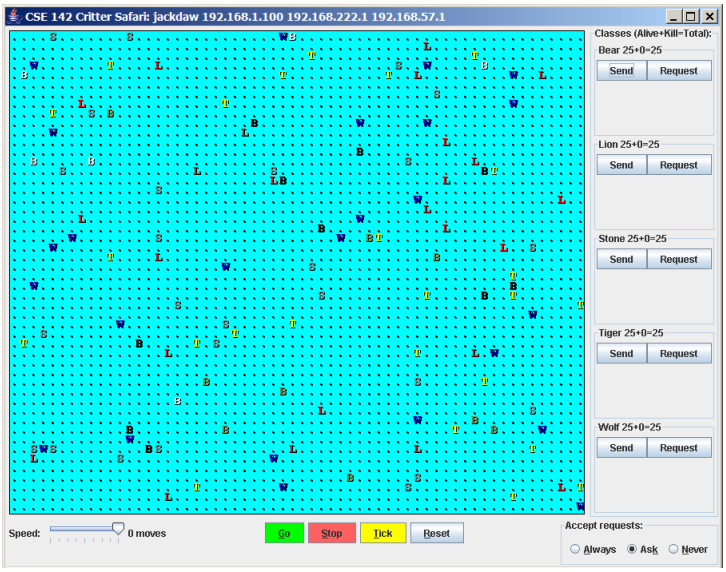
CSE 142, Spring 2007
Programming Assignment #8: Critter Safari (20 points)
Due: Tuesday, May 29, 2007, 4:00 PM

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

This assignment will give you practice with creating classes.
Turn in files named `Mouse.java`, `Tiger.java`, `Elephant.java`, and `Husky.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 (similar to the `DrawingPanel`).

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. The winning animal survives and the losing animal is removed from the simulation. The following table summarizes the possible fighting choices each animal can make and which animal will win in each case. To help you remember which beats which, notice that the starting letters and win/loss ratings of "roar, pounce, scratch" correspond to 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

There are several supporting files you should download on the course web site. Run `CritterMain` to start the simulation. If you try to run a class other than `CritterMain`, you will receive an error such as the following:

```
Error: No 'main' method in 'Tiger' with arguments: ([Ljava.lang.String;)
```

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(char opponent);
    public Color getColor();
    public int getMove(CritterInfo info);
    public char getChar();

    // 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 Mouse 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 `getChar`). 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 to ROAR in a fight.

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

public class Stone implements Critter {
    public int fight(char opponent) {
        return ROAR;
    }

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

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

    public char getChar() {
        return 'S';
    }
}
```

The `Critter` interface defines five constants for the various directions, and three additional constants for the three types of fighting. You can refer to these directly in your code (`NORTH`, `SOUTH`, `ROAR`, 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.

Critters to Implement:

The following are the four critter classes you will implement. Each class must have only 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.

Mouse

constructor	`public Mouse(Color color)`
fighting behavior	always `SCRATCH`
color	the color passed to the constructor
movement behavior	alternates between `EAST` and `SOUTH` in a zigzag pattern (first `EAST`, then `SOUTH`, then `EAST`, then `SOUTH`, ...)
character	`'M'`

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

Tiger

constructor	`public Tiger()`
fighting behavior	always `ROAR`
color	alternates between `Color.ORANGE` and `Color.BLACK` (first `Color.ORANGE`, then `Color.BLACK`, then …)
movement behavior	moves 3 steps in a random direction (`NORTH`, `SOUTH`, `EAST`, or `WEST`), then chooses a new random direction and repeats
character	`'T'`

Elephant

constructor	`public Elephant(int steps)`
fighting behavior	If opponent is a Tiger (`'T'`), then `ROAR`; otherwise `POUNCE`
color	`Color.GRAY`
movement behavior	first go `SOUTH` `steps` times, then go `WEST` `steps` times, then go `NORTH` `steps` times, then go `EAST` `steps` times (a clockwise square pattern), then repeats
character	`'E'`

The `Elephant` constructor accepts a parameter representing the distance the `Elephant` will walk in each direction before changing direction. For example, an `Elephant` constructed with a parameter value of 8 will walk 8 steps south, 8 steps west, 8 steps north, 8 steps east, and repeat. You can assume that the value passed for `steps` is at least 1.

Husky

constructor	`public Husky()`
fighting behavior	*you decide*
color	*you decide*
movement behavior	*you decide*
character	*you decide*

You will decide the behavior of the `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 the `getMove` method is called by the simulator), 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. 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. Whatever character is at that location will be returned.

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

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 `'M'` if you are fighting against a `Mouse`).

You can make your `Husky` return any character you like from its `getChar` method 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 `Husky` that displays itself differently over time. Keep in mind that the `getChar` character is also passed to other animals when they fight your `Husky`; you may wish to strategize to try to fool other animals.

On the last day of class, we will host a Critter tournament consisting of battles in the following format: Two students' `Husky` classes will be placed into the simulator world along with the other standard animals, with 25 of each animal type. The simulator will be started and run until no significant activity is occurring or until 1000 moves have passed, whichever comes first. The student whose `Husky` class has the higher sum of (critters killed + Huskies alive) wins the battle.

No grade points will be awarded for 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 can run even if you haven't completed all of the required critter classes. The first three types of critters increase in difficulty from `Mouse` to `Tiger` to `Elephant`. We recommend that you write the `Mouse` 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 `Mouse`'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 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 any 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.