Homework 04 - Rare Frogs and Speedy Flies

Problem Description

Hello! Please make sure to read all parts of this document carefully.

In this assignment, you will be applying your knowledge of constructor chaining, the `this` keyword, encapsulation, method overloading, visibility modifiers, the toString method, and setters and getters. As we get further into the course, we are starting to learn more elements of Object-Oriented Programming. You will create a **Frog.java, Fly.java,** and **Pond.java** file that will simulate how they interact with each other in the real world.

Remember to test for Checkstyle errors and include Javadoc Comments!

Solution Description

Create files `Fly.java`, `Frog.java`, and `Pond.java` that will simulate how a pond ecosystem works. You will be creating a number of fields and methods for each file. Based on the description given for each variable and method, you will have to decide whether or not the variables/method should be static, and whether it should be private or public. To make these decisions, you should carefully follow the guidelines on these keywords as taught in lecture. In some cases, your program will still function with an incorrect keyword.

`Fly.java`

This Java file defines fly objects that exists within the pond.

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint**: there is a specific visibility modifier that can do this!

The `Fly` class must have these variables.

- `mass` - the mass of the `Fly` in grams (it must allow decimal values)
- `speed` - how quickly this `Fly` can maneuver through the air while flying, represented as a `double`

Constructors

You **must** use constructor chaining in at least two of your constructors. Duplicate code cannot exist in multiple constructors.

- A constructor that takes in `mass` and `speed` of a `Fly`.
- A constructor that takes in only `mass`.
    - By default, the `Fly` will have 10 `speed`.
- A constructor that takes in no parameters.
    - By default, the `Fly` will have 5 `mass` and 10 `speed`.

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- Setters and getters (using appropriately named methods) for all variables in `Fly.java`.

- `toString` - takes in no parameters and returns a String describing the `Fly` as follows:
  (Note: replace the values in brackets [] with the actual value and do not include the double quotes “” in the ouput. Specify all decimal values to 2 decimal points.)
  - If `mass` is 0: “I’m dead, but I used to be a fly with a speed of [`speed`].”
  - Otherwise: “I’m a speedy fly with [`speed`] speed and [`mass`] mass.”
- `grow` - takes in an integer parameter representing the added `mass`. Then it increases the `mass` of the `Fly` by the given number of `mass.` As `mass` increases, `speed` changes as follows:
  - If `mass` is less than 20: increases `speed` by 1 for every `mass` the `Fly` grows until it reaches 20 `mass`.
  - If the `mass` is 20 or more: decreases `speed` by 0.5 for each mass unit added over 20.
- `isDead` – if `mass` is zero, return true. Otherwise, return false.

`Frog.java`

This Java file defines frog objects that exist within the pond.

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint**: there is a specific visibility modifier that can do this!

The `Frog` class must have these variables:

- `name` - the name of this `Frog`, which can be made of any combination of characters
- `age` - the age of the `Frog` as an integer number of months
- `tongueSpeed` - how quickly this `Frog`’s tongue can shoot out of its mouth, represented as a `double`
- `isFroglet` - a value that represents if this `Frog` is young enough to be a froglet (the stage between tadpole and adult frog), which must only have two possible values - `true` or `false`. A `Frog` is a froglet if it is more than 1 month old but fewer than 7 months old. Whenever `age` is changed, this variable must be updated accordingly.
- `species` - the name of the species of this `Frog`, which must be the same for all instances of `Frog` (Hint: there is a keyword you can use to accomplish this). By default, its value must be “Rare Pepe”.

Constructors

You **must** use constructor chaining in at least two of your constructors. Duplicate code cannot exist in multiple constructors.

- A constructor that takes in `name`, `age`, and `tongueSpeed` and sets all variables appropriately.
- A constructor that takes in `name`, `ageInYears` representing the age of the `Frog` in years as a `double`, and `tongueSpeed` and sets all variables appropriately.
  - When converting `ageInYears` to `age` (in an integer number of months), round down to the nearest month without using any method calls (Hint: Java can automatically do this for you with casting).
- A constructor that takes in just a `name`.
  - By default, a `Frog` is 5 months old and has a `tongueSpeed` of 5.

