CSE 8B Spring 2023

Assignment 8

Interfaces

Due: Wednesday, June 7, 11:59 PM

Learning goals:

- Apply knowledge of Concrete Classes, Interfaces, and Inheritance by building an Animal Kingdom Game.
- Apply knowledge of UML diagrams to create files and methods.

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, you will receive an automatic zero on the assignment.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 8B Coding Style Guidelines. These guidelines can also be found on Canvas. Please ensure to have COMPLETE file headers, class headers, and method headers, to use descriptive variable names and proper indentation, and to 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 `cse8b` folder you created in Assignment 1 and create a new folder named `assignment8`.
3. This assignment does NOT have any starter code.
4. You can compile all files in the directory using the single command `javac *.java`.
5. 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.

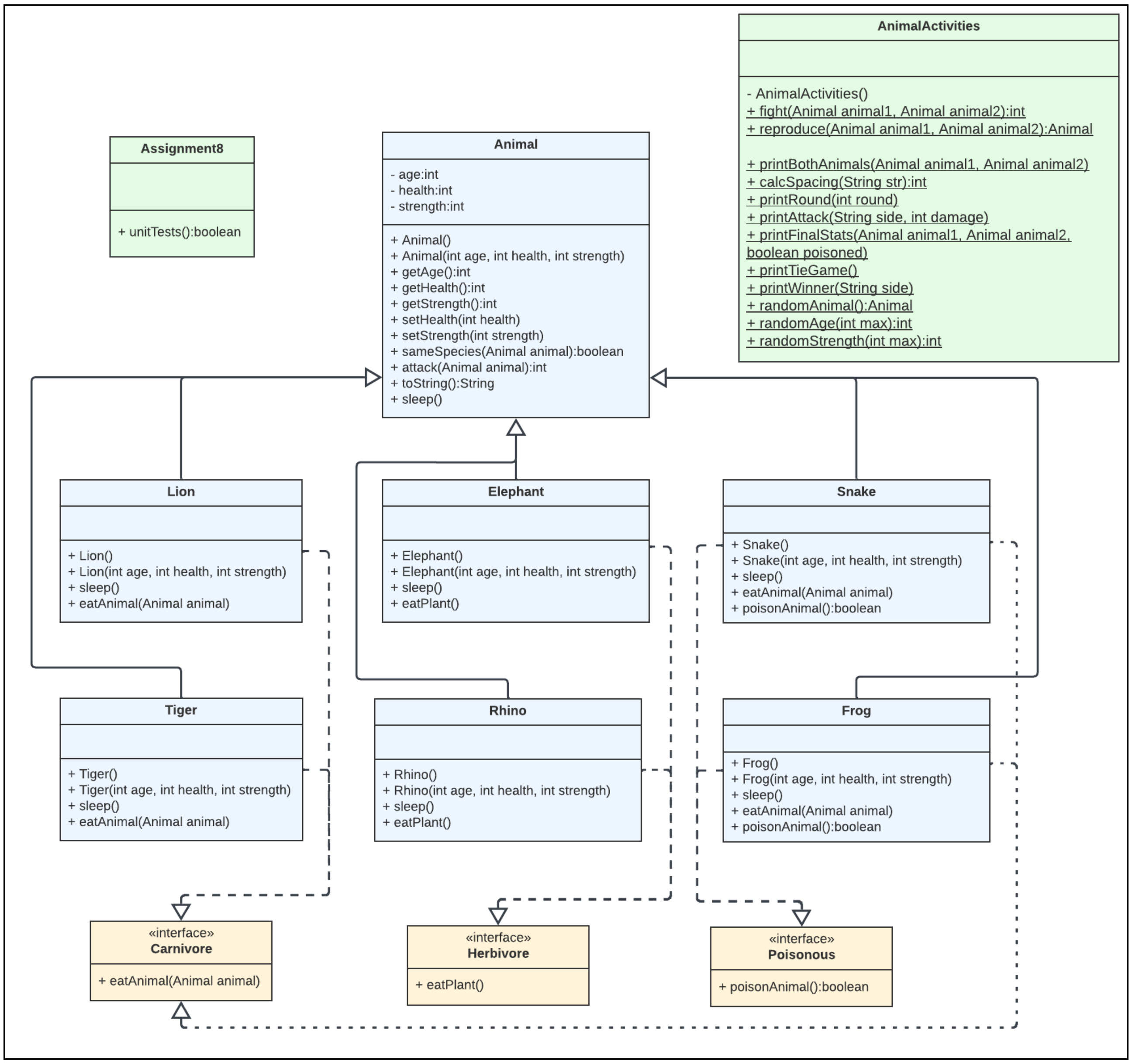
Part 1: Overview

Scenario:

CSE 8B 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 round out 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.


