Homework 4 - D n D Characters

Introduction

In this homework, you will practice:

- implementing classes
- using constructors
- instance methods
- sub and superclasses
- constructor delegation
- polymorphism

Problem Description

After hearing that you have taken the best class at Georgia Tech, CS 1331, an important video game company has hired you to develop the framework for their latest video game, *Fantasy Quest 3: Return of Zardaneck*.

Solution Description

Download the abstract class Character.java (/spring2018/hw4/Character.java). Implement all the missing methods of the abstract class, as well as implementing the subclasses Fighter.java, Rogue.java, Wizard.java, and Cleric.java. You will need to create these other classes from scratch.

Do not modify the code we give you.

Character.java

- A character has several instance variables relating to the characters stats. They are:
  1. name
  2. level
  3. 4 stats (strength, dexterity, intelligence and wisdom)
  4. health
  5. isDead
- Implement the getter and setter methods that have been partially provided.
- Two constructors
  1. Assigning `String name, int level, int strength, int dexterity, int intelligence, int wisdom`. The `health` variable should start equal to five times the character level, and should never go above that. If the `health` variable ever goes below 0, `isDead` should be set to `true` and the `health` variable be reset to 0.
  2. Only taking in a name and a seed, sets the level to one, sets health to 5, and randomly sets the other non-health stats to a number between 1 and 6 (inclusive) by using a Random object using the given seed. Make sure that you generate the other stats in order using this Random object (i.e. `strength`, then `dexterity`, then `intelligence`, then `wisdom`). You do not have to use explicit constructor invocation (aka constructor delegation) for this.

Fighter.java

- This class should extend `Character` and implement all of its abstract methods.
- Implement the `attack(Character c)` method. The attack method of `Fighter` should decrement the health of the parameter character by 10 plus the fighter's strength variable. If the character's `isDead` variable is true, this method should do nothing besides printing `"Cannot attack a dead character"` to the console.
- Implement the `levelUp()` method. This method should increase the character's `level` by 1, reset the `health` to it's maximum (5 times the `level`), increase `strength` by 2, and all other stats by 1.
- a `toString()` method that returns a string with format `"Level (level) fighter named (name) with (strength) strength, (dexterity) dexterity, (intelligence) intelligence, and (wisdom) wisdom."`
- Implement 2 constructors coorresponding with each of the constructors of the superclass.

Rogue.java

- This class should extend `Character` and implement all of its abstract methods.
- Implement the `attack(Character c)` method. The attack method of `Rogue` should decrement the health of the parameter character by 6 plus the rogue's dexterity variable. If the character's `isDead` variable is true, this method should do nothing besides printing `"Cannot attack a dead character"` to the console.
- Implement the `levelUp()` method. This method should increase the character's `level` by 1, reset the `health` to it's maximum (5 times the `level`), increase `dexterity` by 3, and all other stats by 2.
- a `toString()` method that returns a string with format `"Level (level) rogue named (name) with (strength) strength, (dexterity) dexterity, (intelligence) intelligence, and (wisdom) wisdom."`
- Implement 2 constructors coorresponding with each of the constructors of the superclass.

Cleric.java

- This class should extend `Character` and implement all of its abstract methods.
- Implement the `attack(Character c)` method. The attack method of `Cleric` should decrement the health of the parameter character by 6 plus the cleric's wisdom variable. If the character's `isDead` variable is true, this method should do nothing besides printing `"Cannot attack a dead character"` to the console.
- Implement the `levelUp()` method. This method should increase the character's `level` by 1, reset the `health` to it's maximum (5 times the `level`), increase `wisdom` by 2, and all other stats by 1.
- Create a new method called `heal(Character c)` which increases the parameter characters health by 6 plus the clerics wisdom variable, but not beyond their maximum health (5 times their level). If a character is dead, do nothing except print `"Cannot heal a dead character"` to the console.
- a `toString()` method that returns a string with format `"Level (level) cleric named (name) with (strength) strength, (dexterity) dexterity, (intelligence) intelligence, and (wisdom) wisdom."`
- Implement 2 constructors coorresponding with each of the constructors of the superclass.

Wizard.java

- This class should extend `Character` and implement all of its abstract methods.
- Implement the `attack(Character c)` method. The attack method of `Wizard` should decrement the health of the parameter character by a 4 plus the wizard's intelligence variable. If the character's `isDead` variable is true, this method should do nothing besides printing `"Cannot attack a dead character"` to the console.
- Implement the `levelUp()` method. This method should increase the character's `level` by 1, reset the `health` to it's maximum (5 times the `level`), increase `intelligence` by 2, and all other stats by 1.
- Create a new method called `multiAttack(Character ... c)` which decreases each character in the parameter's health by 2 plus the wizard's intelligence variable (different values for each character). If a character is dead, do nothing to that character except print `"Cannot damage a dead character"` to the console.
- a `toString()` method that returns a string with format `"Level (level) wizard named (name) with (strength) strength, (dexterity) dexterity, (intelligence) intelligence, and (wisdom) wisdom."`
- Implement 2 constructors coorresponding with each of the constructors of the superclass.

Grading

- [10] Getter and setter methods for `Character`.
- [5] Properly change `Character.isDead`.
- [5] `health` never goes above `level` * 5.
- [5 * 2 = 10] Each constructor for `Character`.
- [5 * 4 = 20] Each `attack()` method for `Rogue`, `Fighter`, `Wizard`, and `Cleric`.
- [5 * 4 = 20] Each `levelUp()` method for `Rogue`, `Fighter`, `Wizard`, and `Cleric`.
- [5] `Wizard.multiAttack()`
- [5] `Cleric.heal()`.
- [5 * 4 = 20] `toString()` methods for each class.

Running and Testing

Creating a `Game` class with a `main` method to create a simulation to test of all the methods would be the best course of action. **Do not submit this.**

Tips and Considerations

If anything seems confusing, read through the entire description and instructions again. As always, feel free to contact your TAs, post on Piazza, or come in for office hours. In addition, here are some tips specific to this homework:

Import `java.util.Random`, but not anything that trivializes the assignment.

Use the java API when you need help thinking of how to do something.

Test things out in jshell!

Javadocs

You will need to write Javadoc comments and watch for checkstyle errors with your submission.

- Every class should have a class level Javadoc that includes `@author <GT Username>`.
- Every public method should have a Javadoc explaining what the method does and includes any of the following tags if applicable:
    - `@param <parameter name> <brief description of parameter>`
    - `@returns <brief description of what is returned>`
    - `@throws <Exception> <brief explanation of when the given exception is thrown>`

See the CS 1331 Style Guide (http://cs1331.gatech.edu/cs1331-style-guide.html) for details.

Checkstyle

For each of your homwork assignments we will run checkstyle and deduct one point for every checkstyle error.

For this homework the **checkstyle cap is 50**, meaning you can lose up to 50 points on this assignment due to style errors. This limit will increase on the next homework.

- If you encounter trouble running checkstyle, check Piazza for a solution and/or ask a TA as soon as you can!
- You can run checkstyle on your code by using the jar file found on the course website that includes xml configuration file specifying our checks. To check the style of your coed run `java -jar checkstyle-6.2.2.jar *.java` .
- To check your Javadocs run `java -jar checkstyle-6.2.2.jar -j *.java` .
- Note that the command for checking code and the command for checking Javadocs are different. You will have to run both commands to fully test for style errors.
- Javadoc errors are the same as checkstyle errors, as in each one is worth a single point and they are counted towards the checkstyle cap.
- **You will be responsible for running checkstyle on *ALL* of your code.**
- 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 (http://cs1331.gatech.edu/customization-tips.html) page for more information.

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 specification 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:

OKAY: “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?”

BY NO MEANS OKAY: “Hey… the homework is due in like 20 minutes… Can I see your code? I *promise* won’t copy it directly!”

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.

Submission

- Submit all of your Java source files, as well as the `Character.java` file we provided. When we download your submission from Canvas we should have everything we need to test your code. You should do this in an empty directory after submission to ensure that you have submitted complete, working code. You can submit as many times as you want, so feel free to submit as you make substantial progress on the homework. We only grade your **last** submission, meaning we will ignore any previous submissions.

- As always, late submissions will not be accepted and non-compiling code will be given a score of 0. For this reason, we recommend submitting early and then confirming that you submitted ALL of the necessary files by re-downloading your file(s) and compiling/running them.