PA 2: Critters and Inheritance

Read all instructions in this document before starting any coding!

In this assignment, you will be using the idea of inheritance to implement different Critter animals. You will also be creating your own Critter animal, and watch as it battles it out against the others in an arena!

Starter code can be downloaded on Vocareum or here:
https://drive.google.com/a/eng.ucsd.edu/file/d/1xwUqhIDFQPuTzLCQUV83MDuWV8xBq-UU/view?usp=sharing

Helpful Information:

- Getting Help
  - Lab and Office Hours (Always refer to this calendar before posting.)
- Academic Integrity: What You Can and Can't Do in CSE 11
- Setting Up PAs
- How to Use Vim and Eclipse
- Code Style Guidelines
- Submitting on Vocareum and Grading Guidelines

Table of Contents:

Overview

This project enables you to test your game playing strategy by designing a Critter for head-to-head combat in a tournament. Although the rules are fairly simple - the strategy can be highly complex. Although some basic coding skills in Java are required, the real challenge here is in your strategy.

The first part of the project requires you write the code for the Critters who initially populate the world. In the second part of the project, you'll write your own Critter to compete in this world.

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.

Starter Code and Setup

JavaFX works best with Java version 8. You may need to update your Java to **version 8** (if you haven't already) for this assignment for best results.
(Or you could work in one of the CSE labs or ssh into ieng6 server - we have the environment already set up for you)
Here are the instructions to check your java version.

The **starter code** should contain the following files:

```
// code files given in starter code
Critter.java             // do not change - class Critters will extend from
CritterMain.java         // do not change - this class manages gameplay
Easy.class               // Easy Bot
Medium.class             // Medium Bot
Hard.class               // Hard Bot
PA2Tester.java           // Tester
```

How to Compile and Run

For this PA, you will need to compile all your files at once as many of the .java files depend on multiple classes. Thus, you should use the following commands to compile and run your code:

```
> javac *.java
> java CritterMain
```

Note: you will see lots of class files appear when you compile CritterMain.java. This is normal. When you compile, you may also see messages such as these appear on your terminal:

```
Note: CritterMain.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: CritterMain.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Ignore these messages.

Part 1: Understanding the Starter Code [10 Points]

IMPORTANT **Once you're done with the answering all questions:** Draw the class hierarchy for the following classes: Bear, Lion, Tiger, Dragon, Critter. Upload to Vocareum a picture of this class hierarchy in a separate file named **CritterDrawing.png**.

As with PA1 (and as is typical in software development!) we've given you a lot of code to start with. The starter code provides the functionality of PA2. In order to make sure that you understand the starter code, and the underlying concept of Inheritance that the starter files use, answer the following questions – using short answers – in a file called **README.txt** and list your answers as clear as possible.

*(It should be a **plain text file** with file extension **".txt"** - not "README" or "readme" or "README.pdf", but "README.txt")*

General Inheritance Questions

1. What does the line super(5); do when placed in a class's constructor?
2. Assume you have a base class Superclass, and class Subclass that extends Superclass. If you have the following code:

   ```
   Superclass c = new Subclass();
   ```

   will this cause an error? (Assume Subclass has a default constructor). Why or why not?
3. For the classes Superclass and Subclass as described above, which is true:
   a. A Superclass "is a" Subclass
   b. A Subclass "is a" Superclass
   c. Both of these
   d. Neither of these.
4. Now assume you have classes Person and Student as defined in class. Assume that you have a Person type variable that *references a Student type object.* I.e. Person p = new Student("Sally", 18); If the method sayHi() is defined in the Person class and overridden in the student class, which version of the method will be called when I write p.sayHi(), the Person's version or the Student's version?

Inheritance Questions Based on the Implementation of Critters.java

5. Which methods from Critter will you override in the Bear class?
6. Will the following cause an error: Bear b = new Critter(); Why or why not?
7. True or false: you will need to add a line of code that calls the method "eat" critters to eat food during the simulation.
8. True or false: The member variables in the Dragon class can be different and have no relation to the member variables in the Tiger class.
9. Explain how you could use a static member variable in the Bear class to keep track of the total number of Bears that had ever been created. (Note: There's no need to actually do this in the code).

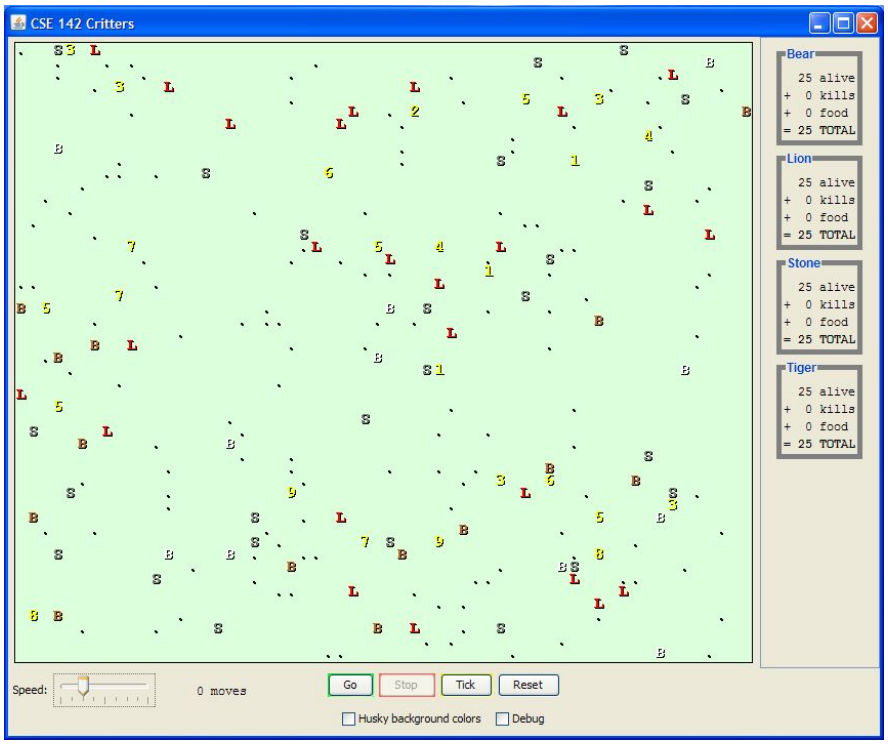
Files to submit for Part 1:

- README.txt
- CritterDrawing.png

Understanding the Arena

You will be provided with several classes that implement a graphical simulation of a 2D world with many animals moving around in it. These classes are defined in the file CritterMain.java. **YOU DO NOT NEED TO LOOK AT OR UNDERSTAND THIS CODE.** You will write a set of classes that define the behavior of those animals, and these classes will be used by the GUI code. Different kinds of animals move and behave in different ways. As you write each class, you are defining those 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.**

IMPORTANT:
Please note that the convention is opposite to that of the convention that we use for a 2D array. It means that if `cols` is the number of columns of cells and `rows` is the number of rows of cells, x will have values from `0 to (cols-1)` and y will have values from `0 to (rows-1)`



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. 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). You might want 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/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 removed from the game. Each animal chooses one of `Attack.ROAR`, `Attack.POUNCE`, or `Attack.SCRATCH`.

Each attack is strong against one other attack (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, Scissors." If the animals make the same choice, the winner is chosen at random.

		Critter #2		
		Attack.ROAR	Attack.POUNCE	Attack.SCRATCH
Critter #1	Attack.ROAR	*random winner*	#2 wins	#1 wins
	Attack.POUNCE	#1 wins	*random winner*	#2 wins
	Attack.SCRATCH	#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 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. **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.**

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.

The provided Critter.java file: The Base Class for your different critters

Each class you write (see next section) will extend a superclass named Critter. This is an example of inheritance, as discussed in the textbook. Inheritance makes it easier for the pre-written GUI code to talk to your critter classes, and it helps us be sure that all your animal classes will implement all the methods we need.

The Critter class contains the following methods, which you need to override in each of your classes:

1. **`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).

2. **`public Attack fight(String opponent)`**
   When two animals of different species 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.

3. **`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.

4. **`public Direction getMove()`**
   Every time the board updates, our code calls this on your animal to ask it which way it wants to move.

5. **`public String toString()`**
   Every time the board updates, our code calls this on your animal to ask what letter it should be drawn as.

Just by writing "`extends Critter`" when you declare your classes in the next section, you receive a default version of these methods. The default behavior is to never eat, to always forfeit in a fight, to use the color **black**, to always stand still (i.e. a move of `Direction.CENTER`), and a `toString` of return value "?".

If you don't want this default, override (i.e. rewrite) the methods in your class with your own behavior. For example, below is a critter class `Lannister`. Lannisters are displayed with the letter "`L`", **red** in color, never move, never eat, and always roar in a fight. Your classes will look like this class, except with fields, a constructor, and more sophisticated code. Note that the `Stone` does not need an eat() or a getMove() method; it uses the default behavior for those operations.

```
import java.awt.*;
// for Color
public class Lannister extends Critter {
  public Attack fight(String opponent) {
    return Attack.ROAR;
  }

  public Color getColor() {
    return Color.RED;
```

```
  }

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

Running the Simulator:

When you press the `Go` button on the simulator, it begins a series of turns. On each turn, the simulator repeats the following steps for each animal in the game:

- 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)
- 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 this program with so many animals on such a large screen. We suggest using a smaller game world and fewer animals – perhaps just 1 or 2 of each species – by adjusting the game's initial settings when you run it. There is also a Debug checkbox that, when checked, prints a large amount of console output about the game behavior.

The code for the simulator is provided in `CritterMain.java` but again, you do not need to look at it or understand it.

NOTE: The GUI populates the board with Critters depending on the class files found in the folder. For example, if you have `Lion.java`, `Bear.java` and `Dragon.java` within your folder, compiling and running the GUI will allow you to populate the arena with only Lions, Bears and Dragons. If you don't want a particular Critter to be available, simply take the file out of the folder.

Part 2: Meet the Critters [40 Points]

This project requires skills with inheritance and polymorphism. In the world will be Critters of four types (Bear, Lion, Tiger, Dragon). You can create instance variables that you need, and any helper methods that you need to fulfill the required functionalities of each class. Here are the classes you have to inherit from the Critter class.

1. 🐻 **Bear**
   Bears are always hungry, scratch when fighting, and either move south or east.

2. 🦁 **Lion**
   Lions get hungry after they fight, they either roar or pounce, and they move in circles.

3. 🐯 **Tiger**
   Tigers have a fixed amount of food they can eat, either scratch or pounce, and move randomly.

4. 🐲 **Dragon**
   Dragons are color changers depending on what they eat. They base their attack on their attack history.

Below are the descriptions of each of each of these Critters. Note that we are not providing you with any starter code for these classes, so you will need to create each from scratch in a separate file (`Bear.java, Lion.java, Tiger.java, Dragon.java`).

Make sure <u>each</u> of these classes extends the Critter class!

Your class headers must indicate the inheritance by writing "extends Critter" keywords, like the following:

```
public class Bear extends Critter { /* code goes here */ }
```

1. 🐻 Bear [10 points]

- Constructor: `public Bear(boolean grizzly)`
- color: brown (`new Color(190,110,50)`) for a grizzly bear (when grizzly is true), and white (Color.WHITE) for a polar bear (when grizzly is false)
  - `Hint: import java.awt.*;  // package for Color class`
- eating behavior: always returns true
- fighting behavior: always scratch (Attack.SCRATCH)
- movement behavior: alternates between south and east in a zigzag pattern (first south, then east, then south, then east, ...)
- toString: "B"

The Bear constructor accepts a parameter representing the type of bear it is: true means a grizzly bear, and false means a polar bear. Your Bear object should remember this and use it later whenever getColor is called on the Bear. If the bear is a grizzly, return a brown color (a `new Color(190, 110, 50)`), and otherwise a white color (`Color.WHITE`).

2. Lion [10 points]

- constructor: public Lion()
- color: **red** (Color.RED)
- eating behavior: returns true if this Lion has been in a fight since it has last eaten (if fight has been called on this Lion at least once since the last call to eat).
- fighting behavior: if opponent is a Bear ("B"), then roar (Attack.ROAR); otherwise pounce (Attack.POUNCE).
- movement behavior: first go south 5 times, then go west 5 times, then go north 5 times, then go east 5 times (a clockwise square pattern), then repeat.
- toString: "L"

Think of the Lion as having a "hunger" that is triggered by fighting. Initially the Lion is not hungry (so eat returns false). But if the Lion gets into a fight or a series of fights (if fight is called on it one or more times), it becomes hungry. When a Lion is hungry, the next call to eat should return true. Eating once causes the Lion to become "full" again so that future calls to eat will return false, until the Lion's next fight or series of fights.

3. Tiger [10 points]

- constructor: `public Tiger(int hunger)`
- color: **yellow** (`Color.YELLOW`)
- eating behavior: returns true the first `hunger` times it is called (variable `hunger` is passed in the constructor), and false after that
- fighting behavior: if this Tiger is hungry (if eat would return true), then scratch (Attack.SCRATCH); else pounce (Attack.POUNCE). NOTE: Based on your code, calling eat() may change the hunger status. So it may not be wise to call eat() to check if the tiger is hungry.
- movement behavior: Chooses a random direction `dir(`north, south, east, or west) and moves in that direction for 3 subsequent calls to move function. Then chooses a new random direction and repeats. If you forget how to use random object, please check the "Generating random numbers" section below.
- toString: the number of pieces of food this Tiger still wants to eat, as a String (i.e. "4")

The Tiger constructor accepts a parameter for the maximum number of food this Tiger will eat in its lifetime (the number of times it will return true from a call to eat). For example, a Tiger 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 for hunger is non-negative.

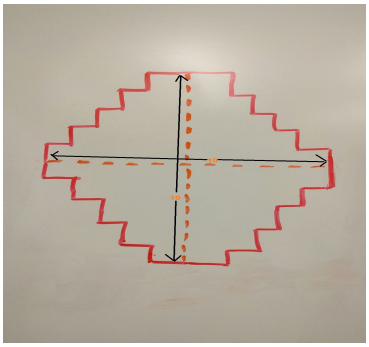
The `toString` method for a Tiger should return its remaining hunger, the number of times that a call to eat would return true for that Tiger. For example, if a new Tiger(5) is constructed, initially that Tiger's toString method should return "5". After eat has been called on that Tiger once, calls to toString should return "4", and so on, until the Tiger is no longer hungry, after which all calls to toString should return "0". Recall that you can convert a number to a string by concatenating it with an empty string. For example, "" + 7 evaluates to "7".

4. Dragon [10 points]

- The dragon should choose a random attack (from ROAR, SCRATCH and POUNCE) when initialized.
- The dragon will change color depending on the food items that it consumes. If it has eaten an even number of food items, it will be black in color. If the dragon has eaten an odd number of food items, it will be white in color.
- When the dragon encounters an animal of a different species, it will fight. The choice of attack will depend on the **PREVIOUS** animal that it encounters:
    - If previous attacker was Bear, attack **ROAR**.
    - If previous attacker was Lion, attack **POUNCE**,
    - Else attack **SCRATCH**.
    - *(no fire attack)*

  The first attack is the randomized attack initialized when a Dragon object is created.
- The initial direction of the dragon is WEST. In the subsequent moves, the dragon goes in a zig zag circle shape of diameter 10, going in counter clockwise direction.
- The eating behavior of a dragon is always true.
- The `toString` method should return "D"

Displayed below is a picture of how the Dragon's movement were to look if you traced it. The two dotted lines are to show that the overall figure is of length 10 and width 10. You **do not** move along the dotted lines!



The move should:

- start at the **middle top**
- and move in **anti**clockwise direction.

Check this video to see how the dragon moves:
https://youtu.be/4HMgxP1dMI0

Generating (pseudo-)random numbers

In this assignment, at least for the Tiger and Dragon classes, you will need to generate random numbers.

The easiest way to do this is to use the class `java.util.Random`. The Random class has a default constructor, which will create a new Random object that is ready to give you random numbers. The method you will most likely want to use `isRandom.nextInt(int n)`, which will return a random integer in the range 0 to (n-1).

For example, you are supposed to choose random directions in the Tiger class. Since there are 4 of them, you can make a call to nextInt(4), which will return either 0, 1, 2, or 3. You can associate each of these with the directions NORTH, SOUTH, EAST, and WEST in whatever way you like.

Optional reading: Random numbers are not really random, they're pseudo-random.

> *Random in Java – and any other programming language afaik – is not truly random, it's pseudo-random. The constructor of Random takes in a numeric value, which is called a **seed**. For a given seed the **exact same** sequence of numbers is **always** generated by the Random object – hence, you get a deterministic sequence.*
>
> *Why's it like that? To allow one to reproduce the results generated by code that uses a random number generator - i.e. you end up getting deterministic results. By allowing you to set a seed and know the sequence that will be produced, your code output can be reproduced elsewhere (this is critical to validate research, for instance.)*
>
> *If you wanna see this happen, check out the following code. No matter how many times the loop runs (100, 1000, a billion), it will **always** print the same thing:*

```
int n_iter = 100;
for(int i = 0; i < n_iter; i++) {
    Random random = new Random(123);
    System.out.println(random.nextInt());
}
```

You may see that happening here: https://ideone.com/9B4ih6

Testing

We have provided you with a file PA2Tester.java that runs some simple tests on the Critter Family (e.g. Bear, Dragon, Lion, Tiger). You may add more tests as you feel are appropriate. These tests for a subset of methods in part2 and additional tests you write will not be graded.

Part 3: Critter DIY [10 Points]

In the latter half 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 need to name your critter file **MyCritter.java**. Don't forget that it must extend the Critter class. Just like the 4 other Critters you implemented, you will override the 5 methods described above. You may also want to override the other (non-final) methods from the critter class so that you can update internal state based on what happens to your critter.

These other methods (**`win()`, `lose()`, `sleep()`** etc.) are all called on your critter when those events occur.

For example the **`sleep()`** method is called on your critter once the Game puts your critter to sleep and **`wakeup()`** is called once the Game wakes your critter up. Note, these methods don't control if your critter is asleep or wakeup. You can override these methods to let your critter behave in a way when they are in a certain state (e.g. change color when they are sleeping).

Your objective is to create**...**

One who can outlive and beat the other critters in the world. It's survival of the fittest!

Some hints and ideas for creating your own Critter:

- Think about having static variables to help the whole class learn from every individual critter's experience
- Randomness can often help make the Critter more unpredictable, and therefore stronger.
- There are many different strategies that can work--maybe you want your critter to try to eat as much as possible. Maybe it should be a good fighter. Maybe it should sit still and try to hide. Try out different approaches to see what works best against the critters you've already created, and then move on to part 4...

Your defined functionality need to be <u>different from</u> the behavior of any of the critters from part 2.

Part 4: SURVIVE! [20 Points + 💫 5 Points for Extra Credits💫]

Now that you have created your own Critter, it is time to put your implementation to the test! For this section, you will face off against the Critters that you implemented in Part 2!

In the starter code, we have provided you with three class files: Easy.class, Medium.class and Hard.class. Other than what is given in the starter code, there will be no further documentation provided about these classes. To find out the functionalities of these classes, and how their Critters work, run it in the simulator and observe its behavior with other classes, including your own class. Use the Tick button if you like.

To receive full credit for this section, your critter will need to be able to win in 4 environments:

1. **(10 points)** A world with Lions, Tigers, Bears and Dragons
2. **(6 points)** A world with Easy Bots, Lions, Tigers and Bears only (no Dragon from here on out)
3. **(4 points)** A world with Medium Bots and Lions, Tigers, and Bears
4. **(5 points for extra credits)** A world with Hard Bots and Lions, Tigers, and Bears
    a. This part is only for extra credits - so don't spend too much time here. It is **hard.** Make sure to get the other parts right first.

Criteria for Winning:

To determine if your critter has successfully beat an environment (The environment will always be of screen size[width = 60, height = 50] and number of critters per type of critter = **25**), **it should have the highest score after 1000 moves have occurred (i.e. your critter's name should be highlighted in yellow on the right hand side of the GUI panel after 1000 moves) and at least 1 of your critters should remain alive. Your critter will have to successfully beat each particular environment 2 out of 3 times in order to be awarded full points. In other words, we will be running each of the 4 environments described above 3 times.** (Yes - some outcomes are random, if you wish you be sure of a victory, be sure you win a significant fraction of the time.)

Style [20 Points]

Refer to the complete guidelines and a set of examples of what your style should look like.

Detailed style requirements:

1. **[2 points]** File header
2. **[1 point]** Class header
3. **[3 points]** Method header
4. **[3 points]** Use proper indenting
5. **[1 point]** Use descriptive variable names
6. **[2 points]** Avoid using magic numbers
   - **When to use variables for magic number?**
   - (1) When you use unique values with unexplained meaning (eg. LARGE_NUMBER for 999999, etc.)
   - (2) When you use a value for multiple times, or a value that may change in the future (eg. GRID_SIZE in a board game, like 2048 or Monopoly)
7. **[3 points]** Write short methods
8. **[3 points]** Write short lines
9. **[2 points]** Write appropriate inline comments

Submitting the Assignment

How to Submit on Vocareum

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

- Bear.java
- Lion.java
- Tiger.java
- Dragon.java
- MyCritter.java
- **CritterDrawing.png -> You should have made this in Part 1**
- **README.txt -> You should have made this in Part 1**

Maximum Score Possible: 100/100 Points plus 5 points for extra credits.