AnimalActivities
- AnimalActivities()
+ fight(Animal animal1, Animal animal2):int
+ reproduce(Animal animal1, Animal animal2):Animal
+ printBothAnimals(Animal animal1, Animal animal2)
+ calcSpacing(String str):int
+ printRound(int round)
+ printAttack(String side, int damage)
+ printFinalStats(Animal animal1, Animal animal2, boolean poisoned)
+ printTieGame()
+ printWinner(String side)
+ randomAnimal():Animal
+ randomAge(int max):int
+ randomStrength(int max):int
Assignment8
+ unitTests():boolean
Animal
- age:int
- health:int
- strength:int
+ Animal()
+ Animal(int age, int health, int strength)
+ getAge():int
+ getHealth():int
+ getStrength():int
+ setHealth(int health)
+ setStrength(int strength)
+ sameSpecies(Animal animal):boolean
+ attack(Animal animal):int
+ toString():String
+ sleep()
Lion
+ Lion()
+ Lion(int age, int health, int strength)
+ sleep()
+ eatAnimal(Animal animal)
Elephant
+ Elephant()
+ Elephant(int age, int health, int strength)
+ sleep()
+ eatPlant()
Snake
+ Snake()
+ Snake(int age, int health, int strength)
+ sleep()
+ eatAnimal(Animal animal)
+ poisonAnimal():boolean
Tiger
+ Tiger()
+ Tiger(int age, int health, int strength)
+ sleep()
+ eatAnimal(Animal animal)
Rhino
+ Rhino()
+ Rhino(int age, int health, int strength)
+ sleep()
+ eatPlant()
Frog
+ Frog()
+ Frog(int age, int health, int strength)
+ sleep()
+ eatAnimal(Animal animal)
+ poisonAnimal():boolean
«interface»
Carnivore
+ eatAnimal(Animal animal)
«interface»
Herbivore
+ eatPlant()
«interface»
Poisonous
+ poisonAnimal():boolean

In the UML diagram above, each rectangle in the UML diagram represents a class. There is an `Animal` superclass that multiple subclasses extend from: `Lion`, `Tiger`, `Elephant`, `Rhino`, `Snake`, and `Frog`. There are three interfaces: `Carnivore`, `Herbivore`, and `Poisonous`. 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 the write-up and to read 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 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 8B 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 `Store` 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 class named `Animal`. This is the superclass for most of the other classes in this assignment (as seen in the UML diagram). The `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 `Animal` class has three data fields:

1. **`private int age`**
2. **`private int health`**
3. **`private int strength`**

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

1. **`public Animal()`**
    - This no-arg constructor sets the instance variables of the object to these values:
        - age → 0
        - health → 0
        - strength → 0
2. **`public Animal(int age, int health, int 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

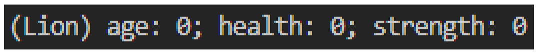
1. **`public int getAge()`**
2. **`public int getHealth()`**
3. **`public int getStrength()`**
4. **`public void setHealth(int health)`**
5. **`public void setStrength(int strength)`**

Then, implement the following methods:

1. **`public boolean sameSpecies(Animal animal)`**
    - This method must return `true` **only** when the current object (referring to this object - this entire writeup will use the same terminology for this) and the input

animal are of the same class. Otherwise, it must return `false`. (Hint: should use `getClass().getName()` to return the `String` of the class name, which means we don't have to continually override this method in the subclasses to check if two `Animal` objects are of the same species.

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



```
(Lion) age: 0; health: 0; strength: 0
```

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

Methods to be overridden by subclasses:

1. **public void sleep()**
   - Declare this method so it can be overridden by subclasses that extend from `Animal`. (Hint: We did this several times in Assignment 6).

