CS145 – PROGRAMMING ASSIGNMENT

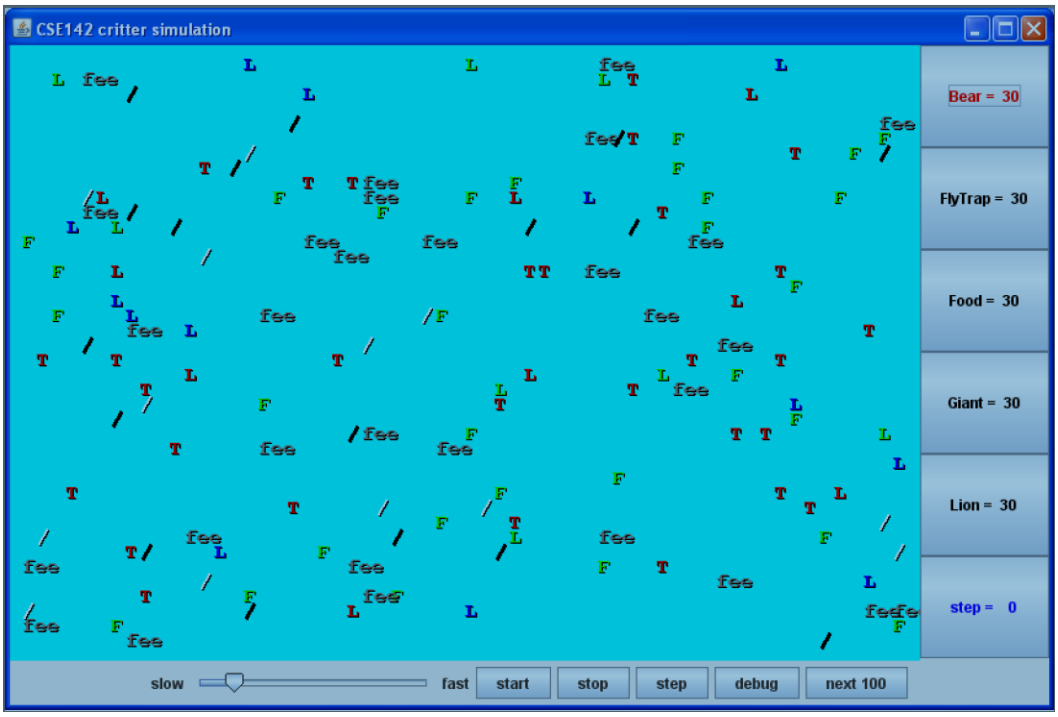
CRITTERS

OVERVIEW

This assignment will give you practice with defining classes and a small introduction to polymorphism. You are to write a set of classes that define the behavior of certain animals. You will be given a program that runs a simulation of a world with many animals wandering around in it. Different kinds of animals will behave in different ways and you are defining those differences. In this world, animals propagate their species by infecting other animals with their DNA, which transforms the other animal into the infecting animal's species.

INSTRUCTIONS

For this assignment you will be given a lot of supporting code that runs the simulation. You are just defining the individual "critters" that wander around this world trying to infect each other. While it is running the simulation will look something like this:



GETTING STARTED

In order to get started with this assignment, start by downloading the Programming Assignment - CS145 – Critters ZIP file from Canvas. Inside that zip file is a folder called JavaCritter that contains 8 files. Move the folder and all 8 files into your data drive.

Of the 8 files included.

File	Actions
CritterFrame.java	Do not touch this file
CriterInfo.java	Do not touch this file
CritterModel.java	Do not touch this file
CritterPanel.java	Do not touch this file
Critter.java	Open this file to look at it, but do not change it.
Food.java	Open this file to look at it, but do not change it. Use this as an example for your files.
Flytrap.java	Open this file to look at it, but do not change it. Use this as an example for your files.
CritterMain.java	This is the file that you will run. You can make very minimal changes to this file to add more critters to the simulation.

YOUR INSTRUCTIONS

You objective is to implement the following 5 classes that are subclasses of critter. You will create 5 new classes and add them to the simulation.

ANIMAL CLASS

Each of your critter classes should extend a class known as Critter. So each Critter class will look like this:

```
public class SomeCritter extends Critter {
...

}
```

The “extends” clause in the header of this class establishes a polymorphic inheritance relationship as discussed in class. The main point to understand is that the Critter class has several methods and constants defined for you.

On each round of the simulation, each critter is asked what action it wants to perform. There are four possible responses each with a constant associated with it.

