CSE 190 D, Winter 2013
Programming Assignment #10: Critters (20 points)
Due: Monday, March 18, 2013, 11:30 PM

This assignment focuses on classes and objects. Turn in `Ant.java` and `Bird.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 moves 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 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, 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. 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**.
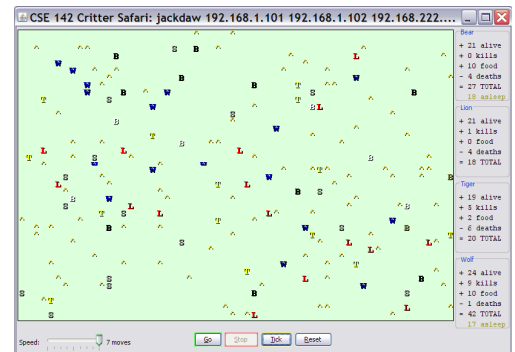
Provided Files:

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 animal classes will implement all the methods we need. Your class headers should indicate the inheritance by writing `extends Critter`, like the following:

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

The `Critter` class contains the following methods, some/all of which you must 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.

Just by writing 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 a `toString` of `"?"`. If you don't want this default, rewrite (override) the 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.

```java
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";
    }
}
```

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 two animal classes to implement. Each has one constructor that accepts exactly the parameter(s) in the table. For random moves, each choice must be equally likely; use a `Random` object or the `Math.random` method.

1. Ant

constructor	`public Ant(boolean walkSouth)`
color	red
eating behavior	always returns `true`
fighting behavior	always scratch

movement	if the Ant was constructed with a `walkSouth` value of `true`, then
	   alternates between south and east in a zigzag (S, E, S, E, ...); otherwise,
	if the Ant was constructed with a `walkSouth` value of `false`, then
	   alternates between north and east in a zigzag (N, E, N, E, ...)

toString	`"%"` (percent)

2. Bird

constructor	`public Bird()`	
color	blue	
eating behavior	never eats (always returns `false`)	
fighting behavior	roars if the opponent looks like an Ant (`"%"`); otherwise pounces	
movement	a clockwise square: first goes north 3 times, then east 3 times, then south 3 times, then west 3 times, then repeats	
toString	`"^"` (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

Development Strategy:

The simulator runs even if you haven't completed all the critters. The classes increase in difficulty from `Ant` to `Bird`. We suggest doing `Ant` first. Look at `Stone.java` 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 the `Critter` superclass as described.

Follow past style guidelines about indentation, spacing, identifiers, and localizing variables. Place comments at the beginning of each class documenting that critter's behavior, and on any complex code. Your critters should not produce any console output.