Part 3: Interfaces (0 points)

Create the three `Interface` files and the methods that 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()`**

3. The `Poisonous` interface has one method:
   - **`public boolean poisonAnimal()`**

Part 4: Create some Animals (45 points)

`Lion`, `Tiger`, `Elephant`, `Rhino`, `Snake`, and `Frog` are all subclasses of `Animal`. **Complete all remaining constructors and methods in these classes.**

Part 4a: Strictly Carnivores (15/45 points)

Lion

Implement the following constructors and methods:

1. **`public Lion()`**
   - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
2. **`public Lion(int age, int health, int strength)`**
   - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
3. **`public void sleep()`**
   - When a `Lion` sleeps, they gain 20 `strength`. Add 20 to the `strength` field.
     - The "Kings of the Jungle" are ferocious when they're fully rested! 🦁

   (**Note:** Remember to use the override annotation (see lecture 12, slide 33).

4. **`public void eatAnimal(Animal animal)`**
    - When a `Lion` eats another animal, they gain **half of the** `strength` of the animal being eaten (in order to remain at the top of the food chain). Take the `strength` of the input `Animal`, divide it by 2 and add it to the current `Lion`’s `strength`. (**Note:** Remember to use the override annotation (see lecture 12, slide 33).

Tiger

Implement the following constructors and methods:

1. **`public Tiger()`**
    - This no-arg constructor calls `Animal`’s no-arg constructor (HINT: review Assignment 6).
2. **`public Tiger(int age, int health, int strength)`**
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
3. **`public void sleep()`**
    - When a `Tiger` sleeps, they gain 15 `strength`. Add 15 to the `strength` field. (**Note:** Remember to use the override annotation (see lecture 12, slide 33).
4. **`public void eatAnimal(Animal animal)`**
    - When a `Tiger` eats another animal, they gain **a third of the** `strength` of the animal being eaten (in order to remain at the top of the food chain). Take the `strength` of the input `Animal`, divide it by 3 and add it to the current `Tiger`’s `strength`.
        - *Fun fact*: Tigers are actually more aggressive and stronger than Lions, but for the purpose of this assignment, Lions will be at the top. 🐯

      (**Note:** Remember to use the override annotation (see lecture 12, slide 33).

Part 4b: Herbivores (15/45 points)

Elephant

Implement the following constructors and methods:

1. **`public Elephant()`**
    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
2. **`public Elephant(int age, int health, int strength)`**
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
3. **`public void sleep()`**
    - When an `Elephant` sleeps, they only gain 10 `strength`. Add 10 to the `strength` field.
        - *Fun fact*: Elephants don't sleep a lot (only 3 to 7 hours a night on average) because they need more time to eat! 🐘

    (**Note:** Remember to use the override annotation (see lecture 12, slide 33).
4. **`public void eatPlant()`**
    - An `Elephant` instance is an Herbivore, so they don't eat animals. Instead, they graze on some plants and will **randomly gain** between **0** (inclusive) and **40** (inclusive) `strength`. You may `import java.util.Random` for this method.

    (**Note:** Remember to use the override annotation (see lecture 12, slide 33).

Rhino

Implement the following constructors and methods:

1. **`public Rhino()`**
    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
2. **`public Rhino(int age, int health, int strength)`**
    - This constructor must set the `age`, `health`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
3. **`public void sleep()`**
    - When a `Rhino` sleeps, they gain 15 `strength`. Add 15 to the `strength` field.
        - *Fun fact*: Rhinos sleep on average 8 hours a day (must be nice right). 🦏

(**Note:** Remember to use the override annotation (see lecture 12, slide 33).

4. **`public void eatPlant()`**
    - An `Rhino` instance is an Herbivore, so they don't eat animals. Instead, they graze on some plants and will **randomly gain** between **0** (inclusive) and **25** (inclusive) `strength`. You may `import java.util.Random` for this method.
    (**Note:** Remember to use the override annotation (see lecture 12, slide 33).

