

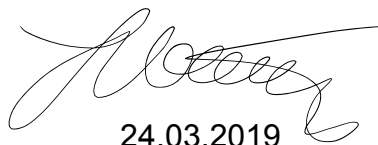
## CS105 Exemption Project 2018-19

Nikolay Tsanov, 201847404, BSc Hons Computer Science

University of Strathclyde

**Except where explicitly marked to the contrary by quotation or reference, what follows is all my own work. I have not knowingly allowed others to copy my work.**

Nikolay Tsanov



24.03.2019

**THE** LEADERSHIP & MANAGEMENT AWARDS 2017 | **Winner**  
Workplace of the Year

**THE** AWARDS 2016 | **WINNER**  
BUSINESS SCHOOL OF THE YEAR

**THE** UK Entrepreneurial University of the Year 2013/14  
UK University of the Year 2012/13

## Brief description of the original project

---

The purpose of the original foxes-and-rabbits system is to simulate interactions between foxes and rabbits within a certain area(window). The main interactions consist of rabbits being eaten by foxes, therefore foxes are considered “predator” and rabbits as “prey”. Moreover, the different animals have different age to which they can live and if they die, they are removed from the simulation. Foxes and rabbits can breed and give birth, however, they need to have a certain age, to begin breeding. Both of the animals have a limit of births and have a different probability of breeding. Apart from death caused by age, the foxes can die when their hunger value goes down, so they need to consume rabbits to stay alive.

The **FnRMain** class is considered as an entry point to the simulation. Its purpose is to make an instance of the **Simulator** and pass the depth and width of the window (area) that will be created for the simulation. If the program runs till this point, a window with a grid containing many small squares will be shown. The yellow square represents a rabbit at that location, the blue one shows a fox and the empty square (the one without a specific colour) is used to denote an empty location (one without an animal). After an object of the **Simulator** class is created, the simulate method is called on that object with an argument representing the steps for which the simulation will run.

The **Field** class is used for creating the rectangular grid, where each location can store only one animal. The grid is represented by a two-dimensional array of type **Object** (it is inherited by all of the classes, therefore the array can store any class). Inside the **Field**'s constructor, the two-dimensional array is initialized with a certain depth and width. Inside the class, there is a method for clearing the whole field or clearing a specific location (it accepts an object of type **Location** which holds the position (row and col) of the animal). It is a common occurrence inside this class to see method overloading. Most of the methods are declared twice but accept different parameters (one of the declarations accepts row and col, the other one accepts an object of type **Location**, the former one makes use of the second one by creating an object of type **Location** from the row and col and passes this to the second one). The class provides methods such as place() that places an animal at a certain position inside the field. There is a method for getting a specific animal from a location or getting a free location that is adjacent to a given one (also it might make use of the **Randomizer** class to get a random location). The **Randomizer** class makes use of the **java.util.Random** class. It uses the singleton design pattern that can restrict the program to have only one instance of this class also it puts a limit to the random number that can be generated.

The **Location** class is used to represent a location in the rectangular grid (field). The constructor accepts a row and col as parameters. The most important method is equals() that check if the current location is the same as the given one.

The **Animal** abstract class is used for declaring common properties of all animals. It must be extended by the **Rabbit** and **Fox** classes (it is a superclass). Common fields shared by animals are age, alive, location. It implements methods that are also shared by all animals such as isAlive(), setDead(), getLocation(), setLocation(), getField(). The class defines a method act() which can be implemented differently by the subclasses. For the rabbit the functionality of run() will be inside the act() method, for the fox that would be hunt(). It is important for the **Simulator** to hide specific classes such as **Rabbit** or **Fox** and show **Animal** instead (code decoupling).

The **Fox** class is the predator of the simulation. They hunt for food in adjacent locations, if they get too hungry they die. The constructor creates a fox at a specific location on the field (it uses the inherited constructor from **Animal** to set the location and the alive status), it accepts a Boolean argument

randomAge that determines if the age should be random or start from 0. On each step by the **Simulator**, the age increments and hunger decrements (if the maximum age or minimum hunger is hit, the fox dies). If the fox is still alive, it can try to breed and give birth (it needs to have a certain age and have less than the maximum number of births), then it will search for food in the adjacent locations and move there (possibly killing a rabbit). If there is no place to move (overcrowded), the fox dies.

