Assignment 8: Critters (20 points)

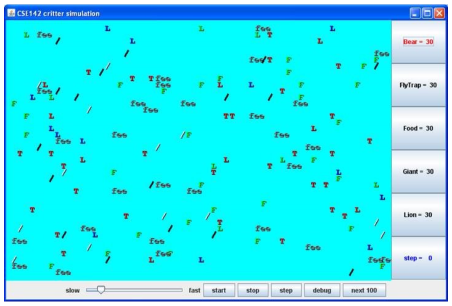
due August 18, 2020, 11:59pm

This assignment focuses on object-oriented programming, classes, and objects Turn in the following **FOUR** Java files using the link on the course website:

- `Bear.java` – A class that implements the Bear critter described below.
- `Lion.java` – A class that implements the Lion critter described below.
- `Giant.java` – A class that implements the Giant critter described below.
- `Husky.java` – A class that implements your Husky critter.

Program Overview

In this assignment, you will *not* be writing a program. Instead, you will be implementing several classes that will be used a client program provided to you. This client represents a graphical simulation of a 2D world of animals, known as critters. In this simulation, critters move around a finite world surrounded by four "walls", attempting to propagate their species by infecting other critters with their DNA. When a critter is infected, it is transformed into the species of the critter that infected it. While it is running, the simulation will look like the image to the right. You will write classes that define the behavior of several types of critters, each of which moves and behaves in different ways.



Critter Class Structure

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 to set up each Critter's state correctly if you would like, although it should generally be a zero-argument constructor (one that takes no arguments). There will be one exception to this rule for your Bear critter, described below.

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 an inheritance relationship. This is discussed in Chapter 9 of the textbook, although you dont need a deep understanding of the concept for this assignment. The main point to understand is that the Critter class has several methods and constants defined for you. So, by saying that you extend the class, you inherit these methods and constants. Inheritance makes it easier for our code to talk to your critter classes, and it helps us be sure that all your critter classes will implement all the methods we need. You should then give new definitions to certain methods to define the behavior of your critters.

There are three key methods in the Critter class that you will redefine (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 you may override for each Critter class are:

- `public Action getMove(CritterInfo info)`
  Each round, the client calls this method on your animal to find out what action it should take on this turn. (See below for more information on CritterInfo and Action.)

- `public Color getColor()`
  Each round, the client calls this method on your animal to find out what color it should appear.

- `public String toString()`
  Each round, the client calls this method on your animal to find out what text it should be displayed as.

By declaring that your class `extends Critter` as shown above, you receive a **default version** of these methods. The default behavior is to always turn left (a move of `Action.LEFT`), to use the color black, and to display as a "?". If you don't want these defaults, rewrite (override) the necessary methods in your class with your own behavior.

The `getMove` method will return one of the four `Action` constants described below. The `getColor` method will return whatever color you want the simulator to use when drawing your critter. And the `toString` method will return whatever text you want the simulator to use when displaying your critter (usually but not always a single character).

For example, below is a definition for a critter called `Food`. `Food` always infects and 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 for access to the `Color` class. All of your Critter classes will have the basic form shown above, except yours may have fields, a constructor, and more sophisticated code.

Critter Actions

The simulation takes place in a series of rounds, with each critter taking a single action in each round. Critters specify the action they wish to take by returning one of the following values from the `getMove` method (see below):

`Action` Constant	Description
`Action.HOP`	move one step forward in the direction the critter is facing
`Action.LEFT`	turn left (90 degrees counter-clockwise)
`Action.RIGHT`	turn right (90 degrees clockwise)
`Action.INFECT`	attempt to infect the critter in front of this critter

To help determine which action the critter should take, the `getMove` method is passed a parameter of type `CritterInfo`. This object provides information about the current status of the critter and the world around it. The `CritterInfo` class contains the following methods:

