**Foxes and Rabbits**

A fundamental function of computers is simulating real-world events, to predict outcomes in situations where real-life tests are not feasible. Examples of useful simulations: traffic optimization, weather prediction, spread of infection, etc. When making a simulation, we attempt to model the behaviors of a subset of the real world.

Every computer-generated simulation is essentially a simplification of real-world activities. The real world is incredibly complex – it's not possible, even with the most powerful computers, to simulate every aspect of something.

In this lab, we're modeling a predator / prey relationship. This simulation will attempt to demonstrate how changes to specific parameters affect the populations of animals. This type of simulation is a very deep and interesting subset of mathematics (for more info, google "Lotka-Volterra model"). This lab will use the following classes, summarized below:

|  |  |
| --- | --- |
| Field | A two-dimensional grid with rows and columns and a fixed number of locations. A single Location in a Field can be empty (null) or store one animal. |
| Location | A two-dimensional position (row, column) in a Field. |
| Fox | A model of the behaviors / characteristics of a fox. |
| Rabbit | A model of the behaviors / characteristics of a rabbit. |
| Simulator | Responsible for storing the state of the simulation. The Simulator has a Field containing Foxes and Rabbits, and repeatedly gives all the animals it contains the chance to live one "step" of their life cycle (they carry out the actions defined by their behaviors). After each step, the current state of the Field is displayed on screen. |

These classes are started for you, but you will be writing most of them. Do not (initially) change the names of the variables or methods if given.

The remainder of the classes (SimulatorView, FieldStats, and Counter) provide a graphical display of the simulation. You shouldn't need to modify the classes (but you may need to use their methods at some point).

**Every method and class is well-documented. Read the comments as you go – don't delete them! In addition to reading the class summaries above, look at each class' instance variables – these variables define what it is to be this particular type.**

**Foxes and Rabbits**

1. To begin, create a class **Runner.java** with a main()method that will (later) start the simulation. Next, create a Simulator object (using the no-parameter constructor). When your code is complete, you will run it by calling the runLongSimulation() method on the Simulator object. At this point you're ready to start adding code!

Location is a simple class that represents a two-dimensional location (row, column) in a two-dimensional array. Complete this class as follows:

1. Complete the two-parameter constructor to initialize row and col.

/\* Remember, when the parameters have the same name as the class' instance variables, the local variables will "hide" the instance variables. You must use the **this** keyword to distinguish between the two and properly initialize the instance variables \*/

1. Override the toString()and equals()methods. The toString() method should return a String in the form of "<row, col>". The equals()method should return true if this location has the same value for row and col as the parameter (i.e. how to determine if two locations are equivalent).

//Check the powerpoints for more info on overriding the equals() method

1. Write getter (accessor) methods for coland row*.*

The Field class is, primarily, a two-dimensional array that contains (at the moment) Objects. Each location (row, column) could contain a fox, a rabbit, or nothing (null). Complete this class as follows:

1. Complete the two-parameter constructor to initialize depthand width to the given depth and width, and initialize the 2D array to a size of depth by width*.*
2. Complete the clear() method that will set every element in the array to null.
3. Complete *both* place()methods, that will place an Object at either the given location or the given row and column. The two-parameter place()method should call the three-parameter place()method, passing in the Location parameter's row and column.
4. Complete *both* getObjectAt() methods, that returns the object (animal) at the given location.

/\* TEST ALL YOUR METHODS AS YOU GO. This program is simply too complex to write all the code and try to find / fix bugs after you've written all the code. \*/

1. Complete the remove() method, that sets the supplied location in the field to null.
2. Complete the move() method, that moves the object in the field from the old location to the new location. Take advantage of methods you've already written!
3. Complete the adjacentLocations() method, that will create a list of locations and return a **shuffled** list (to produce randomness of movement – use a call to the Collections.shuffle() method) of all the locations adjacent (including diagonal) to the given Location. The List should not include the location itself. All locations in the list must be valid (within bounds of the field).

/\* I hope you're using loops for this! To test this method, (in a Tester class) you would make a Field object, make a Location object, and check that the adjacentLocations() method properly returned a list of all the adjacent locations, for that location \*/

1. Complete the freeAdjacentLocation()method, that will return the first "free" (unoccupied) Location found in a list of adjacent locations, given the Location parameter.

/\* This method is called when a fox or rabbit moves or produces babies - babies are born into locations adjacent to the mom. The location returned does not have to be random because the adjacentLocations() method already shuffles the list \*/

1. The depthand widthgetter (accessor) methods have been completed for you. Nice!
2. An overridden toString() has been provided, for printing a quick picture of the state of the field when debugging. Nice! Take a look at it so you have an idea of what it's doing.

