Homework 6: A Trip to the Vet

Problem Description:

Hello! Please make sure to read all parts of this document carefully!
In this assignment, you will be applying your knowledge of abstract classes, inheritance, polymorphism, file I/O and exceptions. For this homework, you will be simulating a veterinary clinic. You will create a **Pet.java**, **Dog.java**, **Cat.java**, **InvalidPetException.java**, and a **Clinic.java** file; the **Clinic.java** file will treat and keep a record of the `Pet` patients it receives!

Remember to check for checkstyle errors and add Javadoc comments!

Solution Description:

Create files `Pet.java`, `Dog.java`, `Cat.java`, `InvalidPetException.java` and `Clinic.java`. Each file will have instance fields, methods, and constructors.

`Pet.java`

This class represents any pet that would seek consultation from the clinic.

Variables:

- `String name`
- `double health`
  - A percentage value ranging from 0.0 to 1.0
- `int painLevel`
  - Ranges from 1 to 10

Constructor:

- `Pet(String name, double health, int painLevel)`
  - health
    - If health passed in is greater than 1.0, set health to 1.0
    - If health passed in is less than 0.0, set health to 0.0
  - painLevel
    - If painLevel passed in is greater than 10, set pain level to 10
    - If painLevel passed in is less than 1, set pain level to 1

Methods:

- getters for all instance fields, which should be camelCase with the variable name, e.g. a variable named `hello` should have a getter `getHello()`
- `int treat()`:
  - Should be an **abstract** method that returns the time taken (in minutes) to treat the pet
- `void speak()`:
  - This method prints "Hello! My name is " with the pet's name
  - If painLevel is greater than 5 prints the message in UPPERCASE
- `boolean equals(Object o)`:
  - Two `Pet` objects are equal if their names are the same
  - Note: You can assume you will not encounter two pets with the same name
- `heal()`:
  - Should be **protected** to prevent access by external classes
  - Sets health to 1.0
  - Sets painLevel to 1

Dog.java

Since a `Dog` is also a `Pet`, this class must inherit from parent class `Pet`. This class is concrete.

Variables:

- `double droolRate`

Constructors:

- `Dog(String name, double health, int painLevel, double droolRate)`
  - droolRate

`If droolRate is less than or equal to zero, set drool rate to 0.5`

- `Dog(String name, double health, int painLevel)`
  - Default droolRate is 5.0

Methods:

- getters for all instance fields, which should be camelCase with the variable name, e.g. a variable named hello should have a getter `getHello()`
- `int treat()`:
  - Should `heal()`
  - Returns the time taken (in minutes) to treat the pet. Round values up.
    - if droolRate is less than 3.5, the minutes for treatment is (painLevel*2)/health
    - if droolRate is in between 3.5 and 7.5 inclusive, the minutes for treatment is painLevel/health
    - If droolRate is greater than 7.5, the minutes for treatment is painLevel/(health*2)
- `void speak()`:
  - Calls parent method
  - Prints "bark" number of times of the painLevel
    - e.g.: if painLevel = 3
      - Prints "bark bark bark"
  - ALL UPPERCASE if painLevel is greater than 5, not inclusive
- `boolean equals(Object o)`:
  - Uses the `equals()` method in `Pet` as part of the decision-making with the additional condition of droolRate being the same

Cat.java

Since a `Cat` is also a `Pet`, this class must inherit from parent class `Pet`. This class is concrete.

Variables:

- `int miceCaught`

Constructors:

- `Cat(String name, double health, int painLevel, int miceCaught)`
  - miceCaught
    - If miceCaught passed in is less than 0, set miceCaught to 0
- `Cat(String name, double health, int painLevel)`
  - Default miceCaught is 0

Methods:

- getters for all instance fields, which should be camelCase with the variable name, e.g. a variable named hello should have a getter `getHello()`
- `int treat()`:
  - Should `heal()`
  - Returns the time taken (in minutes) to treat the pet. Round all values up.
    - if number of miceCaught is less than 4, the minutes for treatment is equal to (painLevel *2)/health
    - if miceCaught is in between 4 and 7 inclusive the minutes for treatment equals painLevel/health
    - If miceCaught is greater than 7, the minutes for treatment equals painLevel/(health*2)

- `void speak()`:
    - Calls parent method
    - Prints “meow” number of times of miceCaught
        - Eg: if miceCaught = 3
            - Print “meow meow meow”
    - ALL UPPERCASE if painLevel is greater than 5, not inclusive
- `boolean equals(Object o)`:
    - Uses the `equals()` method in `Pet` as part of the decision-making with the additional condition of miceCaught being the same

`InvalidPetException.java`

An **unchecked exception** with two constructors

Constructors

- `InvalidPetException()` has message “Your pet is invalid!”
- `InvalidPetException(String s)` has message s

`Clinic.java`

This is a class representing the vet clinic.