Constant	Description
`Action.HOP`	Move forward one square in its current direction
`Action.LEFT`	Turn left (rotate 90 degrees counter-clockwise)
`Action.RIGHT`	Turn right (rotate 90 degrees clockwise)
`Action.INFECT`	Infect the critter in front of you

There are three key methods in the Critter class that you will override in your own classes. When you override these methods, you must use exactly the same method header as what you see below. The three methods to override for each Critter class are:

```
public Action getMove(CritterInfo info) {
...
}
public Color getColor() {
...
}
public String toString() {
...
}
```

There are three key methods in the Critter class that you will override in your own classes, however you may want to implement additional methods to implement your own class actions.

For the getMove method you should return one of the four Action constants described earlier in the writeup. For the getColor method, you should return whatever color you want the simulator to use when drawing your critter. And for the toString method, you should return whatever text you want the simulator to use when
displaying your critter (normally a single character).

For example, below is a definition for a critter that always infects and that displays itself as a green letter F:

```
import java.awt.*;
public class Food extends Critter {
  public Action getMove(CritterInfo info) {
    return Action.INFECT;
  }

  public Color getColor() {
    return Color.GREEN;
  }

  public String toString() {
    return "F";
  }
}
```

Notice that it begins with an import declaration to be able to access the Color class. All of your Critter classes will have the basic form shown above.

The getMove method is passed an object of type CritterInfo. This is an object that provides you information about the current status of the critter. It includes eight methods for asking about surrounding neighbors plus a method to find out the current direction the critter is facing. Below are the methods of the CritterInfo class:

CritterInfo Method	Description
`public Neighbor getFront()`	returns neighbor in front of you
`public Neighbor getBack()`	returns neighbor in back of you
`public Neighbor getLeft()`	returns neighbor to your left
`public Neighbor getRight()`	returns neighbor to your right
`public Direction getDirection()`	returns direction you are facing
`public Direction getFrontDirection()`	returns the direction of the neighbor in front of you
`public Direction getBackDirection()`	returns the direction of the neighbor in back of you
`public Direction getLeftDirection()`	returns the direction of the neighbor to your left
`public Direction getRightDirection()`	returns the direction of the neighbor to your right

The getDirection method tells you what direction you are facing (one of four direction constants):

Constant	Description
`Direction.NORTH`	facing north
`Direction.SOUTH`	facing south
`Direction.EAST`	facing east
`Direction.WEST`	facing west

The return type for the frst group of four CritterInfo methods is Neighbor. There are four different constants for the different kind of neighbors you might encounter:

Constant	Description
`Neighbor.WALL`	The neighbor in that direction is a wall
`Neighbor.EMPTY`	The neighbor in that direction an empty square
`Neighbor.SAME`	The neighbor in that direction is a critter of your species
`Neighbor.OTHER`	The neighbor in that direction is a critter of another species

Notice that you are only told whether critters are of your species or some other species. You can't find out exactly what species they are. The second group of four methods tells you what direction the neighbors on each

This program will probably be confusing at first because this time you are not writing the main method. Your code will not be in control. Instead, you are defining a series of objects that become part of a larger system. For example, you might find that you want to have one of your critters make several moves all at once. You won't be able to do that. The only way a critter can move is to wait for the simulator to ask it for a move. The simulator is in control, not your critters. Although this experience can be frustrating, it is a good introduction to the kind of programming we do with objects. A typical Java program involves many different interacting objects that are each a small part of a much larger system.

Critters move around in a world of finite size that is enclosed on all four sides by walls. You can include a constructor for your classes if you want, although it should generally be a zero-argument constructor (one that takes no arguments).

YOU ARE TO IMPLEMENT FIVE CLASSES.

BEAR

- Constructor
  - `public Bear()`
  - When you create a bear class, you should have it decide if it is a normal bear or a polar bear. 50% chance of each type.
  - Once picked, the bear never changes type.
- getColor
  - `Color.WHITE` for a polar bear
    - (when polar is true),
  - `Color.BLACK` otherwise
    - (when polar is false)
- toString
  - Should alternate on each different move between a slash character (/) and a backslash character (\) starting with a slash.
- getMove
  - always infect if an enemy is in front
  - otherwise hop if possible
  - otherwise turn left

LION

- Constructor
  - `public Lion()`
- getColor
  - Randomly picks one of four colors (`Color.RED`, `Color.GREEN`, `Color.YELLOW`, `Color.BLUE`) and uses that color for three moves, then randomly picks one of those colors again for the next three moves, then randomly picks another one of those colors for the next three moves, and so on.
  - **Make sure the Lion does not pick the same color twice in a row.** So really there are only three choices each round after the first.
- toString
  - `"L"`
- getMove
  - always infect if an enemy is in front
  - otherwise if a wall is in front or to the right, then turn left
  - otherwise if a fellow Lion is in front, then turn right
  - otherwise hop.