The **Rabbit** class consists of the same logic for the constructor as **Fox**. Inside the act() method, on every step, the age increases, it checks whether the rabbit is still alive and tries to give birth (if it has the minimum breeding age) in a free adjacent location. If there is no free nearby location to move to (overcrowding), the rabbit dies.

The **Simulator** class is of main interest in the simulation. If nothing is passed to the **Simulator**, default values will be used for creating an area. Within the constructor, an empty **List** of type **Animal** is created (holds all animals), a new field of type **Field** is created and the graphical interface is instantiated (**SimulatorView**). The field is cleared and populated with animals at random locations, the step counter is set to 0. The method simulateOneStep(), is the main one running the simulation (used by the other simulate methods). It increments the steps by one and loops through all the animals. The act() method on the animal is called on every step with another list (newAnimals) passed by reference as a parameter. The list is filled with newborns and added on each step to the main animals' list. In the end, it calls showStatus() from the **SimulatorView**.

**SimulatorView** is used for graphical presentation of the animals' grid. It creates a window and sets a title. Adds a label on top, the grid in the centre and population stats at the bottom. On each step, when showStatus() is called, it recalculates the stats and re-draws the positions of the animals on the grid.

**Counter** and **FieldStats** can be considered as helper classes, mainly used for statistics for the simulation. **FieldStats** uses **Counter** and creates counters for each entity in the simulation.

## Modifications and new functionality

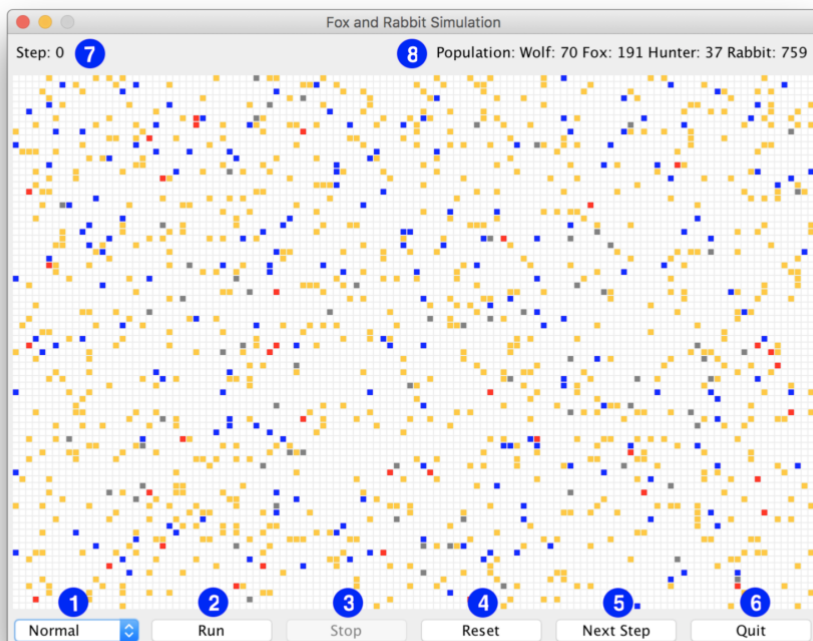
---

The first big feature that has been introduced to the foxes-and-rabbits project is the graphical user interface. Now, the user has more control over the simulation. Different components from the **Swing** library, such as **JButton**, **JPanel**, **JComboBox** and **JFrame** have been used. The simulation is controlled by a timer (**javax.swing.Timer**) with a default delay of 50 milliseconds. The **Timer** expects the current class to be an **ActionListener**, so the **SimulatorView** implements the **ActionListener** interface and also declares the method actionPerformed(). The actionPerformed() method listens for events that occur inside the **SimulatorView** class. Events are triggered when a button is pressed, an option from the **JComboBox** is selected or on every tick of the **Timer**. The **SimulatorView** is now responsible for controlling the **Simulator**. Inside the constructor of the **SimulatorView**, a new **Simulator** is created and assigned to a private field, so that it can be used throughout the class. All of the calls from the **Simulator** to **SimulatorView** have been removed. The logic for setting the colours of the different animals has also been moved to the **SimulatorView**. Currently, the main purpose of the **Simulator** class is to declare the simulateOneStep() method that is used inside the **SimulatorView** to run the simulation on every tick of the **Timer**, it is also used to reset or populate the grid with different organisms based on different probabilities that are defined as constants on the top of the class. The **Simulator** declares two new methods - getField() and getStep(). The former returns the current field with organisms and the latter one - the current step of the simulation. They are both used inside the **SimulatorView** as parameters for the showStatus() method that redraws the positions of the organisms on the grid on every tick of the **Timer**. Since the **SimulatorView** contains default values for width and height, there is no need for

