Übungsaufgabe – Game of Life

1. Installing JavaFX

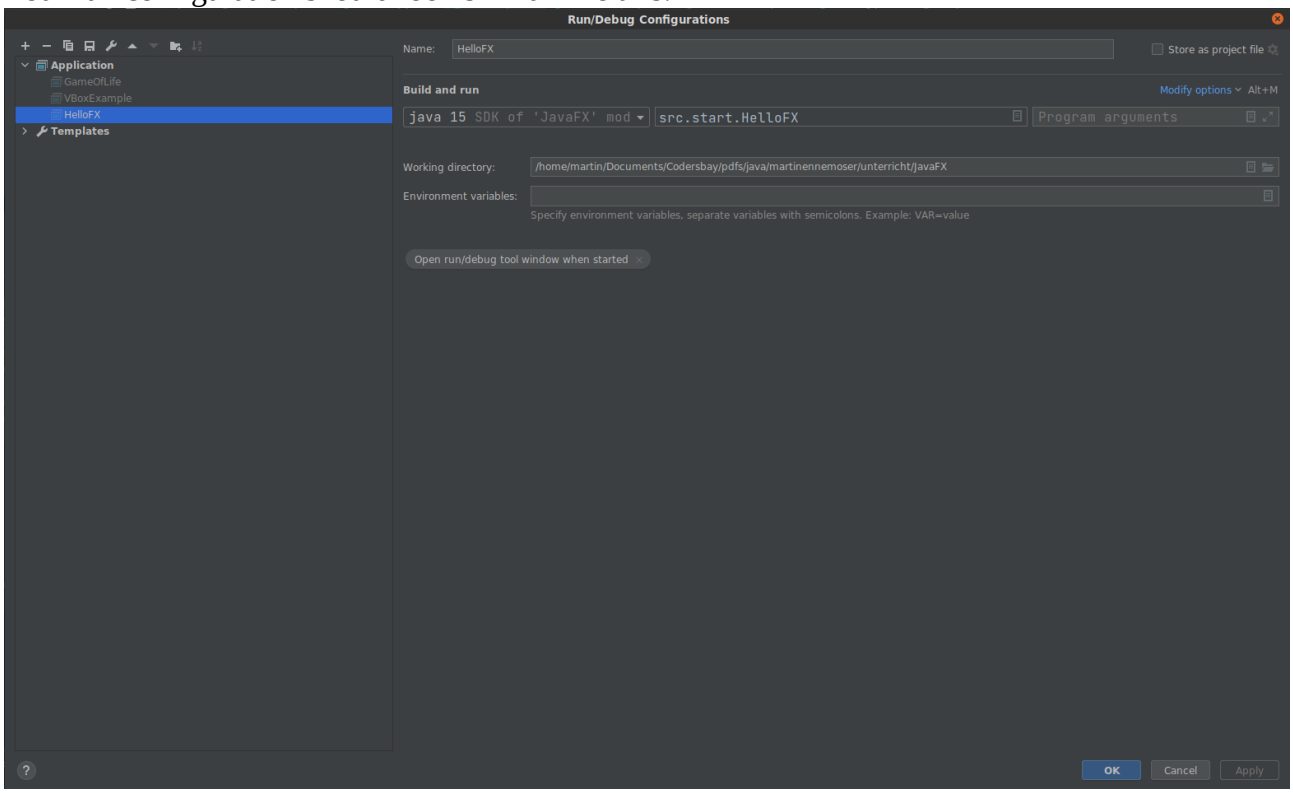
Go to https://gluonhq.com/products/javafx/, download and install the JavaFX SDK.

Open IntelliJ. Click on File → Open and select pdfs/java/martinennemoser/unterricht/JavaFX folder that is located in our Github Repository.

Then go to https://www.jetbrains.com/help/idea/javafx.html#add-javafx-lib and add the JavaFX library as described.

Next try to run the HelloFX.java. This is a tiny Java application that checks if the installation was successful.
Create a new launch configuration and add additional VM options as describe here: https://www.jetbrains.com/help/idea/javafx.html#vm-options.

Your run configuration should look similar like this:



Run the application as described here: https://www.jetbrains.com/help/idea/javafx.html#run.

If everything works as expected, you should get the following output:

A new window opens and shows that the installation was successful and that we have JavaFX support in our project now.

2. Game of Life

In this exercise, we will implement a game called **Game of Life**.
The idea of this game is to simulate the births and deaths of cells over a period of time. Cells are visualized in a 2 dimensional raster and can have only one of two states: live and dead. At every iteration, a new generation of cells gets created. Live cells are visualized as tiny red rectangles in our simulation window (see images below). We assume that our 2 dimensional raster is always square.

Every cell interacts with its eight neighbours, 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 by underpopulation.
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 overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.
5. Any cells outside the boundaries of the cell array are considered dead.
6. In all other cases, the state of the cells does not change.

Births and deaths of cells occur simultaneously, and the discrete moment at which this happens is sometimes called a tick. Each generation is only dependent on the preceding one. The rules continue to be applied repeatedly to create further generations.

This game is a so called zero-player game, meaning that it runs autonomously requiring no user input.

Fun fact: Game of Life is Turing Complete, meaning we can simulate every computer program using Game of Life (including Game of Life itself).

Implementation

Most of the code is already available. Your only task is to implement a new class which implements the `Transitionable` interface. The `Transitionable` interface possesses a method `nextState()`. This method create a new generation of cells and returns them as an array of cells.

An example how this works is already given by a class called `Blinking`. This class simply inverts the state of the cells (from alive to dead and vice versa).
When you are finished with your class, replace the line

```
GameOfLife.this.cells = new
Blinking().nextState(GameOfLife.this.cells);
```

with

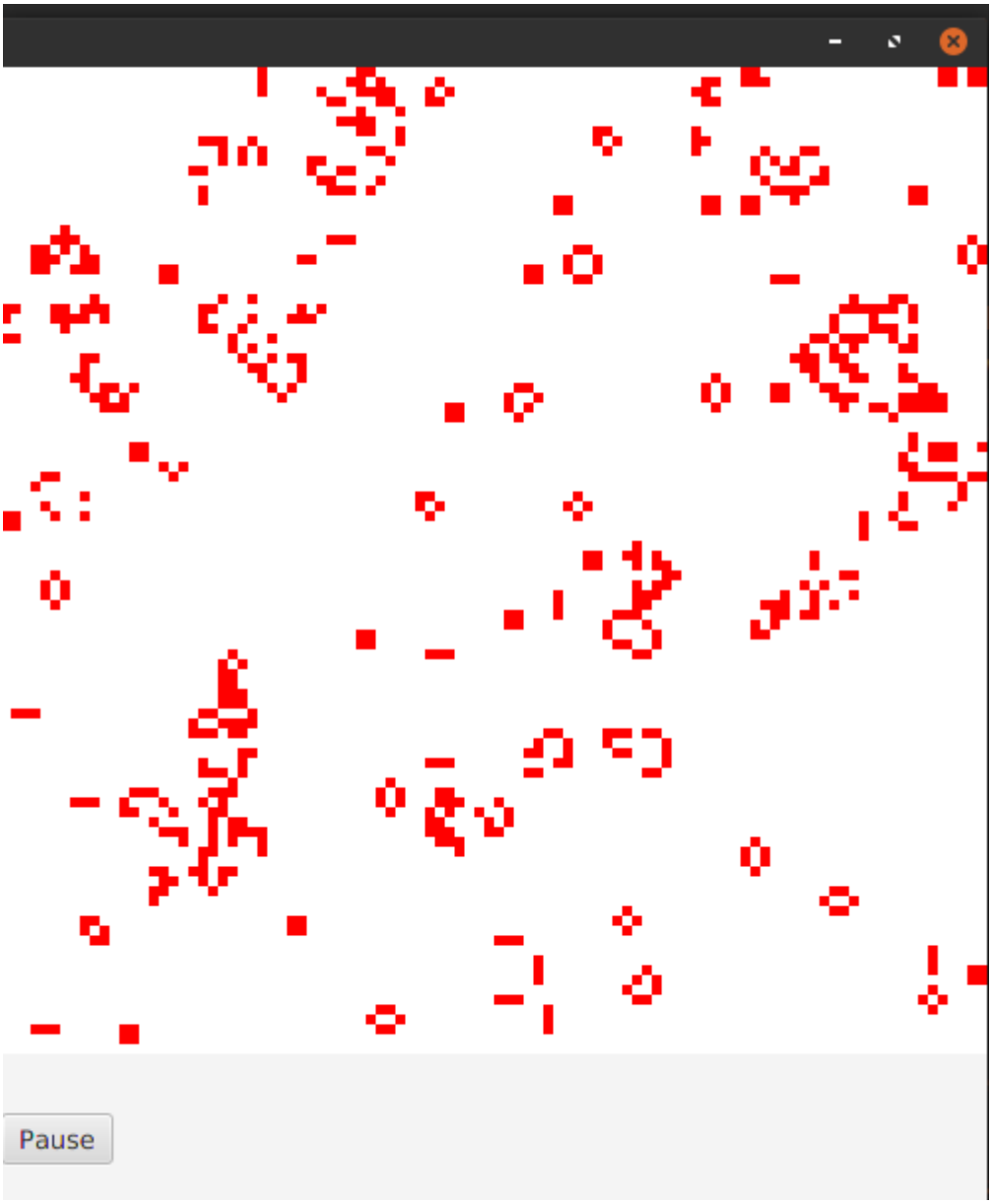
```
GameOfLife.this.cells = new
<yourclass>().nextState(GameOfLife.this.cells);
```

Since we use a seed in the random generator, we always get the same result on every application restart. If you want to get different results on every application restart, you can remove the seed argument in the constructor of `Random`.

IMPORTANT: Note that the next status of each cell is updated/computed concurrently from the old state. This means that you have to create a separate cell array to hold the new state (instead of using in-place operations on the old cell array).

The *Pause* button is simply for your convenience so that you can investigate a generation in more detail if you want.

Here are some screenshots how generations look over a period of time.



Martin Ennemoser