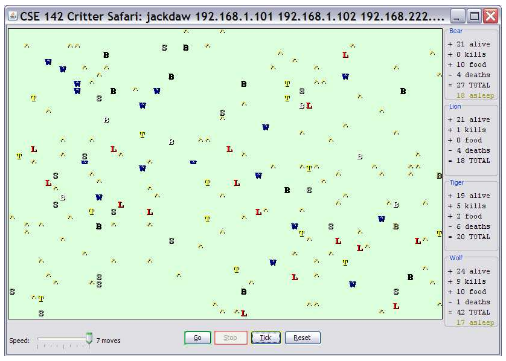
CSE 142, Autumn 2007
Programming Assignment #8: Critters (20 points)
Due: Thursday, December 6, 2007, 8:00 PM

adapted from Critters assignment by Stuart Reges, with ideas from Steve Gribble

This assignment will give you practice with classes. Turn in `Bear.java`, `Lion.java`, `Tiger.java`, and `Husky.java`. There are several supporting files to download on the course web site. Run `CritterMain.java` to start the simulation.

Program Behavior:

You will be provided with several classes that implement a graphical simulation of a 2D world with many animals moving around in it. You will write a set of classes that define the behavior of those animals. Different kinds of animals move and behave in different ways. As you write each class, you are defining those unique behaviors for each animal.


CSE 142 Critter Safari: jackdaw 192.168.1.101 192.168.1.102 192.168.222....
Speed:
7 moves
Go
Stop
Tick
Reset

```
Bear
+ 21 alive
+ 0 kills
+ 10 food
- 4 deaths
= 27 TOTAL
  18 asleep
Lion
+ 21 alive
+ 1 kills
+ 0 food
- 4 deaths
= 18 TOTAL
Tiger
+ 19 alive
+ 5 kills
+ 2 food
- 6 deaths
= 20 TOTAL
Wolf
+ 24 alive
+ 9 kills
+ 10 food
- 1 deaths
= 42 TOTAL
  17 asleep
```

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. 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 probably be confusing at first, because this is the first time where you do not write the `main` method (the client code that uses your animals), so your code is not in control of the overall program's execution. Instead, you are defining objects that become part of a larger system. You might want to have one of your critters make several moves all at once using a loop. But you can't do that. 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 object-oriented programming.

Fighting

As the simulation runs, animals can collide by moving onto the same location. When two animals collide, they fight. The winning animal survives and the losing animal is killed. Each animal chooses to ROAR, POUNCE or SCRATCH its opponent. Each of these attacks is strong against one other attack (e.g. ROAR beats SCRATCH) and weak against another (ROAR loses to POUNCE). The following table summarizes the possible choices and which animal will win in each case. To help remember which beats which, notice that the starting letters and ratings 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		
		ROAR	**POUNCE**	**SCRATCH**
Critter #1	**ROAR**	random winner	#2 wins	#1 wins
	POUNCE	#1 wins	random winner	#2 wins
	SCRATCH	#2 wins	#1 wins	random winner

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 10 pieces of food, that class 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 of the four classes you'll write will implement the following provided `Critter` interface:

```
public interface Critter {
    // methods to be implemented
    public boolean eat();
    public int fight(String opponent);
    public Color getColor();
    public int getMove(CritterInfo info);
    public String toString();

    // constants for directions
    public static final int NORTH = -2;
    public static final int SOUTH = 4;
    public static final int EAST = 3;
    public static final int WEST = 19;
    public static final int CENTER = 11;

    // constants for fighting
    public static final int ROAR = 28;
    public static final int POUNCE = -10;
    public static final int SCRATCH = 55;
}
```

Interfaces are discussed in detail in Chapter 9 of the textbook, but to do this assignment you just need to know a few simple rules about interfaces. Your class headers should indicate that they implement this interface, as in:

```
public class Bear implements Critter {
    ...
}
```

The interface is our GUI's way of being sure that your various animal classes will implement all of the methods we need. If you use a header like the above, in order for your code to compile, you must include in each class a definition for each of the methods in the interface (`eat`, `fight`, `getColor`, `getMove`, and `toString`).

