CSE 11: Introduction to Programming and Computational Problem Solving: Accelerated Pace

Assignment 8

Interfaces

Due: Wednesday, June 5, 11:59 PM

Learning goals:

- Apply knowledge of Inheritance, Polymorphism, Abstract and Concrete Classes, and Interfaces to build an Animal Kingdom Game.
- Apply knowledge of UML diagrams to create files, containing classes and their methods.

NOTE: This programming assignment must be done individually.

Your grade will be determined by your most recent submission. If you submit to Gradescope after the deadline, it will be marked late and the late penalty will apply regardless of whether you had past submissions before the deadline.

If your code does not compile on Gradescope and the autograder fails to execute properly, you will receive an automatic zero on the autograder portion of the assignment. Make sure that you do not change the method headers or import other classes unless otherwise specified.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 11 Coding Style Guidelines. These guidelines can also be found on Canvas. Please ensure you have COMPLETE file headers, class headers, and method headers, you use descriptive variable names and proper indentation, and you avoid using magic numbers.

Part 0: Getting started (0 points)

1. If using a personal computer, then ensure your Java software development environment does not have any issues. If there are any issues, then review Assignment 1, or come to the office/lab hours before you start Assignment 8.
2. First, navigate to the `cse11` folder you created in Assignment 1 and create a new folder named `assignment8`.
3. This assignment does **NOT** have any starter code.
4. The objective of this assignment is to create the necessary files and implement the methods that follow the relationships set up by the UML diagram.

