

# CSE 142, Autumn 2018

## Programming Assignment #9: Critters (20 points)

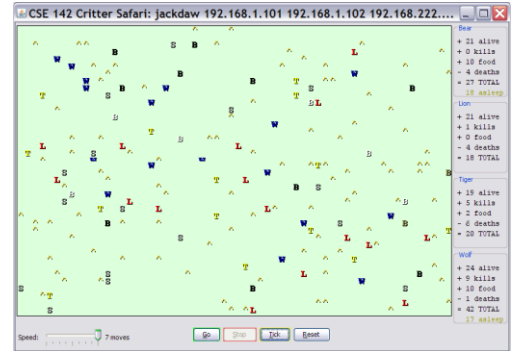
### Due Tuesday, December 4th, 9:00 PM

This assignment focuses on classes and objects. Turn in `Ant.java`, `Bird.java`, `Hippo.java`, `Vulture.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. Your classes will define the unique behaviors for each animal.

The **critter world** is divided into cells with integer coordinates. The world is 60 cells wide and 50 cells tall. The upper-left cell has coordinates (0, 0); `x` increases to the right and `y` increases downward.



### Movement

On each round of the simulation, the simulator asks each critter object which direction it wants to move. Each round a critter can move one square **north**, **south**, **east**, **west**, or stay at its current location (**center**). The world has a finite size, but it wraps around in all four directions (for example, moving east from the right edge brings you back to the left edge).

This program will be confusing at first, because you do not write its `main` method; 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 experience can be frustrating, but it is a good introduction to object-oriented programming.

### Fighting/Mating

As the simulation runs, animals may collide by moving onto the same location. When two animals collide, if they are from different species, they **fight**. The winning animal survives and the losing animal is killed and removed from the game. Each animal chooses one of `Attack.ROAR`, `Attack.POUNCE`, or `Attack.SCRATCH`. Each attack is strong against one other (e.g. roar beats scratch) and weak against another (roar loses to pounce). The following table summarizes the choices and which animal will win in each case. To remember which beats which, notice that the starting letters of "Roar, Pounce, Scratch" match those of "Rock, Paper, Sissors." If the animals make the same choice, the winner is chosen at random.

		<i>Critter #2</i>		
		<code>Attack.ROAR</code>	<code>Attack.POUNCE</code>	<code>Attack.SCRATCH</code>
<i>Critter #1</i>	<code>Attack.ROAR</code>	random winner	#2 wins	#1 wins
	<code>Attack.POUNCE</code>	#1 wins	random winner	#2 wins
	<code>Attack.SCRATCH</code>	#2 wins	#1 wins	random winner

If two animals of the same species collide, they **mate** to produce a baby. Animals are vulnerable to attack while mating: any other animal that collides with them will defeat them. An animal can mate only once during its lifetime.

### Eating

The simulation world also contains **food** (represented by the period character, ".") for the animals to eat. There are pieces of food on the world initially, and new food is gradually added into the world over time. As an animal moves, it may encounter food, in which case the simulator will ask your animal whether it wants to eat it. Different kinds of animals have different eating behavior; some always eat, and others only eat under certain conditions.

Every time one class of animals eats a few pieces of food, that animal will be put to **sleep** by the simulator for a small amount of time. While asleep, animals cannot move, and if they enter a fight with another animal, they will always lose.

### Scoring

The simulator keeps a score for each class of animal, shown on the right side of the screen. A class's score is based on how many animals of that class are **alive**, how much food they have **eaten**, and how many other animals they have **killed**.

## Provided Files:

Each class you write will **extend a superclass** named `Critter`. This is an example of inheritance, as discussed in Ch. 9 of the textbook. 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. Your class headers should indicate the inheritance by using the `extends` keyword, such as:

```
public class Ant extends Critter { ...
```

The `Critter` class contains the following methods, some or all of which you will write in each of your classes:

- `public boolean eat()`  
When your animal encounters food, our code calls this on it to ask whether it wants to eat (`true`) or not (`false`).
- `public Attack fight(String opponent)`  
When two animals move onto the same square of the grid, they fight. When they collide, our code calls this on each animal to ask it what kind of attack it wants to use in a fight with the given opponent.
- `public Color getColor()`  
Every time the board updates, our code calls this on your animal to ask it what color it wants to be drawn with.
- `public Direction getMove()`  
Every time the board updates, our code calls this on your animal to ask it which way it wants to move.
- `public String toString()`  
Every time the board updates, our code calls this on your animal to ask what letter it should be drawn as.