Methods

You **must** use method overloading at least once. Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- `grow` - takes in a whole number parameter representing the number of months.
    - Then it ages the `Frog` by the given number of months and increases `tongueSpeed` by 1 for every month the `Frog` grows until it becomes 12 months old.
    - If the `Frog` is 30 months old or more, then decrease `tongueSpeed` by 1 for every month that it ages beyond 30 months.
    - You must not decrease `tongueSpeed` to less than 5.
    - Remember to update `isFroglet` accordingly
- `grow` - takes in no parameters and ages the `Frog` by one month and updates `tongueSpeed` accordingly as for the other `grow` method
- `eat` – takes in a parameter of a `Fly` to attempt to catch and eat.
    - Check if `Fly` is dead, and if it is dead then terminate the method.
    - The `Fly` is caught if `tongueSpeed` is greater than the `speed` of the `Fly`.
    - If the `Fly` is caught and the `mass` is at least 0.5 times the `age` of the `Frog`, the `Frog` ages by one month using the method `grow`. If the `Fly` is caught, the `mass` of the `Fly` must be set to 0.
    - If the `Fly` is NOT caught, the `mass` of the `Fly` must be increased by 1 while updating the `speed` of the `Fly` using only one `Fly` method.
- `toString` - returns a String describing the `Frog` as follows:
  (Note: replace the values in brackets [] with the actual value and do not include the double quotes "" in the ouput. Specify all decimal values to 2 decimal points.)
    - If frog is a froglet, returns "My name is [`name`] and I'm a rare froglet! I'm [`age`] months old and my tongue has a speed of [tongueSpeed]."
    - Otherwise, returns "My name is [`name`] and I'm a rare frog. I'm [`age`] months old and my tongue has a speed of [`tongueSpeed`]."
- Setter and getter for `species` which must change the value for all instances of the class. Points will be deducted if you include an unnecessary or inappropriate setter/getter.

Pond.java

This Java file is a driver, meaning it will contain and run `Frog` and `Fly` objects and "drive" their values according to a simulated set of actions. You can also use it to test your code. We require the following in your Pond class:

Methods

- Main method must act as such:
    - Must create at least 4 `Frog` objects:
        - `Frog` with `name` Peepo.
        - `Frog` with `name` Pepe, `age` 10 months, and `tongueSpeed` of 15.
        - `Frog` with `name` Peepaw, `age` 4.6 years, and `tongueSpeed` of 5.
        - `Frog` of your liking :)
    - Must create at least 3 `Fly` objects:
        - `Fly` with 1 mass and speed of 3.
        - `Fly` with 6 mass.
        - `Fly` of your liking :)
    - Perform the following operations in this order:

1. Set the `species` of any `Frog` to “1331 Frogs”
2. Print out on a new line the description of the `Frog` named Peepo given by the `toString` method.
3. Have the `Frog` named Peepo attempt to `eat` the `Fly` with a `mass` of 6.
4. Print out on a new line the description of the `Fly` with a `mass` of 6 given by the `toString` method.
5. Have the `Frog` named Peepo `grow` by 8 months.
6. Have the `Frog` named Peepo attempt to `eat` the `Fly` with a `mass` of 6.
7. Print out on a new line the description of the `Fly` with a `mass` of 6 given by the `toString` method.
8. Print out on a new line the description of the `Frog` named Peepo given by the `toString` method.
9. Print out on a new line the description of your own `Frog` given by the `toString` method.
10. Have the `Frog` named Peepaw `grow` by 4 months.
11. Print out on a new line the description of the `Frog` named Peepaw given by the `toString` method.
12. Print out on a new line the description of the `Frog` named Pepe given by the `toString` method.

