L12 Homework: Heroes & Enemies

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, constructor chaining with "this" and "super", equals(Object), method overriding, and calling the super implementation of overridden methods, to continue writing your own classes with Object-Oriented programming! You will create six classes that will simulate a world with heroes and enemies.

Solution Description

The six classes that you will write are Entity, TrainingGround, Hero, Knight, Archer, and Enemy. Each will be written in their own file (named accordingly).

- You must use constructor chaining when it's applicable:
  - If a class has N constructors, it will use "this" constructor chaining N-1 times.
  - A class will use "super" constructor chaining at most once. It may only be used in the constructor that has all the appropriate parameters. You will use it if and only if, in that constructor, the implicit "super" constructor chaining is not appropriate.
- Visibility for required elements:
  - All classes and constructors must be public.
  - All methods must be public unless otherwise stated in the method description
  - All variables must be private.
- You may add to aid you (if you want)
  - Methods, if they are private.
  - Class constants (static final variables), if they are private and their type is a primitive type, a wrapper class, or String.
- You may not add any other classes or constructors.

You will also be provided with two additional classes, EntityConcrete and HeroConcrete (in appropriately named files). These classes are used for autograding purposes, as we need a way to test the types that we cannot directly instantiate. **You need to familiarize yourself with those classes, as it will help you understand autograder error messages related to the types which they are replacing**. These files are **error-free, and error messages related to them are actually errors about the related type**. You must not change them in any way, and you must submit them along with your other files in your submission.

Entity

- It represents entities in the world, each having a name and health. This is an abstract class.
- It will have:
    - The following instance variables: `name` (`String`) and `health` (`int`).
    - 2-arg constructor that takes a `String` and an `int` (in that order) and sets the instance variables. If the health is less than 0, it should be set to 0 (the setter for health, described later, might come in handy)
    - Getters for the two instance variables
    - Setter for name
    - Setter for health with **private** visibility. If the parameter is less than 0, the health should be set to 0, which is the only value used to denote an entity that is dead.
    - `boolean isAlive()` method that returns whether an entity is alive. An entity is alive if it's not dead.
    - `void takeDamage(int)` method that reduces the health of an entity. Note that the health must be set to 0 if it goes below it. The setter for health might come in handy.
    - `void heal(int)` method that increases the health of an entity. The entity may only gain health if it's alive and the parameter is positive.
    - An appropriate `toString` method with the following formatting:
        - If the entity is alive, the format is "I am [name], and I have [health] health" (without the brackets)
        - Otherwise, the format is "I was [name]" (without the brackets)
    - An appropriate `equals` method

TrainingGround

- It represents a training ground, a location in the world in which heroes can train. This is a concrete class.
- It will have:
    - The following instance variables: `name` (`String`), `trainingMultiplier` (`double`), and `isOutdoors` (`boolean`).
    - 3-arg constructor that takes all instance variables (in the order above) and sets the instance variables. If the `trainingMultiplier` is less than 0, it should be set to 0 (the setter for `trainingMultiplier`, described later, might come in handy)
    - 1-arg constructor that takes a name and constructs an indoors training ground with `trainingMultiplier` 1
    - Getters for `name` and `trainingMultiplier`
    - A method `boolean isOutdoors()` that returns `isOutdoors`. Note that we are not considering it a getter since it uses a different naming convention, but it behaves the same way a getter does (some conventions might consider this a getter, but in this course, a getter needs to have a getX naming format). The reasoning behind this decision is that isOutdoors is a better method name than getIsOutdoors.
    - Setter for `trainingMultiplier`. If the parameter is less than 0, the `trainingMultiplier` should be set to 0.
    - An appropriate `toString` method with the following formatting:
        - If the training ground is outdoors, the format is

            - "Outdoors Training Ground: [name]. It has training multiplier [training multiplier with two decimal digits]" (without the brackets). Tip: Use String.format.
        - Otherwise, the format is the same as above but starting with “Indoors” instead of “Outdoors”
    - An appropriate `equals` method

Hero

- It represents heroes in the world, each having an associated name, health, and damage they deal. This is an abstract class and a direct child of Entity.
- It will have:
    - The following instance variables: damage (`int`). Think why we don’t add name and health to this list.
    - 3-arg constructor that takes a `String`, an `int`, and another `int` for the `name`, `health`, and `damage`, respectively. If `damage` is less than 0, it should be set to 0.
    - 2-arg constructor that takes a `String` and an `int`, to create heroes that deal 1 damage.
    - Getter for damage
    - `void increaseDamage(int)` method with **protected** or **default** visibility. It increases the damage the hero deals by the parameter, but only if the hero is alive.
    - `void train(TrainingGround)` method. This method is used to have the hero train to increase the damage they can deal. This method should not be implemented in this class, as the behavior will be fully determined by the subclasses of `Hero`.
    - `boolean hasArmor()` method. This method returns whether the hero is currently wearing their armor. Like the method above, it should not be implemented in this class.
    - `void attack(Enemy)` method. If the hero is alive, it will deal damage to the enemy (the same amount of damage as the hero currently deals).
    - An appropriate `toString` method with the following formatting:
        - If the hero is alive, the format is:
            - "I am [name], and I have [health] health. I deal [damage] damage" (without the brackets)
        - Otherwise, the format is "I was [name]" (without the brackets)
        - Note the prefix matches the one from Entity in both cases. <u>You must call the toString method from Entity for full credit</u>.
    - An appropriate `equals` method
        - <u>You must call the equals method from Entity for full credit</u>.

Knight

- It represents knights in the world, each having an associated name, health, and damage they deal. This is a concrete class and a direct child of Hero.
- It will have:
    - 3-arg constructor that takes a `String`, an `int`, and another `int` for the `name`, `health`, and `damage`, respectively.
    - 2-arg constructor receiving a name and health, to create a knight that deals 2 damage.
    - 1-arg constructor receiving a name to create a knight with 40 health that deals 2 damage.

