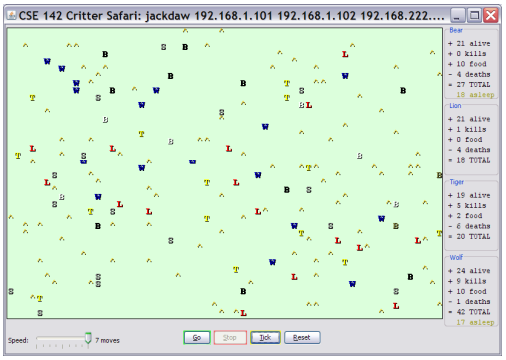
CSE 142, Summer 2008
Programming Assignment #8: Critters (20 points)
Due: Wednesday, August 20, 2008, 11:00 PM

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.

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, your objects become part of a larger system. You might want to have one of your critters make several moves 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 to 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 (represented in the code by values named `Attack.ROAR`, `Attack.POUNCE`, and `Attack.SCRATCH`). 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		
		`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

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 of the four classes you'll write will extend from a superclass named `Critter`. This is an example of inheritance, which is discussed in detail in Chapter 9 of the textbook. The inheritance makes it easier for our code to talk to all of your critter classes, and it also helps us be sure that all your animal classes will implement all of the methods we need. But don't worry; to do this assignment you don't need to understand much at all about inheritance. Your class headers should indicate the inheritance relationship by writing `extends Critter` in their header, like the following:

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

The `Critter` class contains the following five methods, which you must write in each of your four 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 in a fight, 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 behavior, you can write the methods shown above in your class to replace the default behavior with your own. This is called *overriding* the default behavior.

For example, below is a critter class called `Stone`. `Stone` objects are displayed with the letter S, are gray in color, never 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. Note that the `Stone` does not need to write an `eat` or `getMove` method; it uses the default behavior for those operations.

```
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";
    }
}
```

Critters to Implement:

The following are the four classes to implement. Each must have 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.

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"`

Think of the `Lion` as having a "hunger" that is triggered by fighting. Initially the `Lion` is not hungry (so `eat` returns `false`). But if the `Lion` gets into a fight or a series of fights (if `fight` is called on it one or more times), it becomes hungry. When a `Lion` is hungry, the next call to `eat` should return `true`. Eating once causes the `Lion` to become "full" again so that future calls to `eat` will return `false`, until the `Lion`'s next fight or series of fights.

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 hungry (if `eat` would return `true`), then scratch; else 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 `Tiger` 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 its remaining hunger, 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"`. Recall that you can convert a number to a string by concatenating it with an empty string. For example, `"" + 7` evaluates to `"7"`.

Husky

constructor	`public Husky()`	(must accept no parameters)
all other behavior	*you decide*	(see next page)

Husky Class:

You will decide the behavior of your `Husky` class. Part of your grade will be based upon writing creative and non-trivial `Husky` behavior. The following are some guidelines and hints about how to write an interesting `Husky`. There are additional methods that each critter class can use through inheritance from the `Critter` class. Your `Husky` may want to use these methods to guide its behavior. None of the methods below are needed for `Bear`, `Lion`, or `Tiger`.

- `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) {    // check if my x-coordinate is above 10
```

- `public int getWidth()` `public int getHeight()`
  Returns the width and height of the grid world.

- `public String getNeighbor(Direction direction)`
  Returns a `String` representing what is next to your critter in the given direction. `" "` means an empty square.
  For example, to check if your neighbor to the west is a `Bear`, you could write this code in your `getMove` method:

```
if (getNeighbor(Direction.WEST).equals("B")) {    // check if a Bear is 1 square west of me
```

- `public void win()` `public void lose()` `public void sleep()`
  `public void wakeup()` `public void reset()`
  Our code calls these methods on your critter to notify you when you have won a fight, lost a fight, been put to sleep, woken up from sleeping, and when the game world has reset, respectively.

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 advances.

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 grade points because it is too trivial.

Implementation Guidelines:

The provided GUI runs even if you haven't completed all the critters. The classes increase in difficulty from `Bear` to `Lion` to `Tiger`. We suggest doing `Bear` first. Look at `Stone.java` and the lecture/section examples to see the general structure.

Any critter class you write will compile even if you have not written all of the required methods from the `Critter` class. You may want to write and test some of the methods first and leave others for later.

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 for expressing each critter's behavior elegantly. **Encapsulate the data inside your objects.** 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. For reference, our `Bear`, `Lion`, and `Tiger` together occupy ~ 165 lines including comments.