The Rabbit class is a model of the behaviors of a rabbit. Complete it as follows:

1. Complete Rabbit's constructor, initializing all instance variables.
   1. Rabbits are created with a current location and a reference to the field they exist in.
   2. A new rabbit should begin alive and at age 0.
2. Write the void setLocation()method, allowing this rabbit's location to be set to the value of the parameter.
3. Complete the setDead() method that will be called when a Rabbit is no longer alive.
   1. Update the value of alive accordingly.
   2. This rabbit should be removed from the field.
4. Complete the incrementAge() method, that will increase the rabbit's age by one – this can also result in the Rabbit's death.
5. Complete the canBreed() method, that returns true if the rabbit is old enough to breed.
6. Complete the breed() method, that returns the number of babies produced (based upon whether or not it can breed and Rabbit's BREEDING\_PROBABILITY). This number should be random, and based on the value of Rabbit.MAX\_LITTER\_SIZE. It can also be 0.

/\* The Random class has a method nextDouble() that will return a random floating point number between 0.0 and 1.0. This method works like Math.random(), and will be useful given the value of BREEDING\_PROBABILITY (where 0 < BP < 1) \*/

1. Complete the isAlive()getter method.
2. Override the toString()method to print "Rabbit, <age> y/o, at <location>".

/\* You can directly concatenate the rabbit's *location* because the Location class has also overridden the toString() method \*/

1. Every "step" of the simulation, a rabbit does whatever a rabbit does. Complete the run()method as follows:
   1. Increment the rabbit's age (use the incrementAge() method – this could result in the rabbit's death).
   2. If the rabbit is no longer alive, return.
   3. Determine the number of babies this rabbit will have.
   4. For the number of births, do the following:
      1. Attempt to get a free location adjacent to this rabbit.
      2. If that location is not null (if there is room), create a new Rabbit.
      3. Place the rabbit in the field.
   5. Next, this rabbit will move randomly. Attempt to find a new location for this rabbit. If there is a location available, move it in the field, then update this rabbit's location.
   6. If no location was available to this rabbit, it will die due to overcrowding.

The Fox class is a model of the behaviors of a fox, and shares many similarities with the Rabbit class. Repeat all the steps for the Rabbit class, with the following changes:

1. Foxes are the predators in the simulation and have a foodLevel instance variable. It should be initialized to Fox.RABBIT\_FOOD\_VALUE (when a fox is born, it starts off "full").
2. A fox should get hungrier every step. If a fox's foodLevelgoes below 0, it dies due to starvation.
3. Note that while rabbits only move around, foxes hunt for food. They do this by "finding food" (running the findFood()method)*.* This method has been written for you, but you should understand what it does. If a fox eats a rabbit, its foodLevel goes up to Fox.RABBIT\_FOOD\_VALUE.
4. this fox's locationshould be determined by its attempt to find food. If it can't find food, it should move randomly. Like the rabbit, if the fox has nowhere to move, it should die due to overcrowding.

The Simulator class contains the current state of the predator / prey simulation. You should be familiar with these instance variables:

* + Field field – the current state of the field (a Field stores a 2D array of Objects).
  + int step – counts the number of "steps" (life-cycles) the simulation has taken

1. Complete Simulator's constructors as follows:
   1. Complete the two-parameter constructor. This constructor should initialize the field and step variables. Note – some code has been added for you and should not be changed.
   2. Complete the default (no-parameter) constructor. This constructor should call this() with parameters of Simulator.DEFAULT\_WIDTH and Simulator.DEFAULT\_DEPTH. Using a call to this(), rather than repeating everything in multiple constructors, is a really useful technique referred to as "constructor chaining".
2. Complete the simulateOneStep()method, that will run one "step" of the simulation, and will drive the simulate(int steps)and runLongSimulation()methods. This method will iterate through all the foxes and rabbits currently in the field, and allow them to act accordingly. In this simulation, foxes hunt() and rabbits run(). Each step, do the following:
   1. Increment the value of the step instance variable.
   2. Iterate through every location in the field and allow the animals to act accordingly.
      1. If the "current" object is actually a rabbit, it should run(). Use the instanceof operator (seen previously in the findFood() method) to determine if the current object is a rabbit. Example, for an object called animal:

if (animal instanceof Rabbit)

((Rabbit) animal).run(); //cast as a Rabbit

* + 1. Repeat the step above in the case that the current animal is a Fox.

