Homework 05 – Legend of Java

Problem Description

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

Earth. Fire. Air. Water. Only the Avatar can master all four elements and bring balance to the world. You, the master of Java, have brought your talents to Republic City, where you want to simulate a pro-bending practice round where teams are divided by elements (i.e. all water benders are on one team). To do this, you will create **Bender.java**, **FireBender.java**, and **WaterBender.java**, and **EarthBender.java**. You will also create a **BendingPractice.java** that will be used for testing purposes, which will **not be turned in**. 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`, `WaterBender.java`, and `EarthBender.java`.

Based on the description given for each variable and method for each class, you will have to decide whether the variables/method should be static, as well as its appropriate visibility modifier.

Hint: A lot of code you will write for this assignment can be reused. Try to think of what keywords you can use that will help you!

`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

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, represented as a String
- `strengthLevel` – the strength of the Bender, represented as an integer
- `health` – the health of the Bender, represented as an integer

Constructors

You **must use constructor chaining** for your constructors (when applicable). The constructors ***must*** take the variables in the specified order. (Hint: Refer to L10: Constructors as L12: Coding Basics and More Visibility Modifiers for constructor chaining and constructor chaining when a superclass is involved).

- A 3-arg constructor that takes in the `name`, `strengthLevel`, and `health`.
    - You can assume that all strength and health values passed in will always be positive and nonzero integers

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.

- `recover(int i)`
  - Increases health by the integer amount that is passed in as a parameter.
  - If the health of the Bender is equal to 0, they unable to consume a health aid (unable to recover health).
  - Does not return anything
- `attack(Bender b)`
  - Each attack method is unique to the specific bender. There is no implementation for this method within the Bender class.
  - Any concrete class that extends the Bender class must provide a method definition for this method.
  - Does not return anything.
- `equals(Object o)`
  - Two Benders are equal if they have the same name, strengthLevel, and health.
  - Must override equals() method defined in Object class.
- `toString()`
  - Returns a String describing the Bender as follows (Note: replace the values in brackets [] with the actual value):
    - `My name is [name]. I am a bender. My strength level is [strengthLevel] and my current health is [health].`
  - Must override toString() method defined in Object class
- Getters and setters as necessary. No points will be taken off for extra getter and setter methods.

WaterBender.java

This file defines a WaterBender. A WaterBender has power over the element water. This is a type of Bender and should have all the attributes of a Bender..

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 WaterBender class must have these variables:

- `healer` – whether the WaterBender is a healer, represented as a boolean
- `waterPoints` – current points scored by the WaterBender team

Constructors

You **must use constructor chaining** for your constructors. The constructors ***must*** take the variables in the specified order. (Hint: Refer to L10: Constructors as L12: Coding Basics and More Visibility Modifiers for constructor chaining and constructor chaining when a superclass is involved).

- A 4-arg constructor that takes in a `name`, `strengthLevel`, `health`, and `healer` and sets all fields accordingly.
- A 1-arg constructor that only takes in `name` and assigns the following default values:
    - `strengthLevel`: 40
    - `health`: 80
    - `healer`: 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(Bender b)`
    - A WaterBender can only attack if their health is greater than 0.
    - The Bender passed in is attacked by the WaterBender. The Bender's health decreases by the WaterBender's strengthLevel.
        - If the Bender's health is less than or equal to 0 after being attacked, **set both health and strengthLevel to 0**
        - Additionally, the WaterBender team **scores points** equal to the Bender's strengthLevel if the Bender's health drops below 20
    - This method should not return anything.
- `heal(WaterBender wb)`
    - Since it is a competitive game, a WaterBender would only want to heal a fellow WaterBender!
    - If the caller of the method is not a healer (i.e. healer instance variable is false), return from the method (do nothing).
    - Otherwise, if the WaterBender can heal, add 20 points to the passed-in WaterBender's health and 20 points to their strenghLevel.
        - Unlike the recover() method, a WaterBender of health 0 *can be* healed by this method.
- `equals(Object o)`
    - This method must override Bender's equals() method

    - Two WaterBenders are equal if they both have the same name, strengthLevel, health, and healer attributes.
    - You must use the equals() method from the Bender class to receive full credit
    - Returns a boolean value
- `toString()`
    - This method must override the toString() method in the Bender superclass
    - Returns a String describing the WaterBender as follows (Note: Replace the values in brackets with the actual value):
        - `My name is [name]. I am a bender. My strength level is [strengthLevel] and my current health is [health]. With my waterbending, I [can/cannot] heal others.`
    - For example:
        - `My name is Jinbe. I am a bender. My strength level is 100, and my current health is 100. With my waterbending, I cannot heal others.`
    - You must use the toString() method from the Bender class to receive full credit
- Getters and Setters as necessary. No points will be taken off for extra getter and setter methods.

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 a Bender.

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:

- `firePoints` – current points scored by the FireBender team

Constructors

You **must use constructor chaining** for your constructors. The constructors ***must*** take the variables in the specified order. (Hint: Refer to L10: Constructors as L12: Coding Basics and More Visibility Modifiers for constructor chaining and constructor chaining when a superclass is involved).

- A 3-arg constructor that takes in a `name`, `strengthLevel`, and `health` and sets all fields accordingly.
- A 1-arg constructor that only takes in `name` and assigns the following default values:
    - `strengthLevel`: 60
    - `health`: 50

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(Bender b)`

    - A FireBender can only attack if their health is greater than 5.
    - The Bender object passed in is attacked by the FireBender. The Bender's health decreases by the FireBender's strengthLevel
        - If the Bender's health is less than or equal to 0 after being attacked, **set both health and strengthLevel to 0**
        - Additionally, the FireBender team **scores points** equal to the Bender's strengthLevel if the Bender's health drops below 20
    - This method should not return anything.
- `flameCircle(Bender[] b)`
    - The FireBender attacks multiple Benders through a wide attack
    - Each Bender in the array has their health decreased by 10
        - If the Bender's health is less than 0 after being attacked, set it equal to 0.
    - For style points, the FireBender team scores 5 points for each Bender attacked
    - This method should not return anything.
- Getters and Setters as necessary.

`EarthBender.java`

This file defines a EarthBender. A EarthBender has power over the element earth. This is a type of Bender and should have all the attributes of a Bender.

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 EarthBender class must have these variables:

- `earthArmor` – whether the EarthBender has their armor up
- `earthPoints` – current points scored by the EarthBender team

Constructors

You **must use constructor chaining** for your constructors. The constructors ***must*** take the variables in the specified order. (Hint: Refer to L10: Constructors as L12: Coding Basics and More Visibility Modifiers for constructor chaining and constructor chaining when a superclass is involved).

- A 4-arg constructor that takes in a `name`, `strengthLevel`, `health`, and `earthArmor` and sets all fields accordingly.
- A 1-arg constructor that only takes in `name` and assigns the following default values:
    - `strengthLevel`: 40
    - `health`: 100
    - `earthArmor`: 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(Bender b)`

    - An EarthBender can only attack if their health is greater than 0.
    - An EarthBender has immense control – if the Bender passed in is also an EarthBender, do nothing and return the method.
        - Hint: Check out the instanceof operator
    - The Bender passed in is attacked by the EarthBender. The Bender's health decreases by the EarthBender's strengthLevel and decreases by an additional 20 points if the EarthBender's earthArmor is active (i.e. earthArmor is true).
        - If the Bender's health is less than or equal to 0 after being attacked, **set both health and strengthLevel to 0**
        - Additionally, the EarthBender team **scores points** equal to the Bender's strengthLevel if the Bender's health drops below 20
    - After attacking, if the EarthBender has earthArmor, the armor weakens and disintegrates
    - This method should not return anything.
- `buildArmor()`
    - The EarthBender takes a turn to build up their earth armor.
    - This method sets their earthArmor attribute to true.
    - This method should not return anything.
- Getters and Setters as necessary. No points will be taken off for extra getter and setter methods.

