Homework 6 - Game Design

Problem Description

After drinking a copious amount of coffee from your newly created machine, you feel inspired to design the world's next best-selling video game. First, you need to define a class structure for the different characters in the game.

Solution Description

The Character Class

We have provided the very basic abstract class `Character`, which you will use as your stepping-off point for the rest of your class structure. The `Character` class includes just three instance variables: `int health`, `int maxHealth`, and `String name`. You should implement getters and setters for these variables, keeping in mind what should be immutable (would it make sense for another character to change a character's name or max health?). One method is also defined, `abstract void interact(Character otherCharacter)`. Feel free to add any other instance variables or methods that you think may be helpful, but do not edit what is already given. For instance, if you think keeping track of some other kind of information, or defining some other behavior via a new method will be useful, you are free to define new instance variables or methods. Remember this class is abstract, meaning you shouldn't be able to create an instance of this class.

Your Implementation

You will be required to design and implement four other classes to achieve the specifications below. These classes are all concrete, meaning one should be able to create an instance of them.

Friendly Character Subclass

- Create a class called `Friendly` which is a subclass of `Character`.
- Implement a constructor to fully instantiate the object.
  - Take in a `String` representing the characters name.
  - This class should always have a max health of 10.
  - Upon creation `"Hi my name is [name]! I am friendly!"` should be printed.
- Override and implement the `interact` abstract method from `Character`
  - Whenever this class interacts, `"[name]: Hi [other character name]!"` should be printed out.
  - An instance of this class cannot interact with itself.

Unfriendly Character Subclass

- Create a class called `Unfriendly` which is a subclass of `Character`.
- Implement a constructor to fully instantiate the object.
  - Take in a `String` representing the characters name.
  - This class should always have a max health of 15.
  - Upon creation `"The name's [name]. I'm not friendly."` should be printed.
- Override and implement the `interact` abstract method from `Character`
  - Whenever this class interacts, `"[name]: What are you looking at [other character name]!?"` should be printed out.
  - An instance of this class cannot interact with itself.

Healer Character Subclass

- Create a class called `Healer` which is a subclass of `Friendly`.
- This class should be able to do everything a friendly class could, but it should also be able to heal other characters. How you choose to implement this healing function is up to you as long as it does the following:

  - A character that is at 0 health cannot be healed. `"[name]: I'm sorry [other character name]. Nothing is working!"` should be printed.
  - A character that is at their max health cannot be healed. `"[name]: Hey [other character name]. You look perfectly fine to me!"` should be printed.
  - A character that can be healed will have a health between 0 and their max health. They will be healed different amounts based on what subclass they are:
    * A friendly subclass should be healed for 4 health. Their health should be capped at their max health.
    * An unfriendly subclass should be healed for 2 health. Their health should be capped at their max health.
    * After healing, `"[name]: Got you up to [other character health] health. Hope you feel better [other character name]!"` should be printed.
  - A healing character cannot heal themselves.
- This class should heal other classes when interacting with them.
  - Don't forget to greet the other class first like a normal `Friendly`.

Attacker Character Subclass

- Create a class called `Attacker` which is a subclass of `Unfriendly`.
- This class should be able to do everything an unfriendly class could, but it should also be able to attack other characters. How you choose to implement this attacking function is up to you as long as it does the following:
  - A character that is at 0 health cannot be attacked. `"[name]: You show such weakness [other character name]."` should be printed.
  - A character that can be attacked will take different amounts of damage based on what subclass they are:
    * A friendly subclass should take 5 damage. Their health should not fall below 0.
    * An unfriendly subclass should take 3 damage. Their health should not fall below 0.
    * If the other character has above 0 health after being attacked, `"[name]: Now you're only at [other character health] health. How does it feel [other character name]!?"` should be printed.
    * If the other character is at 0 health, `"[name]: You were no match for me [other character name]."` should be printed.
  - An attacking character cannot attack themselves.
- This class should attack other classes when interacting with them.
  - Don't forget to greet the other class first like a normal `Unfriendly`.

General Guidelines

The open-ended nature of this homework is intended to give you more freedom as a programmer. In quite a few places, you have been told what to do, but not exactly how to do it. This leaves room for different implementations, some of which may be better than others. Specifically, the attacking and healing subclasses allow for the most variation. How you go about seeing if the passed in character is friendly or unfirendly is completely up to you, but try to make your code as modular as possible. Implementations that do not use `instanceof` and handle subclass recognition in a better manner will be awarded extra credit. Note that creating another instance variable in `Character` that explicitly holds the type of the object is not neccesarily considered a better manner. It can be argued that this is even worse.

Testing

We have provided the extremely basic `Driver.java`. It includes a few statements designed to guide you into testing what your different classes should do. However, we expect that you will create your own tests to achieve coverage of the unique combinations. Keep in mind that Gradescope tests are also not comprehensive - *test your code*!

Allowed Imports

To prevent trivialization of the assignment, you are not allowed to import anything.

Feature Restrictions

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

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

Rubric

[30] **Class Structure**

- [5] Correctly defined friendly class
- [5] Correctly defined unfriendly class
- [10] Correctly defined healing class
- [10] Correctly defined attacking class

[70] **Correct Functionality**

- [15] `Friendly`
  - [5] Correct constructor
  - [10] Correct interaction
- [15] `Unfriendly`
  - [5] Correct constructor
  - [10] Correct interaction
- [20] `Healer`
  - [5] Correct constructor
  - [15] Correct healing and interaction
- [20] `Attacker`
  - [5] Correct constructor
  - [15] Correct attacking and interaction

[5] **BONUS: Not using *instanceof* in any of your code.**

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.

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. Just like Checkstyle, **one point will be deducted for each Javadoc error detected by running checkstyle**. These deductions will also be limited by the checkstyle cap, which applies to the sum of checkstyle and javadoc errors.

Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **20** points. Review the style guide and download the checkstyle jar. Run checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.2.jar -a *.java
Audit done. Errors (potential points off):
0
```

Make sure to include the -a flag included for testing both checkstyle and javadocs

The message above means there were no Checkstyle or javadoc errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the checkstyle cap mentioned above). The Java source files we provide contain no Checkstyle errors. In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Depending on your editor, you might be able to change some settings to make it easier to write style-compliant code. See the customization tips page for more information.

Submission

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

- `Character.java`
- `Friendly.java`
- `Unfriendly.java`
- `Attacker.java`
- `Healer.java`

Make sure you see the message stating "HW## 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. non-compiling code)
2) Provide basic formatting and usage validation

In other words, the test cases on Gradescope are not 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 and run your code; you can do that locally on your machine.

Other portions of your assignment are also autograded once the submission deadline has passed. The autograders used are often dependent on specific output formats, so **make sure that your submission passes all test cases marked "FORMAT:"**.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Ensure you pass all "FORMAT:" tests
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications