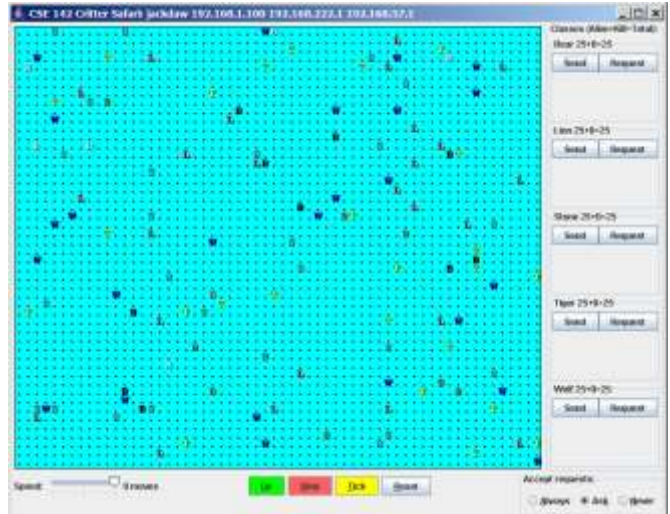


**Note 1:** this assignment is the 2<sup>nd</sup> easiest to code of all our assignments – once you understand what needs to get done. In class, we will walk through the simulation to help you visualize how the pieces work together. Do not miss this.

**Note 2:** include **all** of the support files in the folder with your code to compile.

**Note 3:** Bring working yourInitialsDolphin.class versions to class Monday week 11 to compete with your classmates.

**Program Behavior:** You will write a set of classes to **add** to a simulation program. Your classes will define the behavior of various animals. You will be given 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 by only one square -- north, south, east, west, or stay at its current location. Critters move around in a world of finite size, but the world 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 may 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 want to have one of your critters make several moves all at once—but you can't do that. The only way a critter can move at all is to wait for the simulator to ask it for a move—and then it can only move by 1 cell. Although this experience can be frustrating, it is a good introduction to the kind of programming we typically do with objects.

As the simulation runs, 2 animals can collide by moving onto the same location. When 2 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. 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 fight choice, the winner is chosen at random. Technically, when there's a collision, the simulation program asks the 2 colliding animals how each will fight. The simulation uses their responses to apply the collision rules.

		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. `CritterMain` will run using whatever `Critter` classes are in the same folder. Note: if you try to run a class other than `CritterMain`, you can expect an error such as the following:

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

Bring other error messages, if any, to class for us to troubleshoot together.

**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. (1) Your class headers should indicate that they implement this interface, as in:

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

(2) Because your classes implement the interface, you **must** include in each class a definition for each of the methods that are specified in the interface (`fight`, `getColor`, `getMove`, and `getChar`). These methods must have the exact signature given in the interface. 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 in their current location (they return `CENTER` for their move), and always "choose" to `ROAR` in a "fight." Notice how the code below meets the combined requirements for a `Critter` and for a `Stone`.

```
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';
    } }
}
```

(3) Interfaces never specify the signature for constructor methods. In fact, you can never "new" an interface; that is, you can never ask the compiler to construct an object from the definition of the interface itself, only from the definitions of classes that **implement** the interface. A class that implements an interface can have 0 or more explicit constructor methods.

(4) Interfaces never specify instance variables. In most cases, a class that implements an interface will need to set up and manipulate one or more instance variables. That's how each instance will be able to keep track of its own **state**. Recall that a local variable is erased when its method ends. But an object may need to "remember" where it is, or what it did last, or other status data, to respond appropriately the next time one of its methods is called.

The simulation (`CritterMain`) has already been written to call methods with those 4 signatures for any `Critter` in the simulation. The actual value returned and the code to produce the return value will depend on the behavior of the specific `Critter` (see below).

Also, the `Critter` interface defines five constants for the various directions plus three constants for the three types of fighting. You can – and should -- refer to these by name 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. Assume they will always be of type `int`. There does not appear to be any logical reason for the specific values, but the simulation code uses them as described. I think the designers purposely chose those particular integer values to discourage you from using the integers in your code. Regardless of what integers have been assigned, your code will be easy to read if you use the names – and hard to read if you use the integers. 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 must 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. Each animal class should be in its own .java file.

