



# Week 6 Discussion

Wednesday, 11/6/19



# Reminders

PSA4 Submission due Tuesday, November 12 11:59pm

For students that are missing discussion participation and you clicked in, we will handle it all at once at the end of the quarter.

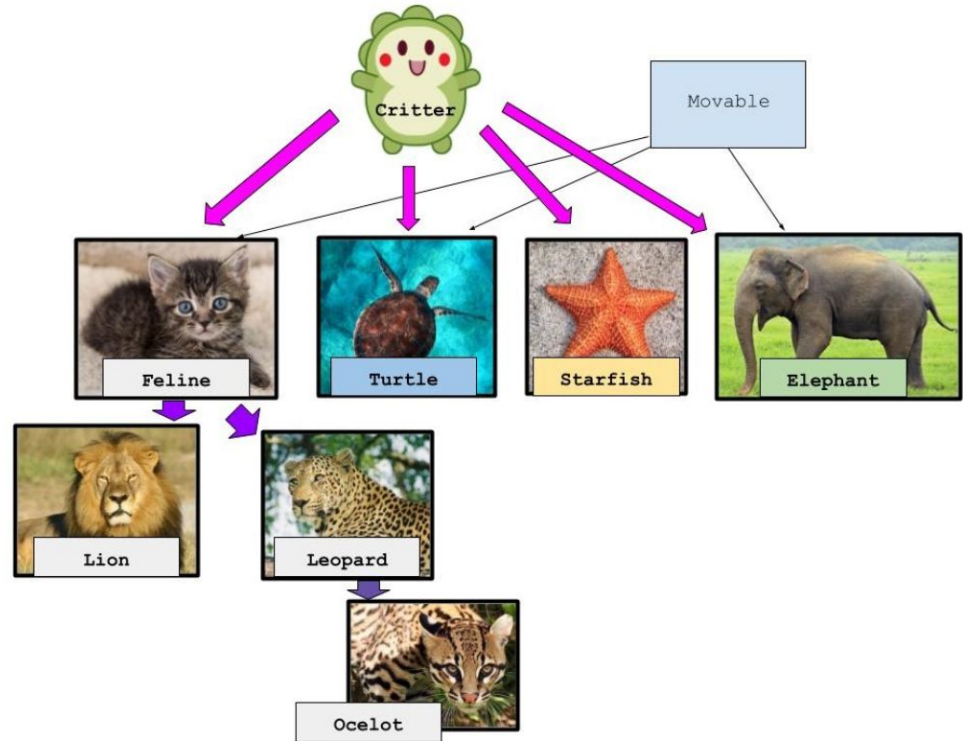
Extra credit: Beating the raccoon, but fix your code first!

# Today's agenda

- Inheritance Review
- Overview of the PA
- Game mechanics
- Critters you have to make
- Inheritance Practice

# PSA4 Overview

- Complex hierarchy of classes
- All subclasses inherit from `Critter` and have their own implementation
- No need to worry about abstract classes for now



