CSE 142, Autumn 2006
Programming Assignment #8: Critter Safari (20 points)
Due: Wednesday, December 6, 2006, 9:00 PM

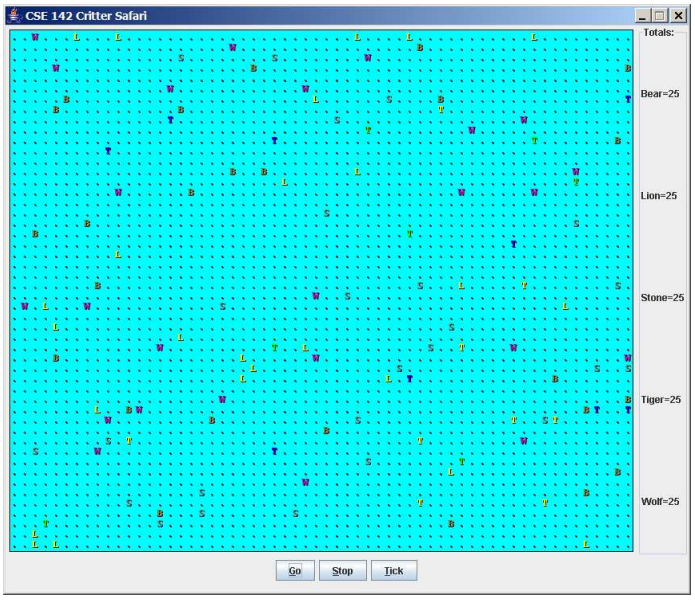
shamelessly stolen from "Critters" assignment by UW instructor Stuart Reges, with ideas from UW Prof. Steve Gribble

Note: All assignment submissions must be received by Sunday, December 10, 2006, 11:59 PM to receive credit.

This assignment will give you practice with creating classes.
Turn in files named `Bear.java`, `Lion.java`, `Tiger.java`, and `Wolf.java`.

Program Behavior:

You will write a set of classes that define the behavior of various animals. You will be provided with several classes that implement a graphical simulation of a 2D world with many animals moving around in it. Different kinds of animals move in different ways; as you write each class, you are defining those differences.



On each round of the simulation, each critter is asked which direction it wants to move. On each round, each critter can move one square north, south, east, west, or stay at its current location. Critters move around in a world of finite size, but the word is toroidal (going off the end to the right brings you back to the left and vice versa; going off the end to the top brings you back to the bottom and vice versa). The critter world is divided into cells that have integer coordinates. There are 60 cells across and 50 cells up and down. The upper-left cell has coordinates (0, 0), increasing x values move you right and increasing y values move you down.

This program will probably be confusing at first because this is the first time where you are not writing the `main` method (the client code that uses your animal objects), therefore your code will not be in control of the overall program's execution. Instead, you are defining a series of objects that become part of a larger system. For example, you might find that you want to have one of your critters make several moves all at once. You won't be able to do that. The only way a critter can move is to wait for the simulator to ask it for a move. Although this experience can be frustrating, it is a good introduction to the kind of programming we do with objects.

As the simulation runs, animals can collide by moving onto the same location. When two animals collide, they fight to the death in a deadly game of rock-paper-scissors. The winning animal survives and the losing animal is removed from the simulation. The following table summarizes the possible choices each animal can make and which animal will win in each case. If the animals make the same choice, the winner is chosen at random.

		Critter #2		
		ROCK	**PAPER**	**SCISSORS**
Critter #1	**ROCK**	random winner	#2 wins	#1 wins
	PAPER	#1 wins	random winner	#2 wins
	SCISSORS	#2 wins	#1 wins	random winner

There are several supporting files you should download on the course web site. Run `CritterMain` to start the simulation.