Rubric

[45] `Fly.java`

- [4] Correct variables `mass` and `speed` with proper visibility, encapsulation, and data types
- [10] Correctly creates constructors with constructor chaining
    - [3] Constructor with params mass and speed
    - [3] Constructor with params mass
    - [4] Constructor with no params
- [12] Correctly implements setters and getters
    - [3] setter method for `speed`
    - [3] setter method for `mass`
    - [3] getter method for `speed`
    - [3] getter method for `mass`
- [5] Correctly implements `toString` method
- [10] Correctly implements `grow` method
- [4] Correctly implements `isDead`

[45] `Frog.java`

- [10] Correct variables with proper visibility, encapsulation, and data types
    - [2] `name`
    - [2] `age`
    - [2] `tongueSpeed`
    - [1] `isFroglet`
    - [3] `species`
- [10] Correctly implements `grow` methods
    - [8] `grow` method with one parameter
    - [2] `grow` method with no parameters
- [10] Correctly implements `eat` method

- [10] Correctly implements `toString` method
- [5] Correctly implements setters and getters for `species`

[10] `Pond.java`

- [4] Correct `Frog` objects
- [3] Correct `Fly` objects
- [3] Correct testing of the objects

We reserve the right to adjust the rubric, but this is typically only done for correcting mistakes.

Allowed Imports

To prevent trivialization of the assignment, you are *not* allowed to import any classes or packages.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`

Javadoc

For this assignment, you will be commenting your code with Javadoc. Javadoc comments are a clean and useful way to document your code's functionality. For more information on what Javadoc comments are and why they are awesome, the online overview for them is extremely detailed and helpful.

You can generate the Javadoc overview for your code using the command below, which will put all the files into a folder named "javadoc". Note you should execute this after adding Javadoc comments to your code:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to include are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 *This class represents a Dog object.
 *@author George P.Burdell
 *@version 1.0
 */
 public class Dog {

    /**
     *Creates an awesome dog(NOT a dawg!)
     */
     public Dog() {
     ...
    }

    /**
     *This method takes in two ints and returns their sum
     *@param a first number
     *@param b second number
     *@return sum of a and b
     */
     public int add(int a,int b) {
     ...
     }
}
```

A more thorough tutorial for Javadoc Comments can be found here.

Take note of a few things:

1. Javadoc comments begin with `/**` and end with `*/`.
2. Every class you write must be Javadoc'd and the @author and @verion tag included. The comments for a class should start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an @param tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

Checkstyle can check for Javadoc comments using the -a flag, as described in the next section.

Checkstyle

For this assignment, we will be enforcing style guidelines with Checkstyle, a program we use to check Java style. Checkstyle can be downloaded here.

To run Checkstyle, put the jar file in the same folder as your homework files and run

```
java -jar checkstyle-6.2.2.jar -a *.java
```

The Checkstyle cap for this assignment is **25 points**. This means that up to 25 points can be lost from Checkstyle errors.

Collaboration

Collaboration Statement

To ensure that you acknowledge a collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one .java file that you submit**. That collaboration statement should say either:

I worked on the homework assignment alone, using only course materials.

or

In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].

Allowed Collaboration

When completing homeworks for CS1331 you may talk with other students about:

- What general strategies or algorithms you used to solve problems in the homeworks
- Parts of the homework you are unsure of and need more explanation
- Online resources that helped you find a solution
- Key course concepts and Java language features used in your solution

You may **not** discuss, show, or share by other means the specifics of your code, including screenshots, file sharing, or showing someone else the code on your computer, or use code shared by others.

Examples of approved/disapproved collaboration:

- **approved**: "Hey, I'm really confused on how we are supposed to implement this part of the homework. What strategies/resources did you use to solve it?"
- **disapproved**: "Hey, it's 10:40 on Thursday... Can I see your code? I won't copy it directly I promise"

In addition to the above rules, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- Frog.java
- Fly.java
- Pond.java

Make sure you see the message stating "HW04 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your last submission: be sure to **submit every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications