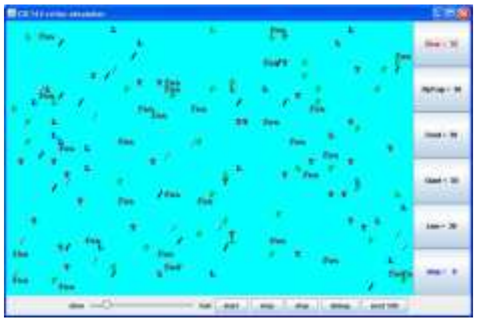
CSE 142, Winter 2020
Programming Assignment #8: Critters (20 points)
Due Wednesday, March 11, 2020, 11:59 PM

This assignment focuses on classes and objects. Turn in `Bear.java`, `Lion.java`, `Giant.java`, and `Husky.java`. Download supporting files on the course web site. Run `CritterMain.java` to start the simulation.

Program Behavior:



You are provided with several client program classes that implement a graphical simulation of a 2D world of animals. You will write classes that define the behavior of those animals. Animals move and behave in different ways. 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. Your classes will define the unique behaviors for each animal.

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 like the image on the right.

`Critter`s 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 don't 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.

Movement:

On each round of the simulation, the simulator asks each critter object what action it wants to perform (by calling the `getMove` method). Each round, a critter will take one of the following actions:

`Action` Constant	**Description**
`Action.HOP`	Move forward one square in the critter's 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

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 Direction getDirection()`	returns the direction you are facing
`public Neighbor getFront()`	returns the neighbor in front of you
`public Neighbor getBack()`	returns the neighbor in back of you
`public Neighbor getLeft()`	returns the neighbor to your left
`public Neighbor getRight()`	returns the neighbor to your right
`public boolean frontThreat()`	returns whether there is an enemy facing you in front of you
`public boolean backThreat()`	returns whether there is an enemy facing you in back of you
`public boolean leftThreat()`	returns whether there is an enemy facing you to your left
`public boolean rightThreat()`	returns whether there is an enemy facing you to your right

The `getDirection` method returns what direction you are facing as one of four `Direction` constants. The next group of four `CritterInfo` methods (the "`get`" methods) return one of four `Neighbor` constants to represent the different kind of neighbors you might encounter. Below are the values of these constants:

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

`Neighbor` 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 cannot find out exactly what species they are. The final group of four methods (the "`threat`" methods) tell you whether there is an enemy in each direction that is facing you (which means they could potentially infect you). An enemy is a critter of a different species.

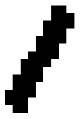
This assignment will be confusing at first, because you do not write the `main` method of the executed program; your code is not in control of the overall execution. Instead, your objects are **part of a larger system**. You might want your critters make several moves at once using a loop, but you can't. The only way a critter can move is to wait for the simulator to ask it for a single move and return that move. This is typical of object-oriented programming, though it might be frustrating at first.

Critter Classes:

The following are the four critter classes to implement. **Each has one constructor that should accept exactly the parameter(s) in the table.** For random moves, each choice should be equally likely; use a `Random` object to make these choices. **You must create exactly the behavior described below for each critter.** See the "Debugging Strategy" section for more information on how to test your critters.

Bear

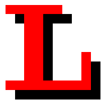
constructor	`public Bear(boolean polar)`
`getColor`	`Color.WHITE` for a polar bear (when polar is `true`), `Color.BLACK` otherwise (when polar is `false`)
`toString`	Alternate on each move between a forward slash (`"/"`) and a backslash (`"\"`) starting with a forward slash (`"/"`).
`getMove`	Infect 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 won't necessarily match the other giants if they are somewhere in the middle of their pattern. Doesn't 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 hadn't already completed its move, it won't get a turn.
- Critters that infect after a hop are slightly favored in that around 20% of the time an infect action will fail if the critter did 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 doesn't 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 shouldn't 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:

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. **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.

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.

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

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.

Follow past style guidelines about indentation, spacing, identifiers, and localizing variables. Place comments at the beginning of each class documenting that critter's behavior, as well as on the top of each method and on any complex code. 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).

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.