Homework 05 – Avatar

Problem Description

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

Water. Earth. Fire. Air. Long ago, the four nations lived together in harmony. Then, everything changed when the Fire Nation attacked. Only you, the master of Java, can stop them. To do this, you will create **Bender.java**, **FireBender.java**, **AirBender.java**, and **PowerUp.java** that will simulate the respective "Benders" of the Fire Nation and Air Nomads and the battles between the AirBenders and FireBenders. You will also create a **BenderDriver.java** that will be used for testing purposes and will not be turned in to us. To complete this assignment, you will use your knowledge of inheritance, class hierarchies, method overriding, and abstract classes.

Remember to test for Checkstyle Errors and include Javadoc Comments!

Solution Description

Create files `Bender.java`, `FireBender.java`, `AirBender.java`, and `PowerUp.java` that will simulate battles between the FireBenders and AirBenders. 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.

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!

`PowerUp.java:`

This file defines an interface with the name PowerUp .

Methods

- `fireArmy()` - abstract method that does not take in anything and does not return anything.
  (Note: Any class that implements PowerUp must provide a method definition for this method)
- `strengthDoubles()` - concrete method that takes in a FireBender and doubles their strength level

`Bender.java:`

This file defines a Bender of the elements. You must not be able to create an instance of this class (there is a keyword that prevents us from creating instances of a class).

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 `Bender` class must have these variables.

- `name` - the name of the Bender, which can be made of any combination of characters
- `element` - the element of the Bender, which can be made of any combination of characters
- `strengthLevel` - the strength of the Bender as an integer number
- `health` - the health of the Bender as an integer number

Constructors

- A constructor that takes in the `name`, `element`, `strengthLevel`, and `health`. You should not have any other constructors. Make sure that health is not below 0. If `health` is a negative number, set it to 0.

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.

- `healthIncrease` - takes in an integer
    - Increases `health` by that integer level
    - If the health of the bender is equal to 0 then they are unable to consume a health aid.
    - Does not return anything
- `attack` - takes in a Bender object
    - Each attack method is unique to the specific bender.
    - Any concrete class that extends the Bender class must provide a method definition for attack.
    - Does not return anything.
- `equals`
    - Two Benders are equal if they have the same name, element, strengthLevel, and health.
- `toString` - returns a String describing the `Bender` as follows:
  (Note: replace the values in brackets [] with the actual value)
    - `“My name is [name]. I have control over [element]. My strength level is [strengthLevel] and my current health is [health].”`
- Getters and setters as necessary.

`FireBender.java`

This file defines a FireBender. A FireBender has power over the element fire. This is a type of Bender and should have all the attributes of one. FireBenders implement the PowerUp interface.

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 `FireBender` class must have these variables:

- `fireStrength` - the strength a FireBender has over their element as an integer number
- `hasArmy` - a value that represents if the FireBender has an army or not, represented by a boolean

Constructors

- A constructor that takes in a `name`, `strengthLevel`, `health`, `fireStrength`, and `hasArmy` and sets all fields accordingly. Remember all FireBenders use the element “Fire”.
- A constructor that takes in just a name and assigns the following default values:

        - `element:` Fire
        - `strengthLevel`: 30
        - `health`: 15
        - `fireStrength`: 15
        - `hasArmy`: true

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.

- `fireArmy` - does not take in anything
    - Checks the value of `hasArmy`
    - If `hasArmy` is `true` then it doubles the FireBender's current health value and triples the FireBender's `fireStrength` level.
    - If `hasArmy` is `false` then it does not do anything.
- `attack`
    - A FireBender can only attack if their health is greater than or equal to 5.
    - The Bender object being passed in is attacked. Their health decreases by the FireBender's strengthLevel * fireStrength. If the Bender's health is less than 0 after being attacked, set it equal to 0.
- `equals`
    - Two FireBenders are equal if they have the same name, element, strengthLevel, health, and fireStrength.
    - You must use the `equals` method from the Bender class to receive full credit.
- `toString` - returns a String describing the `FireBender` as follows:
  (Note: replace the values in brackets [] with the actual value)
    - `My name is [name]. I have control over [element]. My strength level is [strengthLevel] and my current health is [health]. My fire strength level is [fireStrength].`
    - You must use the `toString` method from the Bender class to receive full credit.
- Getters and Setters as necessary .

`AirBender.java`

This file defines an AirBender. An AirBender has power over the element air. This is a type of Bender and should have all the attributes of one. AirBenders **do not** implement the PowerUp interface.

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 `AirBender` class must have these variables:

- `avatar` - a value that represents if the AirBender is the Avatar or not, represented by a boolean

Constructors

- A constructor that takes in a `name`, `strengthLevel`, `health`, `avatar` and sets all fields accordingly. Remember all AirBenders use the element "Air".

- A constructor that takes in just a name and assigns the following default values:
    - `element:` Air
    - `strengthLevel`: 40
    - `health`: 35
    - `avatar`: false

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.

- `attack`
    - An AirBender can attack only if their health is greater than 0.
    - The Bender object being passed in is attacked. Their health decreases by the AirBender's strengthLevel divided by 2. If the Bender's health is less than 0 after being attacked, set it equal to 0.
- `spiritualProjection` - takes in a Bender object
    - AirBenders have the ability to spiritually project to other locations, and can form spiritual connections with other Benders.
    - Two Benders have a strong connection if their health levels are equal.
    - Return a boolean value to represent if the AirBender has established a spiritual connection.
- `avatarState` - takes in an element represented by a String and a Bender object
    - If the AirBender is not an avatar, this method does not do anything
    - If the element passed in is `Earth`:
        - The AirBender can decrease the health of the defending Bender.
        - The AirBender decreases the defending Bender's health by a half.
    - If the element passed in is `Fire`:
        - The AirBender can now decrease the defending Bender's strengthLevel.
        - The AirBender decreases the defending Bender's strengthLevel by 10. If the Bender's strengthLevel is under 10, set it equal to 0. If the Bender's strengthLevel falls under 0, set it equal to 0.
    - If the element passed in is `Air`:
        - The AirBender now has the ability to remove the defending Bender's ability to bend.
        - The AirBender can set the defending Bender's strengthLevel to 0 if the defending benders strength level is less than 15.
    - If the element passed in is `Water`:
        - The Air Bender can set the defending Bender's health to 0 if the defending Bender's health is less than 5 and their strengthLevel is less than 15.
- Getters and Setters as necessary.

`BenderDriver.java`

This Java file is a driver, meaning it will contain and run `FireBender` and `AirBender` objects and "drive" their values according to a simulated set of actions. Use this to test your code. You will not be turning this into us. Here are some tips to help you get started:

- Create a FireBender with a name of your choice
- Create an AirBender with a name of your choice
- FireBender attacks the Air Bender

- AirBender uses avatarState on the Fire Bender
- FireBender calls `fireArmy`
- FireBender calls `healthIncrease`
- Print out AirBender
- Print out FireBender

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] `PowerUp.java`

- [5] Correctly creates interface
- [10] Correctly implements specified methods

[30] `Bender.java`

- [5] Correct instance variables
- [5] Correctly creates constructors
- [20] Correctly implements specified methods

[30] `FireBender.java`

- [5] Correct instance variables
- [10] Correctly creates constructors
- [5] Correctly implements interface
- [10] Correctly implements specified methods

[25] `AirBender.java`

- [5] Correct instance variables
- [5] Correctly creates constructors
- [15] 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 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 Gradecope:

- Bender.java
- FireBender.java
- AirBender.java
- PowerUp.java

Make sure you see the message stating "HW05 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