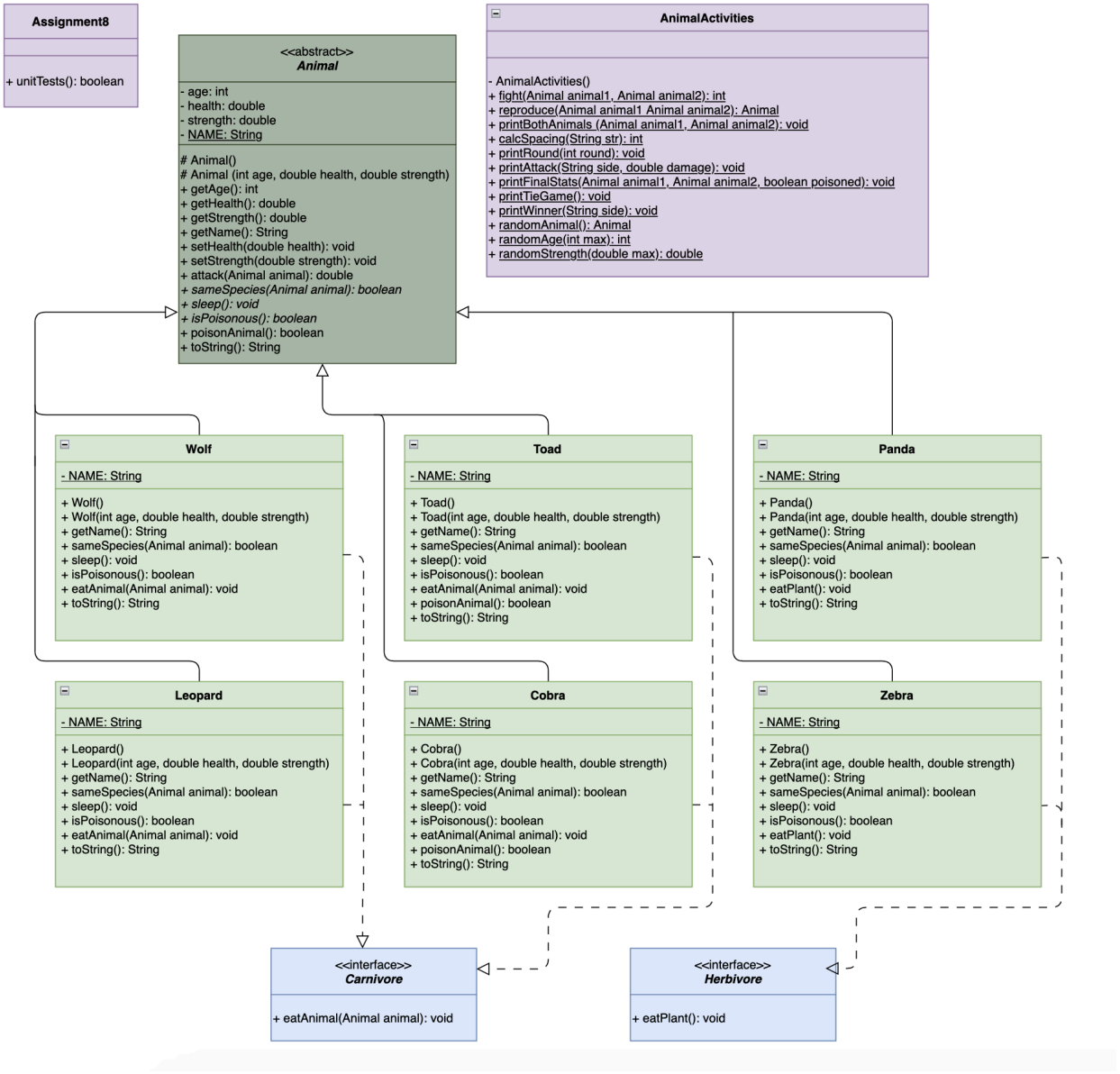
Part 1: Overview

Scenario:

CSE 11 is coming to an end, and we've learned a lot this quarter! Let's have some fun with this last assignment and create a simple text-based animal fighting game, additionally implementing extra methods along the way to develop our animal kingdom! Basically, the game involves two animals fighting each other in an endless number of rounds until one of them (or both) run out of health! All the details will be explained below, but let's first take a high-level look at the structure.

Logistics:

For this assignment, you will be implementing the classes shown in the following UML diagram.



In the UML diagram above, each rectangle in the UML diagram represents a class. There is an abstract `Animal` superclass that multiple subclasses extend from: `Wolf`, `Leopard`, `Panda`, `Zebra`, `Toad`, and `Cobra`. There are two interfaces: `Carnivore` and `Herbivore`. Remember, the solid line with hollow triangle represents inheritance (extends) and the dotted line with hollow triangle represents implementing an interface. If the image looks blurry in the write-up, then open `PA8_UML.pdf`.

Before you start programming, please take some time to review this write-up, reading the instructions below **CAREFULLY**. Some of the methods have been provided for you, but because we are not giving any starter code, you will have to copy and paste the necessary code

into your own file. You will still need to supply those methods with a method header for coding style points. You should fully understand the purpose of each variable and the usage for each method before you implement anything.

NOTE 1: You must NOT change any data field or method signature defined in the write-up, as it will fail to compile on the Gradescope autograder. As such, do NOT add any additional parameters to methods. Feel free to add any helper methods if desired.

NOTE 2: Especially because we are not providing the starter code, do NOT forget to adhere to the CSE 11 style guidelines. Declare any necessary constants to adhere to guidelines.

NOTE 3: You can assume that all inputs will be valid, unless specified otherwise.

Be sure to compile your code often, so that you can catch compile errors early on! Recall, to compile multiple Java files, use:

```
> javac *.java
```

You will be implementing methods in every provided Java class, with the `AnimalActivities` class having the majority of the functionality of this program.

Notice how each member field is declared `private`**.** This means that the member is only visible *within* the class, not from any other class. In other words, you will need to use accessors (i.e., getter methods) and mutators (i.e., setter methods) to access and modify, respectively, these `private` members. **You must also use the** `this` **keyword to access member variables hidden by local variables.**

Part 2: `Animal.java` (10 points)

First, you need to implement the abstract class named `Animal`. This is the superclass for most of the other classes in this assignment (as seen in the UML diagram). The abstract `Animal` class initializes the core characteristics of an `Animal` and defines the default behavior of specific methods (some of which are overridden by subclasses).

The abstract `Animal` class has the following data fields:

- `private int age`
- `private double health`
- `private double strength`
- `private static final String NAME`
    - Must be string literal `"Animal"`

First, in `Animal.java`, you need to implement the following constructors:

- `protected Animal()`
    - This no-arg constructor sets the instance variables of the object:
        - `age` must be set to 0
        - `health` must be set to 0
        - `strength` must be set to 0
- `protected Animal(int age, double health, double strength)`
    - This constructor sets the corresponding instance variables of the object to what the caller of the constructor passed in as arguments.

Remember, you must use the `this` keyword to access member variables hidden by local variables.

Next, complete the getters and setters to access and mutate the data fields

- `public int getAge()`
- `public double getHealth()`
- `public double getStrength()`
- `public String getName()`
- `public void setHealth(double health)`
- `public void setStrength(double strength)`

Then, implement the following methods:

- `public double attack(Animal animal)`
    - This method will be a crucial component of the game.
        - Generate a random `double` from the range: 1 (inclusive) to the `strength` of the current object (inclusive). You may import `java.util.Random` for this method.
        - Decrease the `health` of the input `animal` by the `double` generated based on the current object's `strength`.
        - Return the random `double` you generated.
    - There are many ways to implement this method. It doesn't matter how you choose to tackle it, just ensure that the `double` is between the specified range. For example, if the current `Animal`'s `strength` is 100. We **do not** want the current `Animal` object to be able to generate an `attack` that is greater than 100.