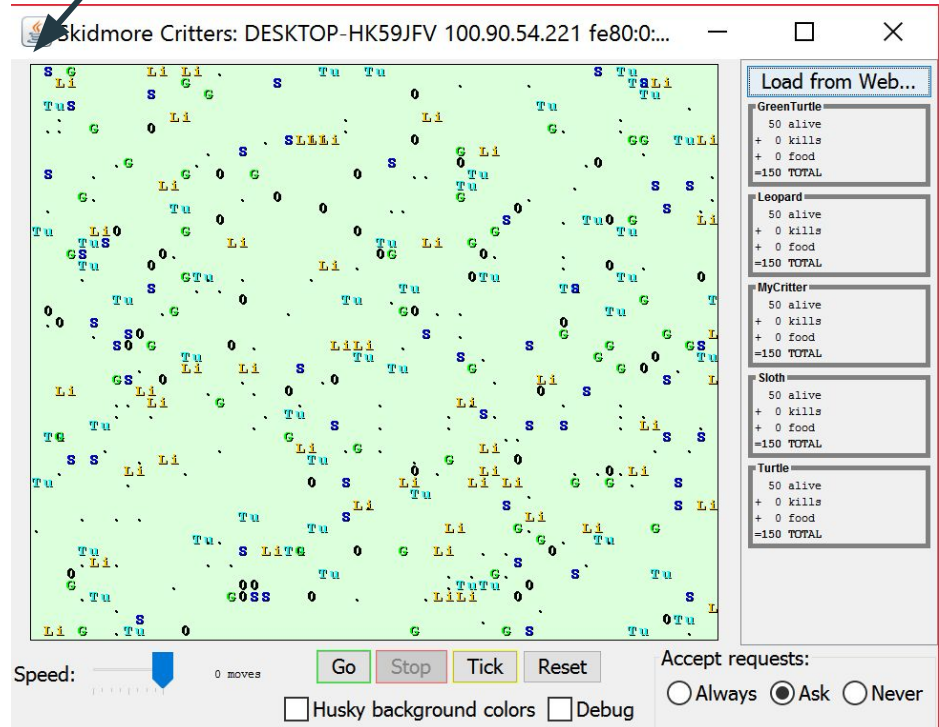
# Overriding

- Methods like `eat()`, `getColor()`, and `getAttack()` are **inherited** from the `Critter` superclass.
- The `Critter`'s implementation doesn't do anything useful, so a `Critter` object would be useless and lose in the arena.
- We must override these methods and make them be more useful to ensure the survival of our `Critters`.

# Critter World

- World is divided into cells with coordinates
- Upper-left cell is (0, 0)
- x increases to the right, y increases downward

X: 0, Y: 0



Skidmore Critters: DESKTOP-HK59JFV 100.90.54.221 fe80:0:...

Load from Web...

GreenTurtle  
50 alive  
+ 0 kills  
+ 0 food  
-150 TOTAL

Leopard  
50 alive  
+ 0 kills  
+ 0 food  
-150 TOTAL

MyCritter  
50 alive  
+ 0 kills  
+ 0 food  
-150 TOTAL

Sloth  
50 alive  
+ 0 kills  
+ 0 food  
-150 TOTAL

Turtle  
50 alive  
+ 0 kills  
+ 0 food  
-150 TOTAL

Speed: 0 moves

Go Stop Tick Reset

Husky background colors  Debug

Accept requests:  
 Always  Ask  Never

# Meet the Methods

# Movement





# Movement

- On each round of the simulation, the simulator asks each `Critter` object which direction it wants to move by calling its `getMove` method
- A `Critter` can move `North`, `South`, `West`, `East`, or `Center` once per round
- The world is finite but there is wrap around



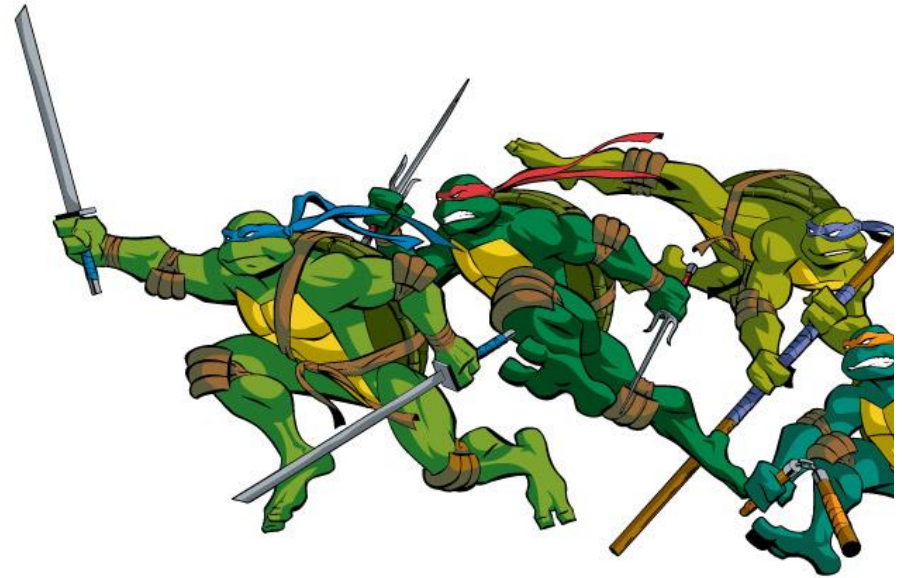
# Eating



# Eating

- Critters encounter food in the simulation world
- A Critter's eat method will decide whether or not it eats
- Critters who eat too much go into sleep mode temporarily, where they become vulnerable