BendingPractice.java

This Java file is a driver, meaning it will contain and run WaterBender, FireBender, and EarthBender objects and "drive" their values according to a simulated set of actions. This class is purely an example of what an output could look like. Its purpose is for you to test your code to produce the correct provided output. **You do not need to submit this file.**

In addition, keep in mind that getting the same output does not guarantee full points on this assignment, as this is only one example. You are encouraged to test out different scenarios.

- Create 2 WaterBender instances
    - A WaterBender with the following attributes:
        - name: Katara
        - strengthLevel: 80
        - health: 100
        - healer: true
    - A WaterBender created using the 1-arg constructor
        - name: Mermaid Man
- Create 2 FireBender instances
    - A FireBender with the following attributes:
        - name: Ace
        - strengthLevel: 120
        - health: 20
    - A FireBender created using the 1-arg constructor
        - name: Mushu
- Create 1 EarthBender instance
    - An EarthBender with the following attributes:
        - name: Whitebeard
        - strengthLevel: 100

        - health: 80
        - earthArmor: true
- Perform the following method calls:
    - Each in a separate line, print the following (Note: When printing, the print method automatically calls the object's toString() method)
        - Print the "Katara" object
        - Print the "Mushu" object
        - Print the "WhiteBeard" object
    - Ace attacks Mermaid Man
    - Whitebeard attacks Mermaid Man
    - Katara heals Mermaid Man
    - Whitebeard calls buildArmor()
    - Mermaid Man attacks Mushu
    - Mushu uses flameCircle() to attack Whitebeard and Katara
    - Katara recovers by 5 points using the recover() method
    - To conclude the battle, once again print the following:
        - Print the "Katara" object
        - Print the "Mushu" object
        - Print the "WhiteBeard" object
    - Print the number of points scored by each team (formatting is up to you, as this driver class will not be submitted)

The sample output is as follows:

```
1  My name is Katara. I am a bender. My strength level is 80 and my current health
   is 100. With my waterbending, I can heal others.
2  My name is Mushu. I am a bender. My strength level is 60 and my current health
   is 50.
3  My name is Whitebeard. I am a bender. My strength level is 100 and my current
   health is 80.
4  My name is Katara. I am a bender. My strength level is 80 and my current health
   is 95. With my waterbending, I can heal others.
5  My name is Mushu. I am a bender. My strength level is 60 and my current health
   is 30.
6  My name is Whitebeard. I am a bender. My strength level is 100 and my current
   health is 70.
7  Earth: 0
8  Water: 0
9  Fire: 50
```

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

[30] `Bender.java`

- [3] Correct class header
- [4] Correct instance variables
- [4] Correctly creates constructor
- [5] Correctly sets up abstract methods
    - [5] attack()
- [14] Correctly implements specified methods
    - [3] recover()
    - [4] equals()
    - [4] toString()
    - [3] Necessary setters and getters

[30] `WaterBender.java`

- [1] Correct class header
- [4] Correct instance variables
- [5] Correctly creates constructors
- [10] Correctly overrides and implements equals() and toString() methods
- [10] Correctly implements specified methods
    - [4] attack()
    - [4] heal()
    - [2] Necessary setters and getters

[20] `FireBender.java`

- [1] Correct class header
- [4] Correct instance variables
- [5] Correctly creates constructors
- [10] Correctly implements specified methods
    - [4] attack()
    - [4] flameCircle()
    - [2] Necessary setters and getters

[20] `EarthBender.java`

- [1] Correct class header
- [4] Correct instance variables
- [5] Correctly creates constructors
- [10] Correctly implements specified methods
    - [4] attack()
    - [4] earthArmor()
    - [2] Necessary setters and getters

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 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 **15 points**. This means that up to 15 points can be lost from Checkstyle errors. **In addition to past checks, the autograder will also include checks for Javadocs.** Any missing or incorrect Javadoc comments will count as 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:

- Bender.java
- WaterBender.java
- FireBender.java
- EarthBender.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