**NOTE**: Technically, if the "current" animal moves to a free adjacent location to the "right" (column + 1) or "down" (row + 1), it will be able to (continually) act again in the current "step" of the simulation. This is definitely not ideal, but tolerated for the sake of simplicity. Luckily, the simulation runs fast enough that you probably won't notice.

If this bothers you, you can prevent this from happening by maintaining Lists of the animals in the simulation, such that each animal in the list gets one chance to act, rather than simply iterating through all locations in the field in order (there are other ways this can be accomplished too). Animals that are no longer alive after they act should be removed from the List (as well as the field).

Give this modification a shot if you'd like, but do so at your own risk (it's not necessary to understand the material this lab covers, and not required to receive maximum points on this project).

1. Complete the populate()method (note – this method is called in the reset()method, which is called when a Simulation object is created). This method adds foxes and rabbits to the field, based upon a pre-established creation probability and random number generation. Do the following:
   1. Create a new Random object, then clear the field*.*
   2. Iterate through every position in field. Generate a random number and, based on the probability of creation, create (or don't) a new fox and add it to the simulation:
      1. Create a new fox object, supplying a reference to the field and a new location object (given the current row and column).
      2. Place the fox object into the field*.*
   3. Repeat the previous steps to generate Rabbits (assuming no Fox was added).

Done properly, your simulation should run like [this](https://youtu.be/M7mf3U33Xzw).

At this point, you should have a very good understanding of how the simulation works. It should come as no surprise that this program needs some work – a large portion of the code in Fox and Rabbit is repeated, and adding new animals to the simulation would be a significant chore.

A better design would make use of an abstract Animal super-class that Fox and Rabbit will inherit from.

In addition, what if you wanted to add another type of being to your simulation, for example a Hunter? A hunter is not exactly an animal, and therefore won't have all the characteristics animals have. They would, however, act in the simulation.

Further refine your project by adding an Actor interface. An actor is anything in the simulation that can act, and would have three main behaviors: acting (polymorphically called based on the object type), determining if it's active (alive), and removing it from the simulation. Add the following interface to your project:

public interface Actor

{

void act(); //do whatever the Actor does

boolean isActive(); //determine if this Actor is still active in the simulation

void remove(); //remove the Actor from the field / simulation

}

The Animal class should implement the Actor interface, as would a new class like Hunter.

Refactor the Rabbit, Fox, Simulator, and Field classes to make use of an abstract Animal super-class that Fox and Rabbit will both extend (which implements the Actor interface). Rather than Object types, the Actor type can be used to polymorphically store both fox and rabbit objects (along with other futures types, like hunter).

Look for similarities in the Fox and Rabbit classes and refactor those (instance variables / methods) into Animal. Of particular interest:

1. Methods that won't change for different types of animal (e.g. getters, the setDead() method, etc.) should be concrete and will be inherited as-is.
2. The obvious methods that will change for each type of animal are Rabbit's run()and Fox's hunt(). Concrete animal classes should now override the void act() method, given how that particular animals behaves.

Rather than having to check if the current Object in the field is a rabbit or a fox and calling separate methods, the act() method is simply called polymorphically – much cleaner!

1. Note – in the naïve version of this project, we had no actor super-type, and therefore had to declare everything of type Object. Now that a super-class exists, all methods should take Actor type variables (and the field should store Actor objects).

Finally, add some interesting new actors (something that implements the Actor interface) to your simulation (with new colors). The behavior of the classes is up to you. Examples:

* UntrainedHunter – fires randomly and occasionally into its own location.
* Grass – a food source for Rabbits (which should now be able to eat and die of starvation).
* ZombieRabbit – a small chance of mutation could create a Zombie Rabbit that could then affect other Rabbits.

With your remaining time, think of ways you can improve your hierarchy of classes. For example, suppose you have an Actor interface, an Animal abstract class that implements Actor, and Fox / Rabbit concrete classes that extend Animal. If you added a Hunter to the simulation (something that isn't an Animal), it would require much of the same code as Animals. If you added Grass to the simulation, it would require some of the code found in Animal, but not all.

Should you refactor the hierarchy to have Predator and Prey interfaces (or abstract classes)? Or maybe a Moveable interface for types that move? How about parent classes or interfaces for types that produce offspring? There is no right or wrong answer - good solutions will be highly modular with little duplicate code and easily extensible. A very good solution will require very little code and no refactoring when adding a new type (think about how easy it was to make the Thirteens game, in the Elevens lab).

*This lab is adapted from the Foxes and Rabbits project*

*Objects First with Java, Barnes and Kolling*