CSC207: Java inheritance, abstract classes and polymorphism.

To earn the credit for this lab, you must be present in the lab class. For this lab, and for the rest of the labs in this class, please make sure to submit by end of the lab class.

1 Overview

This week, you are going to practice inheritance, abstract classes and polymorphism.

2 Choose a driver and a navigator

In all the labs, we will use the terms *driver* and *navigator*. Here are the definitions of the two roles:

- **Driver**: Types at the keyboard. Focuses on the immediate task at hand.
- **Navigator**: Thinks ahead and watches for mistakes.

In lab handouts, we'll often refer to you as `s1` and `s2`, and `s1` will be the first driver.

3 Log in and get things set up.

Use MarkUs to form a group with your lab partner. The MarkUs item for this lab is Lab2. This will require `s1` to log in to MarkUs and invite `s2` to form a group. Then `s2` will need to log in and accept the invitation from `s1`. Take a note of the group MarkUs created for you. It will be of the form `group_nnnn`.

All repositories in this course will have a URL of the form:

`httsp://markus.cdf.toronto.edu/svn/csc207-2016-05/repo-name`

`s1` drives and `s2` navigates.

Now `s1` logs in and opens a new terminal window.

1. Change to `s1`'s home directory.
2. Check out/update your group's repository.
3. In your repository, you will find a newly created directory called Lab3.
4. Open up Eclipse. Point your workspace to the folder `repo-name`.
5. Using Eclipse, create a new Java project called `Lab3`. Make sure you open the Java perpective.
6. Create a new package called `lab3`.
7. From the course web site, download the starter classes and add them to the lab3 package.

4 Artificial life

In this lab we will simulate a simple interaction between two living organisms in a habitat represented as a grid. In practical terms, the world we are simulating, is made up of a rectangular grid, where in any of the cells of the grid may live a critter (represented by lower case `o`) or a plant (represented by a `*`):

```
....o...o*.....
...............
......o.......o
....o.o........
...o...*.*.....
```

When the world is instantiated, a number (chosen randomly) of plants and critters is placed on the grid. The initial energy level of each plant or critter is (initially) 10 units. On every cycle of life, a plant absorbs a constant amount of energy from the environment (say 0.1 units). The energy supply of the environment is unlimited. The critters, unlike plants, can move around and eat plants. A critter can move to a neighboring cell at random, given the cell is available (that is, there is no plant or critter living there). On every move, the critter spends some energy (say 0.1 units). If a critter discovers a plant in a neighboring cell, the critter consumes it by absorbing the whole energy of it. If a critter's energy level drops to 0, then the critter dies (that is, its reference gets removed). For instance, after one life cycle, the world pictured above, has been transformed to:

```
.......o.......
.....o........o
...............
...o.o.........
..o...o..*.....
```

5 The `Organism` class

The `Organism` class is an abstract class that represents a living entity. As such it has the following properties:

> `private final double INITIAL_ENERGY = 10`
>
> `protected int xpos, ypos` : The coordinates of the entity in the grid
>
> `protected double energy` : The current energy of the entity
>
> `protected World world` : The `World` instance where the entity lives
>
> `protected String type` : It is `o` or `*` (see above)

The methods of `Organism` class are:

> `protected ArrayList<int[]> getNeighbors()` returns a list of eight pairs of coordinates (a cell in a rectangular grid can have up to 8 neighbors). For example the neighbors of the cell `(1,2)` are: `[[0,1],[0,2],[0,3],[1,1],[1,3],[2,1],[2,2],[2,3]]`.
>
> `protected abstract void eat()` : to be implemented in the derived classes.
>
> `protected abstract void move()`: to be implemented in the derived classes.

6 The `World` class

The `World` class objects have the following properties:

> `protected int width, height`: The dimensions of the grid
>
> `protected Organism[][] map`: The grid itself. The unoccupied cells are set to null, the occupied cells contain `Plant` or `Critter` objects.
>
> `protected ArrayList<Organism> living`: Contains a list of plants and critters that inhabit the world. Observe that an abstract type can also serve as parameter to a generic class (in that case to an `ArrayList`).

The constructor and the `toString` methods have already been provided. Your task is to implement the `turn()` method (its signature is provided in the starter code) so each object of `living` is processed polymorphically with the appropriate `eat()` and `move()` method. Your implementation should contain no more than three lines of code!

7 The `Plant` and `Critter` classes

- Both classes extend `Organism`. Please implement `eat()` method for both. You may need to make use of `getNeighbors` method (provided). Note plants `eat()` method simply increments the energy of it by a constant percentage on each lifecycle as opposed to the same method for critters should work by identifying neighboring plants and consuming them.
- Switch diver and navigator roles. Implement the method `move()` for both classes.
- Test your work using `Life` class (provided). Show your work to your TA for marking.

8 Exercises - on your own at home - do not submit for marking

1. Add some code in the `Life` class so the simulation runs until all the food (plants) is consumed or until all critters have starved. Make sure to limit the number of loops to avoid the infinite loop situation.
2. Add a new class `Predator` that extends `Organism` and feeds on critters. Add smarts so predators chase critters.
3. Change the behaviors of `Critter` objects so they avoid predators and at the same time feed so they don't starve.