

Assignment #2

Richard Forrester



February 6, 2020

Everett Community College

raforrester@students.everettcc.edu

# Description

This assignment was meant to test our knowledge of Inheritances, Interfaces, Abstract Classes, and Array lists by creating a simple farm simulation. The idea of an animal was represented by an abstract class named Animal, and as a result, the following animal classes extended the superclass animal. This allowed us to store the animal class in one location, an ArrayList, thereby creating a virtual farm. Furthermore, the while loop symbolized a night and day cycle as we called the various functions that virtually fed and checked the status of the animals while the tick function represented the virtual passing of time which made the animals hungrier(lowered their hunger levels represented by Integers). Finally, whether or not an animal became sick or fell asleep was determined by generating random numbers, and if they equaled a certain value the animal's sleep or sick status, represented by a Boolean, would become true or false.

# The Code

**import** java.util.\*;

**public** **class** Simulation {

**static** **int** *satisfiedAnimals* = 0;

**public** **static** **void** main(String[] args) {

List<Animal> farm = **new** ArrayList<>(

List.*of*(**new** Cow("Tom"), **new** Cow("Jeff"), **new** Dog("Fido"), **new** Dog("Blue"), **new** Chicken("Pecker"),

**new** Chicken("Sam"), **new** Horse("Boxer"), **new** Horse("Ben"), **new** Sheep("luv"), **new** Sheep("Ian")));

**int** farmSize = farm.size();

**int** cycle = 1;

**while** (*satisfiedAnimals* < farmSize) {

System.***out***.println("FEEDING " + cycle + "\n");

*satisfiedAnimals* = 0;

cycle += 1;

Collections.*sort*(farm, **new** Comparator<Animal>() {

@Override

**public** **int** compare(Animal first, Animal second) {

**return** second.getHungerLevel() -first.getHungerLevel();

}

});

farm.forEach((animal) -> {

**if** (animal.getHungerLevel() == 0) {

*satisfiedAnimals* += 1;

} **else** {

*satisfiedAnimals* = 0;

}

//"Allow each animal to speak, print it out and then feed it if it is not full."

System.***out***.println(animal.speak());

System.***out***.println(animal);

**if** (animal.getHungerLevel() != 0) {

animal.feed();

}

animal.tick();

});

System.***out***.println("\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_\n");

}

}

}

# The output

In Folder

# Comments/Notes (Extra Credit)

**Struggles**

The only issue I had was from overriding the compreTo function from the comparable interface. From my understanding, returning a positive integer would indicate that the current object is in fact larger than the other object it is being compared against. Thus, my original formula was “this.getHungerLevel-other.getHungerLevel.” However, this was giving me the opposite of my desired ordering. As a result, I had to inverse the formula. Apparently, the compareTo function was swapping the objects only when a negative value was returned.

**Additions**

A) Used Lambda Notation for the forEach loop

B) Used an anonymous function to implement the compare function of the Comparator interface.

C) Added a sick feature:

\* Animals had a 5% chance to get sick when the tick function was called.

\* sick animals gained 2 additional hunger units as time passed.

\*The toString function showed if they were sick.

\*Animals had a 33% chance to become healthy again when fed.

D) Made sure the hungerUnits was reset to 24 or 0 if it fell outside of the range.

E) Added getters and setters for all the fields.

**Q&A**

1. If we are never going to instantiate an Animal directly what type of class could we make it? **Explain**.

If you are not going to instantiate an Animal directly we make it an abstract class. There is no such things as an “animal” in real life, but types of animals. Abstract classes cannot be instantiated because they contain function that are not fully implemented yet (abstract functions). Thus, the animal abstract class acts as a blueprint for the rest of the animal classes. We make some methods abstract such as the speak function because all animals make sounds, but they make different sounds.

1. It is possible that when you run the simulation it will loop infinitely\*.

Yes.

**Can you explain why? Why does this stop infinite loops?**

\* For larger sets of animals this would almost always happen.

You could put this in the constructor:

hungerUnits = hungerUnits-(hungerUnits%5);

What does this do?

That makes it so that no animal will start with the max hungerUnits possible by subtracting by 4. This makes 20 the max hungerUnit for animals when they are first constructed. This would prevent the hungerUnits from going above 24. If an animal was instanced at 24 and ticked was the first function that was called, it would result in their hungerUnits exceeding the desired range which may cause the program to loop infinitely if that was not accounted for. Also, depending on how you programed the simulation, if there were a large set of animals to feed and time passed for all of them as you fed each animal there hungerUnits may decrease faster than you could increase it. This would cause them never to be full and as a result the loop’s end which is contingent upon this event would continue indefinitely.