Variables

- `File patientFile`
    - File with patient information
- `int day`

Constructors

- `Clinic(File file)`
    - File that contains patient info - assign to `patientFile`
        - Name
        - Type of pet (includes pet info)
        - Appointment Info
            - timeIn(military time)
            - health(before Treatment)
            - painLevel(before Treatment)
            - TimeOut(military time)
        - See example file “Patients.csv” for the format.
    - `Day` initialized to 1
- `Clinic(String fileName)`
    - String includes filename extension – don't add “.csv”
    - Chains to the other constructor

Methods

- `String nextDay(File f) throws FileNotFoundException`
  `String nextDay(String fileName) throws FileNotFoundException`
    - Reads File f that contains the name, type of pet, and time of the appointments for the day
        - See example file “Appointments.csv” for the format
        - Eg: If there was a `Cat` Chloe, with a miceCaught count of 5, scheduled for 2:30 pm, Chloe’s information in Appointments.csv would look like:
          Chloe,Cat,5,1430
        - You will have one file for each different day
    - Use a Scanner object to take in user input
    - Print “Consultation for [name] the [typeOfPet] at [time].\nWhat is the health of [name]?\n”
    - If typeOfPet is not valid (i.e. not a `Dog` or `Cat`, case-sensitive) throw `InvalidPetException`

        - Do not catch the exception in your code! The caller of the method should handle the exception.
    - Take in user input for health
        - If input is not a number, continue prompting user until they provide a number
    - Print "On a scale of 1 to 10, how much pain is [name] in right now?\n"
    - Take in user input for painLevel
        - If input is not a number, continue prompting user until they provide a number
    - Call speak()
    - Treat pet
    - Calculate time out (there exists a method for this)
    - Note: Don't try to read the file and write to it at the same time – this method is intended only to read the file.
    - Don't forget the increment the day!
    - Returns a String with patient information to be used when treating patients and updating the file.
    - The string being returned should hold the updated information for all patients seen in the day separated by a newline character.
    - Each appointment should be formatted as follows:
        - [Name],[Species],[DroolRate/MiceCaught],[Day],[EntryTime],[ExitTime],[InitialHealth],[InitialPainLevel]
        - E.x.: If there are 2 appointments on day 2:
            - Appointment 1 on Day 2:
                - Dog Dobie with droolRate 2.7
                - Entry time: 1715 (5:15 pm) and Exit time: 1735 (5:35 pm)
                - Health was 0.5 and painLevel was 5 before treating
            - Appointment 2 on Day 2:
                - Cat Marlin with miceCaught 84
                - Entry time: 1655 (4:55 pm) and Exit time: 1700 (5:00 pm)
                - Health was 0.4 and painLevel was 4 before treating

        The output of nextDay would be:

```
Dobie,Dog,2.7,Day 2,1715,1735,0.5,5
Marlin,Cat,84,Day 2,1655, 1700,0.4,4
```

- `boolean addToFile(String patientInfo)`
    - Consumes a string representing a single appointment
        - Eg. In format:
          [Name],[Species],[DroolRate/MiceCaught],[Day],[EntryTime],[ExitTime],[InitialHealth],[InitialPainLevel]
    - Write info to patientFile
        - If old patient, only the **appointment info** should be added to the patient file, which includes:
            - Day #
            - Time in and time out
            - Health and pain
        - If new patient, all info should be added to the clinic's patient file
        - Assume the vet will never see two different pets with the same name
        - See Patients.csv for an example
    - Returns true if the appointment info was successfully written, and false if an error occurs or a checked exception is caught
    - Note (cont'd): Don't try to read the file and write to it at the same time – this method is intended to rewrite the file.
- `String addTime(String timeIn, int treatmentTime)`
    - This method should only be accessible in the Clinic class
    - This method should calculate the time the patient's appointment ends
    - Return timeOut
    - Remember: timeIn and timeOut should be represented in military time

- You can assume that timeIn and timeOut will **NOT** go across multiple days (ex. timeIn = “23:30” and timeOut = “00:30”)

Example Output

User input is **bolded**

Example output for this entry: Chloe,Cat,5,1430

Consultation for Chloe the Cat at 1430.

What is the health of Chloe?

0.6

On a scale of 1 to 10, how much pain is Chloe in right now?

Six

Please enter a number

On a scale of 1 to 10, how much pain is Chloe in right now?

6

HELLO! MY NAME IS CHLOE

MEOW MEOW MEOW MEOW MEOW

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. These tests and the ones on Gradescope are by no means comprehensive so be sure to create your own!

Rubric

[15] **`Pet.java`**

- [4] Correct constructors
- [3] Correct variables
- [3] treat Method
- [3] speak Method
- [2] equals Method

[20] **`Dog.java`**

- [4] Correct constructors
- [2] Correct variables
- [6] treat Method
- [6] speak Method
- [2] equals Method

[20] **`Cat.java`**

- [4] Correct constructors
- [2] Correct variables
- [6] treat Method
- [6] speak Method
- [2] equals Method

[40] **`Clinic.java`**

- [15] nextDay method
- [15] addToFile
- [3] addTime

- [3] Correct constructors and variables
- [4] Throws exception correctly

[5] **`InvalidPetException.java`**
- [2.5] Both constructors are correct
- [2.5] Exception is an unchecked exception

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 only allowed to import the following classes or packages.

```
java.util.Scanner;
```

```
java.io.File;
```

```
java.io.FileNotFoundException;
```

```
java.io.IOException;
```

```
java.io.PrintWriter;
```

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 **35 points**. This means that up to 35 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:

- Pet.java
- Dog.java
- Cat.java
- InvalidPetException.java
- Clinic.java

Make sure you see the message stating "HW06 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