Homework 06 - Dungeon

Purpose

To complete this assignment, you will apply your knowledge of inheritance, class hierarchies, abstract classes, and method overriding. This assignment covers inheritance, one of the core concepts of Object-Oriented programming. When doing this homework, be sure adhere to principles of good class design!

Description

Taking that quest to travel deep into the dungeons of a 10,000 year old dragon was probably not your best decision. Nor was coming alone with only your laptop and about three hours of battery life. Luckily, since you are an excellent programmer, you can use your code writing skills to conjure up some help! In this exercise, you will write classes for various characters out of a fantasy dungeon exploration game to help you out before you are stuck in the dungeon forever!

Directions

For this assignment, you will be creating four classes to simulate brave adventurers: one class representing the archetypal adventurer (`Adventurer.java`), one class representing the noble knight (`Knight.java`), one class representing the feisty archer (`Archer.java`), and one class to test your code (`DungeonDriver.java`). You will also javadoc your code for this assignment, so be sure to read the "Checkstyle and Javadocs" section.

`Adventurer.java`

This Java file represents an adventurer from the side of good. You should not be able to create an instance of this class.

- Instance Variables:
  - Since adventurers are not nameless souls, they should have a name attribute. Adventurers are a secretive bunch, so this variable should be private with no getters or setters.
  - All adventurers come ready to fight, so include an attack attribute to show their power! Once assigned, the attack value should never change.
  - No adventurers like to quit, but some are more durable than others, shown by a health attribute
  - You may include more variables if you'd like in this class
- Constructors:
  - Include a constructor that takes in name, health, and attack values
  - You should not have any other constructors
- Methods:
  - Since the dungeon is a confusing and dangerous place, adventurers should be able to attack each other
    - Every adventurer has their own unique attack, so this method should be abstract
    - This method should take in another `Adventurer` as its only parameter
  - An adventurer should also be able to drink a magical health potion.
    - This should increase their health by a value of 15
    - An adventurer is allowed to have a health value that is higher than their original health value
    - Even if an adventurer has no health remaining, they can still drink a potion. This is because 0 health doesn't mean they're dead, they're just too exhausted to keep fighting
  - `equals` method
    - Override from `Object`
    - Two adventurers are equal if they have the same health, attack and name values
  - `toString` method
    - Override from `Object`

        * Should return `"Name: {name}, Health: {health}, Attack: {attack}"`
    - Getters and setters as necessary. Remember, do not create getters or setters for the name attribute.
- Class must follow the rules of encapsulation and abstraction

`Knight.java`

This Java file represents a noble knight.

- A knight is a type of `Adventurer` and should have all the behaviors of one
- Instance Variables:
    - A squire is a knight's helper. Some knights have a squire while others are less fortunate, so knights should have an attribute for the presence of one
    - No other instance variables should be created in `Knight`
- Constructors:
    - A constructor that sets a knight's name, health, attack, and squire existence.
    - A constructor that takes in a name and assigns the following default values:
        * Health: 100
        * Attack: 15
        * hasSquire: true
    - No other constructors should be created
- Methods:
    - A knight is a powerful warrior who can attack with great might
        * Attack must override the method from `Adventurer`
        * If any null parameters are passed in, this method should not do anything
        * A knight may only attack if they have more than 0 remaining health
        * If the knight does not have a squire, damage dealt should be equal to the knight's attack value. If the knight has a squire, damage dealt should be twice the knight's attack value.
        * The damage should be inflicted on the defending adventurer's health. If the attack brings the defending adventurer's health below 0, set it to 0 instead
    - A knight should be able to challenge another knight for their squire
        * This method should take in another knight as its only parameter
        * If any null parameters are passed in, this method should not do anything
        * A knight can only attempt to steal a squire if the following are true: the knight has more than 0 health, the knight doesn't have a squire, and the other knight (who's being stolen from) does have a squire.
        * The challenge can still occur if the defending knight has 0 health
        * When the conditions above are met, the squire-stealing knight makes one attack on the other.
            · If the attacked knight's health hits 0, it loses its squire. The squire-stealing knight gets the squire!
            · If the attacked knight's health doesn't hit zero, it should attack the honorless squire-stealing knight in revenge. Don't change any squire attributes in the revenge attack. Only make one revenge attack.
    - `equals` method
        * Must override
        * Two knights are equal if they have the same health, attack, name and squire values.
    - `toString` method
        * Must override
        * Returns `"Name: {name}, Health: {health}, Attack: {attack}, Squire: {true/false}"`
    - Getters and Setters as appropriate
- Class must follow the rules of encapsulation and abstraction

`Archer.java`

This Java file represents a stealthy archer.

- An archer is a type of adventurer and should have all the behaviors of one

- Instance Variables:
  - Archers always carry arrows, and they should have an attribute for how many they currently have. By default, archers carry 10 arrows.
  - No other instance variables should be created in `Archer`
- Constructors:
  - A constructor that takes in name, health, and attack values
  - A constructor that takes in a name and assigns the following default values:
    - Health: 75
    - Attack: 40
  - No other constructors should be created
- Methods:
  - An archer's attacks their enemies with aid from their bow
    - Attack must override from `Adventurer`
    - If any null parameters are passed in, this method should not do anything
    - An archer is only able to attack if they have at least 1 arrow left and more than 0 health
    - Damage dealt should be equal to the archer's attack value
    - The damage should be inflicted on the defending adventurer's health. If the attack brings the defending adventurer's health below 0, set it to 0 instead
    - After the attack, reduce the amount of arrows the archer has by 1
  - `equals` method
    - Must override
    - Two archers are equal if they have the same health, attack and name values.
  - `toString` method
    - Must override
    - Returns `"Name: {name}, Health: {health}, Attack: {attack}, Arrows: {# arrows}"`
  - Getters and Setters as appropriate
- Class must follow the rules of encapsulation and abstraction

`DungeonDriver.java`

This Java file is a driver, meaning it will run the simulation. You can also use it to test your code. Here are some basic tests to get you started with `DungeonDriver`. These tests are my no means comprehensive, so be sure to create your own!

Create a `Knight` with the following fields: Name="Galahad", Health=7, Attack=6, Squire=true
Create a `Knight` with the following fields: Name="Lancelot", Health=30, Attack=5, Squire=false
Create an `Archer` with the following fields: Name="Archie", Health=10, Attack=2, Arrows=10

1. Have the `Archer` Archie attack the `Knight` Lancelot. After the attack:
   - Archie should have: Name="Archie", Health=10, Attack=2, Arrows=9
   - Lancelot should have: Name="Lancelot", Health=28, Attack=5, Squire=false
2. Now have the `Knight` Galahad attack the `Archer` Archie. After the attack:
   - Galahad should have: Name="Galahad", Health=7, Attack=6, Squire=true
   - Archie should have: Name="Archie", Health=0, Attack=2, Arrows=9
3. Have the `Archer` Archie attack the `Knight` Galahad.
   - Nothing happens, because Archie has Health=0, and so is too tired to attack
4. Have the `Knight` Lancelot attack the `Knight` Galahad. After the attack…
   - Galahad should have: Name="Galahad", Health=2, Attack=6, Squire=true
   - Lancelot should have: Name="Lancelot", Health=28, Attack=5, Squire=false
5. Have the `Knight` Lancelot challenge the `Knight` Galahad in an attempt to steal his squire. After the challenge,
   - Galahad should have: Name="Galahad", Health=0, Attack=6, Squire=false
   - Lancelot should have: Name="Lancelot", Health=28, Attack=5, Squire=true

Tips and Tricks

- Reuse code when possible! The `equals` and `toString` methods in particular have a lot of reuse potential.
- Test your code yourself! The `DungeonDriver` tests are by no means comprehensive.
- When overriding a method, if you use the `@Override` tag, you don't need to javadoc that method!
- Unless specified, you are free to choose access modifiers for your variables. Just ensure they follow the rules of encapsulation!

Note: Make sure your classes are all in separate files and that all of your files are in the same directory.

Allowed Imports

To prevent trivialization of the assignment, no imports are allowed.

Checkstyle and Javadocs

You must run checkstyle on your submission. The checkstyle cap for this assignment is **20** points. If you don't have checkstyle yet, download it from Canvas -> Files/Resources. Place it in the same folder as the files you want checkstyled. Run checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle 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!

Additionally, you must javadoc your code.

Run the following to only check your javadocs.

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both javadocs and checkstyle.

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or breaks our auto grader. For that reason, do not use any of the following in your final submission: * var (the reserved keyword) * System.exit * Runtime.getRuntime.halt * Runtime.getRuntime.exit

Collaboration

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 `//`.

Turn-In Procedure

Submission

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

- `Adventurer.java`
- `Knight.java`
- `Archer.java`

Make sure you see the message stating "HW06 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.

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 all official clarifications