Part 4c: Poisonous-Carnivores (15/45 points)

Snake

Implement the following constructors and methods:

1. **`public Snake()`**
    - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
2. **`public Snake(int age, int health, int strength)`**
    - This constructor must set the `age`, and `strength` in its superclass (HINT: use `super` to call the superclass constructor!).
3. **`public void sleep()`**
    - When a `Snake` sleeps, they gain 15 `strength`. Add 15 to the `strength` field.
4. **`public void eatAnimal(Animal animal)`**
    - When a `Snake` eats another animal, they gain **all of the** `strength` of the animal being eaten. Take the `strength` of the input `Animal` and add it to the current `Snake`'s `strength`.
        - Snakes swallow their prey whole, so they gain all the `strength`. 🐍
5. **`public boolean poisonAnimal()`**
    - A `Snake` has a 40% chance to poison the other animal. If that animal is poisoned, they will die at the end of the round even if they win the battle. If the `Snake` wins the battle it doesn't matter (this will be implemented later). Basically, return `true` 40% of the time.
        - (Implementation Hint: The **nextDouble()** method from the **Random** package 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.)

Frog

Implement the following constructors and methods:

1. **`public Frog()`**
   - This no-arg constructor calls `Animal`'s no-arg constructor (HINT: review Assignment 6).
2. **`public Frog(int age, int health, int strength)`**
   - This constructor must set the `age`, and `strength` in its superclass (HINT: use [`super`]() to call the superclass constructor!).
3. **`public void sleep()`**
   - When a `Frog` sleeps, they gain 10 `strength`. Add 10 to the `strength` field.
4. **`public void eatAnimal(Animal animal)`**
   - A `Frog` can't really eat any of the animals here… so they'll eat a bug instead. If the `Frog` defeats the input `Animal`, then they will eat one of the bugs swarming the dead carcass. The bug will be worth **a quarter of the** `strength` of the input `Animal`. Take the `strength` of the input `Animal`, divide it by 4 and add it to the current `Frog`'s `strength`.
     - *Fun fact*: Poison frogs are the most brightly colored frogs in the world. 🐸
5. **`public boolean poisonAnimal()`**
   - A `Frog` has a 20% chance to poison the other animal. If that animal is poisoned, they will die at the end of the round even if they win the battle. If the `Frog` wins the battle it doesn't matter (this will be implemented later). Basically, return `true` 20% of the time.
     - (Implementation Hint: The **nextDouble()** method from the **Random** package 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.)

Part 5: AnimalActivities.java (25 points)

Finally, the cool part! You will be implementing an `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 worry about implementing the two methods above the space in the `AnimalActivities` class, the rest will be provided for you.

First, implement the AnimalActivities constructor:

1. **`private AnimalActivities()`**
    - Our `AnimalActivities` class will primarily be two static methods. Therefore, to prevent instantiation of a class with no instance methods, 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 `int` and deal damage to the other `animal`.
    - Once one of the animals' health hits 0 or goes below 0 (`<= 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.
        - **Carnivores**: will call their `eatAnimal()` method on the losing animal because they get to eat their prey.
        - **Herbivores**: will call their `eatPlant()` method because they did not die.
    - 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. They can still poison the other animal and win the fight.

    - 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:
        - 0 → tie game (both died)
        - 1 → `animal1` won
        - 2 → `animal2` won

- **Full logical walkthrough of the method:**
    1. Check if either `animal1` and `animal2` are poisonous (**instanceof** Frog or Snake)
        - Invoke the respective `poisonAnimal()` method. Have some `booleans` 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`)

REMEMBER: When you use **instanceof** to check the subclass of the Animal, you must **explicitly cast** the animal to that subclass in order to invoke its corresponding method.

These are the necessary methods you need to **copy and paste** into your `AnimalActivities` class, to get the game working. We will provide as much detail as possible. You will need to **copy and paste** the code we provide for you below at the bottom of your `AnimalActivities` class. You will also need to **copy and paste** the constants for these methods.

1. **`public static void printBothAnimals(Animal animal1, Animal animal2)`**
2. **`public static int calcSpacing(String str)`**
3. **`public static void printRound(int round)`**
4. **`public static void printAttack(String side, int damage)`**
5. **`public static void printFinalStats(Animal animal1, Animal animal2, boolean poisoned)`**
6. **`public static void printTieGame()`**
7. **`public static void printWinner(String side)`**

Copy and paste these methods into your `AnimalActivities` class (preferably at the bottom). You may have to reformat and provide proper indentation, refer back to the write-up if needed.