# Fighting



# Fighting

- When two animals collide in the world (at the same location), they fight (if they are different species)
- The loser dies and is removed from the game
- Critters can ROAR, POUNCE, or SCRATCH
- ROAR beats SCRATCH, SCRATCH beats POUNCE, POUNCE beats ROAR -
  - Rock-Paper-Scissors logic
- If there is a tie, determine the winner randomly
- Critters can also FORFEIT, always lose unless the opponent FORFEIT as well



# Mating



# Mating

- If two animals of the same species collide (at the same location), they "mate" to produce a baby
- Both `Critters` are vulnerable to attack while mating: any other animal that collides with them will defeat them
- An animal can mate only **once** during its lifetime.
- The "baby" will be a full adult by birth and will spawn next to the parent critters when they finish mating.

# Scoring

- The simulator keeps a score for each class of animal, shown on the right side of the screen
- A `Critter`'s score is based on how many of that `Critter` are alive, how much food they have eaten, and how many other `Critters` they have defeated.



# Meet the Critters

# Critter.java

- Provided for you
- Other classes will inherit from this class
- A useful template for all of its descendants

# Starfish extends Critter

- Does not move
- Does not eat
- FORFEITS every time



# Turtle extends Critter

- Always moves WEST
- eat () only when no hostile Critter adjacent to it
- Attacks with ROAR 50% and FORFEIT the other 50%.
  - How can we calculate this?



# Feline extends Critter

- Moves new random direction every third move
  - Example: N,N,N,W,W,W,S,S,S, ...
- eats every third time of encountering food and not eating
  - Example: false, false, true, false, false, true, ...
- Always POUNCES on attack



# Lion extends Feline

- Overrides most of Feline's functionality except `getAttack()`
- Moves in a clockwise square pattern
- Only eats food if hungry (won a fight and hasn't eaten yet since the win)
  - Sleeping restores hunger





# Leopard extends Feline

- Lots of method overriding, just like with `Lion`
- Shares confidence between all Leopards
  - What keyword does this for us?
- $(\text{confidence} * 10) \% \text{ chance of eating}$
- Confidence changes based on wins and losses



# Ocelot extends Leopard

- Only override `getColor()` and `generateAttack()`
- Override `generateAttack()` to change what attack will be selected
- **Don't** override `getAttack()`





# Elephant extends Critter

- Shared goalX and goalY variables
  - All Elephants move to this point
- getMove ()
  - Once the first Elephant reaches the goal, the goal changes for all Elephants
- Be sure to check if an Elephant has reached the goal **at the beginning** of getMove ()



# buffBehavior() & debuff()

- These methods are defined in `Critter` class and you will need to override them in each `Critter` class based on how the `Critter` should behave for the buffer effect
- `buffBehavior()` adds the buffer effect for that `Critter`
- `debuff()` removes the buffer effect for that `Critter`
- Remove the buff effects in the next round after you add the buffer effects

# Lion's Buff Effect

- Buff Effect: Change the display name to LION since Lions are the kings of the Animal Kingdom
- Remove the Buff Effect: Change the display name back to Lion



# Leopard's Buff Effect

- Buff Effect: Change its display name to lalalala~~~~ since Leopards are happily running fast
- Remove Buff Effect: Change its display name back to Lpd



# Ocelot's Buff Effect

- Buff Effect: Follow Leopard's buff action - they run fast too!
- Remove Buffer Effect: Change its name back to Oce



# Starfish's Buff Effect

- Buff Effect: Transparency by changing the display name to an empty string so no one can see them
- Remove Buff Effect: Change the display name back to original
- These are already implemented for you as an example, but Starfish has another effect...



# Starfish's Teleportation

- `void teleport(Point currentLocation, Critter[][] arena)`
  - Modify the point (location of a Starfish) to a random coordinate within the arena
    - The arena will update this automatically
  - Set the previous location of the Starfish in the arena to `null`



# More Inheritance Practice Problems



```
public class Vehicle {
    String name;
    int year;
    public void startEngine() {
        System.out.println("Engine");
    }
    public String getName() {
        return name;
    }
}
```

```
public class Car extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("VROOM");
    }
}
```

```
public class Motorcycle extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("ZOOM");
    }
}
```

```
public static void main(String[] args) {
    Vehicle v = new Vehicle();
    v.startEngine();
    Vehicle c = new Car();
    c.startEngine();
    Motorcycle m = new Motorcycle();
    m.startEngine();
}
```

// What is the output?

```
public class Vehicle {
    String name;
    int year;
    public void startEngine() {
        System.out.println("Engine");
    }
    public String getName() {
        return name;
    }
}
```

```
public class Car extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("VROOM");
    }
}
```

```
public class Motorcycle extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("ZOOM");
    }
}
```

```
public static void main(String[] args) {
    Vehicle v = new Vehicle();
    v.startEngine();
    Vehicle c = new Car();
    c.startEngine();
    Motorcycle m = new Motorcycle();
    m.startEngine();
}
```

// What is the output?

Engine

VROOM

ZOOM

# Inheritance Practice 2

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Starting Engine");
    }
}

public class Car extends Vehicle {
    public void startEngine(int x) {
        System.out.println("Car " + x);
    }
}

public static void main(String[] args) {
    Vehicle c = new Car();
    c.startEngine();
}
```

**What gets printed?**

A:  
Starting Engine

B:  
Car 0

C:  
compiler error

# Inheritance Practice 2

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Starting Engine");
    }
}

public class Car extends Vehicle {
    public void startEngine(int x) {
        System.out.println("Car " + x);
    }
}

public static void main(String[] args) {
    Vehicle c = new Car();
    c.startEngine();
}
```

**This gets printed.**

A:  
Starting Engine

B:  
Car 0

C:  
compiler error

# Inheritance Practice 3

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Starting Engine");
    }
}

public class Car extends Vehicle {
    public void startEngine(int x) {
        System.out.println("Car " + x);
    }
}

public static void main(String[] args) {
    Vehicle c = new Car();
    c.startEngine(1);
}
```

**What gets printed?**

A:  
Starting Engine

B:  
Car 1

C:  
compiler error

# Inheritance Practice 3

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Starting Engine");
    }
}

public class Car extends Vehicle {
    public void startEngine(int x) {
        System.out.println("Car " + x);
    }
}

public static void main(String[] args) {
    Vehicle c = new Car();
    c.startEngine(1);
}
```

**This gets printed.**

A:  
Starting Engine

B:  
Car 1

C:  
compiler error

# Inheritance Practice 4

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Starting Engine");
    }
}

public class Car extends Vehicle {
    public void startEngine(int x) {
        System.out.println("Car " + x);
    }
}

public static void main(String[] args) {
    Car c = new Car();
    c.startEngine(1);
}
```

**What gets printed?**

A:  
Starting Engine

B:  
Car 1

C:  
compiler error

# Inheritance Practice 4

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Starting Engine");
    }
}

public class Car extends Vehicle {
    public void startEngine(int x) {
        System.out.println("Car " + x);
    }
}

public static void main(String[] args) {
    Car c = new Car();
    c.startEngine(1);
}
```

**This gets printed.**

A:  
Starting Engine

B:  
Car 1

C:  
compiler error



## // Inheritance Constructor Example 1

```
public class Dog {
    public String name;
    public Dog() {
        name = "Dog";
    }
}

public class Husky extends Dog {
    public Husky() {
        name = "Husky";
    }
}

// in main()
Dog sydney = new Husky();
System.out.println(sydney.name);
```

## // Inheritance Constructor Example 1

```
public class Dog {
    public String name;
    public Dog() {
        name = "Dog";
    }
}
public class Husky extends Dog {
    public Husky() {
        name = "Husky";
    }
}
```

```
// in main()
```

```
Dog sydney = new Husky();
```

```
System.out.println(sydney.name);
```

