Homework 07 - Puppy Shelter

Authors: Tushna Eduljee, Emma Barron, Ryan Miles, Manny Sonubi, Allan Nguyen, Shishir Bhat

Problem Description

You are walking down the street one day when you stumble across a puppy. Like, literally stumble. And so like the guilty good person you are, you take the now-injured puppy and take him to the nearby veterinarian, who directs you to a local puppy shelter. At the puppy shelter, you start chatting with the owner, and before you know it you've volunteered to write code for the puppy shelter! You will help the owner modernize his store by writing a Java program that allows him to track each puppy in his care.

Solution Description

For this assignment, create a file with the following name:

- `Puppy.java`

Puppy Class:

Instance Variables

- `String name` holds the name of the puppy. It should have private access.
- `int age` holds the age of the puppy. It should have private access.
- `int health` holds the health of the puppy. It should have private access.

Constructors: Be sure to use constructor chaining.

- `Puppy(String name, int age, int health)`
- `Puppy(String name)`
  - Assign the puppy a random age between 0 and 15, inclusive.
  - Assign the puppy a random health between 5 and 35, inclusive.
  - Again, make sure to use constructor chaining! HINT: Try creating a static Random field that you can use.

Methods:

- Correct public getters & setters for `name`, `age`, and `health`
- `String toString()` Returns "[name]: a puppy [age] years old with [health] health"
- `boolean canAdopt()` Returns true when the puppy can be adopted and false otherwise. A puppy can be adopted if its health is 50 or above.
- `void fetch()` Increases the puppy's health by 1 (because it got exercise)
- `void fetch(boolean inside)` The parameter `inside` signals whether or not you're playing fetch inside or outside. If inside, increase health by 5. If not inside, increase health by 10.
- `void fetch(int distance)` Distance represents how far the toy is thrown. Increase the puppy's healthy by distance divided by 10. Truncate decimals if necessary.

Testing your code:

If you want to test out your own code before submitting we recommend creating a separate java file and implementing a main method there. You can then create new animals and test your methods. Below is some sample testing code. **These tests are not comprehensive!**

```
Puppy pup = new Puppy("pup");
System.out.println("age: " + pup.getAge());
System.out.println("health: " + pup.getHealth());

pup.fetch(true);
System.out.println("Play fetch outside");

System.out.println("health: " + pup.getHealth());
```

This could print out

```
age: 15
health: 7
Play fetch outside
health: 12
```

Feel free to create your own tests in the `main` method! Try to write tests for the remaining methods in the file!

Rubric

- [100]
  - [5] Instance variables
  - [10] Constructor with three parameters assigns properly
  - [10] Constructor with one parameter assigns properly
  - [10] Creates random values correctly
  - [10] Constructor Chaining
  - [10] Setters & Getters correct for `name`, `age`, and `health`
  - [10] `toString()`
  - [5] `canAdopt()`
  - [10] `fetch()`
  - [10] `fetch(boolean)`
  - [10] `fetch(int)`

Allowed Imports

- To prevent trivialization of the assignment, you are only allowed to import the following class:
  - java.util.Random

Javadocs

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

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ 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 Javadocs can be found here. Take note of a few things:

1. Javadocs are begun with `/**` and ended 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 Javadocs 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 **50 points**. This means that up to 50 points can be lost from Checkstyle errors.

Collaboration Statement

To ensure that you acknowledge 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].

Recall that comments are special lines in Java that begin with `//`.

Submission

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

- `Puppy.java`

Make sure you see the message stating "HW07 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:

1. Prevent upload mistakes (e.g. forgetting checkstyle, non-compiling code)
2. 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.