]

Implementation (60% of total)

- Operational correctness [20%]
- Clarity of documentation (i.e. comments) [15%]
- Correct use of threads [15%]
- Code 'quality' (e.g. sensible variable names, correct use of `final`, etc [10%]

Some hints

Probabilites:

To decide if a child should be placed in an **empty** square (assuming the creature's fitness is stored in 'fitness':

```
If(Math.random() <= fitness) {
    // create child
}
```

To decide if a child should be placed in a square occupied by another creature (‘creature’):

```
If(Math.random() <= fitness – creature.getFitness()) {
    // create child
}
```

Murdering:

When you murder a creature, try not to leave its thread running in the ether somewhere. In the course we have covered how you can interrupt a sleeping thread and how the sleeping thread can detect it has been interrupted.