By declaring that your class `extends Critter` as shown above, you receive a **default version** of these methods. The default behavior is to never eat, to always forfeit fights, to use the color black, to always stand still (a move of `Direction.CENTER`), and to display as a "?". If you don't want these defaults, rewrite (override) the necessary methods in your class with your own behavior.

For example, below is a critter class `Stone`. A `Stone` is displayed with the letter `S`, is gray in color, never moves or eats, and always roars in a fight. Your classes will look like this class, except with fields, a constructor, and more sophisticated code. The `Stone` does not need an `eat` or `getMove` method; it uses the default behavior for those operations.

```
import java.awt.*;    // for Color

public class Stone extends Critter {
    public Attack fight(String opponent) {
        return Attack.ROAR;
    }

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

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

NOTE: You will **not** necessarily write `extends Critter` on every single critter class you write. If you find that two critter classes are very similar to each other, you should have one extend the other to reduce redundancy.

## Running the Simulator:

When you press the Go button, it begins a series of turns. On each turn, the simulator does the following for each animal:

- move the animal once (calling its `getMove` method), in random order
- if the animal has moved onto an occupied square, fight! (call both animals' `fight` methods) or mate
- if the animal has moved onto food, ask it if it wants to eat (call the animal's `eat` method)

After moving all animals, the simulator redraws the screen, asking each animal for its `toString` and `getColor` values. It can be difficult to test and debug with many animals. We suggest adjusting the initial settings to use a smaller world and fewer animals. There is also a **Debug checkbox** that, when checked, prints console output about the game behavior.

## Critter Classes:

The following are the five 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 or the `Math.random` method.

### 1. Ant

<b>constructor</b>	<code>public Ant(boolean walkSouth)</code>
<b>color</b>	red
<b>eating behavior</b>	always eats
<b>fighting behavior</b>	always scratches
<b>movement</b>	if the Ant was constructed with a <code>walkSouth</code> value of <code>true</code> , then alternates between south and east in a zigzag (S, E, S, E, ...); otherwise, if the Ant was constructed with a <code>walkSouth</code> value of <code>false</code> , then alternates between north and east in a zigzag (N, E, N, E, ...)
<b>toString</b>	"%" (percent)



### 2. Bird

<b>constructor</b>	<code>public Bird()</code>
<b>color</b>	blue
<b>eating behavior</b>	never eats (always returns <code>false</code> )
<b>fighting behavior</b>	roars if the opponent looks like an Ant ("%"); otherwise pounces
<b>movement</b>	a clockwise square: first goes north 3 times, then east 3 times, then south 3 times, then west 3 times, then repeats
<b>toString</b>	"^" (caret) if the bird's last move was north or it has not moved; ">" (greater-than) if the bird's last move was east; "v" (uppercase letter v) if the bird's last move was south; "<" (less-than) if the bird's last move was west



### 3. Hippo

<b>constructor</b>	<code>public Hippo(int hunger)</code>
<b>color</b>	gray if the hippo is still hungry (would still eat food); otherwise white
<b>eating behavior</b>	eats the first <code>hunger</code> times it is called; never eats after that
<b>fighting behavior</b>	if this <code>Hippo</code> is hungry (would still eat food), then scratches; otherwise pounces
<b>movement</b>	moves 5 steps in a random direction (north, south, east, or west), then chooses a new random direction and repeats
<b>toString</b>	the number of pieces of food this <code>Hippo</code> still wants to eat, as a <code>String</code>



The **Hippo constructor** accepts a parameter for the maximum amount of food that `Hippo` will eat in its lifetime (the number of times it will return `true` from a call to `eat`). For example, a `Hippo` constructed with a parameter value of 8 will return `true` the first 8 times `eat` is called and `false` after that. Assume that the value passed is non-negative.

The **toString method** for a `Hippo` returns the number of times that a call to `eat` that would return `true` for that `Hippo`. For example, if a new `Hippo(4)` is constructed, initially its `toString` return "4". After `eat` has been called on it once, calls to `toString` return "3", and so on, until the `Hippo` is no longer hungry, after which all calls to `toString` return "0". You can convert a number to a string by concatenating it with an empty string. For example, "" + 7 makes "7".

#### 4. Vulture

<b>constructor</b>	public Vulture()
<b>color</b>	black