Methods to be overridden by subclasses:

- `public abstract boolean sameSpecies(Animal animal)`
    - Declare this method without a body, so it can be overridden by subclasses that extend from `Animal`. (Hint: We did this several times in Assignment 6).
- `public abstract void sleep()`
    - Declare this method without a body, so it can be overridden by subclasses that extend from `Animal`. (Hint: We did this several times in Assignment 6).
- `public abstract boolean isPoisonous()`
    - Declare this method without a body, so it can be overridden by subclasses that extend from `Animal`. (Hint: We did this several times in Assignment 6).
- `public boolean poisonAnimal()`
    - As default, return `false.`
- `public String toString()`
    - This method must return the string representation of the `Animal` object. Don't forget the override annotation (see lecture 12, slide 33). This method will give you the `String` representation of an `Animal`. You may copy and paste the implementation below, as this method is mainly to help you test and debug your methods.

```
@Override
public String toString() {
    return "(Animal)" + " age: " + getAge() +
            "; health: " + getHealth() + "; strength: " + getStrength();
}
```

Part 3: Interfaces (0 points)

Create the `Interface` files and the methods they declare, based on the UML diagram.

Remember, you must NOT change the existing signature or the fields.

1. The `Carnivore` interface has one method:
    - `public void eatAnimal(Animal animal)`

2. The `Herbivore` interface has one method:
    - `public void eatPlant()`

Part 4: Create some Animals (45 points)

`Wolf`, `Leopard`, `Panda`, `Zebra`, `Toad`, and `Cobra` are all subclasses of `Animal`. **Complete all remaining constructors and methods in these classes.**

Part 4a: Carnivores (30/45 points)

You will be writing methods that override methods from the superclass `Animal`.

Note: Remember to use the override annotation everytime you override a method (see lecture 12, slide 33).

`Wolf`

Add the following field:

- `private static final String NAME`
    - Must be string literal `"Wolf"`

Implement the following constructors and methods:

- `public Wolf()`

    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
- `public Wolf(int age, double health, double strength)`
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
- `public String getName()`
    - This method overrides the `getName()` method in `Animal`. Simple getter method that returns the `NAME` static variable.
- `public boolean sameSpecies(Animal animal)`
    - This method overrides the `sameSpecies()` method in `Animal`. This method must return `true` only when the current `Wolf` has the same `NAME` as the input `animal`. Otherwise, it must return `false`.
- `public void sleep()`
    - This method overrides the `sleep()` method in `Animal`. When a `Wolf` sleeps, their `strength` increases by 1.6 times. For example, if their `strength` was 10, it becomes 16 = 10 * 1.6.
- `public boolean isPoisonous()`
    - This method overrides the `isPoisonous()` method in `Animal`. `Wolf`s are **not** poisonous, so return `false`.
- `public void eatAnimal(Animal animal)`
    - When a `Wolf` eats another animal, it gains 60% of that animal's `strength`. To calculate the new strength of the `Wolf`, multiply the eaten animal's `strength` by 0.60 and add it to the `Wolf`'s current `strength`. For example, if a `Wolf` with a `strength` of 10 eats an animal with a `strength` of 20, the `Wolf`'s new `strength` becomes 22 = 10 + (20 * 0.60).
- `public String toString()`
    - This method must return the string representation of the `Wolf` object. This method will give you the `String` representation of a `Wolf`. You may copy and paste the implementation below, as this method is mainly to help you test and debug your methods.

```
    @Override
    public String toString() {
        return super.toString() + "; species: Wolf";
    }
```

`Leopard`

Add the following field:

- `private static final String NAME`
    - Must be string literal `"Leopard"`

Implement the following constructors and methods:

- `public Leopard()`
    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
- `public Leopard(int age, double health, double strength)`
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
- `public String getName()`
    - This method overrides the `getName()` method in `Animal`. Simple getter method that returns the `NAME` static variable.
- `public boolean sameSpecies(Animal animal)`
    - This method overrides the `sameSpecies()` method in `Animal`. This method must return `true` only when the current `Leopard` has the same `NAME` as the input `animal`. Otherwise, it must return `false`.
- `public void sleep()`
    - This method overrides the `sleep()` method in `Animal`. When a `Leopard` sleeps, their `strength` increases by 1.5 times. For example, if their `strength` was 10, it becomes 15 = 10 * 1.5.
- `public boolean isPoisonous()`
    - This method overrides the `isPoisonous()` method in `Animal`. `Leopard`s are **not** poisonous, so return `false`.
- `public void eatAnimal(Animal animal)`

    - When a `Leopard` eats another animal, it gains 55% of that animal's `strength`. To calculate the new `strength` of the `Leopard`, multiply the eaten animal's `strength` by 0.55 and add it to the `Leopard`'s current `strength`. For example, if a `Leopard` with a `strength` of 10 eats an animal with a `strength` of 20, the `Leopard`'s new `strength` becomes 21 = 10 + (20 * 0.55).
- `public String toString()`
    - This method must return the string representation of the `Leopard` object. This method will give you the `String` representation of a `Leopard`. You may copy and paste the implementation below, as this method is mainly to help you test and debug your methods.

```
@Override
public String toString() {
    return super.toString() + "; species: Leopard";
}
```

`Toad`

Add the following field:

- `private static final String NAME`
    - Must be string literal `"Toad"`

Implement the following constructors and methods:

- `public Toad()`
    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
- `public Toad(int age, double health, double strength)`
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
- `public String getName()`
    - This method overrides the `getName()` method in `Animal`. Simple getter method that returns the `NAME` static variable.
- `public boolean sameSpecies(Animal animal)`

    - This method overrides the `sameSpecies()` method in `Animal`. This method must return `true` only when the current `Toad` has the same `NAME` as the input `animal`. Otherwise, it must return `false`.
- `public void sleep()`
    - This method overrides the `sleep()` method in `Animal`. When a `Toad` sleeps, their `strength` increases by 1.2 times. For example, if their `strength` was 10, it becomes 12 = 10 * 1.2.
- `public boolean isPoisonous()`
    - This method overrides the `isPoisonous()` method in `Animal`. `Toad`s are poisonous, so return `true`.
- `public void eatAnimal(Animal animal)`
    - A `Toad` will eat bugs off a dead animal carcass instead of the carcass itself. Each bug increases the `Toad`'s `strength` by 30% of the carcass's strength. The number of bugs eaten is randomly chosen between 0 and 10 (inclusive). You may import `java.util.Random` for this method.

      Concrete Example: Suppose there is a dead animal carcass with a `strength` of 20. The toad will eat bugs off this carcass. Each bug increases the toad's strength by 30% of the carcass's strength, which is 6 = 20 * 0.3. The number of bugs eaten is randomly chosen between 0 and 10. Let's say the random number generated is 4. So, the toad's strength increase would be 24 = 4 * 6. If the toad initially has a strength of 10, its new strength will be 34 = 10 + 24. You may import `java.util.Random` for this method.

- `public boolean poisonAnimal()`
    - This method overrides the `poisonAnimal()` method in `Animal`. A `Toad` has a 30% chance to poison the other animal when fighting. If that animal is poisoned, they will die at the end of the round, even if they win the battle. If the `Toad` wins the battle it doesn't matter (this will be implemented later). Basically, return `true` 30% of the time. (Implementation Hint: The `nextDouble()` method from the `Random` class is used to get the next pseudorandom, uniformly distributed `double` value between 0.0 and 1.0 from this random number generator's sequence. You may import `java.util.Random` for this method.)

- `public String toString()`
    - This method must return the string representation of the `Toad` object. This method will give you the `String` representation of a `Toad`. You may copy and paste the implementation below, as this method is mainly to help you test and debug your methods.

```
@Override
public String toString() {
    return super.toString() + "; species: Toad";
}
```

`Cobra`

Add the following field:

- `private static final String NAME`
    - Must be string literal `"Cobra"`

Implement the following constructors and methods:

- `public Cobra()`
    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
- `public Cobra(int age, double health, double strength)`
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
- `public String getName()`
    - This method overrides the `getName()` method in `Animal`. Simple getter method that returns the `NAME` static variable.
- `public boolean sameSpecies(Animal animal)`
    - This method overrides the `sameSpecies()` method in `Animal`. This method must return `true` only when the current `Cobra` has the same `NAME` as the input `animal`. Otherwise, it must return `false`.
- `public void sleep()`
    - This method overrides the `sleep()` method in `Animal`. When a `Cobra` sleeps, their `strength` increases by 1.7 times. For example, if their `strength` was 10, it becomes 17 = 10 * 1.7.

