Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

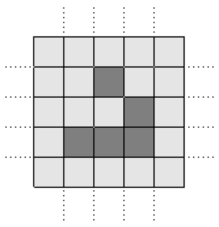
Lab Handout

Outline

In this lab you will revisit the implementation of Conway's *Game of Life* seen in an earlier lab. The purpose of this lab is to consider the implementation of that program using Java 8 *lambda expressions*, rather than using *inheritance*. You will find that the implementation using lambda's is more concise and simpler to write. Before the end of the lab, you should submit your solutions via the *online submission system* which will automatically mark it. You may submit as many times as you like in order to improve your mark and the final deadline will be Friday @ 23:59.

Conway's Game of Life

Recall that Conway's *Game of Life* is a simple cellular simulation devised by John Horton Conway (a British mathematician) in 1970. The Game of Life has been studied extensively since then for both scientific interest, and also just for fun. For example, it has been shown that the problem of deciding whether a given state of the game will ever reach the empty board is *undecidable*. A simple example is:



Here, cell's marked in black are "alive" whilst those in white are "dead". Each cell has *eight* neighbours, which are the cells that are *horizontally*, *vertically*, or *diagonally* adjacent. There are four rules for governing the transition between life and death in the Game of Life:

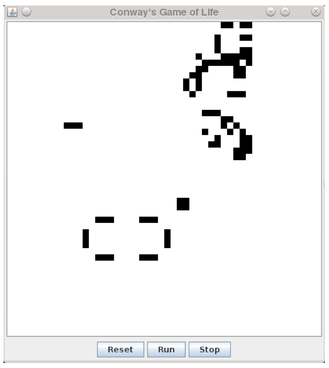
1. Any live cell with fewer than two live neighbours dies, as if caused by *under-population*.
2. Any live cell with two live neighbours continues on to the *next generation*.
3. Any live cell with more than three live neighbours dies, as if by *over-population*.
4. Any cell with exactly three live neighbours becomes a live cell (dead cell becomes alive, and alive cell continues), as if by *reproduction*.

In this lab, we are going to play with an extensible implementation of the Game of Life. The implementation is extensible because new rules can easily be written to control the game in different ways. The implementation also supports more complex forms of the Game, as it allows more than two states ("alive" or "dead") for each cell. Specifically, each cell has 10 different "aliveness"/"deadness" states, represented as integers between 0 and 9. As the state number increases, the cell becomes more towards to "dead". State 9 corresponds roughly to "dead" and state 0 indicates that the cell is the most "alive". The meaning of the intermediate states is left up to the rule implementor to simulate *age*, *sickness*, *happiness*, etc.

Getting Started

To get started, download the `lambda-conway.jar` file from the course website and import this into an Eclipse project. The Game of Life is provided with a Graphical User Interface, which you can run by right-clicking on `GameOfLife` and selecting "Run As⟶Java Application" (ignore `CellDecayGameOfLife` for now). You will see a white board with no alive cell.

Then, you can place live cells on the board by clicking on the corresponding locations. For example, you can keep clicking on the board to obtain a window similar to the following:



After placing the live cells, you can start the simulation by pressing "run". Take a few minutes to play around with the game. You should notice that not all rules are implemented and we will begin by fixing them in Activity 1.

NOTE: it is useful to take the time and revisit the implementation of Conway using inheritance from an earlier lab. You will notice that the `rules` package is no longer required in the implementation for this lab. *In fact, the number of classes required has reduced by four in total.*

Activity 1: Conway's Missing Rules

The implementation of the Game of Life provided does not properly implement rules (3) and (4) from above. The goal of this activity is to complete these rules. The rules themselves are implemented using lambda expressions. For example, Conway's *underpopulation* rule is implemented as follows:

```
public static final Rule[] ConwaysOriginalRules = {
      // The underproduction rule
      (Pair<Point,Board> p) -> neighbours(p) < 2 ? DEAD : null,
      ...
};
```

This implementation of the rule is very easy to compare with the original English description. You will find this rule in the class `GameOfLife` and it should provide a useful guide which you can follow. Having completed rules (3) and (4), you should find that all tests in `GameOfLifeTests` now pass.

Activity 2: Cell Decay

The goal now is to implement a variation on the basic rules for the Game Of Life which models the age of a cell. Our updated rules for the game are:

1. Any live cell with fewer than two live neighbours gets *older*, as if caused by *under-population*.
2. Any live cell with two live neighbours continues *in stasis* to the *next generation*.
3. Any live cell with more than three live neighbours gets *older*, as if by *over-population*.
4. Any cell with exactly three live neighbours gets *younger*, as if by *happiness*.

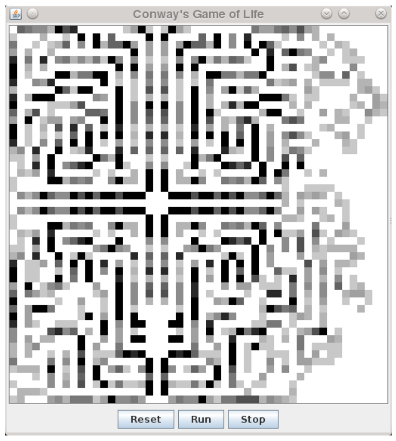
The age of a cell is determined by its current state which, if you recall, is a value between 0 and 9. Here, 0 is youngest whilst 9 is oldest. We consider that cells of age 9 are *dead*, whilst all others are varying forms of *alive*.

When a cell gets older, its state value is increased by 1. When a cell gets younger, its state value is decreased by 1.

This variation should be implemented in `CellDecayGameOfLife`. The corresponding test cases are in `CellDecayTests`. You can run the simulation as a Java Application with `CellDecayGameOfLife`, though at this stage you should find that it does *nothing*.

What to do. Your aim is to implement the above rules for cell decay. You should add the rules you create to the `CellDecayRules` array in `CellDecayGameOfLife`, as was done for the original `GameOfLife`. Having done this, all tests in `CellDecayTests` should now pass.

Having completed this part, you should find that the balance has tipped and cell growth expands *quickly*. This is because cells stay "alive" for much longer than before. For example, the system will often produce something like this:



Submission

Your lab solution should be submitted electronically via the *online submission system*, linked from the course homepage. The minimum set of required files is:

```
swen221/lab9/CellDecayGameOfLife.java
swen221/lab9/GameOfLife.java
swen221/lab9/model/Board.java
swen221/lab9/model/BoardView.java
swen221/lab9/model/Rule.java
swen221/lab9/model/Simulation.java
swen221/lab9/testing/CellDecayTests.java
swen221/lab9/testing/GameOfLifeTests.java
swen221/lab9/util/Pair.java
swen221/lab9/util/Point.java
```

You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable.* See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.