passing any values to the constructor when making an instance of the class. Therefore, the main method inside the **FnRMain** class can be rewritten to contain only one line where an instance of the **SimulatorView** is created. However, if the user passes width and height, they will overwrite the default values and will be used when creating the **Simulator** inside **SimulatorView**. It is important to note, that now the **Simulator** can be used as a stand-alone class without the need for an instance of the **SimulatorView** class. The **SimulatorView** is responsible for displaying the data (this includes the stats for the simulation too) to the user, as well as controlling the simulation based on the user's interactions.

The **Animal** abstract class has been renamed to **Organism** since now there are two more entities that are displayed in a simulation - **Wolf** and **Hunter**. Wolves are animals and hunters are humans, so it is natural to say that they have common qualities of a living organism. All references to the **Animal** class has been replaced with **Organism**. Moreover, since most of the classes - **Rabbit**, **Fox**, **Hunter** and **Wolf** share the same functionality for breeding and ageing, some of the methods have been moved to the abstract class - canBreed(), getAge(), setAge() and incrementAge(), also the class defines getBreedingAge() and getMaxAge() as abstract methods that need to be declared in all child classes.

The new GUI for the simulation is presented on the next page with explanation on every component.



1. The simulation's speed can be controlled by a **JComboBox** located on the lower left corner of the window. By default, the simulation will run with its normal speed which is a 50ms delay, however, the user can choose from a variety of options such as "Very Slow" (500ms), "Slow" (100ms), "Fast" (25ms) and "Very Fast" (5ms). This option works by changing the delay of the simTimer that is responsible for the simulation to run. It changes the delay by using the setDelay() method from the **javax.swing.Timer** class. To present the options inside the **JComboBox**, they first need to be added to an array of strings

that is then passed to the **JComboBox** constructor. While the simulation is running, the box is disabled. To enable it, the user needs to stop the simulation (using the "Stop" button), then the box will be selectable again.

2. The "Run" button is the core of the simulation. After it is pressed, the simulation starts running. Since the button is represented by a **JButton** object, the actionPerformed() listens for an event that will be triggered by it. If the event is triggered, then the speed box, the "Run" button itself, the "Reset" button and the "Next step" button are disabled, immediately after that, the timer is started by calling the start() method on the simTimer object that contains the **Timer**. When the timer starts running, the actionPerformed() listens for events with source simTimer, then it checks if the simulation should continue to run by using the isViable method inside the **SimulatorView**. If the answer is no, then a message dialog is shown with the population data, otherwise it simulates one step.

3. The “Stop” button is used when the simulation needs to be halted. It is disabled when the simulation is not running and enabled back when it starts. It works by calling the stop() method on the simTimer, which causes the **Timer** to stop sending events.
4. The “Reset” button can be used to randomly populate the grid. It is enabled only when the simulation is not running, otherwise the simulation needs to be stopped first. When the event is caught by the actionPerformed() it calls the reset() method on the simulator object that is used inside the **SimulatorView**.
5. The “Next Step” button is used to simulate only one step. It works by calling the simulateOneStep() from the **Simulator** class. It is disabled if the simulation is running and enabled when it is not.
6. The “Quit” button has the same functionality as hitting the “X” on the window. It halts the program and closes the window.
7. The step counter has been moved to the top left corner of the window. It shows the current step of the simulation.
8. The population stats have been moved to the top right corner. They show the how many organisms are still alive in the simulation.

The content of the window has been divided into three **JPanels** – topPanel, middlePanel and bottomPanel. The topPanel contains the current step and population stats. It uses **GridLayout** to position them on left and right. The middlePanel contains the grid for the simulation, whereas the bottomPanel contains all of the buttons. For the bottomPanel, **GridLayout** is used to position the buttons. All of the panels are added to the content of the window with a **BorderLayout**. The window is not allowed to be resizable and when opened it is positioned at the center of the user’s screen by using setLocationRelativeTo(null).