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

    /**
     * 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(Integer.toString(animal1.getHealth()));
        int strSpacing = calcSpacing(Integer.toString(animal1.getStrength()));
        int animalSpacing = calcSpacing(animal1.getClass().getName());
        String str = "(" + animal1.getClass().getName() + ")" +
                " ".repeat(animalSpacing) + "(" +
                animal2.getClass().getName() + ")\n" +
                "----------" + "         " + "----------\n" +
                "A: " + animal1.getAge() + " ".repeat(ageSpacing) +
                "A: " + animal2.getAge() + "\n" +
                "H: " + animal1.getHealth() + " ".repeat(healthSpacing) +
                "H: " + animal2.getHealth() + "\n" +
                "S: " + animal1.getStrength() + " ".repeat(strSpacing) +
                "S: " + animal2.getStrength() + "\n";
        System.out.println(str);
    }
```

```java
/**
 * 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 + ":");
}

/**
 * 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, int damage) {
    System.out.println(side + " does " + damage + " 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);
```

```java
        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).

```java
    // 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 **instanceof** the `Animal` object, return an `Animal` with the **same declared type** as the input `Animal` objects.
        - For example, if the input `Animal` objects are **both** Lions, then return a new Lion().
    - The baby being returned should have these default characteristics:
        - Age = 0
        - Health = 100
        - Strength = half of the average strength of both parents
- If none of the above conditions are met, return a new `Animal()` object of the superclass type, using the no-arg constructor.

(OPTIONAL) Additional Fun (0 points)

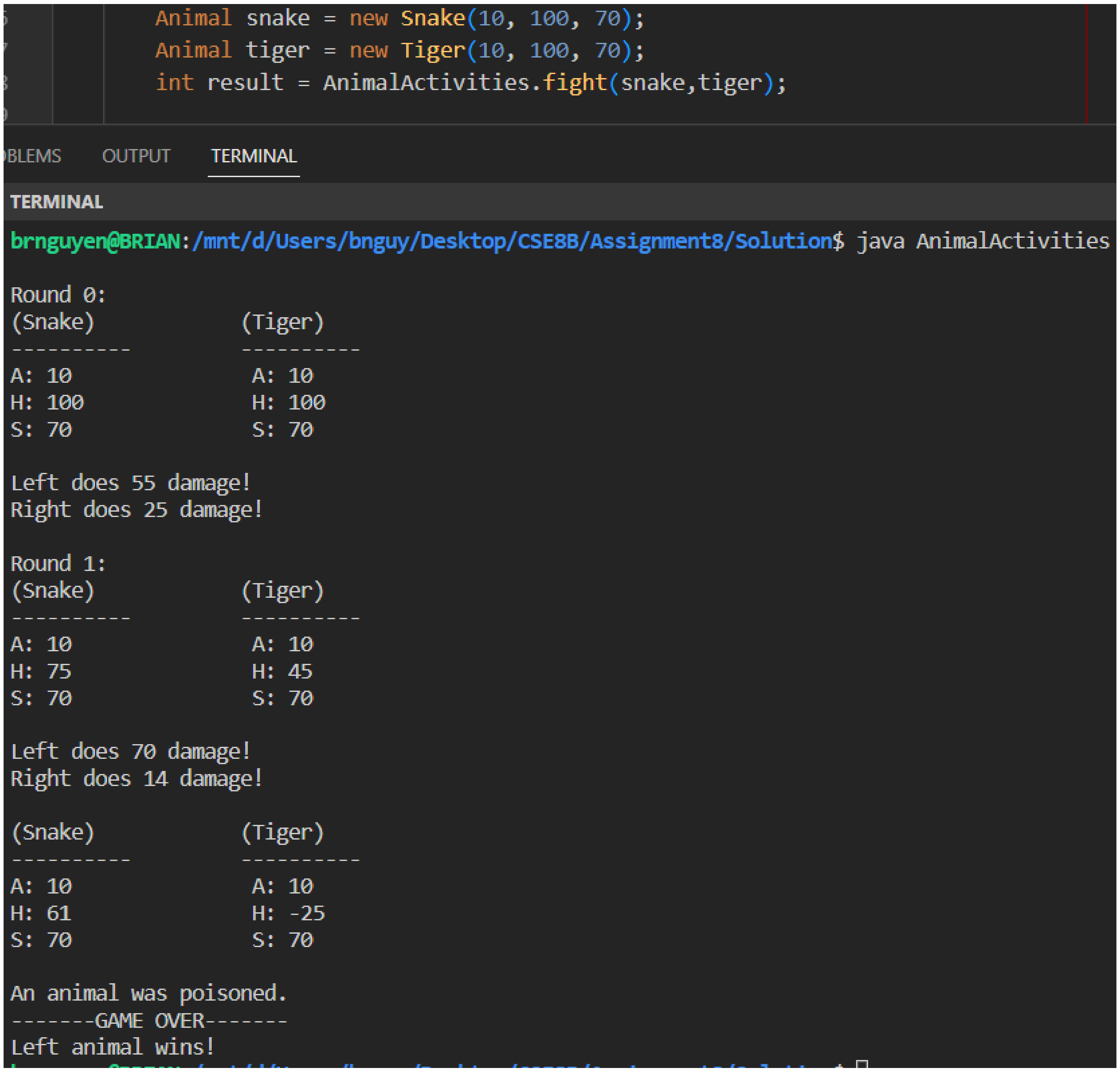
If you would like to have some more fun, you can **copy and paste** this code to create **random** animals! This is **completely optional,** but in case you don't want to keep instantiating your own animals. You will have to define your own `MAX_AGE`, `MAX_AGE`, and `MAX_STRENGTH` constants to what you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

