Take-home Assessment 8: Critters

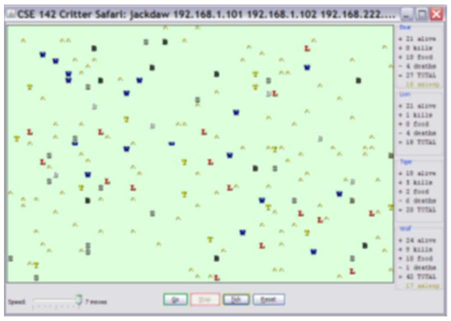
due June 2, 2021, 11:59pm

This assignment will assess your mastery of the following objectives:

- Write a class to define objects with specified behavior.
- Use fields to store state necessary for a class's operation.
- Practice good object-oriented design, including abstraction and encapsulation.
- Utilize inheritance and choose an appropriate base class for each new class.
- Follow prescribed conventions for spacing, indentation, naming methods, and header comments.

Program Behavior

Overview



In this assignment, you will *not* be writing a program. Instead, you will be implementing several classes that will be used in a client program provided to you. This client represents a graphical simulation of a 2D world of animals, known as critters. While it is running, the simulation will look like the image to the right. The client will only display the graphical output shown here and described below; it *will not* produce any console output. You will write classes that define the behavior of several types of critters, each of which moves and behaves in different ways.

Your code should **not** produce *any* console output.

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. The world has a finite size, but it wraps around in all four directions (e.g., moving east from the right edge brings you back to the left edge).

This assessment will be confusing at first, because you will not write a `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 to make several moves at once using a loop, but you can't. Critters can only make one move at a time, and only when the simulator asks. This experience may be frustrating because it is a new way of programming, but it is important. In most real-world projects, you will not be in control of the entire environment and will have to work with existing code that you cannot change.

Critter Movement

The simulation takes place in a series of rounds, with each critter making a single move in each round. Critters specify the direction they wish to move by returning one of the following values from the `getMove` method (see below):

`Direction` **Constant**	**Description**
`Direction.NORTH`	move one cell up
`Direction.SOUTH`	move one cell down
`Direction.EAST`	move one cell right
`Direction.WEST`	move one cell left
`Direction.CENTER`	do not move; stay in the same location

Fighting/Mating

As the simulation runs, critters may collide by moving onto the same location. When two critters collide, if they are from different species, they **fight**. Each critter 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 one (e.g. roar loses to pounce) according to the following rules:

> `Attack.ROAR` beats `Attack.SCRATCH` and loses to `Attack.POUNCE`
> `Attack.POUNCE` beats `Attack.ROAR` and loses to `Attack.SCRATCH`
> `Attack.SCRATCH` beats `Attack.POUNCE` and loses to `Attack.ROAR`

(To help you remember which attack beats which, notice that the starting letters of "Roar, Pounce, and Scratch" match those of "Rock, Paper, and Scissors.") The winning critter survives and the losing critter is killed and removed from the game. If the critters make the same choice, the winner is chosen randomly.

If two critters of the *same* species collide, they **mate**. A critter can mate only once during its lifetime. Mating critters will not move for several rounds, after which a new "baby" critter of the same type will be produced and begin moving around the world. Critters are vulnerable to attack while mating: they will always lose a fight if a critter of a different type collides with them.

Eating/Sleeping

The simulation world also contains **food** (represented by the period character, `"."`) for the critters to eat. Food is initially placed in the world randomly, and new food is gradually added over time. As a critter moves, it may encounter food, in which case the simulator will ask the critter whether it wants to eat. Different kinds of critters will have different eating behavior.

Every time a critter eats a few pieces of food, that critter will be put to **sleep** by the simulator for a small amount of time. While asleep, critters cannot move and will always lose fights, though they can mate if another critter of the same type moves into their location.

Scoring

The simulator keeps a score for each type of critter. This score is the sum of how many critters of that type are alive, how much total food they have eaten, and how many other critters they have killed.

Implementing Critters

You will be provided with a **base class** called `Critter` which defines the default behavior of all critters. Each critter you implement will be a **subclass** of `Critter` and override some of the default behaviors. This is an example of inheritance, as discussed in class and in Chapter 9 of the textbook, and will use the `extends` keyword. (Note that not all critters will extend `Critter` directly. If two types of critters are similar, it may be more appropriate for one to extend the other.)

When you override a method, be sure to use the *exact same* name and parameters. In particular, you cannot add parameters.

Each critter class you write will override some or all of the following methods:

`public Direction getMove()`
Called each round to choose direction to move. (Default behavior: Always stand still.)

`public boolean eat()`
Called when food is encountered. Return `true` to eat or `false` to not eat. (Default behavior: Never eat.)

`public Attack fight(String opponent)`
Called when colliding with a different critter. The appearance of the other critter is given as the parameter. (Default behavior: Always forfeit.)

`public Color getColor()`
Called each round to determine color to display critter as. (Default behavior: Always display as black.)

`public String toString()`
Called each round to determine text to display critter as. (Default behavior: Always display as ?.)

An example critter called `Stone` is shown below. A `Stone` is always displayed as a gray capital "S", never moves or eats and always roars in a fight. Notice how the `Stone` class does not implement the methods for which it is using the default behavior. Your critter classes will look similar to this, but will be more sophisticated, and may include fields and constructors.

You should not override methods that will use the default critter behavior.

`Sample Critter: Stone`

```
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";
    }
}
```

Required Critters

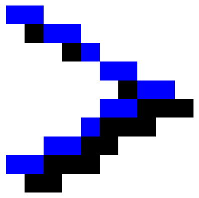
You will implement the following **five** critters for this assessment. Each critter will have one constructor, which must accept *exactly* the parameters shown below. (Any changes to the constructor will cause the client to not compile.) For any behavior described as random, all possibilities should be equally likely, and random values should be generated using a `Random` object.

Ant

constructor	`public Ant(boolean walkSouth)`
movement	If constructed with `walkSouth` as `true`, alternate between south and east; otherwise, alternate between north and east.
eating	Always eat.
fighting	Always scratch.
color	Always red.
string	Always "%" (percent sign).

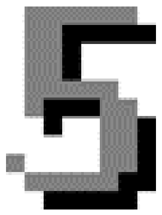
Bird

constructor	`public Bird()`
movement	North three (3) times, then east three (3) times, then south three (3) times, then west three (3) times, then repeat. (A clockwise square.)
eating	Never eat.
fighting	Roar if opponent looks like an ant ("%"); otherwise, pounce.
color	Always blue.
string	`"^"` (caret) if last move was north, or never has never moved; `">"` (greater than) if last move was east; `"V"` (uppercase V) if last move was south; `"<"` (less than) if last move was west.



Hippo

constructor	`public Hippo(int hunger)`
movement	Five (5) steps in a randomly chosen direction, then five (5) steps in another randomly chosen direction, then repeat. The same direction can be chosen multiple times in a row.
eating	Eat if still hungry (see below).
fighting	Scratch if still hungry; otherwise, pounce.
color	Grey if still hungry; otherwise, white.
string	Amount of remaining hunger.

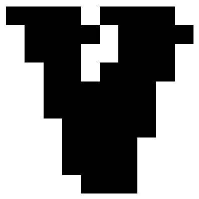


Hippos have a concept of hunger, which is the amount of food the hippo will eat in its lifetime. Each hippo is constructed with an initial hunger. For example, a hippo constructed by calling `new Hippo(8)` will eat the first eight (8) times it encounters food, then will never eat again. Each time a hippo eats, its hunger is reduced by one until it reaches zero, meaning hippo is no longer hungry. You may assume that hunger is always a non-negative integer.

Each hippo should display as its *current* hunger, not as its original hunger. (This means the hippo will change how it is displayed as it eats.) You can convert a number to a string by concatenating it with the empty string. For example, if `n` is an integer variable, then `"" + n` will be the value of `n` as a string.

Vulture

constructor	`public Vulture()`
movement	North three (3) times, then east three (3) times, then south three (3) times, then west three (3) times, then repeat. (A clockwise square.)
eating	Eat the first food encountered after being created or after each fight (see below).
fighting	Roar if opponent looks like an ant ("%"); otherwise, pounce
color	Black
string	`"^"` (caret) if last move was north, or never has never moved; `">"` (greater than) if last move was east; `"V"` (uppercase V) if last move was south; `"<"` (less than) if last move was west.



Vultures are a special type of bird, and most behavior is the same as other birds. Vultures are born "hungry" and will eat the first food they encounter. After eating, a vulture will become "full" until it fights, at which point it will become hungry again. When a vulture is hungry, it only needs to eat once to become "full" and will not eat again until it becomes hungry again (by fighting). Note that you do not need to confirm if the vulture wins or loses when it fights— if the vulture loses a fight, it will be dead.

Husky

constructor	`public Husky()`
movement	Your choice.
eating	Your choice.
fighting	Your choice.
color	Your choice.
string	Your choice.

Your Husky may have any behavior you like, as long as it does not exactly duplicate any example critter you have been shown in class (lecture, section, or lab) or any of the four required critters (Ant, Bird, Hippo, Vulture). In addition you may use advanced material if you choose, as long as it does not break or attempt to "hack" the simulation. Your Husky will only contribute to the Behavior dimension grade. It will not factor in to grading on the other dimensions.

On the last day of class, we will hold a competition pitting students' Husky critters against each other. More details about the tournament and how to participate will be released later. The tournament will be for fun, and participation will be entirely optional—your grade will not be affected by your participation or performance in the tournament.

If you'd like, you can also use or override some of the following additional `Critter` method in your Husky to get more interesting behavior. Note that **none of these methods are required or should be used in `Ant`, `Bird`, `Hippo`, or `Vulture`.**

`public int getX()`, `public int getY()`
Returns the critter's current x or y coordinate.

`public int getWidth()`, `public int getHeight()`
Returns the overall width or height of the critter world.

`public String getNeighbor(Direction direction)`
Returns the string representation of the critter next to yours in the specified direction. If there is no critter in that direction, " " is returned.

`public void win()`, `public void lose()`, `public void sleep()`, `public void wakeup()`, `public void mate()`, `public void mateEnd()`
These methods are called by the simulator when your critter wins a fight, loses a fight, goes to sleep, wakes up, starts mating, or finishes mating (respectively). You can override these methods to allow your critter to react to these events occuring.

`public void reset()`
This method is called by the simulator when the simulation is reset.

Development Strategy and Hints

We recommend attempting the classes in the order they are presented above (`Ant`, then `Bird`, then `Hippo`, then `Vulture`). Within each class, you can test each part of the behavior without having implemented them all. (Any methods that are not implemented will use the default critter behavior.) You will likely be able to debug your work more easily if you test each method individually.

As you work on your code, be sure to think carefully about the **state** (i.e. the fields) necessary for each class. It will be difficult or impossible to achieve the correct behavior if you are not storing the correct state in your objects. In particular, you will likely need to "remember" values passed to constructors in fields to ensure they are available for later use.

Remember that you will *not* be able to control when or how often the methods of your class are called—that is controlled by the simulator. You will need to ensure that your critters behave properly no matter when, how often, or in what order the simulator class your methods. For example, you should not assume that `getColor()` is always called before `toString()`, or that `eat()` will only be called once between calls to `getMove()`. You will also only be able to return one value (move, attack, etc.) each time the method is called. You will need to use fields (see above) to keep track of each object's current state and remember what your next choice should be. You should be sure that any updates to your state take place in the appropriate method. For example, if a change in state is based on the critter eating, the update should occur in the `eat()` method.

Code Quality Guidelines

In addition to producing the desired behavior, your code should be well-written and meet all expectations described in the grading guidelines and the Code Quality Guide. For this assessment, pay particular attention to the following elements:

Object-oriented Design

Your code must follow good object-oriented design principles as described and demonstrated in class, the textbook, and the Code Quality Guide. In particular, 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.** You should also **not create any unnecessary objects.** When possible, create an object once and use it as a field or parameter rather than repeatedly recreating the object.

Your class should also utilize inheritence effectively. Your critter classes should properly extend a superclass, and should use inheritance to remove redundancy between classes that are similar. In other words, if two of your critter classes are very much alike, you should have one extend the other rather than having both simply extend `Critter`.

Code Aesthetics

Your code should be properly indented, make good use of blank lines and other whitespace, and include no lines longer than 100 characters. Your class, methods, variables, and constant should all have meaningful and descriptive names and follow the standard Java naming conventions. (e.g. `ClassName`, `methodOr-VariableName`, `CONSTANT_NAME`) See the Code Quality Guide for more information.

Commenting

Your code should include a header comment on each class, following the same format described in previous assessments. Your code should also include a comment at the beginning of each method that describes that method's behavior. Method comments should also explicitly name and describe all parameters to that method and describe the method's return value (if it has one). Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implementation details (such as describing loops or expressions included in the code). See the Code Quality Guide for examples and more information.

Since you are not writing a *program*, your comments will be slightly different than on previous assessments, but should follow the same guidelines. Your class comments will describe the general behavior of a type of object rather than the function of a specific program. Your method comments should describe the behavior of the method in the context of the larger class. The commenting section of the Code Quality Guide has some examples specific to object-oriented programming.

Running and Submitting

You can run the critter simulation by clicking the "Check" button in Ed. The simulator will recognize any classes you have implemented that extend `Critter` (or extend another class that extends `Critter`). You can choose which of your critters to include when you run the program using the checkboxes in the menu that appears when you click Check. (You should not need to modify any other options in that menu.) If you do not see one of your critters, you most likely either have compiler errors in that critter class or have not properly extended `Critter`.

The critter simulation uses Ed's resources rather quickly, so it will only be able to run for a limited time. You should be able to run the simulation for plenty of time to test your critters' behavior, but you should make sure to **close the simulation window** as soon as you are done testing. Leaving the window open could cause problems in Ed.

If you believe your behavior is correct, you can submit your work by clicking the "Mark" button in the Ed lesson. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA.**

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. Note the due date and time carefully—**work submitted after the due time will not be accepted**.

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from lessons, section, and lab
- Reading the textbook
- Visiting Support Hours
- Posting a question on the message board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.

Reflection

In addition to your code, you must submit answers to short reflection questions. These questions will help you think about what you learned, what you struggled with, and how you can improve next time. The questions are given in the file Reflection slide in the Ed lesson; type your responses directly into those textboxes.