Homework 07 – Vet Visit

Problem Description

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

Welcome to the 1331 Veterinarian! We have lots of pets that get treated at our clinic! To complete this assignment, you will use your knowledge of class hierarchies, interfaces, and polymorphism.

Remember to test for Checkstyle Errors and include Javadoc Comments!

Solution Description

You will create one abstract Pet class, three concrete subclasses for the types of Pets, one interface that will guarantee the Object is Treatable, and a Vet class that will interact with them. 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 the modules.

Hint: A lot of the code you will write for this assignment can be reused. Try to think of what keywords you can use that will help you! You should be able to put @Override on the line before the method header for any methods you override.

`Pet.java`

This class represents a generic Pet. You must not be able to create an instance of this class (Hint: there is a keyword that prevents us from creating instances of a class).

Variables

The visibility of these variables can be any that support encapsulation!

- `String name` - the name of the pet
- `int age` - the age of a pet ranging from 1 – 100 **Any time the age would be set to a value of out of bounds, set it to the closest bound** (I.e. -2 would be set to 1 and 120 would be set to 100)
- `int painLevel` – the level of pain the pet is in ranging from 1 – 10. **Any time the pain level would be set to a value of out of bounds, set it to the closest bound** (I.e. -2 would be set to 1 and 20 would be set to 10)

Constructors

- A constructor that takes in the name, age, and painLevel, and sets all fields accordingly. It must accept the variables in the specified order.

Methods

Do not create any other public methods than those specified. Any extra public methods will result in point deductions. However, private helper methods are allowed. All methods must have the proper visibility to be used where specified.

- `playWith(Pet p)`
    - Abstract method that represents this Pet playing with the other Pet
    - Any concrete class that extends the Pet class must provide a method definition
    - Does not return anything.
- `toString()`
    - Returns “My name is [name] and I am [age]. On a scale of one to ten my pain level is [painLevel].”
- `equals(Object o)`
    - Two Pets are equal if they have the same name, age, and painLevel.
    - Must override equals() method defined in Object class
- Include getters and setters when applicable.

`Treatable.java`

This is an interface that guarantees an Object is Treatable, or more specifically that they have a treat method.

Methods

Do not create any other public methods than those specified. Any extra public methods will result in point deductions. However, private helper methods are allowed. All methods must have the proper visibility to be used where specified.

- `convertDogToHumanYears(int dog_age)`
    - This static method returns the dog’s age in human years.
    - Human age = 16 * ln(dog_age) + 31 (Fun Fact - this is the actual formula to calculate a dog’s age in human years)
    - Returns the above number floored as an int
    - **Hint:** Look at Math.log()
- `convertCatToHumanYears(int cat_age)`
    - This static method returns the cat’s age in human years.
    - Human age = 9* ln(cat_age) + 18 (Disclaimer – this isn’t the actual formula to calculate a cat’s age in human years but is rather a made up one)
    - Returns the above number floored as an int
    - **Hint:** Look at Math.log()
- `treat()`
    - Implementations of this method should improve the condition of the Treatable object in some way
    - This method shouldn’t return anything

`Dog.java`

This class represents a Dog object. A Dog is a Pet and should contain all its attributes. A Dog is also Treatable.

Variables

The Dog class must have these variables. Similarly to Pet, any visibility is fine as long as it supports encapsulation. Do NOT re-declare any of the instance variables declared in the Pet class:

- String breed - the breed of the dog

Constructors

- A constructor that takes in the name, age, painLevel, and breed and sets all fields accordingly. It must accept the variables in the specified order.
    - Hint: There is a specified keyword in L12 to access the superclass's constructor.
- A constructor that takes in just breed and assigns the following default values:
    - name: Buzz
    - age: 6
    - painLevel: 3

Methods

Do not create any other public methods than those specified. Any extra public methods will result in point deductions. However, private helper methods are allowed. All methods must have the proper visibility to be used where specified.

- `playWith(Pet p)`
    - The Dog absolutely LOVES playing with other Dogs
        - If the Dog plays with another Dog decrease the Dog that called the method's painLevel by 3
        - Print out "Woof! I love playing with other dogs so much that my pain level went from [oldPainLevel] to [newPainLevel]"
    - The Dog isn't as enthusiastic about playing with Cats but still enjoys some Cats' company
        - If the Dog plays with a Cat without stripes decrease the Dog that called the method's painLevel by 1
            - Print out "Woof. Cats without stripes are okay since they made my pain level go from [oldPainLevel] to [newPainLevel]"
        - If the Dog plays with a Cat with stripes increase Dog that called the method's painLevel by 2
            - Print out "AHHH! I thought you were a tiger!"
    - The Dog CANNOT play with Narwhals
        - Don't print out anything or adjust the Dog's painLevel
- `treat()`
    - Reduces the pain level by 3.
    - This method does not return anything
- `bark()`
    - Prints out "bark bark"
    - This method does not return anything
- `toString()`
    - Returns "My name is [name], I am [age], and I am a [breed]. On a scale of one to ten my pain level is [painLevel]. My age in human years is [age in human years]."
    - Hint: Remember that we made a helpful converter method for dog to human years
- `equals(Object o)`
    - Two Dogs are equal if they have the same name, age, painLevel, and breed.
    - Must override equals() method defined in Pet class
- Include getters and setters when applicable.

`Cat.java`

This class represents a Cat object. A Cat is a Pet and should contain all its attributes. A Cat is also Treatable.

Variables

The Cat class must have these variables. As always, don’t violate encapsulation. Do NOT re-declare any of the instance variables declared in the Pet class:

- boolean hasStripes - Indicator for whether the cat has stripes or not

Constructors

- A constructor that takes in the name, age, painLevel, and hasStripes and sets all fields accordingly. It must accept the variables in the specified order.
    - Hint: There is a specified keyword in L12 to access the superclass’s constructor.
- A constructor that takes in just hasStripes and assigns the following default values:
    - name: "Purrfect”
    - age: 4
    - painLevel: 9

Methods

Do not create any other public methods than those specified. Any extra public methods will result in point deductions. However, private helper methods are allowed. All methods must have the proper visibility to be used where specified.

- `playWith(Pet p)`
    - The Cat absolutely LOVES playing with other Cats - Specifically, a cat with stripes loves playing with other cats with stripes and a cat without stripes loves playing with other cats without stripes
        - If a cat plays with another cat with the same pattern as them, decrease the Cat that called the method’s painLevel by 4
            - Print out “Meow! I love playing with other cats with the same pattern as me”
        - If a cat plays with another cat without the same pattern as them, decrease the Cat that called the method’s painLevel by 2
            - Print out “Meow! I like playing with other cats without the same pattern as me”
    - The Cat doesn’t like playing with Dogs at all
        - If the Cat plays with another Dog increase the Cat that called the method’s painLevel by 1
        - Print out “Meow. Go away [Dog’s name]! I don’t like playing with Dogs!”
    - The Cat CANNOT play with Narwhals
        - Don’t print out anything or adjust the Cat’s painLevel
- `treat()`
    - Reduce `painLevel` by 1

- `toString()`
  - Returns “My name is [name] and I am [age]. On a scale of one to ten my pain level is [painLevel]. My age in human years is [age in human years].”
  - Hint: Remember that we made a helpful converter method for cat to human years
- `equals(Object o)`
  - Two Cats are equal if they have the same name, age, painLevel and hasStripes.
  - Must override equals() method defined in Pet class
- Include getters and setters when applicable.

`Narwhal.java`

This class represents a Narwhal object. A Narwhal is a Pet and should contain all its attributes. Note that Narwhals are not Treatable since they cannot fit into a clinic!

Variables

- int hornLength – the length of the horn in feet

Constructors

- A constructor that takes in the name, age, painLevel, and hornLength and sets all fields accordingly. It must accept the variables in the specified order.
  - Hint: There is a specified keyword in L12 to access the superclass’s constructor.
- A constructor that takes in nothing and assigns the following default values:
  - name: "Jelly”
  - age: 19
  - painLevel: 2
  - hornLength: 7

Methods

- `playWith(Pet p)`
  - The Narwhal can only play with other Narwhals
    - If another Narwhal is passed in print “Who needs dogs and cats when we have each other”
      - Decrease the Narwhal that called the method’s painLevel by 2
    - If another Pet is passed in print “I live in the ocean so I can’t play with you”
      - Increase the Narwhal that called the method’s painLevel by 1
- `toString()`
  - Returns “My name is [name] and I am [age]. On a scale of one to ten my pain level is [painLevel]. I have a horn that is [hornLength] feet long.”
- `equals(Object o)`
  - Two Narwhals are equal if they have the same name, age, painLevel, and hornLength.
  - Must override equals() method defined in Pet class
- Include getters and setters when applicable.

`Vet.java`

This class represents a veterinarian clinic that can treat Pet objects. **It doesn't need to be instantiated since all the methods should be static.** (You can just leave the default no-args constructor)

Methods

Do not create any other public methods than those specified. Any extra public methods will result in point deductions. However, private helper methods are allowed. All methods must have the proper visibility to be used where specified.

- `inspectPet(Pet pet)`
  - Static method that prints the toString method of the pet
  - If the pet is a Dog, they will bark so call their bark method
  - Does not return anything
- `treatPet(Pet pet)`
  - Static method that accepts any Pet object.
  - If the Pet object is Treatable, print "Welcome to the vet [name]" and call their treat method
    - Afterwards, if the Pet is a Dog, print "Wow what a cute dog!" and call giveDogTreat
  - If the Pet is not Treatable, print "Sorry, we cannot treat [name]"
  - Does not return anything.
- `giveDogTreat(Dog dog)`
  - Static method that decreases the dog's pain by 2.
  - Does not return anything.

General Notes

- Reuse your code when possible. Certain methods can be reused using certain keywords.
- If you use the `@Override` tag for a method, you will not have to write the Javadocs for that method.
- As we progress further in the semester, you should get more familiar with testing your own code! Remember to compile it incrementally to make sure you are on the right track.
- Gradescope tests are not comprehensive.

Rubric

[10] `Pet.java`

- [1] Correct class header
- [1] Correct instance variable
- [3] Correctly creates constructors
- [5] Correctly implements specified methods

[20] `Dog.java`

- [2] Correct class header
- [2] Correct instance variables
- [4] Correctly creates constructors
- [12] Correctly implements specified methods

[20] `Cat.java`

- [2] Correct class header

- [2] Correct instance variables
- [4] Correctly creates constructors
- [12] Correctly implements specified methods

[13] `Treatable.java`

- [5] Correct interface header
- [8] Correctly implements specified methods

[15] `Narwhal.java`

- [2] Correct class header
- [2] Correct instance variables
- [4] Correctly creates constructors
- [7] Correctly implements specified methods

[22] `Vet.java`

- [2] Correct class header
- [20] Correctly implements specified methods

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 anything. You are *not* allowed to import any other classes or packages.

Feature Restriction

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`

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.

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.

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:

- Pet.java
- Dog.java
- Cat.java
- Treatable.java
- Narwhal.java
- Vet.java

Make sure you see the message stating "HW08 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