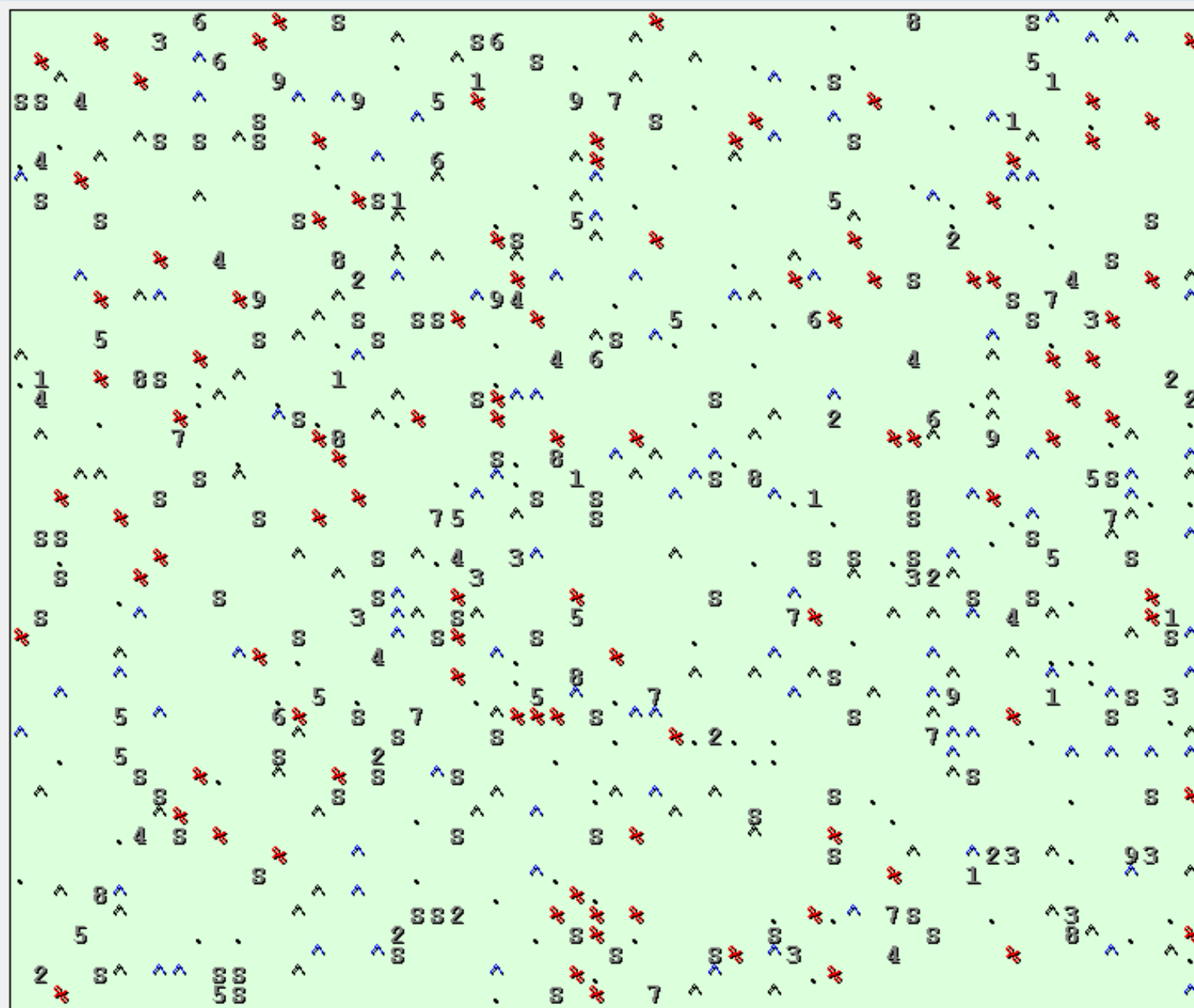


Assignment 11: Critters

HW11 Assignment Overview

Critters

- A simulation world with animal objects with behavior:
 - `fight` animal fighting
 - `getColor` color to display
 - `getMove` movement
 - `toString` letter to display
 - `eat` eat food?
- You must implement:
 - Ant
 - Bird
 - Vulture
 - Hippo
 - Longhorn (Wins Overall and Creative)

**Ant**

100 alive (- 0)
+ 0 kills
+ 0 food
=100 TOTAL

Bird

100 alive (- 0)
+ 0 kills
+ 0 food
=100 TOTAL

Hippo

100 alive (- 0)
+ 0 kills
+ 0 food
=100 TOTAL

Stone

100 alive (- 0)
+ 0 kills
+ 0 food
=100 TOTAL

Vulture

100 alive (- 0)
+ 0 kills
+ 0 food
=100 TOTAL

Speed:



0 moves

Go

Stop

Tick

Reset

How the simulator works

- When you press "Go", the simulator enters a loop:
 - move each animal once (`getMove`), in random order
 - if the animal has moved onto an occupied square, `fight`!
- Key concept: The simulator is in control, NOT your animal.
 - Example: `getMove` can return only one move at a time.
`getMove` can't use loops to return a sequence of moves.
 - It wouldn't be fair to let one animal make many moves in one turn!
 - Your animal must keep state (as fields, instance variables) so that it can make a single move, and know what moves to make later.

Scoring

- Score for each species:
- For all animals of that species
- Number of animals alive
- Number of fights won
- Pieces of food eaten

Food

- Simulator places food randomly around world
- Eating food increases score for species, but ...
- Critters sleep after eating
 - simulator (CriticMain) handles this
- A Critter that gets in a fight while sleeping always loses
 - simulator handles this

Mating

- Two Critters of same species next to each other mate and produce a baby Critter
- Simulator handles this
- Critters not asked if they want to mate
- Critters vulnerable while mating (heart graphic indicates mating)
 - automatically lose fight
- The Simulator handles all of this
 - You don't write any code to deal with mating

Critter Class

```
public abstract class Critter {  
    public boolean eat() {  
        return false;  
    }  
    public Attack fight(String opponent) {  
        return Attack.FORFEIT;  
    }  
    public Color getColor() {  
        return Color.BLACK;  
    }  
    public Direction getMove() {  
        return Direction.CENTER;  
    }  
    public String toString() {  
        return "?";  
    }  
}
```


Enums

- Critter class has two ***nested*** Enums for Direction of movement and how to fight

```
// constants for directions
public static enum Direction {
    NORTH, SOUTH, EAST, WEST, CENTER
};
```

```
// constants for fighting
public static enum Attack {
    ROAR, POUNCE, SCRATCH, FORFEIT
};
```

Nested Enums

- To access a Direction or Attack a class external to Critter would use the following syntax:
- Critter.Direction.NORTH
- Critter.Attack.POUNCE
- Classes that are descendants of Critter (like the ones you implement) do not have to use the Critter.
 - it is implicit
- Direction.SOUTH, Attack.ROAR

A Critter class

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

- `extends Critter` tells the simulator your class is a critter
- override methods from Critter based on Critter spec
- Critter has a number of methods not required by the 4 simple Critter classes (Ant, Bird, Vulture, Hippo)
- ... but you should use them to create an interesting and successful Longhorn

Critter exercise: Stone

- Write a critter class `Stone`(the dumbest of all critters):

Method	Behavior
constructor	<code>public Stone()</code>
fight	Always <code>Attack.ROAR</code>
getColor	Always <code>Color.GRAY</code>
getMove	Always <code>Direction.CENTER</code>
toString	<code>"S"</code>
eat	Always <code>false</code>



Ideas for state

- You must not only have the right state, but update that state properly when relevant actions occur.
- Counting is helpful:
 - How many total moves has this animal made?
 - How many times has it fought?
- Remembering recent actions in fields is helpful:
 - Which direction did the animal move last?
 - How many times has it moved that way?

Clicker 1

- We want to implement a Critter that moves West until it is in a fight. After the fight the Critter moves East until it is in another fight. Each time the Critter is in a fight it shifts its direction for the next move from West to East or East to West.
- Will the move method have a loop?
 - A. No
 - B. Yes
 - C. Maybe

Keeping state

- How can a critter move west until it fights?

```
public Direction getMove() {  
    while (animal has not fought) {  
        return Direction.EAST;  
    }  
    while (animal has not fought a second time) {  
        return Direction.EAST;  
    }  
}
```

```
private int fights;    // total times Critter has fought  
  
public Direction getMove() {  
    if (fights % 2 == 0) {  
        return Direction.WEST;  
    } else {  
        return Direction.EAST;  
    }  
}
```

Testing critters

- Use the MiniMain to create String based on actions and print those out
- Focus on one specific critter of one specific type
 - Only spawn 1 of each animal, for debugging
- Make sure your fields update properly
 - Use `println` statements to see field values
- Look at the behavior one step at a time
 - Use "Tick" rather than "Go"

Critter exercise: Snake

Method	Behavior
constructor	<code>public Snake(boolean northSnake)</code>
fight	alternates between SCRATCH and POUNCE
getColor	Yellow
getMove	north bound snakes: 5 steps north, pause 5 ticks, 5 steps north, pause 5 ticks, ... otherwise: 5 steps west, pause 5 ticks, 5 steps west, pause 5 ticks, ...
eat	always eats
toString	"K"



Determining necessary fields

- Information required to decide what move to make?
 - Direction to go in
 - Length of current cycle
 - Number of moves made in current cycle
- Remembering things you've done in the past:
 - an `int` counter?
 - a `boolean` flag?