```java
/* 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 (Lion, Tiger, Elephant, Rhino, Snake, Frog)
 */
public static Animal randomAnimal() {
        int randAge = randomAge(MAX_AGE);
        int randStrength = randomStrength(MAX_STRENGTH);
        int randClass = new Random().nextInt(NUM_ANIMALS);
        switch (randClass) {
            case 0: return (new Lion(randAge, MAX_HP, randStrength));
            case 1: return (new Tiger(randAge, MAX_HP, randStrength));
            case 2: return (new Elephant(randAge, MAX_HP, randStrength));
            case 3: return (new Rhino(randAge, MAX_HP, randStrength));
            case 4: return (new Snake(randAge, MAX_HP, randStrength));
            case 5: return (new Frog(randAge, MAX_HP, randStrength));
            default: return new Animal();
        }
}

/**
 * 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 int randomStrength(int max) {
    int randStrength = (int)(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 set-up and run the game in **your test file** and what it looks like.



```java
        Animal snake = new Snake(10, 100, 70);
        Animal tiger = new Tiger(10, 100, 70);
        int result = AnimalActivities.fight(snake,tiger);
```

```
brnguyen@BRIAN:/mnt/d/Users/bnguy/Desktop/CSE8B/Assignment8/Solution$ java AnimalActivities

Round 0:
(Snake)             (Tiger)
----------          ----------
A: 10                A: 10
H: 100               H: 100
S: 70                S: 70

Left does 55 damage!
Right does 25 damage!

Round 1:
(Snake)             (Tiger)
----------          ----------
A: 10                A: 10
H: 75                H: 45
S: 70                S: 70

Left does 70 damage!
Right does 14 damage!

(Snake)             (Tiger)
----------          ----------
A: 10                A: 10
H: 61                H: -25
S: 70                S: 70

An animal was poisoned.
-------GAME OVER-------
Left animal wins!
```

The Snake poisoned the Tiger, but the Snake won anyways!

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()`**,** `encounter()`**, and** `reproduce()`. In other words, do whatever you need to do to make at least **one test for each of the methods above** (it doesn't have to be a formal or well-written test case, just have a minimum of five test cases in `Assignment8.java` that should help test the correctness for full points).

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.
- `fight()`
  - Set up a game where one of the animals WILL win (giving an animal very low strength), check the `int` you return.
- `reproduce()`
  - Make two animals, call `reproduce()` , check the return value.