**Answer: Husky**

## // Inheritance Constructor Example 2

```
public class Dog {
    public String name;
    public Dog() {
        name = "Dog";
    }
}

public class Husky extends Dog {
    public Husky() {
        // do nothing
    }
}

// in main
Dog sydney = new Husky();
Husky cindy = new Husky();
System.out.println(sydney.name + " " + cindy.name);
```

## // Inheritance Constructor Example 2

```
public class Dog {
    public String name;
    public Dog() {
        name = "Dog";
    }
}

public class Husky extends Dog {
    public Husky() {
        // do nothing call super() by default
    }
}

// in main
Dog sydney = new Husky();
Husky cindy = new Husky();
System.out.println(sydney.name + " " + cindy.name);
Answer: Dog Dog
```

### // Inheritance Constructor Example 3

```
public class Dog {
    public String name;
    public Dog() {
        name = "Dog";
    }
    public Dog(String name) {
        this.name = name;
    }
}

public class Husky extends Dog {
    public Husky() {
        // do nothing
    }
    public Husky(String name) {
        // do nothing
    }
}

// in main
Dog sydney = new Husky();
Husky cindy = new Husky("husky");
System.out.println(sydney.name + " " + cindy.name);
```

### // Inheritance Constructor Example 3

```
public class Dog {
    public String name;
    public Dog() {
        name = "Dog";
    }
    public Dog(String name) {
        this.name = name;
    }
}
```

```
public class Husky extends Dog {
    public Husky() {
        // do nothing
    }
    public Husky(String name) {
        // do nothing
    }
}
```

// in main

```
Dog sydney = new Husky();
```

```
Husky cindy = new Husky("husky");
```

```
System.out.println(sydney.name + " " + cindy.name);
```

**Answer: Dog Dog**

## // Inheritance Constructor Example 4

```
public class Dog {
    public String name;
    public Dog() {
        name = "Dog";
    }
    public Dog(String name) {
        this.name = name;
    }
}
```

```
public class Husky extends Dog {
    public Husky() {
        // do nothing
    }
    public Husky(String name) {
        super(name);
    }
}
```

```
// in main
```

```
Dog sydney = new Husky();
Husky cindy = new Husky("Husky");
System.out.println(sydney.name + " " + cindy.name);
```

## // Inheritance Constructor Example 4

```
public class Dog {
    public String name;
    public Dog() {
        name = "Dog";
    }
    public Dog(String name) {
        this.name = name;
    }
}
```

```
public class Husky extends Dog {
    public Husky() {
        // do nothing
    }
    public Husky(String name) {
        super(name);
        // super() not called
    }
}
```

// in main

```
Dog sydney = new Husky();
Husky cindy = new Husky("Husky");
System.out.println(sydney.name + " " + cindy.name);
```

**Answer: Dog Husky**



# Inheritance Practice 5

```
public class Vehicle {  
    public Vehicle(int x) {  
        System.out.println("NEW VEHICLE " + x);  
    }  
}
```

```
public class Plane extends Vehicle {  
    public Plane() {  
        super(1);  
        System.out.println("NEW PLANE");  
    }  
}
```

```
public static void main(String[] args) {  
    Vehicle v = new Vehicle(2);  
    Plane c = new Plane();  
}
```

## What gets printed?

A:  
NEW VEHICLE 2  
NEW PLANE

B:  
NEW VEHICLE 2  
NEW VEHICLE 1  
NEW PLANE

C:  
compiler error

# Inheritance Practice 5

```
public class Vehicle {  
    public Vehicle(int x) {  
        System.out.println("NEW VEHICLE " + x);  
    }  
}
```

```
public class Plane extends Vehicle {  
    public Plane() {  
        super(1);  
        System.out.println("NEW PLANE");  
    }  
}
```

```
public static void main(String[] args) {  
    Vehicle v = new Vehicle(2);  
    Plane c = new Plane();  
}
```

**This gets printed.**

A:  
NEW VEHICLE 2  
NEW PLANE

B:  
NEW VEHICLE 2  
NEW VEHICLE 1  
NEW PLANE

C:  
compiler error

# Inheritance Practice 6

```
public class Vehicle {
    public Vehicle(int x) {
        System.out.println("NEW VEHICLE " + x);
    }
}

public class Plane extends Vehicle {
    public Plane() {
        super(1);
        System.out.println("NEW PLANE");
    }
}

public static void main(String[] args) {
    Vehicle v = new Vehicle(2);
    Plane c = new Plane(3);
}
```

## What gets printed?

A:  
NEW VEHICLE 2  
NEW VEHICLE 1  
NEW PLANE

B:  
NEW VEHICLE 2  
NEW VEHICLE 3  
NEW PLANE

C:  
compiler error

# Inheritance Practice 6

```
public class Vehicle {
    public Vehicle(int x) {
        System.out.println("NEW VEHICLE " + x);
    }
}

public class Plane extends Vehicle {
    public Plane() {
        super(1);
        System.out.println("NEW PLANE");
    }
}

public static void main(String[] args) {
    Vehicle v = new Vehicle(2);
    Plane c = new Plane(3);
}
```

**This gets printed.**

A:  
NEW VEHICLE 2  
NEW VEHICLE 1  
NEW PLANE

B:  
NEW VEHICLE 2  
NEW VEHICLE 3  
NEW PLANE

C:  
compiler error

**-We don't inherit the constructor method-**

# Inheritance Practice 7

```
public class Vehicle {  
    public Vehicle(int x) {  
        System.out.println("NEW VEHICLE " + x);  
    }  
}
```

```
public class Plane extends Vehicle {  
    public Plane() {  
        System.out.println("NEW PLANE");  
    }  
}
```

```
public static void main(String[] args) {  
    Vehicle v = new Vehicle(2);  
    Plane c = new Plane();  
}
```

## What gets printed?

A:  
NEW VEHICLE 2  
NEW PLANE

B:  
NEW VEHICLE 2  
NEW VEHICLE 0  
NEW PLANE

C:  
compiler error

# Inheritance Practice 7

```
public class Vehicle {  
    public Vehicle(int x) {  
        System.out.println("NEW VEHICLE " + x);  
    }  
}
```

```
public class Plane extends Vehicle {  
    public Plane() {  
        super(); // why?  
        System.out.println("NEW PLANE");  
    }  
}
```

```
public static void main(String[] args) {  
    Vehicle v = new Vehicle(2);  
    Plane c = new Plane();  
}
```

**This gets printed.**

A:  
NEW VEHICLE 2  
NEW PLANE

B:  
NEW VEHICLE 2  
NEW VEHICLE 0  
NEW PLANE

C:  
compiler error

# Inheritance Practice 8

```
public class Vehicle {
    public Vehicle() {
        System.out.println("NEW VEHICLE");
    }
}

public class Plane extends Vehicle {
    public Plane() {
        this(1);
        System.out.println("YAY");
    }
    public Plane(int x) {
        System.out.println("NEW PLANE");
    }
}

public static void main(String[] args) {
    Vehicle v = new Vehicle();
    Plane c = new Plane();
}
```

What gets printed?

A:  
NEW VEHICLE  
NEW PLANE  
YAY

B:  
NEW VEHICLE  
NEW VEHICLE  
NEW PLANE  
YAY

C:  
NEW VEHICLE  
NEW VEHICLE  
NEW VEHICLE  
NEW PLANE  
YAY

# Inheritance Practice 8

```
public class Vehicle {
    public Vehicle() {
        System.out.println("NEW VEHICLE");
    }
}

public class Plane extends Vehicle {
    public Plane() {
        this(1);
        System.out.println("YAY");
    }
    public Plane(int x) {
        System.out.println("NEW PLANE");
    }
}

public static void main(String[] args) {
    Vehicle v = new Vehicle();
    Plane c = new Plane();
}
```

**This gets printed.**

A:  
NEW VEHICLE  
NEW PLANE  
YAY

B:  
NEW VEHICLE  
NEW VEHICLE  
NEW PLANE  
YAY

C:  
NEW VEHICLE  
NEW VEHICLE  
NEW VEHICLE  
NEW PLANE  
YAY