- `public boolean isPoisonous()`
    - This method overrides the `isPoisonous()` method in `Animal`. `Cobra`s are poisonous, so return `true`.
- `public void eatAnimal(Animal animal)`
    - When a `Cobra` eats another animal, it gains 90% of that animal's `strength`. To calculate the new `strength` of the `Cobra`, multiply the eaten animal's `strength` by 0.90 and add it to the `Cobra`'s current `strength`. For example, if a `Cobra` with a `strength` of 10 eats an animal with a `strength` of 20, the `Cobra`'s new `strength` becomes 28 = 10 + (20 * 0.90).
- `public boolean poisonAnimal()`
    - This method overrides the `poisonAnimal()` method in `Animal`. A `Cobra` has a 80% chance to poison the other animal when fighting. If that animal is poisoned, they will die at the end of the round even if they win the battle. If the `Cobra` wins the battle it doesn't matter (this will be implemented later). Basically, return `true` 80% of the time. (Implementation Hint: The `nextDouble()` method from the `Random` class is used to get the next pseudorandom, uniformly distributed `double` value between 0.0 and 1.0 from this random number generator's sequence. You may import `java.util.Random` for this method.)
- `public String toString()`
    - This method must return the string representation of the `Cobra` object. This method will give you the `String` representation of a `Cobra`. You may copy and paste the implementation below, as this method is mainly to help you test and debug your methods.

```
@Override
public String toString() {
    return super.toString() + "; species: Cobra";
}
```

Part 4b: Herbivores (15/45 points)

You will be writing methods that override methods from the superclass `Animal`.

Note: Remember to use the override annotation everytime you override a method (see lecture 12, slide 33).

`Panda`

Add the following field:

- `private static final String NAME`
    - Must be string literal `"Panda"`

Implement the following constructors and methods:

- `public Panda()`
    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
- `public Panda(int age, double health, double strength)`
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!)
- `public String getName()`
    - This method overrides the `getName()` method in `Animal`. Simple getter method that returns the `NAME` static variable.
- `public boolean sameSpecies(Animal animal)`
    - This method overrides the `sameSpecies()` method in `Animal`. This method must return `true` only when the current `Panda` has the same `NAME` as the input `animal`. Otherwise, it must return `false`.
- `public void sleep()`
    - This method overrides the `sleep()` method in `Animal`. When a `Panda` sleeps, their `strength` increases by 1.4 times. For example, if their `strength` was 10, it becomes 14 = 10 * 1.4.
- `public boolean isPoisonous()`
    - This method overrides the `isPoisonous()` method in `Animal`. `Panda`s are **not** poisonous, so return `false`.
- `public void eatPlant()`
    - A `Panda` instance is an herbivore, so they don't eat animals. Instead, they feed on some plants and will randomly gain between 0 (inclusive) and 50 (inclusive) strength. You may import `java.util.Random` for this method.
- `public String toString()`
    - This method must return the string representation of the `Panda` object. This method will give you the `String` representation of a `Panda`. You may copy and

paste the implementation below, as this method is mainly to help you test and debug your methods.

```
@Override
public String toString() {
    return super.toString() + "; species: Panda";
}
```

`Zebra`

Add the following field:

- `private static final String NAME`
    - Must be string literal `"Zebra"`

Implement the following constructors and methods:

- `public Zebra()`
    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
- `public Zebra(int age, double health, double strength)`
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
- `public String getName()`
    - This method overrides the `getName()` method in `Animal`. Simple getter method that returns the `NAME` static variable.
- `public boolean sameSpecies(Animal animal)`
    - This method overrides the `sameSpecies()` method in `Animal`. This method must return `true` only when the current `Zebra` has the same `NAME` as the input `animal`. Otherwise, it must return `false`.
- `public void sleep()`
    - This method overrides the `sleep()` method in `Animal`. When a `Zebra` sleeps, their `strength` increases by 1.3 times. For example, if their `strength` was 10, it becomes 13 = 10 * 1.3.
- `public boolean isPoisonous()`
    - This method overrides the `isPoisonous()` method in `Animal`. `Zebra`s are **not** poisonous, so return `false`.

- `public void eatPlant()`
    - A `Zebra` instance is an herbivore, so they don't eat animals. Instead, they feed on some plants and will randomly gain between 0 (inclusive) and 40 (inclusive) strength. You may import `java.util.Random` for this method.
- `public String toString()`
    - This method must return the string representation of the `Zebra` object. This method will give you the `String` representation of a `Zebra`. You may copy and paste the implementation below, as this method is mainly to help you test and debug your methods.

```
@Override
public String toString() {
    return super.toString() + "; species: Zebra";
}
```

Part 5: `AnimalActivities.java` (25 points)

Finally, the cool part! You will be implementing a **final** `AnimalActivities` class with two unique static methods related to animals. The first method `fight()` will be our game and the second method `reproduce()` is to add some depth to our animal-related assignment. Based on the UML diagram above, you only need to implement `fight()` and `reproduce()` in the `AnimalActivities` class, the code for the rest will be provided for you.

First, implement the AnimalActivities constructor:

- `private AnimalActivities()`
    - Our `AnimalActivities` class will consist only of static methods. Therefore, to prevent instantiation of this class, declare a private no-arg constructor.

Method 1 - `fight` (15 points)

The first method is `fight`. The method signature for it is:

`public static int fight(Animal animal1, Animal animal2)`

- This method is a bit intricate, as it involves printing things to the terminal to display our game. We will provide you with a lot of the necessary methods to get this method working, but it is your job to weave in the logic required to tie all the pieces together.

- Overview of the method:
    - Every round, both `animal1` and `animal2` will call their `attack()` method to generate a random `double` and deal damage to the other animal.
    - Once one of the animal's health is less than or equal to 0, then that animal has died and the other one is the winner.
    - The winner gets to eat something and will call their respective method.
        - `Carnivore`s will call their `eatAnimal()` method on the losing animal because they get to eat their prey.
        - `Herbivore`s will call their `eatPlant()` method because they did not die.
        - **Hint**: `NAME` can be useful for checking if the animal is a `Carnivore` or an `Herbivore`
    - At the start of a fight, `Poisonous` animals will call their `poisonAnimal()` method, such that in case they lose the fight, the other animal will also die and it will be a "tie".
    - If both animals die, it is a **tie game**.
        - An animal has won, but has been poisoned as well.
        - Both animals do enough damage to each other, such that the `health` of both animals are less than or equal to 0 **in the same round**.
    - Return value:
        - Return 0 if it's a tie game (both died)
        - Return 1 if `animal1` won
        - Return 2 if `animal2` won

- **Full logical walkthrough of the method:**
    1. Check if either `animal1` and `animal2` are poisonous
        - If the animal is poisonous, invoke the respective `poisonAnimal()` method. Have some `boolean` to keep track if the other animal is successfully poisoned, which is when `poisonAnimal()` returns `true`. (Note: both animals could be poisonous!)

2. Keep track of the round number (starting at 0)
3. **While** the `health` of **both** animals are above 0:
    - `printRound(<your variable>)`
    - `printBothAnimals(animal1, animal2)`
    - `printAttack(LEFT, <animal1's attack()>)`
    - `printAttack(RIGHT, <animal2's attack()>)`
    - Increment round number

   *– At this point, one or both the animals have died and you **exit loop** –*
4. `printFinalStats(animal1, animal2, poisoned)`
5. Check if both animals died from attacking each other → `printTieGame()` → `return 0`
6. Check if `animal1` wins, if they won:
    - Check if `animal1` has been poisoned → `printTieGame()` → `return 0`
    - Invoke their respective eating method: (`eatAnimal()` or `eatPlant()`)
    - `printWinner(LEFT)`
    - `return 1`
7. Repeat step 6 but change for `animal2` (`printWinner(RIGHT)` and `return 2`)

Copy and paste the following necessary methods and constants into your `AnimalActivities` class to get the game working. Place the provided code at the bottom of your `AnimalActivities` class.

