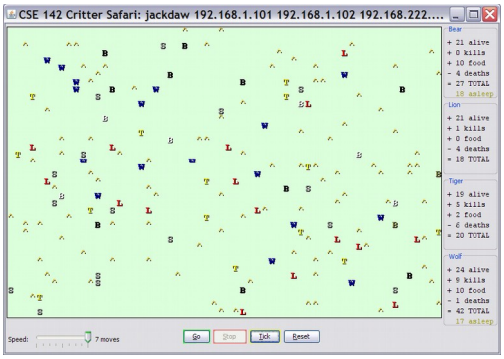
Stanford University, CS 106A, Homework Assignment 6
Critters (PAIR ASSIGNMENT)

This assignment was co-created by Stuart Reges and Marty Stepp at University of Washington.

This assignment focuses on classes, objects, and inheritance. You will **turn in** the following files:

- **Ant.java**, **Bird.java**, **Crab.java**, **FireAnt.java**, **Hippo.java**, **Vulture.java**, and **Wolf.java**

Get starter files on the course web site. Run **CritterMain** to start the simulation, or **MiniMain** for a quick test.



Program Description:

You are provided with several client classes that implement a simulation of a 2D world of animals. You write classes to 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, 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).

You do not write the overall `Program` class or `run` 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: 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 "**R**oar, **P**ounce, **S**cratch" match those of "**R**ock, **P**aper, **S**cissors." If the animals make the same choice, the winner is chosen at random. (Technically there is a fourth type of attack called `Attack.FORFEIT` that always loses the fight, unless both animals forfeit, in which case a random winner is chosen.)

		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

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 defeats them. An animal can mate once during its life.

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 an animal eats a few pieces of food, it 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** in fights.

Provided Files:

Each class you write will **extend a superclass** named `Critter`. This is an example of **inheritance**, as discussed in Ch. 6 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 finds 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()`
  Each time the board updates, our code calls this on your animal to ask 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 (always return `false`), to always forfeit fights (always return `Attack.FORFEIT`), to use the color black (returns `Color.BLACK`), to always stand still (returns `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 provided critter class named `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, except with fields, a constructor, and more complex code. The `Stone` does not need an `eat` or `getMove`; it uses the default behavior for them.

```
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 are not required to write `extends Critter` on every single animal class you write. If you find that two animal 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`. It can be difficult to test and debug with many animals. We suggest adjusting the settings to use a smaller world and fewer animals. There is also a **Debug checkbox** that prints console output about the game behavior.

Critter Classes:

The following are the **seven 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 `RandomGenerator` object.

1. Ant

constructor	`public Ant(boolean walkSouth)`
color	black
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

3. Crab

constructor	`public Crab(Color color)`
color	returns whatever color was passed to the constructor
eating behavior	randomly decides to eat or not eat each time with 50/50% probability
fighting behavior	always scratch
movement	moves back-and-forth in the following pattern: • west 1 time, east 2 times • west 3 times, east 4 times • west 5 times, east 6 times • west 7 times, east 8 times • west 7 times, east 6 times • west 5 times, east 4 times • west 3 times, east 2 times • *(entire pattern repeats)*
toString	`"w"` (lowercase W)

(continued on next page)

4. FireAnt

constructor	`public FireAnt()`
color	red
eating behavior	always returns `false`
fighting behavior	always scratch
movement	alternates between south and east in a zigzag (S, E, S, E, ...)
toString	`"%"` (percent)

5. Hippo

constructor	`public Hippo(int hunger)`
color	gray if the hippo is still hungry (if `eat` would return `true`); otherwise white
eating behavior	returns `true` the first `hunger` times it is called, and `false` after that
fighting behavior	if this `Hippo` is hungry (if `eat` would return `true`), then scratches; else pounces
movement	moves 5 steps in a random direction (north, south, east, or west), then chooses a new random direction and repeats
toString	the number of pieces of food this `Hippo` still wants to eat, as a `String`

The **`Hippo` constructor** accepts a parameter for the max number of food that `Hippo` will eat in its lifetime (the number of times it will return `true` from `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 `eat` would return `true` for that `Hippo`. For example, if a `new Hippo(4)` is made, initially its `toString` is `"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"`.

6. Vulture

constructor	`public Vulture()`
color	black
eating behavior	Returns `true` if vulture is hungry. A vulture is initially hungry, and he remains hungry until he eats *once*. After eating once he will become non-hungry until he gets into a fight. After one or more fights, he will be hungry again. *(see below)*
fighting behavior	roars if the opponent looks like an Ant (`"%"`); otherwise pounces
movement	a clockwise square: first goes north 5 times, then east 5 times, then south 5 times, then west 5 times, then repeats
toString	`"^"` (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. Think of the `Vulture` as having a "hunger" enabled when he 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, he 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.

(continued on next page)

6. Wolf

constructor	`public Wolf()` (must not accept any parameters)
all behavior	you decide (see below)

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

Your `Wolf`'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 `Wolf` 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 `Wolf` 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 `Wolf` can use any **advanced material** you happen to know in Java. If your Wolf uses additional classes you have written, contact your section leader, TA, or instructor to make sure it will be compatible with our system.

Each critter class has **additional methods** that it receives by inheritance from `Critter`. Your `Wolf` may want to use these methods. None of the methods below are needed for `Ant`, `Bird`, `Crab`, `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 of what is next to your critter in the given direction. `" "` (space) is 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, been put to sleep/ wake up, start/end 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 roughly increase in difficulty from `Ant` to `Bird` to `Crab` to `Hippo` to `Vulture`. We suggest doing `Ant` first. Look at `Stone.java` and the lecture/section examples of critter classes to see the general structure.

It will be impossible to implement each behavior if you don't have the right state (fields) 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" information and 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 (unwritten ones will use default behavior). Add a method, run the simulator to see that it works, then add another.

Critter Tournament:

In our last lecture, we will host a **Critter tournament**. In each battle, two students' `Wolf` classes will be placed into the simulator along with the other standard animals. The student with the higher score in the right sidebar advances.

A **"battle"** is defined as the following: We run `CritterMain` with a standard 60x50 world, using 25 of each kind of animal. The animals present are `Ant`s, `Bird`s, `FireAnt`s, `Hippo`s, `Vulture`s, `Stone`s, and two students' Wolves. We start the simulator and let it run until either one student's Wolves are completely eliminated or until roughly 1000 moves have passed. At this point whichever student's `Wolf` species has the more points wins the battle.

Think of the tournament like one of the major pro sports. We will run a "**regular season**" in which every student's `Wolf` species will play many battles against randomly chosen opponent `Wolf` classes. We run a season of many games (at least 64 for each student), then we grab the top 16 students that have the best win/loss records. They advance to the "playoffs", which will take place live in lecture on the last day of class.

The playoffs consist of a 16-Wolf bracket like part of an NCAA basketball tournament bracket. Wolf #1 battles #16, #2 battles #15, and so on. Winners advance and losers are eliminated. Eventually one Wolf is the champion.

No grade points will be based on tournament performance. For example, a `Wolf` that sits completely still might fare well in the tournament, but it will not receive full grade points because it is too trivial.

Grading and 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 is also based on 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. 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, have B extend A rather than having both simply extend `Critter`.

Some of your style grade will be awarded on the basis of how much **energy and creativity** you put into defining an interesting `Wolf` class. These points allow us to reward the students who spend time writing an interesting critter definition. Your `Wolf`'s behavior should not be trivial or closely match that of an existing animal shown in class.

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 (`println`).

The `Wolf` is **not graded on style** at all. Its code does not need to be commented, can be redundant, and can use any advanced material you like, so long as it works properly and obeys the other constraints described previously.

As always, review the "Grading" section of prior specs for reminders of our general grading expectations.

Follow the style guidelines taught in class and listed in the course **Style Guide**. Show us a good procedural decomposition of code into methods to indicate structure and avoid redundancy. Minimize the use of **data fields** and prefer local variables as much as possible. If there are important fixed values used in your code, declare them as final **constants**. Use descriptive **names**. **Format** your code cleanly. Avoid **redundancy**. Use descriptive **comments**, including the top of each .java file, atop each method, inline on complex sections of code, next to each field, and a **citation of all sources**. If you complete any **extra features**, list them in your comments to make sure the grader knows what you completed.

Honor Code: Follow the **Honor Code** when working on this assignment. Submit your own (pair's) work; do not look at others' solutions. Do not give out your solution. Do not place a solution to this assignment on a public web site or forum. Solutions from this quarter, past quarters, and any solutions found online, will be electronically compared.