For example, below is a critter class called `Stone`. `Stone` objects are displayed with the letter S, are gray in color, always stay on the current location (returning `CENTER` for their move), never eat, and always choose to ROAR in a fight. Your classes will look like the class below, except with fields, a constructor, and more sophisticated behavior code.

```
import java.awt.*;   // for Color

public class Stone implements Critter {
    public boolean eat() {
        return false;
    }

    public int fight(String opponent) {
        return ROAR;
    }

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

    public int getMove(CritterInfo info) {
        return CENTER;
    }

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

The `Critter` interface defines five constants for the directions and three constants for the types of fighting. You can refer to these directly in your code (`NORTH`, `ROAR`, etc). Your critter can stay on its current location by returning `CENTER`.

Your code should not depend upon the specific values assigned to these constants, although you may assume they will always be of type `int`. You will lose style points if you fail to use the named constants when appropriate.

Critters to Implement:

The following are the four critter classes you will implement. Each class must only have one constructor, and that constructor must accept exactly the parameter(s) described in the table. For random moves, each possible choice must be equally likely. You may use either a `Random` object or the `Math.random` method to obtain pseudorandom values.

Bear

constructor	`public Bear(`**`boolean grizzly`**`)`
color	brown (`new Color(190, 110, 50)`) for a grizzly bear (when `grizzly` is `true`), white (`Color.WHITE`) for a polar bear (when grizzly is `false`)
eating behavior	always returns `true`
fighting behavior	always `SCRATCH`
movement behavior	alternates between `NORTH` and `WEST` in a zigzag pattern (first `NORTH`, then `WEST`, then `NORTH`, then `WEST`, ...)
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`).

Lion

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`; otherwise `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 repeats
toString	`"L"`

Tiger

constructor	`public Tiger(int `**`hunger`**`)`
color	yellow (`Color.YELLOW`)
eating behavior	returns `true` the first *`hunger`* times it is called, and `false` after that
fighting behavior	if this `Tiger` is still hungry (if a call to `eat` would return `true`), then `SCRATCH`; otherwise `POUNCE`
movement behavior	moves 3 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 `Tiger` still wants to eat, as a `String`

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 `Lion` 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 the `Tiger`'s remaining hunger; in other words, the number of times that a call to `eat` that 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"`.

Husky

constructor	`public Husky()`
all other behavior	*you decide*

You will decide the behavior of your `Husky` class. **Your constructor must accept no parameters, as shown above.**

Husky Class:

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

Each time a critter is asked to move (each time `getMove` is called), the critter is passed a parameter of type `CritterInfo` that provides useful information; your `Husky` may wish to make use of this information to guide its movement behavior:

```
public interface CritterInfo {
    public int getX();
    public int getY();
    public int getWidth();
    public int getHeight();
    public String getNeighbor(int direction);
}
```

For example, you can find out the critter's current x and y coordinates by calling `getX` and `getY` on `info`. The `getWidth` and `getHeight` methods return information about the size of the simulation. You can find out what is around the critter by calling `getNeighbor` and passing a direction constant as a parameter. You will be told the display character for whatever is next to your animal in that location. A blank space, `" "` means an empty cell. For example, to check whether your critter's x-coordinate is greater than 10, you would write code such as:

```
    if (info.getX() > 10) {                       // check if my x-coordinate is above 10
```

To check if your neighbor to the west is a `Bear`, you could write this code in your `Husky`'s `getMove` method:

```
    if (info.getNeighbor(WEST).equals("B")) {    // check if there's a Bear 1 square west of me
```

Your `Husky`'s fighting behavior may want to utilize the parameter to the `fight` method, `opponent`, which tells you what kind of critter you are fighting against (such as `"B"` if you are fighting against a `Bear`).

Your `Husky` 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 `Husky` that displays differently over time. The `toString` text is also passed to other animals when they fight your `Husky`; you may want to try to fool other animals.

On the last day of class, we will host a Critter tournament. In each battle, two students' `Husky` classes will be placed into the simulator along with the other standard animals, with 25 of each type. The simulator will run until no significant activity occurs or 1000 moves have passed. The student whose `Husky` has the higher score in the right sidebar wins.

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

Implementation Guidelines:

The provided GUI runs even if you haven't completed all the critter classes. The classes increase in difficulty from `Bear` to `Lion` to `Tiger`. We recommend that you write `Bear` first. Look at `Stone.java` as an example of the general structure.

Any critter class you write will not compile without having implementations of all methods from the `Critter` interface. However, if you want to write some of the methods and leave others for later, write a "stub" version of the others that returns a meaningless value (for example, always return `CENTER` if you don't want to write the `Bear`'s movement code yet).

In the case of each animal, it will be impossible to implement the 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.

Stylistic Guidelines:

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

Style points will also be awarded on your ability to express each critter's operations elegantly. **Your objects should be encapsulated.** Follow past stylistic guidelines about indentation, whitespace, identifiers, and localizing variables. Place comments at the beginning of each class. Each class should be in its own file. Document each critter's behavior in comments at the top of its file and/or at the top of each method. Your critters should not produce any console output.