GIANT

- Constructor
  - `public Giant()`
- getColor
  - `Color.GRAY`

- toString
  - `"fee"` for 6 moves,
  - then `"fie"` for 6 moves,
  - then `"foe"` for 6 moves,
  - then `"fum"` for 6 moves,
  - then repeat.
- getMove
  - always infect if an enemy is in front
  - otherwise hop if possible
  - otherwise turn right.

TITAN

The titan class is an extension/subclass of the Giant class. As such it should **only** have the following 3 methods. Also, it should have **ZERO** fields, since it is a subclass of Giant.

- Titan()
  - A constructor.
- getColor
  - `Color.BLACK`
    - `When saying "fee"`
  - `COLOR.GRAY`
    - `When saying "fie" or "fum".`
  - `Color.WHITE`
    - `When saying "foe"`
- getMove
  - it should have the same pattern as a Giant, except instead of turning right, it should turn left.

- Note that the Titan Class should NOT have any other methods, they will be inherited that method from the Giant class.
- **VERY IMPORTANT**
  - You should NOT have any private variables in Titan, your goal with this class is to have the parent/(Giant) class do all the hard work, and then this class will look a the data, and return the new/modified result.
  - You will need to use your `super`power here!

GATOR

- Constructor
  - `public Gator()`
- getColor
  - You decide
- toString
  - You decide
- getMove

- You decide

As noted above, you will determine the behavior of your Gator class. Some of the style points for this assignment will be awarded on the basis of how much energy and relativity you put into defining an interesting Gator class. These points allow me to reward the students who spend time writing an interesting critter definition. Your Gator's behavior should not be trivial or closely match that of an existing animal shown in class.

You can also get credit for providing a careful description of the experiments you have performed or the thought process you have used to arrive at your Gator strategy.

NOTES

Style points will also be awarded for expressing each critter's behavior elegantly. Encapsulate the data inside your objects. Follow past style guidelines about indentation, spacing, identifiers, and localizing variables. Place comments at the beginning of each class documenting that critter's behavior, and on any complex code.

For the random moves, each possible choice must be equally likely. You must use Random object and NOT the Math.random() method to obtain pseudorandom values.

Each of your critter classes has a pattern to it and at first all of your critters will be in synch with each other. For example, all of the bears will be displayed as slashes and all of the giants will be displayed as “fee.” But as critters become infected, they will get out of synch. A newly constructed giant will display itself as “fee” for its first six moves, so it won’t necessarily match the other giants if they are somewhere in the middle of their pattern. Doen’t worry about the fact that your critters end up getting out of synch in this way.

Your classes should be stored in files called Bear.java, Lion.java, Giant.java, Titan.java, and Gator.java. The files that you need for this assignment will be included in a zip file available from the class web page. All of these files must be included in the same folder as your Critter files. You should download the zip file, then add your four classes to the folder, compile CritterMain and run CritterMain.

The simulator has several supporting classes that are included in the ZIP file. (CritterModel, CritterFrame, etc). You can in general ignore these classes. When you compile CritterMain, these other classes will be compiled. The only classes you will have to modify and recompile are CritterMain (if you change what critters to include in the simulation) and your own individual Critter classes.

You will be given two sample critter classes. The first is the Food class that appears earlier in the write-up. The second is a particularly powerful strategy called FlyTrap. Both of these class definitions appear in the zip file and should serve as examples of how to write your own critter classes.

You will notice that CritterMain has lines of code like the following:

```
// frame.add(30, Lion.class);
```

You should uncomment these lines of code as you complete the various classes you have been asked to write. Then critters of that type will be included in the simulation.

When a critter is infected, it is replaced by a brand new critter of the other species, but that new critter retains the properties of the old critter. So it will be at the same location and facing in the same direction.

The simulator provides great visual feedback about where critters are, so you can watch them move around the world. But it doesn't give great feedback about what direction critters are facing. The simulator has a "debug" button that makes this easier to see. When you request debug mode, your critters will be displayed as arrow characters that indicate the direction they are facing.