<b>eating behavior</b>	Eats if vulture is hungry. A vulture is initially hungry, and remains hungry until it eats <i>once</i> . After eating, it will become non-hungry until he gets into a fight. After one or more fights, it will be hungry again. ( <i>see below</i> )
<b>fighting behavior</b>	roars if the opponent looks like an Ant ("%"); otherwise pounces
<b>movement</b>	a clockwise square: first goes north 3 times, then east 3 times, then south 3 times, then west 3 times, then repeats
<b>toString</b>	"^" (caret) if the vulture's last move was north or has not moved; ">" (greater-than) if the vulture's last move was east; "v" (uppercase letter v) if the vulture's last move was south; "<" (less-than) if the vulture's last move was west



A Vulture is a specific sub-category of bird with some changes. Think of the Vulture as having a "hunger" that is enabled when it is first born and also by fighting. Initially the Vulture is hungry (so eat would return true from a single call). Once the Vulture eats a single piece of food, it becomes non-hungry (so future calls to eat would return false). But if the Vulture gets into a fight or a series of fights (if fight is called on it one or more times), it becomes hungry again. When a Vulture is hungry, the next call to eat should return true. Eating once causes the Vulture to become "full" again so that future calls to eat will return false, until the Vulture's next fight or series of fights.

#### 5. Husky

<b>constructor</b>	public Husky() (must not accept any parameters)
<b>all behavior</b>	you decide (see below)



**You will decide** the behavior of your Husky class. Part of your grade will be based upon writing creative and non-trivial Husky behavior. The following are some guidelines and hints about how to write an interesting Husky.

Your Husky's fighting behavior may want to utilize the opponent parameter to the fight method, which tells you what kind of critter you are fighting against (such as "%" if you are fighting against an Ant). Your Husky can return any text you like from toString (besides null) and any color from getColor. Each critter's getColor and toString are called on each simulation round, so you can have a Husky that displays differently over time. The toString text is also passed to other animals when they fight you; you may want to try to fool other animals.

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.

Each critter class has some **additional methods** that it receives by inheritance from Critter. Your Husky may want to use these methods to guide its behavior. (None of the methods below are needed for Ant, Bird, Hippo, or Vulture.)

- public int getX()                    public int getY()  
Returns your critter's current x and y coordinates.  
For example, to check whether your critter's x-coordinate is greater than 10, you would write code such as:  
    if (getX() > 10) {
- public int getWidth()    public int getHeight()  
Returns the width and height of the simulation grid.
- public String getNeighbor(Direction direction)  
Returns a String representing what is next to your critter in the given direction. " " means an empty square.  
For example, to check if your neighbor to the west is a "Q", you could write this in your getMove method:  
    if (getNeighbor(Direction.WEST).equals("Q")) {
- public void win()    public void sleep()    public void mate()    public void reset()  
  public void lose()    public void wakeup()    public void mateEnd()  
The simulator calls these methods on your critter to notify you when you have won/lost a fight, when you have been put to sleep/woken up, when you have started/ended mating, or when the game world has reset, respectively.

## Development Strategy:

The simulator runs even if you haven't completed all the critters. The classes increase in difficulty from `Ant` to `Bird` to `Hippo` to `Vulture`. We suggest doing `Ant` first. Look at `Stone.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 will be about how well you follow proper **object-oriented** programming style. You should **encapsulate** the data inside your objects, 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.

Another aspect of the style of this program is **inheritance**. Your critter classes should properly extend a superclass as described. Inheritance can also be used to remove redundancy between classes that are similar, and you should make use of this concept in your classes as appropriate. In other words, if two of your critter classes *A* and *B* are very much alike, you should have *B* extend *A* rather than having both simply extend `Critter`.

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. 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 `Ant`, `Bird`, `Hippo`, and `Vulture` together occupy just under 199 lines including blank lines and comments (117 "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.