- `public static void printBothAnimals(Animal animal1, Animal animal2)`
- `public static int calcSpacing(String str)`
- `public static void printRound(int round)`
- `public static void printAttack(String side, double damage)`
- `public static void printFinalStats(Animal animal1, Animal animal2, boolean poisoned)`
- `public static void printTieGame()`
- `public static void printWinner(String side)`

```
/* Below are helper methods to make fight() work */

/**
```

```java
    * Use this method in fight() to display the stats of both animals together
    *
    * @param (animal1) Animal on the left side to display stats
    * @param (animal2) Animal on the right side to display stats
    */
   public static void printBothAnimals(Animal animal1, Animal animal2) {
      int ageSpacing = calcSpacing(Integer.toString(animal1.getAge()));
      int healthSpacing = calcSpacing(String.format("%.2f",animal1.getHealth()));
      int strSpacing = calcSpacing(String.format("%.2f",animal1.getStrength()));
      int animalSpacing = calcSpacing(animal1.getName());
      String str = String.format( "(%s) %s  (%s)\n" +
              "----------" + "            " + "----------\n" +
              "A: %d %s A: %d\n" +
              "H: %.2f %s H: %.2f\n" +
              "S: %.2f %s S: %.2f\n", animal1.getName(),
              " ".repeat(animalSpacing),animal2.getName(),
              animal1.getAge()," ".repeat(ageSpacing),animal2.getAge(),
              animal1.getHealth(), " ".repeat(healthSpacing), animal2.getHealth(),
              animal1.getStrength(), " ".repeat(strSpacing), animal2.getStrength()
              );
      System.out.println(str);
   }

   /**
    * Helper method for printBothAnimals()
    *
    * @param (str) String on the left - used to calculate spacing
    * @return An int describing how many spaces to put between strings
    */
   public static int calcSpacing(String str) {
       int totalWidth = SPACING;
       int str1Width = str.length();
       int spacing = (totalWidth - str1Width);
       if (spacing < 0) {
           return 0;
       }
       return spacing;
   }

   /**
    * Use this method in fight() to display the current round.
    * @param (round) An int of the round (should start at 0)
    */
   public static void printRound(int round) {
       System.out.println();
       System.out.println("Round " + round + ":");
   }
```

```java
/**
 * Use this method in fight() to display the damage each round.
 *
 * @param (side) The side of the Animal that invoked the attack().
 * @param (damage) The int (damage) returned from an attack() call
 */
public static void printAttack(String side, double damage) {
    System.out.printf("%s does %.2f damage!\n",side, damage);
}

/**
 * Use this method in fight() to display final stats and poison status.
 *
 * @param (animal1) Left animal
 * @param (animal2) Right animal
 * @param (poisoned) If either animal was poisoned
 */
public static void printFinalStats(Animal animal1, Animal animal2,
                                   boolean poisoned) {
    System.out.println();
    printBothAnimals(animal1, animal2);
    if (poisoned) {
        System.out.println("An animal was poisoned.");
    }
}

/**
 * Use this method in fight() to display a tie game.
 */
public static void printTieGame() {
    System.out.println("-------GAME OVER-------");
    System.out.println("TIE: Both animals died!");
}

/**
 * Use this method in fight() to display the winner.
 * @param (side) The side of the Animal that won.
 */
public static void printWinner(String side) {
    System.out.println("-------GAME OVER-------");
    System.out.println(side + " animal wins!");
}
```

Copy and paste the helpful constants needed for these methods to work properly (at the top).

```
    // Necessary constants
    private final static int NUM_ANIMALS = 6;
    private final static int SPACING = 17;
    private final static String LEFT = "Left";
    private final static String RIGHT = "Right";
```

Method 2 - `reproduce` (10 points)

The second method is `reproduce`. The method signature for it is:

`public static Animal reproduce(Animal animal1, Animal animal2)`

- Based on the two `Animal` objects being passed into this method, return a new `Animal` object (the baby) if the following conditions are met:
    - Both `Animal` objects are of age, such that `animal1` and `animal2` are both strictly older than **5**.
    - Both `Animal` objects are of the same species (Hint: we made a method for this)
- Follow these rules:
    - Based on the species of the `Animal` object, return an `Animal` with the same species as the input `Animal` objects.
        - For example, if the input `Animal` objects are **both** `Wolf`s, then return a new `Wolf`.
    - The baby being returned must have these default characteristics:
        - Age = 0
        - Health = 100
        - Strength = half of the average strength of both parents
- If either of the above conditions are not met, return `null`.

(OPTIONAL) Additional Fun (0 points)

If you would like to have some more fun, you can **copy and paste** this code to create **random** animals! This is **completely optional,** but in case you don't want to keep instantiating your own animals. You will have to define your own `MAX_AGE`, `MAX_AGE`, and `MAX_STRENGTH` constants to what you want them to be.

```
/* Below are helper methods to make a random Animal object*/
/**
 * Use this method to create a random Animal object of a random subclass.
 * @return random new Animal (Wolf, Leopard, Panda, Zebra, Toad, Cobra)
 */
public static Animal randomAnimal() {
        int randAge = randomAge(MAX_AGE);
        double randStrength = randomStrength(MAX_STRENGTH);
        int randClass = new Random().nextInt(NUM_ANIMALS);
        switch (randClass) {
            case 0: return (new Wolf(randAge, MAX_HP, randStrength));
            case 1: return (new Leopard(randAge, MAX_HP, randStrength));
            case 2: return (new Panda(randAge, MAX_HP, randStrength));
            case 3: return (new Zebra(randAge, MAX_HP, randStrength));
            case 4: return (new Toad(randAge, MAX_HP, randStrength));
            case 5: return (new Cobra(randAge, MAX_HP, randStrength));
            default: return null;
        }
}

/**
 * Use this method for randomAnimal()
 * @param (max) Max age acceptable
 * @return age
 */
public static int randomAge(int max) {
    int randAge = (int)(Math.random()*(max+1));
    return randAge;
}

/**
 * Use this method for randomAnimal()
 * @param (max) Max strength acceptable
 * @return strength
 */
public static double randomStrength(int max) {
    double randStrength = Math.random()*(max+1);
    return randStrength;
}
```

Part 6: Compile, Run and UnitTest Your Code (10 points)

Just like in previous assignments, **in this part of the assignment, you need to implement your own test cases in the method called `unitTests` in `Assignment8.java`. You might want to refer to previous assignments as a framework.**

Example: how you can setup and run the game in **your test file** and what it looks like.

```
291
292         Animal cobra = new Cobra(10, 100, 100);
293         Animal wolf = new Wolf(10, 100, 80);
294         int result = AnimalActivities.fight(cobra,wolf);
```

```
PROBLEMS 1   OUTPUT   TERMINAL   DEBUG CONSOLE

MacBook-Air-7:PA8 sarasafa$ java AnimalActivities

Round 0:
(Cobra)              (Wolf)
----------           ----------
A: 10                A: 10
H: 100.00            H: 100.00
S: 100.00            S: 80.00

Left does 72.16 damage!
Right does 74.47 damage!

Round 1:
(Cobra)              (Wolf)
----------           ----------
A: 10                A: 10
H: 25.53             H: 27.84
S: 100.00            S: 80.00

Left does 84.64 damage!
Right does 32.32 damage!

(Cobra)              (Wolf)
----------           ----------
A: 10                A: 10
H: -6.78             H: -56.80
S: 100.00            S: 80.00

An animal was poisoned.
-------GAME OVER-------
TIE: Both animals died!
```

In `Assignment8` you can unit test individual methods. You are encouraged to create as many test cases as you think to be necessary to cover all the edge cases. The `unitTests` method returns `true` only when all the test cases are passed. Otherwise, it returns `false`.

To get full credit for this section, you must create at least five test cases that cover different situations for these methods - `sleep()`, `eatAnimal()`, `poisonAnimal()`, `fight()`, and `reproduce()`. In other words, do whatever you need to do to make at least **one test for each of the methods above**.

Here are some potential ideas:

- `sleep()`
    - Make an animal, make it sleep, check the `strength`
- `eatAnimal()`
    - Make two animals, make one eat the other, check `strength`
- `poisonAnimal()`
    - Run this method several times, make sure the probability is always within range, check that it returns `true`/`false` accordingly.
- `reproduce()`
    - Make two animals, call `reproduce()`, check the return value.
- `fight()`
    - Set up a game where one of the animals WILL win (giving an animal very low strength), check the `int` you return.

Remember to use our `toString()` method to display your `Animal` objects. You can compile all the files present in the directory and run your unit tests from `main()` using the following commands: (Make sure you are in the correct directory, else navigate to the starter code using `cd`)

```
> javac *.java
> java Assignment8
```

Remember that writing unit tests will help you find bugs in your code and ensure that it is correct for different inputs.

Submission

You're almost there! Please follow the instructions below carefully and use the **exact submission format**. Because we will use scripts to grade, **you may receive a zero** if you do not follow the same submission format.

1. Open Gradescope and login. Then, select this course → Assignment 8.
2. Click the DRAG & DROP section and directly select the required files: `Assignment8.java`, `Animal.java`, `AnimalActivities.java`, `Carnivore.java`, `Herbivore.java`, `Wolf.java`, `Leopard.java`, `Panda.java`, `Zebra.java`, `Toad.java`, and `Cobra.java`. Drag & drop is fine. Do not submit a zip, just all the files in one Gradescope submission. Make sure the names of the files are correct.
3. You can resubmit unlimited times before the due date. Your score will depend on your final (most recent) submission, even if your former submissions have higher scores.
4. Your submission should look like the below screenshot. If you have any questions, feel free to post on Piazza!

Submit Programming Assignment

Upload all files for your submission

Submission Method

Upload

Add files via Drag & Drop or Browse Files.

Name	Size	Progress	
Zebra.java	**1.2** KB		
Panda.java	**1.2** KB		
Cobra.java	**1.8** KB		
Toad.java	**1.8** KB		
Leopard.java	**1.2** KB		
Wolf.java	**1.2** KB		
AnimalActivities.java	**12.3** KB		
Animal.java	**1.6** KB		
Assignment8.java	**0.7** KB		
Carnivore.java	**67** b		
Herbivore.java	**53** b		

Submitting For

Sara Mccoy

Cancel Upload