If you want to test specific methods, you can just instantiate animals to be the correct subclass to avoid needing to cast everytime. For example, just declare a `new Lion()` to test Lion's eatAnimal() method. 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: <u>(Make sure you are in the correct directory, else navigate to the starter code using `cd`)</u>

```
> 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.

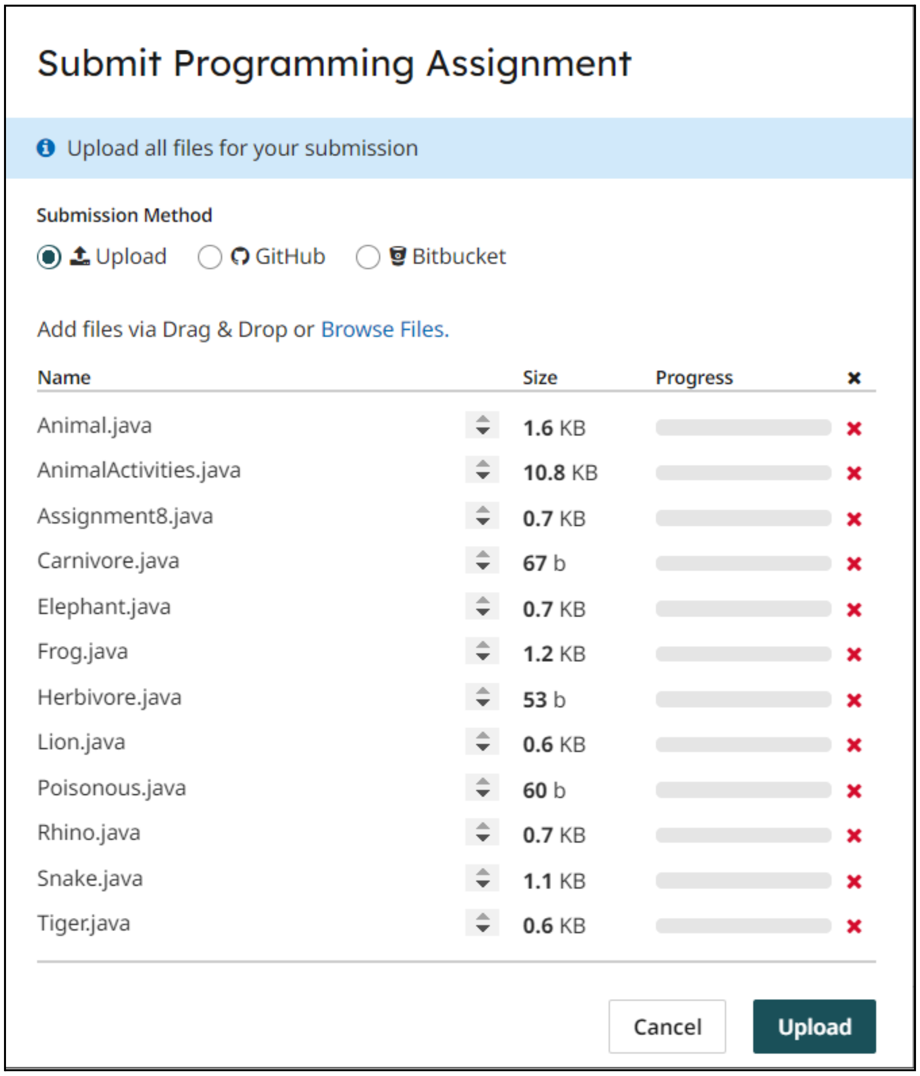
Submission

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

1. Open Gradescope and login. Then, select this course → Assignment 6.
2. Click the DRAG & DROP section and directly select the required files: `Assignment8.java`, `Animal.java`, `AnimalActivities.java`, `Carnivore.java`,

`Herbivore.java`, `Poisonous.java`, `Lion.java`, `Tiger.java`, `Elephant.java`, `Rhino.java`, `Snake.java`, and `Frog.java`. Drag & drop is fine. Do not submit a zip, just the nine 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] GitHub Bitbucket

Add files via Drag & Drop or Browse Files.

Name	Size	Progress
Animal.java	1.6 KB	
AnimalActivities.java	10.8 KB	
Assignment8.java	0.7 KB	
Carnivore.java	67 b	
Elephant.java	0.7 KB	
Frog.java	1.2 KB	
Herbivore.java	53 b	
Lion.java	0.6 KB	
Poisonous.java	60 b	
Rhino.java	0.7 KB	
Snake.java	1.1 KB	
Tiger.java	0.6 KB	