The two new classes that have been introduced are **Wolf** and **Hunter**. They are both predators and have all of the qualities of the **Organism** class, however, they have additional functionality which is considered as “battle”. They extend the new abstract class **BattleOrganism** which defined new properties that are relevant to all organisms that are expected to fight. The **BattleOrganism** class extends the **Organism** abstract class, so all properties previously available in the **Organism**, are available in the **BattleOrganism** too. There is a new field called strengthLevel that is used for representing the strength of the current entity (currently, this field is randomly populated for both **Wolf** and **Hunter**). There are a setter and a getter for this field inside the abstract class. Something that needs to be declared by all child classes is the getMaxStrengthLevel() method, which returns the maximum strength of an organism; a value that cannot be exceeded. The reason behind this is that every organism can have different values for the maximum strength. Additional methods that are provided are incrementStrength() and decrementStrength(). The former is for incrementing the strength and accepts a parameter that is added to the strength field; there is a check inside the method where the maximum value for strength cannot be exceeded. The decrementStrength() has similar functionality but in reverse.

The main food source for the wolves is foxes. The wolf first checks if there are any foxes in an adjacent location. If the answer is yes, the wolf kills the fox, increments its strength and fills up its food levels, and moves to that location. Otherwise, if the wolf is too hungry and it is about to die (foodLevel is less than or equal to 2), it checks if there is a rabbit around. In that case, it kills the rabbit, increments its strength with only 1, since rabbits are not as powerful as a fox, it increments its food levels (not fully, only with a small amount) and moves to that location. The wolves have different breeding probability and breeding age than the foxes.

The **Hunter** class is the most complex of the organisms. It has similar abilities as the **Wolf** since it extends the **BattleOrganism** abstract class. The main objective of a **Hunter** is to find and kill a **Wolf**. However, wolves are strong animals, so the hunters need to fight them first. The **Hunter** searches for a

**Wolf** in all adjacent locations. If there are no wolves around, then the hunter checks if there is a **Rabbit**. If so, it kills the rabbit, increments its strength and food levels and moves to that location. If there are more than one wolf, they are considered as a pack, so the hunter needs to fight them all. The pack's strength is equal to the total sum of the strengths of all wolves. If the pack's strength is greater than the hunter's strength then the hunter is killed, and all of the wolves receive food and strength. Otherwise, all of the wolves are killed, and the hunter receives strength and food. Output in the console:

```
---  
The pack wins! Hunter: 99 Pack: 106  
---
```

If only one wolf is found nearby, then the stronger of them (the hunter and the wolf) wins. If they have the same strength, the victory is randomly chosen. Console output:

```
---  
Wolf: 59  
Hunter: 64  
Hunter wins!  
---
```

```
---  
Wolf: 79  
Hunter: 44  
Wolf wins!  
---
```

```
---  
Wolf: 10  
Hunter: 10  
Hunter wins!  
---
```

## List of lessons

---

1. **Multiple new classes, each with multiple methods** - new **Hunter** and **Wolf** classes
2. **New functionality that includes conditional statements and iteration / loop statements** – the findFood() method inside the **Hunter** class
3. **Use of collections (such as ArrayLists)** - The findFood() method inside the **Hunter** class, for storing wolves that are in adjacent locations.
4. **Use of library classes** - `javax.swing.Timer` and **Swing** inside the **SimulatorView**
5. **Use of GUI classes** – Components such as **JButton**, **JPanel**, **JComboBox** etc. inside the **SimulatorView**
6. **New abstract classes** – The **BattleOrganism** abstract class that inherits the **Organism** class
7. **Inheritance** – **Fox**, **Hunter**, **Wolf**, **Rabbit** inherit the functionality of **Organism**
8. **Using Eclipse** – Even though I used IntelliJ for the project, the lectures for Eclipse were helpful for understanding how IDE works and to set up my own project inside IntelliJ
9. **Constants** – Almost all of the organism classes use constants to define values that are shared.
10. **Testing and Debugging** – Something that I had to do was to test the new functionality that I added, I used few strategies for debugging, that were introduced by Isla Ross. The main one was to use System.out.println() on certain places to see what the values are.

The project also uses version control (git). By writing `git log` the commits that have been done can be displayed.