PSA 4: Inheritance and Critter Simulation

Prepare for battle. Write several lineages of Critters and test your gameplay strategy by designing your own Critter for combat. The rules are generally simple, but there are some nuances to the rules. Your strategies can be highly complex. You will learn both a new object-oriented design paradigm using inheritance as well as entering challenges against both our provided and your classmates' Critters.

The code will have effects that are mostly visual. **Therefore, testing out your programs will be doing the simulation and observing the behaviors**. In addition to writing five classes to our specifications, you will implement one more Critter. You may iteratively improve its strengths in order to beat the five aptly named Critters and the Mystery Critter and the Raccoon for extra credit. The implementation of Mystery and Raccoon will be hidden from you. Finally, your Critter will be entered into a large scale Critter competition with your classmates to see whose Critter will come out the strongest and luckiest. The top 3 winners from each section of the competition will receive a 3D printed sculpture of an animal. Below are some of the critters you could win:

For this assignment, **we will not purposefully test edge cases**, but the program (simulation) must still run without crashing. The program IS the simulation! If your Critter works for the program on your computer, it will work for ours.

Thank you to the developers of *Critters*. The original version of *Critters* was developed at the University of Washington by Stuart Reges and Marty Stepp.

Follow the code of academic integrity. May everyone enjoy this assignment and compete in a fair game.

Helpful Information:

- **Online Communication: Using Piazza, Opening Regrade Requests**
- **Getting Help from Tutor, TA, and Professor's Hours**
    - **Lab and Office Hours** (Always refer to this calendar before posting.)
- **Academic Integrity: What You Can and Can't Do in CSE 8B**
- **Setting Up the PSA**
- **How to Use Vim**
- **Style Guidelines**
- **Submitting on Vocareum and Grading Guidelines**

Table of Contents:

Part 1: .vimrc [5 Points]

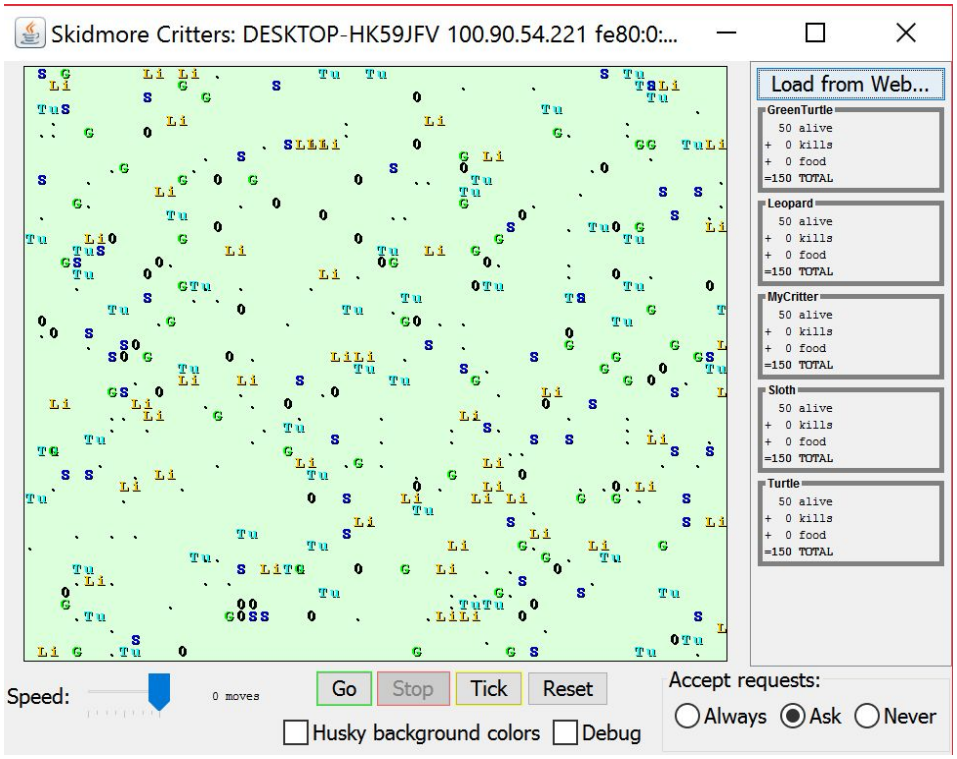
If you have not created a custom configuration for Vim yet, here is your chance to make Vim a little more friendly. If you have already created a vimrc file in the past, you're already good to go. Submit that vimrc. In the file header, comment whether you made this before or made this for the assignment.

In your home directory (~), make a file called **.vimrc** with the dot in front. The file must do at least:

1. Make Vim use a different color scheme from the default. Pick your favorite theme. Past favorites are desert and elflord.
2. Always show the line numbers in any file opened in vim.
3. Insert 4 spaces instead of a tab when you press the tab key (it is **not** enough to simply set the width of each tab to 4 - they must be actual spaces, not tabs).

The Arena

Several classes in the starter code implement a graphical simulation of a 2D world with many animal moving around in it. You will write a set of classes that define the behavior of these animals. As you write each class, you are defining the unique behaviors for each animal. Notice that there may be similarities between animals of closer family relations, such as the Turtle and GreenTurtle. The Critter World is divided into cells with integer coordinates. The world is 60 cells wide and 50 cells tall by default. The upper-left cell has coordinates (0, 0). The x coordinate increases to the right. The y coordinate increases downwards.



Movement

On each round of the simulation, the simulator asks each Critter object which direction it wants to move by calling its getMove method. Each round a Critter can move one square north, south, east, west, OR stay at its current location (i.e. 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). It might be tempting to allow your critters to make several moves at once using a loop, but you can't. The only way a critter moves is to wait for the simulator to ask it for a single move and return that move.

Fighting

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 removed from the game. Each animal chooses one of Attack.FORFEIT, Attack.ROAR, Attack.POUNCE, or Attack.SCRATCH as their attack mode. Each strong against one other attack (e.g. roar beats scratch) except for forfeit.

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, Scissors." If the animals make the same choice, the winner is chosen with a coin flip.

The relationship of different attack actions are listed in the following table.

Critter #2

Critter #1		**ROAR**	**POUNCE**	**SCRATCH**	**FORFEIT**
	ROAR	random (50% chance)	#2	#1	#1
	POUNCE	#1	random (50% chance)	#2	#1
	SCRATCH	#2	#1	random (50% chance)	#1
	FORFEIT	#2	#2	#2	random (50% chance)

Mating

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. The “baby” will be a full adult by birth and will spawn next to the parent critters when they finish mating.

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 slowly grows 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. Once an animal has eaten a few pieces of food, that animal will be put to "sleep" by the simulator for a small amount of time. During the sleeping period the animal will automatically forfeit all fights, meaning it will lose to all other critters that attack it.

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 defeated.

The Assignment

Each class you write in this section will inherit from a superclass, and may be inherited by a subclass. We take advantage of inheritance in two ways: since subclasses automatically inherit methods from their superclass, if we want a certain method to be uniform across a family of classes, we can simply define the method in the superclass. The other way we take advantage of inheritance is that we minimize the amount of code that we have to write. This reduces the possibility of error. Inheritance provides the programmer assistance in streamlining the code writing process.

There is a UML diagram of the Critter family tree at the beginning of Part 1. In this diagram, we tell you in addition to what bloodline that the Critters have, which classes are provided, which classes you do not need to edit, and which classes require you to write the file from scratch.

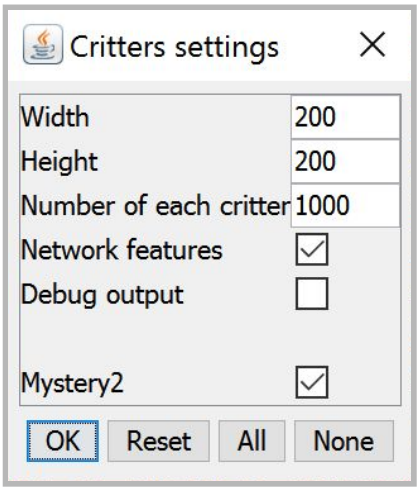
First, look at the Critter.java class. It contains helpful documentation from various previous writers of the Critter class and may prove very useful to knowing what each method does, which ones are provided, and which ones you might be working on. This class is an abstract class.

Just by defining a class to "extends Critter", you receive a default version of the methods defined in Critters.java. Look inside the file to see what the default behaviors are for each of the following methods: eat, fight, getColor, getMove, and toString. If you don't want this default behavior, you will then override the inherited methods in your class through your own definition of the method. The method will have the exact same signature and return type, but your own implementation will be used over the inherited one. Here is an example of a class that inherits from Critter:

```
import java.awt.*;
public class Stone extends Critter {

	@Override	/* You MUST use @Override for every method you override. We
			require it because it introduces you to using annotations and
			ensures that you will correctly override methods! */

	public Attack fight (String opponent) {
		return Attack.ROAR;
	}

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

	@Override
	public String toString() {
		return "S";
		//This double quote may not compile correctly if you copy it into Java.
	}
}
```

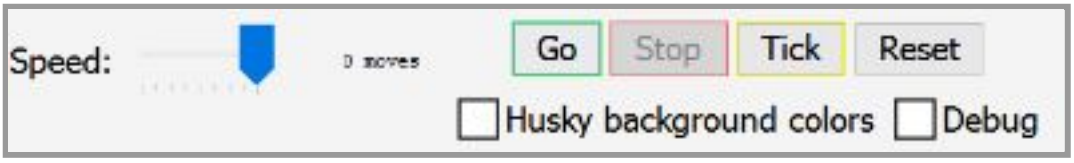
Running the Simulator

The steps to getting the simulator running are slightly longer than 2048's steps. First, compile all necessary classes. You can compile separately if needed, but **`javac *.java`** is most convenient. Next, run the simulator with **java CritterMain**. The following screen will appear. Enabling"Debug output" will print the actions your critters take to the terminal. You can also choose the size and amount of critters that initially appear in the world, and select which critters that will start in the world.



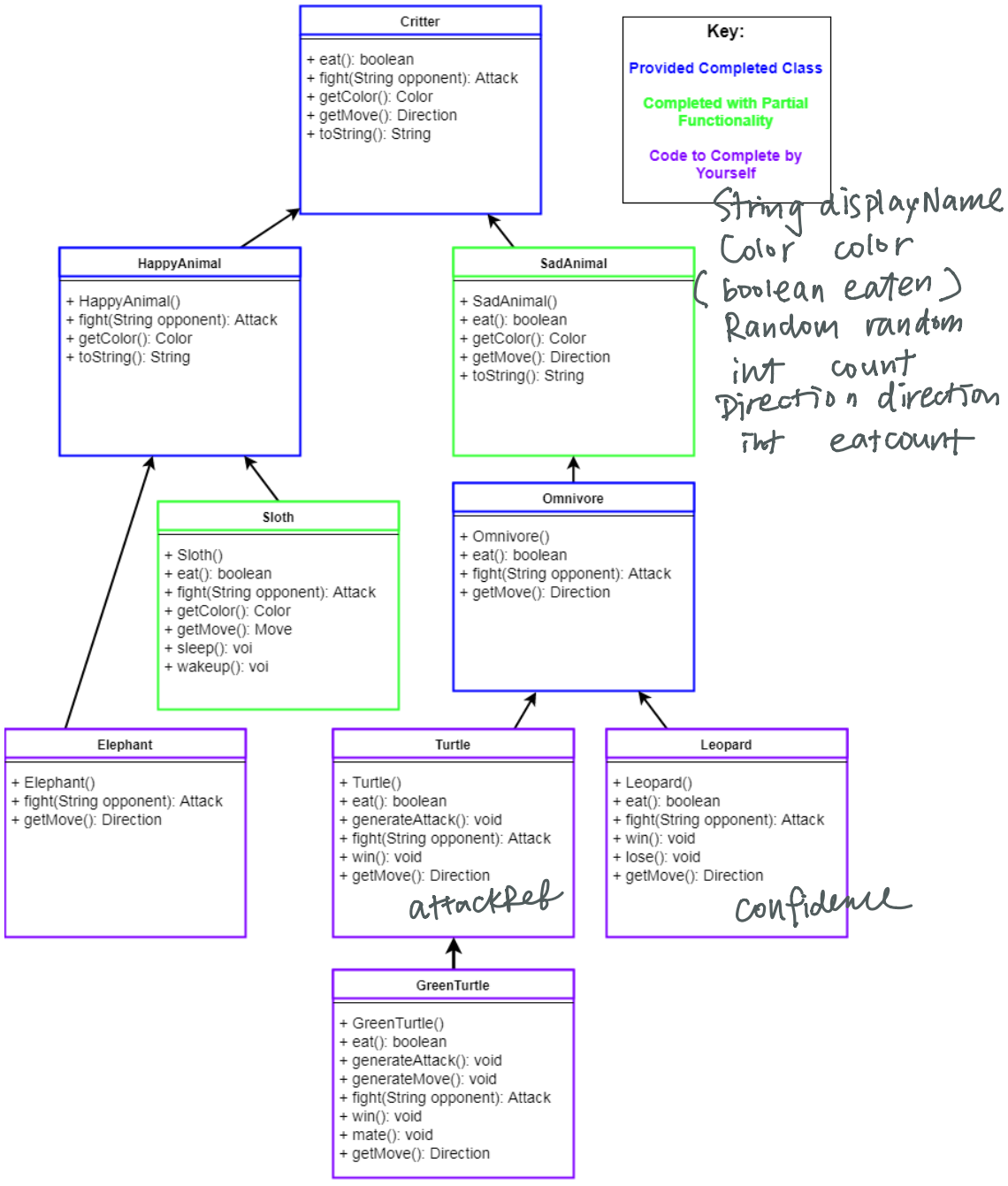
You may see some warnings of deprecated APIs when you run the simulator. You can ignore these warnings.

You have many options in the next screen. You can adjust the speed of the simulator. When you press Go, the simulation will take off and run. You can Stop the simulation. For visual testing, you can click Tick, which will run one round of the simulation. If you would like to toggle debug messages on your terminal as your simulation runs, click the check box for "Debug".



Part 2: Getting to Know the Critter Family [10 Points]

This portion introduces you to writing classes with inheritance. If you have not read the Critter.java file, **do so now.** Critter is the abstract class from which all subclasses will be derived, directly or indirectly. This file will also contain enums that are important for you to use in the following tasks. There are five classes in total you will work on, hence there will be five classes you will write with style. See the UML diagram below for a graph of the family. Note: the instance variables are not listed. You can have additional instance variables if needed for all the class that you implement.



Inheriting from Critter

Next, take a look at HappyAnimal.java, which is a provided class that you do not need to edit. Its functions are incomplete though. So what would happen if you instantiated HappyAnimals on the simulation? Try it now by compiling and run the simulation on HappyAnimal. You don't have to modify the HappyAnimals class for this PSA.

By writing a HappyAnimal class, we can define a subset of Critters that are "happy". The subclasses of HappyAnimal will inherit its characteristics, and you will have the choice of overriding any of those characteristics.

Your task is to implement a class that is parallel to the HappyAnimal: the SadAnimal class (SadAnimal.java in the starter code). SadAnimal is a partially implemented Critter whose behavior can be seen in the simulation as incomplete. Notice that there is already a provided class that extends SadAnimal - the Omnivore. The methods you write in SadAnimal will not only be affecting SadAnimal.java, but also all subclasses that choose to extends SadAnimal but chooses to not override the methods. Because of this, writing code for SadAnimal is very important to get right. The definition will affect many other classes!