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 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 rock-paper-scissors game
    public static final int ROCK = 28;
    public static final int PAPER = -10;
    public static final int SCISSORS = 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 {
    ...
}
```

Because your classes implement the interface, you must include in each class a definition for each of the methods in the interface (`fight`, `getColor`, `getMove`, and `toString`). For example, below is a definition for a class called `Stone` that is part of the simulation. `Stone` objects are displayed with the letter S, are gray in color, always stay on the current location (returning `CENTER` for their move), and always choose the rock in the rock-paper-scissors game.

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

public class Stone implements Critter {
    public int fight(String opponent) {
        return ROCK;
    }

    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 various directions, and three additional constants for the three weapons in the rock-paper-scissors game. You can refer to these directly in your code (`NORTH`, `SOUTH`, `ROCK`, etc) because you are implementing the interface. 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. Your critter can stay on its current location by returning `CENTER`.

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()`
fighting behavior	always scissors
color	brown (R=128, G=128, B=64)
movement behavior	alternates between north and west in a zigzag pattern (first north, then west, then north, then west, ...)
toString	B

Lion

constructor	`public Lion(int steps)`
fighting behavior	always paper
color	yellow
movement behavior	moves the given number of steps in a random direction, then chooses a new random direction and repeats
toString	L

The `Lion` constructor accepts a parameter representing the distance the `Lion` will walk in a straight line before choosing a new random direction. For example, a `Lion` constructed with a parameter value of 8 will use its first 8 moves to walk in a single random direction, then after the 8th move, the `Lion` will choose a new random direction (which might be the same as the original one), and repeat.

Tiger

constructor	`public Tiger(Color color)`
fighting behavior	alternates between scissors and paper (first scissors, then paper, then scissors, then paper, ...)
color	the color passed to the constructor
movement behavior	first 5 steps south, then 5 steps west, then 5 steps north, then 5 steps east (a clockwise square pattern), then repeats
toString	T

The `Tiger` constructor accepts a parameter representing the color in which the `Tiger` should be drawn. This color should be returned each time the `getColor` method is called on the `Tiger`. For example, a `Tiger` constructed with a parameter value of `Color.RED` will return `Color.RED` from its `getColor` method and will therefore appear red on the screen.

Wolf

constructor	`public Wolf()`
fighting behavior	*you decide*
color	*you decide*
movement behavior	*you decide*
toString	*you decide*

You will decide the behavior of the `Wolf` class. (Your constructor must accept no parameters as shown above.)

Wolf Class:

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

Each time a critter is asked to move (each time the `getMove` method is called), the critter is passed a parameter of type `CritterInfo` that provides useful information; your `Wolf` may wish to make use of this information to guide its movement behavior. For example, you can find out the critter's current x and y coordinates by calling the `getX` and `getY` methods, while the `getWidth` and `getHeight` methods return information about the size of the simulation world. You can find out what is around the critter by calling `getNeighbor` and passing one of the direction constants as a parameter. You will be told the display character for whatever is in that location (a `"."` for an empty cell):

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

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

You can make your `Wolf` return any text you like from its `toString` method (other than `null`) and any color you like from the `getColor` method. In fact, critters are asked what display color and character to use on each round of the simulation, so you can have a `Wolf` that displays itself differently over time. Keep in mind that the `toString` text is also passed to other animals when they fight your `Wolf`; you may wish to strategize to try to fool other animals.

For those of you who are interested, we will run a tournament at the end of class where packs of your wolves can take on packs of wolves designed by other students. (No points will be awarded for tournament performance.)

Implementation Guidelines:

The provided GUI can run even if you haven't completed all of the required critter classes. To do this, edit the file `CritterMain.java` and comment out lines that refer to critter types you have not yet written. For example, commenting out the following line would prevent the GUI from trying to place its usual 25 `Tiger` objects in the simulation:

```
model.add(25, Tiger.class);
```

The first three types of critters increase in difficulty from `Bear` to `Lion` to `Tiger`. We recommend that you write the `Bear` first. Look at the `Stone.java` file as an example of the general structure of your classes.

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 `Wolf` class. These points allow us to reward the students who spend time writing an interesting critter definition.

Style points will also be awarded on the basis of your ability to express each critter's operations simply and clearly. Your objects should be well encapsulated. You should follow past stylistic guidelines about indentation, whitespace, identifiers, and localizing variables. You should 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 or at the top of each method.