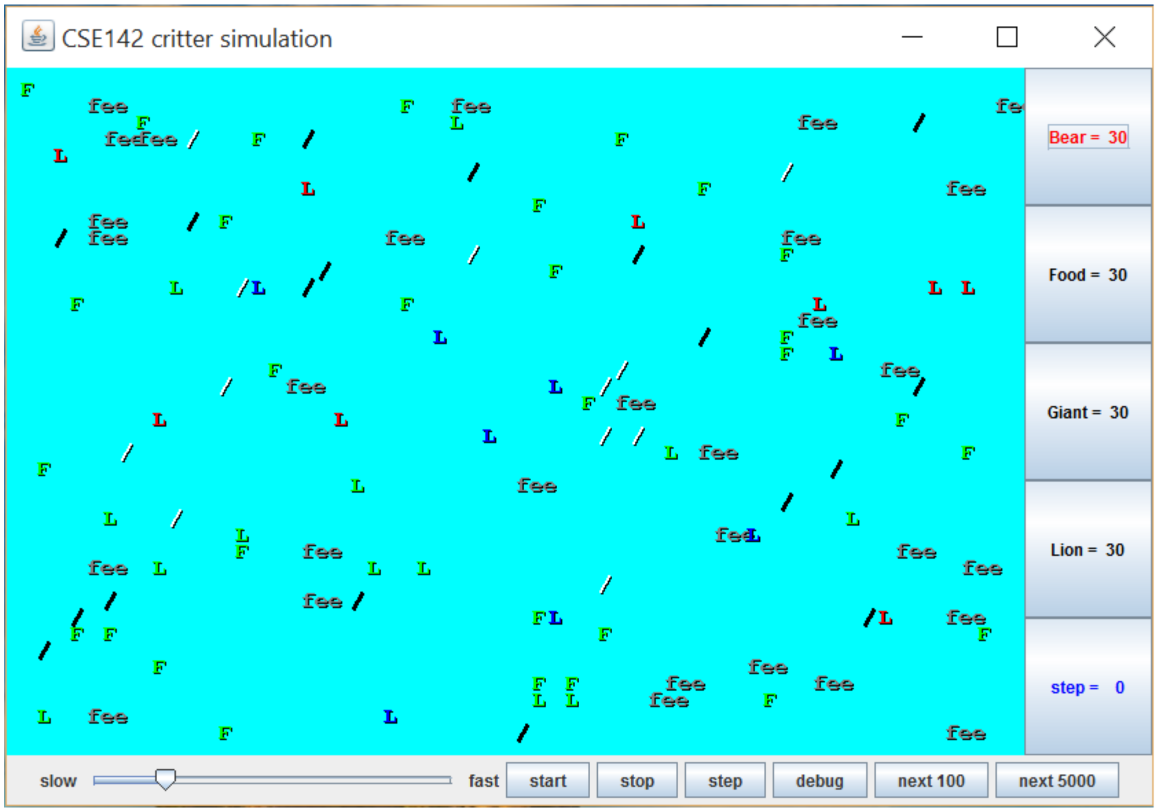
CSE142—Computer Programming I
Programming Assignment #8
due: Tuesday, 8/18/15, 9 pm

Please note that solutions to this homework will not be accepted after 9 pm on Friday, August 21th. This assignment will give you practice with defining classes. 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. This idea of animals transforming into a different species appeared in many Star Trek episodes, particularly the Next Generation episode called "Identity Crisis."

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:



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 it 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 automatically get access to these methods and constants. You then give new definitions to certain methods to define the behavior of your critters.

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

```
public Action getMove(CritterInfo info) {
    ...
}

public Color getColor() {
    ...
}

public String toString() {
    ...
}
```

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 plus a method to find out how much your critter has been moving lately. Below are the methods of the CritterInfo class:

Method	Description
`public Direction getDirection()`	returns direction you are facing
`public int getMoveScore()`	returns # of successful hops in the last 10 turns
`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 getFrontDirection()`	returns the direction of the neighbor in front
`public Direction getBackDirection()`	returns the direction of the neighbor in front
`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 of the CritterInfo class tells you what direction you are facing. There are 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 first group of four 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 side of you are facing. Those methods return north if there is a wall or empty on the side you are asking about.

This program will probably be confusing at first because this is the first time where 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). The one exception is that you are to define a class called Bear that takes a boolean parameter specifying whether it is a black bear or a polar bear, which will change how its color is displayed. The simulator treats bears in a special way, basically flipping a coin each time it constructs one to decide whether or not to create a white bear or black bear. Your Bear class does not have to figure this out. You just have to make sure that each bear pays attention to the boolean value passed to the constructor by the simulator.

You are to implement four classes.

Bear

Constructor	`public Bear(boolean polar)`
Color	`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()`
Color	Randomly picks one of three colors (`Color.RED`, `Color.GREEN`, `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.
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()`
Color	`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.

Husky

Constructor	`public Husky()`
Color	You decide
toString	You decide
getMove	You decide

As noted above, you will determine the behavior of your Husky class. On the last day of class, we will host a contest where students will be allowed to compete to see which has the best survival rate.

Some of the style 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. 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 Husky strategy. **There is a bonus point available for particularly good solutions (you would get 1 extra point, for a possible score of 21).**

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. One of the style points will be awarded based on whether you make each field of each class private and whether you initialize all fields in constructors (you only need to include a constructor if you are setting fields to values other than the default zero-equivalent values for each type). You are limited to the features of chapters 1 through 9 of the textbook.

For the random moves, each possible choice must be equally likely. You may use either a Random object or the Math.random() method to obtain pseudorandom values, although if you use a Random object, you are required to make it a field of your object so that you don't have to construct a new one every time you use it.

Be sure that your code is tied to actual moves made by a critter. For example, the bear is supposed to be displayed alternately as a slash or backslash. This should happen as the getMove method is called. The simulator alternates between calling toString and getMove, but your code shouldn't depend on that. 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 applies also to the behavior of giants and lions because they are defined in terms of changes happening after a given number of moves. To accomplish this, make sure that all updates to fields occur only in the getMove method, not in toString or getColor.

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. Don'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, and Husky.java. The files that you need for this assignment will be included in a zip file called ass8.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 ass8.zip, then add your four classes to the folder, compile CritterMain and run CritterMain.

The simulator has several supporting classes that are included in ass8.zip (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 writeup. The second is a strategy called FlyTrap that was discussed in lecture. Both of these class definitions appear in ass8.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.

For those who want to produce critters that "win" in the sense of taking over the world, you will want to understand some subtleties about the order in which actions are performed. At the start of each round, the simulator asks each critter what it wants to do. The simulator then shuffles the critters and has each one try to complete its move. This means that your critter may decide to infect another other critter in front of it, but that other critter might hop out of the way before your critter's turn. 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 someone else during the round, and if the infected critter hadn't already completed its move, it won't get a turn.

The shuffling is done by choosing a random number based on the move score for each critter. The move score is the number of successful hops in the last 10 turns. The random numbers are chosen in the range 0–19. The higher the number a critter gets, the earlier it will get its turn. A critter with a move score of 0 chooses a number in the range 0–9. A critter with a move score of 1 chooses from a range 0–10. This continues up to a critter with a move score of 10 choosing a number in the range 0–19. This gives a small advantage to critters that move more. Taking advantage of this ordering can allow you to write more competitive critters.

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, facing in the same direction, and with the same move score as the critter it is replacing.

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.

The simulator also indicates the "step" number as the simulation proceeds (initially displaying a 0). Below are some suggestions for how you can test your critters:

- **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.