### Mouse

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

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 a `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

<b>constructor</b>	<code>public Tiger()</code>
<b>fighting behavior</b>	alternates between <code>ROAR</code> and <code>POUNCE</code> (first <code>ROAR</code> , then <code>POUNCE</code> , then ...)
<b>color</b>	<code>Color.ORANGE</code>
<b>movement behavior</b>	moves 3 steps in the same randomly chosen direction ( <code>NORTH</code> , <code>SOUTH</code> , <code>EAST</code> , or <code>WEST</code> ), then chooses a new random direction and repeats
<b>character</b>	<code>'T'</code>

### Elephant

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

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

### Dolphin

<b>constructor</b>	<code>public Dolphin()</code>
<b>fighting behavior</b>	<i>you decide</i>
<b>color</b>	<i>you decide</i>
<b>movement behavior</b>	<i>you decide</i>
<b>character</b>	<i>you decide</i>

You will decide the behavior of the `Dolphin` class. (Your constructor must accept no parameters.)

**Dolphin Class:** A `Dolphin` that stays completely still may fare well in the simulation, but it will not receive full points because it is too trivial. Do not worry about how a dolphin can survive in a safari-type setting. The dolphin is the Shoreline Community College mascot – it can thrive anywhere! Part of your grade will be based upon writing creative and non-trivial behavior for your `Dolphin` class. The following are some guidelines and hints about how to write an interesting `Dolphin`.

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 `Dolphin` may wish to make use of this information to guide its movement behavior. For example, your code 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.

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

For example, when a tiger's `getMove` method is called, that tiger can query the parameter `info` to find out just where the tiger is and how large its world is. To find out what is on each side of the tiger, the code can call `getNeighbor` and sending one of the direction constants as a parameter. Whatever character is at that location will be returned.

Your `Dolphin`'s fighting behavior may want to utilize the parameter sent to the `fight` method, `opponent`, which tells what kind of critter you are fighting against (such as 'M' if you are fighting against a `Mouse`). You can make your `Dolphin` 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 `Dolphin` that displays itself differently over time. Keep in mind that the `getChar` character is also passed to other animals when they fight your `Dolphin`; you may wish to strategize to try to fool other animals.

You will demonstrate your solution and explain your `Dolphin` behavior our penultimate day of class.

**Implementation Guidelines:** The provided GUI can run even if you haven't completed any of the required critter classes. The first three types of critters increase in difficulty from `Mouse` to `Tiger` to `Elephant`. Look at the `Stone.java` file as an example of the most basic structure of your classes. Then write the `Mouse` class.

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 for a critter class and leave others for later, write "stub" versions of the other methods that always return a simple value (for example, always return `CENTER` if you don't want to write the `Mouse`'s movement code yet). Your code will compile if the return value is merely of the expected data type. Your temporary code will probably run if your return value is one of the expected constants.

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. Your critters should not produce any console output in the completed version, but you may want to print during the debugging phase.

To reward students who spend time writing an interesting critter definition some points for this assignment will be awarded on the basis of how much creativity you put into defining an interesting `Dolphin` class. Your `Dolphin`'s behavior should not be trivial or closely match that of any existing animal shown in class. Points will also be awarded on your ability to express each critter's operations elegantly. Your objects must be **encapsulated**.

**Topics:** **ch. 9**, Inheritance, Polymorphism, super interaction, Interfaces ...

**You must also meet these Specifications:**

1. Create programs **named YourInitials** `Mouse`, **YourInitials** `Tiger`, **YourInitials** `Elephant` and **YourInitials** `Dolphin.java` (e.g. Ed Bob Hyde would use `EBH_Dolphin.java`).
2. You must meet all requirements in *Java\_Program\_Style\_Requirements\_GradingComponents* for full credit.
3. Submit the `.java` file(s), printed, easy to read listing with line numbers & screen-captures of 3 runs.