`CritterInfo` Method	Description
`public Direction getDirection()`	returns the direction this critter is facing
`public Neighbor getFront()`	returns the type of neighbor in front of this critter
`public Neighbor getBack()`	returns the type of neighbor in back of this critter
`public Neighbor getLeft()`	returns the type of neighbor to the left of this critter
`public Neighbor getRight()`	returns the type of neighbor to the right of this critter
`public boolean frontThreat()`	returns `true` if there is a critter of a different species facing you from the front
`public boolean backThreat()`	returns `true` if there is a critter of a different species facing you from the back
`public boolean leftThreat()`	returns `true` if there is a critter of a different species facing you from the left
`public boolean rightThreat()`	returns `true` if there is a critter of a different species facing you from the right

The `getDirection` method returns one of the four `Direction` values listed below, indicating which direction the critter is facing. (`Direction.NORTH` is the top of the world.) The four `get` methods each return one of the four `Neighbor` values listed below, indicating what is in the space in the direction requested (e.g. `getLeft` returns what type of neighbor is to the critter's left). The four `threat` methods each return whether or not the critter is at risk of being infected from the requested direction (e.g. `backThreat` determines if the critter could be infected from the back). A critter is threatened if there is a critter of a different species in that direction, and the other critter is facing the threatened critter. Critters of other species are referred to as "enemies."

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

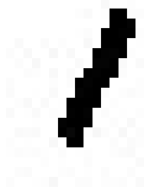
`Neighbor` Constant	Description
`Neighbor.WALL`	a wall (i.e. the edge of the world)
`Neighbor.EMPTY`	an empty space
`Neighbor.SAME`	a critter of the same species as this critter
`Neighbor.OTHER`	a critter of a different species than this critter

Notice that you are only able to determine whether or not a neighboring critter is of the same species as your critter; you cannot determine what specific species any critter is.

Required Critters

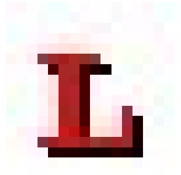
You will implement the following **four** critters for this assignment. Each critter will have one constructor, which must accept *exactly* the parameters shown below. (Any changes to the constructor will cause the client to not compile.) For any behavior described as random, all possibilities should be equally likely, and random values should be generated using a `Random` object.

Bear	
constructor	`public Bear(boolean polar)`
getColor	`Color.WHITE` for a polar bear (when polar is `true`), `Color.BLACK` otherwise (when polar is false)
toString	Alternate on each move between a forward slash ("/") and a backslash ("̈) starting with a forward slash ("/").
getMove	Infect if an enemy is in front Otherwise hop if possible Otherwise turn left

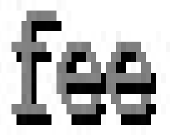


The Bear constructor accepts a parameter indicating whether or not the bear is a polar (white) bear. When the parameter is true, the resulting bear will be white. When the parameter is false, the resulting bear will be black. The simulator deals with deciding whether each bear will be white or black; your class just needs to ensure that you create the right color bear based on the parameter.

Lion	
constructor	`public Lion()`
getColor	Randomly pick one of `Color.RED`, `Color.GREEN`, or `Color.BLUE` and use that color for three moves, then randomly pick one of those colors again for the next three moves, then randomly pick one of those colors for the next three moves, and so on.
toString	"L"
getMove	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	Infect if an enemy is in front Otherwise hop if possible Otherwise turn right.



Husky	
constructor	`public Husky() (must not accept any parameters)`
`all behavior`	you decide (see below)

As noted above, you will determine the behavior of your Husky class. Part of your grade will be based upon writing creative and non-trivial Husky behavior. **Your Husky's behavior should not be trivial or closely match that of an existing animal shown in class.**

Unlike on most assignments, your Husky **can use advanced material** you happen to know in Java (with some restrictions). If you wish to use particularly complex material in your Husky, contact your TA or instructor to make sure it will be compatible with our system.