- `void train(TrainingGround)` method. The knight will increase the damage they deal by the result of rounding down: 5 times the multiplier from the training ground. Tip: cast the result to int.
- `boolean hasArmor()` method. A knight always wears his armor.
- An appropriate `toString` method with the following formatting:
  - If the knight is alive, the format is:
    - "I am [name], and I have [health] health. I deal [damage] damage. I am a knight" (without the brackets)
  - Otherwise, the format is "I was [name]. I was a knight" (without the brackets)
  - Note the prefix matches the one from Hero in both cases. <u>You must call the toString method from Hero for full credit</u>.
- An appropriate `equals` method
  - <u>You must call the equals method from Hero for full credit</u>.

<u>Archer</u>

- It represents archers in the world, each having an associated name, health, damage they deal, and a value indicating whether they are currently wearing armor. This is a concrete class and a direct child of Hero.
- It will have:
  - The following instance variables: `hasArmorEquipped` (`boolean`)
  - 4-arg constructor that takes a `String`, `int`, `int`, `boolean`, for the `name`, `health`, `damage`, and `hasArmorEquipped`, respectively.
  - 2-arg constructor receiving a name and health, to create an archer that deals 4 damage and is not wearing their armor.
  - 1-arg constructor receiving a name to create an archer with 20 health that deals 4 damage and is not wearing their armor.
  - `void equipArmor()` and `void unequipArmor()` methods. An archer can equip or unequip their armor if they are alive.
  - `void train(TrainingGround)` method. If the training ground isn't outdoors, it does nothing. Otherwise, the archer will increase the damage they deal by the result of rounding down: 3 times the multiplier from the training ground. Tip: cast the result to int.
  - `boolean hasArmor()` method. Indicates whether the archer is currently wearing their armor.
  - An appropriate `toString` method with the following formatting:
    - If the archer is alive and wearing their armor, the format is:
      - "I am [name], and I have [health] health. I deal [damage] damage. I am an archer with my armor equipped" (without the brackets)
    - If the archer is alive but not wearing their armor, the format is the same as above but "equipped" is replaced with "unequipped"
    - If the archer is dead, the format is "I was [name]. I was an archer" (without the brackets)
    - Note the prefix matches the one from Hero in all cases. <u>You must call the toString method from Hero for full credit</u>.
  - An appropriate `equals` method
    - <u>You must call the equals method from Hero for full credit</u>.

Enemy

- It represents enemies in the world, each having an associated name, health, and value indicating whether they can pierce armor. This is a concrete class and a direct child of Entity.
- It will have:
    - The following instance variables: `canPierceArmor` (`boolean`).
    - 3-arg constructor that takes a `String`, an `int`, and `boolean`, in that order.
    - 2-arg constructor that takes `String`, `int` for enemies that cannot pierce armor.
    - A method `boolean canPierceArmor()` that returns `canPierceArmor`. Note that we are not considering it a getter since it uses a different naming convention, but it behaves the same way a getter does (some conventions might consider this a getter, but in this course, a getter needs to have a getX naming format). The reasoning behind this decision is that `canPierceArmor` is a better method name than get`CanPierceArmor`.
        - As you may have noticed, boolean values tend to have very poor getter names, and getters are frequently replaced by methods with the same name instead.
    - `void attack(Hero)` method. Note that while `Hero` is abstract, it can be used as a parameter type to indicate any concrete class that directly or indirectly extends from `Hero`. We will explore more about this in L15.
        - If the enemy isn't alive, it does nothing
        - Otherwise, if the hero isn't wearing their armor or the enemy can pierce armor, it deals 3 damage to the hero
        - Otherwise, it deals 1 damage to the hero
    - An appropriate `toString` method with the following formatting:
        - If the enemy is alive and can pierce armor, the format is:
            - "I am [name], and I have [health] health. I am an enemy that can pierce armor" (without the brackets)
        - If the enemy is alive but cannot pierce armor, the format is the same as above, but "can" is replaced with "cannot"
        - Otherwise, the format is "I was [name]. I was an enemy" (without the brackets)
        - Note the prefix matches the one from Entity in all cases. You must call the toString method from Entity for full credit.
    - An appropriate `equals` method
        - You must call the equals method from Entity for full credit.

Testing Your Solution

You will notice that none of the classes have a `main` method. That is because not all classes have to be Java programs, and in this case, none are. To help test your implementations, you can create more classes in which you can create instances of your concrete classes, invoke methods to have them interact with each other, and print to console information that helps verify the correctness of your solution.

Rubric

[40] Type Review (it is very important for your submission that these tests pass. All of these are visible in every submission)

- [10] All classes compile, all class headers are correct, all static variables are correct
- [5x6] Check constructors (1), methods (3), variables (1) presence in each class

[10x6] Implementations in each of the student-written classes

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 packages.

Feature Restriction

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)

Checkstyle

The Checkstyle deduction limit for this assignment is 15 points, counting Javadoc errors. Refer to the course guide for the use of Checkstyle and review the autograder Checkstyle report on your submissions.

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].

If the autograder encounters issues processing your collaboration statement, begin it with "Collaboration Statement: ".

Allowed Collaboration

When completing homeworks for CS 1331 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

Upload the files listed below to the corresponding assignment on Gradescope:

- Archer.java
- Enemy.java
- Entity.java
- EntityConcrete.java (provided but must also be submitted)
- Hero.java
- HeroConcrete.java (provided but must also be submitted)
- Knight.java
- TrainingGround.java

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.

<u>Important Notes</u>

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit .class files or .jar 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