SadAnimal class

- **Instance variables:** The SadAnimal knows the following things: a String that it will use to identify itself, a Color it will display itself with, a boolean to know whether it has eaten yet, a Random object when it needs to be able to make arbitrary decisions or some other purpose, and an int by which the SadAnimal or its descendants will count. **All these fields should be protected. You can have more instance variables if you think they are necessary.**
- **No-arg constructor:** Instantiate all instance variables except the Random and count variables. You will notice that SadAnimal in particular did not have a use for Random or count but its subclasses might. The boolean indicator variable should be initialized as false, the color should be red, and the name of the SadAnimal is the three character String: ":-(".
- **Override getMove method:** SadAnimal has the ability to move but it is only able to alternate between moving south, or west. By default, it will always move west first, and then when it eats, it changes direction to south. When it eats again, it will go west, and the pattern continues. If the World runs out of food, it's stuck going in one direction!
- **Override eat method:** When presented with food, SadAnimal will always eat.
- **Override getColor method:** Return the color field. Don't return red directly as a hard-coded color. Make sure you return the field.
- **Override toString method:** Return the string field. Similarly like the getColor method, don't return the hardcoded name. Return the correct field.

Inheriting from Classes that Inherited from Critter

Now that you have finished implementing SadAnimal, we will move one level lower. The Sloth is a fully implemented animal! **Be careful though, that does not mean that Sloth overrides all the methods it inherited from HappyAnimal**; it means simply that all the functions will have some kind of implementation, simple or hard, and inherited or overridden. Also, remember that the Sloth inherits instance variables from its superclasses too.

Sloth class

- **Instance Variables:** Specific to the Sloth, it is very important to remember whether it has eaten previously, and whether it moved north. **You should design how many instance variables you need and their types. The goal is to make sure your methods work correctly with the help from these instance variables. All instance variables should be protected.**
- **Constructor:** Upon birth, Sloths are named "S", know that they have not yet eaten previously, and intuitively decide to move north first.
- **Override eat method**: Sloth loves to eat. When encountering food, the Sloth will always eat. A sloth should keep track of the number of times it has eaten, because it will use this to determine what attack it is going to use.
- **Override fight method**: In general, a sloth will scratch in self defense if it encounters another critter. However, some sloths have grown large enough to pounce instead. If a sloth has eaten 10 food or more, then it will pounce instead.
- **Override getMove method:** The Sloth will alternate in moving north and east, starting with north when it is first born.
- **Override sleep and wakeup methods**: Sloth's identity is an "S" whenever it wakes up, but whenever it sleeps, it will appear to other Critters as "Zzz".

Now, let's look at the Omnivore, which extends from SadAnimal that you implemented. Notice that this is not a particular animal! Although the Sloth is a fully implemented animal inherited from HappyAnimal, we do not necessarily need to have the same rate of implementation completeness going down the inheritance family. We fully implemented Sloth extending from HappyAnimal because **we could**. But we can also implement a slightly complete, but not fully complete, animal from SadAnimal, because **we can**.

Part 3: Completing the Critter Family [40 Points]

This part continues directly from Part 2. In this section, you will define Critter classes from scratch, and then override certain methods inherited from Critter, SadAnimal, and Omnivore. In addition, there will be certain new methods that will be implemented as part of the three classes you will implement. There will be two lineages that inherit from the Omnivore: Turtle and GreenTurtle, and Leopard.

Let's implement the Leopard first. The Leopard is the last subclass in its lineage, which goes Critter → SadAnimal → Omnivore → Leopard. The Leopard inherits from Omnivore, which is a provided file, but has inherited some fields from SadAnimal which you implemented. So, while some fields inherited from Omnivore were inherited directly, others were inherited from SadAnimal. The Leopard has one unique instance variable that needs to be implemented, which results in a more unique behavior for eat and fight.

Leopard Class

- **Instance variables:** Each Leopard, in addition to the instance variables inherited from its superclasses, will all telepathically keep track of their confidence together. The confidence starts at 10 when the simulation starts. When the confidence of one Leopard is affected, All Leopard's confidence will be affected in the exact same way. (Hint: What type of modifier can you apply to a variable to make that variable shared across all instances?)
- **No-arg constructor**: Initialize the variable count (inherited from SadAnimal). The value you use is up to you though it will be wise to see how you might need to use count in the getMove method. Leopards' random (inherited from SadAnimal) will initialize with a seed of 2018. Additionally, the Leopards' self-identity as "Li" and are orange.
- **Override win and lose method**: If a Leopard wins a fight, all Leopards' confidence will increment if their confidence is less than 10. If a Leopard loses, all Leopards will reduce their confidence by 1 if their confidence is greater than zero. The minimum confidence they have is 0, and the maximum is 10. (Think: where do these two methods come from?)
- **Override getMove method**: The Leopard will move south five times, west five times, north five times, and east five times, prowling around this square motion for the remainder of its life. Leopard remembers this by using its inherited memory to incrementally count.
- **Override eat method**: The Leopards will always have (confidence * 10)% chance of eating. If confidence is at 2, then there is a 20% chance of eating.
- **Override fight method**: When fighting, if the opponent is the Omnivore class (NOT a subclass of Omnivore) OR when all Leopards have a confidence HIGHER than 5, then the Leopard will roar. Else if all the Leopards have a confidence less than 2, they are too scared to attack and forfeit. Otherwise the Leopard will scratch.

The Turtle starts a lineage that diverges from Leopard. The Turtle extends from Omnivore and is in general slow and indecisive. The Turtle often likes to think about what attack it will use, and so it makes a decision after a fight and when it encounters food. Turtles are able to walk in one directly only, and gets tired easily so it always waits two times before moving again. The Turtle is thoughtful, but slow and indecisive.

Turtle Class

- **Instance Variables**: an Attack type reference and any other instance variables you may need.
- **No-arg constructor**: Self-identifies as "Tu" and is color cyan. Initialize count to the value you choose, and the random will instantiate with a seed of 8.
- **New method to decide attack - generateAttack()**: Turtle will often think about what attack it wants to use next, and this method sets the attack mode of turtle (i.e. change the attack field of Turtle). When the Turtle chooses a move, it will cycle between roar, scratch and pounce, in that order. For example, the first time that generateAttack() is called, the Turtle will decide to use roar on its first attack. The second time, scratch, and the third, pounce, and then it repeats. The Turtle will decide its attack mode in two different scenarios (i.e. call this generateAttack method): (1) when it wins a fight and (2) when it encounters food. The thought process is the same in all of these actions. Be careful with this description, as it affects more than one method.
- **Override getMove() method**: Turtle will stand still two times, and then move north, repeating this pattern for the duration of the simulation. You can use the inherited count variable to keep track of the pattern.
- **Override fight() method:** The Turtle will always use the attack that it has decided on using. Simply return the attack field.
- **Override win() method:** Upon winning a fight, Turtle decides a new attack.
- **Override eat() method**: The Turtle will never eat, but upon encountering food, the Turtle decides a new attack.

Next, the **GreenTurtle, which should extends from Turtle.** GreenTurtle is even more thoughtful than Turtle. It not only thinks about what attack is should do next, but also which direction to move. GreenTurtle always moves in some direction, and is decisive on what attack it should do. GreenTurtle goes through various thoughts upon doing various actions as described below. In short, although GreenTurtle is indecisive about which direction to move, it is decisive on its attack, fast, and versatile.

GreenTurtle Class

- **Instance Variables**: should remember the next intended Direction to take. You can design any other fields that deem appropriate. (Think: Why did we intentionally leave out the information that GreenTurtle should be able to remember what attack it wants to use?)
- **No-arg constructor**: self-identity is "G" and is green. Random seed is 9. Right after it is born, it has a natural sense of direction to know it should go north first. It will be able to start counting from a value you set. (Where did the count instance variable come from?) It will also make the initial choice to roar.
- **New method to deciding the move - generateMove()**: the GreenTurtle will cycle between the four non-center directions, starting with north, then south, then east, then west. It will go through this thought process (i.e. call the generateMove method) if it wins a fight, and when it eats. Since the GreenTurtle eats only three times, there will be only three times when GreenTurtle thinks of its next move while eating. This method shouldn't take any parameters.
- **Override generateAttack() method:** The GreenTurtle will go through the thought process of choosing its next Attack when it mates, and when it wins a fight. When it chooses a new Attack, GreenTurtle will alternate between roaring and scratching.
- **Override eat() method**: GreenTurtle will eat three times and then never eats again. This means that there is only three times where encountering food will cause the GreenTurtle decide its next direction. You should use the inherited count variable to keep track of it.
- **Override fight() method**: If the opponent the GreenTurtle is facing is Sloth, the GreenTurtle will roar. BUT, if the Sloth is sleeping, then the GreenTurtle will forfeit. Otherwise, the GreenTurtle will use the attack that it had decided on earlier.
- **Override win() method**: Upon winning, the GreenTurtle will decide its next direction and its next attack.
- **Override mate() method**: Upon mating, the GreenTurtle will decide its next attack.
- **Override getMove() method**: The GreenTurtle will move in the direction that it has decided on.

In the **Elephant class, which should extend HappyAnimal**, all elephants will try to travel together in a herd. Additionally, elephants engage on long treks to reach locations they have going to. All elephants have a common location that they all travel towards at once, in order to eventually meet up.

Elephant Class

- **Instance Variables:** Elephants have an x and y goal to which they all travel towards in a pack. This goal is for all elephants (think: what modifier should be applied to these variables). At the beginning you should mark, in some way, that these have not yet been set. **Important:** in order for us to grade the behavior of the Elephant class, it is required that these variables be named **goalX** and **goalY**. If you give these variables a different name, we **may be unable to grade your Elephant class**.
- **No-arg constructor:** The elephants self-identity is "El" and it is colored gray. Notice that Elephant inherited the "random" reference variable from HappyAnimal. Elephant will instantiate a new Random with the seed 2048, and assign this new instance to that inherited reference variable.
- **Override fight() method:** Elephants should always pounce.
- **Override getMove() method:** When move() is called, the Elephant will check if it has reached the goal location it wanted to reach. If it has, or the goal has not been set for the first time, set a new goal as a random location in the simulation (Hint: the method getHeight() and getWidth() in the Critter class will give you the size of the similuation). This new goal applies for **all elephants**. When the Elephant moves, it should always move in a direction towards its goal. If it is further away in the x direction, it should move east or west towards its goal. If the elephant is further away in the y direction, it should move north or south.

Part 4: Inheritance Concepts [10 Points]

This part requires that the previous two parts are implemented and correct to our specification. In README.md, for each of the following questions, provide information about what happened, why you think that is happening, and use the concept of inheritance to support your answer.

You should create a new class file that will not be submitted for the sole purpose of having a main method defined in a class outside the Critter family. Declare and instantiate various Critters inside this new main method, and then try to find out as much as possible, based on your understanding of inheritance, for each of the questions. Each answer must be at least four sentences long, and at most six sentences long.

1. Suppose in the Critter.java file, I add an additional method. It is defined with the following pseudocode:

```
method sayStuff():

      System.out.println("I am a Critter.");
```

How does that affect other Critters?

2. What is the fight behavior of SadAnimal? Explain why it is the case.

3. Suppose that I comment out getColor in HappyAnimal.java. What happens? Why does it happen?

4. In SadAnimal.java, suppose I add one more instance variable that is a public and static String. Who is affected, and what happened? Please also clearly explain why. Experiment (change it, add it, bop it) with this variable in your subclasses.

Part 5: Your Critter [10 Points]

In this part of your assignment, you will create your own Critter from scratch. There is no starter code, of course, because the implementation is entirely up to you. You may choose to write a class that extends from Critter, or any of the Critters from Parts 1 and 2.

There will be a competition between every Critter that all students submit. Therefore, we need a viable way of identifying your Critter. The filename needs to include your first name and your UCSD email address without the domain to be entered into the competition. **(FirstName_email.java)**. However, you may make your Critter's display name (inside the simulation) anything. **The display name must be appropriate.**

You critter must, at a minimum, have its own display name and color choice, and override the **getMove()** and **fight()** methods to receive full credit for this part. You should aim to make it as strong as possible. Be sure to test out your critter's strength with various critter combinations that you implemented from the previous parts.

In the competition, your Critter will compete against other student's Critters. We will run a simulation with many Critters from every student on a powerful machine with 64 Gigabytes of RAM to see whose Critter will come out on top. While strategy may be a large portion of probable success, luck is also a major portion. There will be only one run of the entire competition, with multiple instances of every student's Critter inside the simulation.

The top 3 contestants of each section of the competition will receive 3D printed animals as prizes. May the odds be ever in your favor.

Part 5.1 Well-Adapted Critters [10 Extra Credit]

At this point, you have implemented Critters to our specification with our guidance in starter code, other Critters with hints from the writeup, and your own Critter you added to the Critter family. For the extra credit, your critter needs to be able to win against some more difficult critters that are adapted to their environments.

The conditions for a "win" is the following:

1. The screen size is width = 60 and height = 50. There will be CritterCount = 25 of each Critter.
2. At 1000 moves, the game is considered over. Even though the game will continue to run, we specify that 1000 moves is the minimum time of the simulation.
3. Your Critter must have the highest score and least one of your Critters must still be alive in the World.

Two out of three simulation executions done by the grader must result in a Win for your Critter for full credit. Any less will be considered for partial credit based on the performance as seen by the simulator.

Your Critter must win in the following environments:

1. A World with Sloths, Leopards, Turtles, GreenTurtles, Elephants and **Mystery**. [2 Points]
2. A World with Sloths, Leopards, Turtles, GreenTurtles, Elephants and **Raccoons**. [8 Points]

Only the class file of Mystery and Raccoons are provided. There will be no documentation about these classes. To find out what is the functionalities of this class, run them in the simulator and observe their behavior with other classes, including your own class. Use the Tick button if you like.

Part 6: Program Descriptions [5 Points]

Write a paragraph on how you tested your Critters. Examples of your testing procedures might include what features of the Critter simulation you used to help you run your tests.

Write a paragraph about the entire Critters program itself. This program description should NOT be about each class related to the Critters. That's the file and class headers' job. Rather, the description should talk about what the simulator does, how the simulator reads files into its program, and how you change the settings to your needs.

Style Guidelines (Link) [20 Points]

#1-10 on the class website.

Submission and Grading

The grading for the Extra Credit will be slightly different. **Your Critter may not win the simulation every time.** Therefore, you need to ensure that your Critter performs with great strength in order to beat the odds. We run each student's Critter with the scenarios as listed above. **Regrade requests for rerunning the simulation will be rejected.** There will be three runs of the simulation. To be confident that your critter will win two times, you should ensure that your Critter can win in your simulations most, if not all, of the time. If the Critter loses twice in our test runs, that is the fail-safe for full points. Some partial credits may be given if your critter wins between zero to two times.

Start early, as most of the challenges might be in getting your code to match our specifications.

Submission Files (Make sure your submission contains all of the files and that they work on the ieng6 lab machines!)

- **README.md**
- **.vimrc**
- **SadAnimal.java**
- **Sloth.java**
- **Turtle.java**
- **GreenTurtle.java**
- **Leopard.java**
- **Elephant.java**
- **FirstName_email.java**

Maximum Score Possible: 110/100 Points