Running the Simulator

Each of your critter classes has a pattern to it and at first all of your critters will be in sync 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 sync. A newly constructed giant will display itself as "fee" for its first six moves, so it wont necessarily match the other giants if they are somewhere in the middle of their pattern. Doesnt worry about the fact that your critters end up getting out of sync in this way.

Your classes should be stored in files called `Bear.java`, `Lion.java`, `Giant.java`, and `Husky.java`. The files that you need for this assignment will be included in a zip file called `assign8.zip` available from the class web page. All of these files must be included in the same folder as your Critter files. You should download and unzip `assign8.zip`, then add your four classes to the folder, compile `CritterMain` and run `CritterMain`.

The simulator has several supporting classes that are included in `assign8.zip` (`CritterModel`, `CritterFrame`, etc). In general, you can ignore these classes. When you compile `CritterMain`, these other classes should be compiled. If they are not, you can compile them manually in the following order: `Critter.java`, `CritterModel.java`, `CritterPanel.java`, `CritterMain.java`. Once you have compiled these classes once, 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 also be given two sample critter classes. The first is the `Food` class that appears earlier in the writeup. The second is a class called `FlyTrap` that was discussed in lecture. Both of these class definitions appear in `assign8.zip` 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.

Husky Strategy

For those who want to produce huskies that "win" in the sense of taking over the world, you will want to understand some subtleties about the simulation.

- In general, the simulator allows each critter to make a move and it scrambles the order in which it does that. This becomes important, for example, when two critters are facing each other and each wants to infect the other. The simulator will randomly decide which critter goes first and the one that goes first will win.

- If a critter successfully hops during its turn, it cannot be infected for the rest of the round. Once a critter has been infected, it cannot be infected again by a different critter during the same round, and if the infected critter hadnt already completed its move, it wont get a turn.

- Critters that infect after a hop are slightly favored in that around 20did not hop on its previous move.

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. That is, the new critter will be placed at the same location and be facing in the same direction.

Debugging Strategy

The simulator provides great visual feedback about where critters are, so you can watch them move around the world. But it doesnt give very helpful 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. The simulator also indicates the step number as the simulation proceeds (initially showing a 0).

Since the behavior of this assignment is visual, there is no Output Comparison Tool. Instead, below are some suggestions for how you can test your critters and descriptions of what the general critter behavior should look like:

- **Bear:** Try running the simulator with just 30 bears in the world. You should see about half of them being white and about half being black. Initially they should all be displayed with slash characters. When you click step, they should all switch to backslash characters. When you click step again they should go back to slash characters. And so on. When you click start, you should observe the bears heading towards walls and then hugging the walls in a counterclockwise direction. They will sometimes bump into each other and go off in other directions, but their tendency should be to follow along the walls.
- **Lion:** Try running the simulator with just 30 lions in the world. You should see about one third of them being red and one third being green and one third being blue. Use the step button to make sure that the colors alternate properly. They should keep these initial colors for three moves. That means that they should stay this color while the simulator is indicating that it is step 0, step 1, and step 2. They should switch colors when the simulator indicates that you are up to step 3 and should stay with these new colors for steps 4 and 5. Then you should see a new color scheme for steps 6, 7, and 8. And so on. When you click start you should see them bouncing off of walls. When they bump into a wall, they should turn around and head back in the direction they came. They will sometimes bump into each other as well. They shouldnt end up clustering together anywhere.
- **Giant:** Try running the simulator with just 30 giants in the world. They should all be displayed as fee. This should be true for steps 0, 1, 2, 3, 4, and 5. When you get to step 6, they should all switch to displaying fie and should stay that way for steps 6, 7, 8, 9, 10, and 11. Then they should be foe for steps 12, 13, 14, 15, 16, and 17. And they should be fum for steps 18, 19, 20, 21, 22, and 23. Then they should go back to fee for 6 more steps, and so on. When you click start, you should observe the same kind of wall-hugging behavior that bears have, but this time in a clockwise direction.

Development Strategy

The simulator runs even if you haven't completed all the critters. The classes increase in difficulty from Bear to Lion to Giant. We suggest doing Bear first. Look at Food.java and the lecture/section examples to see the general structure.

It will be impossible to implement each behavior if you don't have the right state in your object. As you start writing each class, spend some time thinking about what **state** will be needed to achieve the desired behavior.

One thing that students in the past have found particularly difficult to understand is the various constructors for each type of animal. Some of the constructors accept parameters that guide the behavior of later methods of the animal. It is your job to **store data from these parameters** into fields of the animal as appropriate, so that it will "remember" the proper information and will be able to use it later when the animal's other methods are called by the simulator.

Test your code **incrementally.** A critter class will compile even if you have not written all of the methods (the unwritten ones will use default behavior). So, add one method, run the simulator to see that it works, then add another.

Style Guidelines

You should follow all guidelines in the Style Guide and on the General Style Deductions page of the course website. Pay particular attention to the following elements:

Since this assignment is largely about classes and objects, much of the style grading for this assignment will be about how well you follow proper object-oriented programming style.

Appropriate Object Encapsulation

You should encapsulate the data inside your objects, initialize fields in constructors, and you should not declare unnecessary data fields to store information that isn't vital to the state of the object. Style points will also be awarded for expressing each critter's behavior elegantly.

Appropriate State Updates

Be sure that your code is tied to actual moves made by a critter (i.e., that changes to state occur in the getMove method). For example, the bear is supposed to be displayed alternately as a slash or backslash. **This should happen as the getMove method is called.** Your code should work properly even if toString is called twice and then getMove is called. The alternation should happen for each move, not for each call on toString. This also applies to the behavior of giants and lions because they are defined in terms of changes happening after a given number of moves.

Inheritance

Another aspect of the style of this program is inheritance. Your critter classes should properly extend a superclass as described.

Husky Creativeness

Some of the points for this assignment will be awarded on the basis of how much **energy and creativity** you put into defining an interesting Husky class. These points allow us to reward the students who spend time writing an interesting critter definition. **Your Husky's behavior should not be trivial or closely match that of an existing animal shown in class.** A **single point** of extra credit will be awarded to exceptionally creative or impressive Husky implementations. Note that this point will be awarded very sparingly, and that writing complex code or using advanced material is neither necessary nor sufficient to earn this point

Husky Internal Correctness

The Husky is **not graded on internal correctness** at all. Its code does not need to be commented, can be redundant, and can use advanced material, so long as it works properly and obeys the other constraints described previously.

Other General Style

Follow past style guidelines about indentation, spacing, identifiers, and localizing variables. Your critters should not produce any console output. For reference, our Bear, Lion, and Giant together occupy just under 140 lines including blank lines and comments (85 "substantive" lines, according to our Indenter tool).

Code Aesthetics

Your code should be properly indented, make good use of blank lines and other whitespace, and include no lines longer than 100 characters. Your class, methods, variables, and constant should all have meaningful and descriptive names and follow the standard Java naming conventions. (e.g. `ClassName`, `methodOrVariableName`, `CONSTANT_NAME`). See the Style Guide for more information.

Commenting

Place comments at the beginning of each class documenting that critters behavior, as well as on the top of each method. You should also include inline comments for any complex or confusing code to further explain what that code is doing. Comments should be written in your own words (i.e. not copied and pasted from this spec) and header comments should not include implementation details. See the Style Guide for examples.

Getting Help

If you find you are struggling with this assignment, make use of all the course resources that are available to you, such as:

- Reviewing relevant lecture examples
- Reviewing this week's section handouts
- Reading the textbook
- Visiting the IPL
- Posting a question on the message board

Academic Integrity

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assignment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details, and ask the course staff if you are unclear on whether or not a resource is OK to use.