Homework 06 – The RPG Game

Authors: Brandon, Vishal, Chloe, Joshua, Helen, Ricky, Sang Yoon
Topics: Polymorphism, File IO, Handling Exceptions

Problem Description

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

You and your friends were bored the past spring break, so you decided to try out the new popular table-top RPG game, *Computers & Compilers*! However, you are overwhelmed with the options and variety of this RPG game, so you decided to create a database with the heroes from this game.

Solution Description

You have been given the .class files for the following classes:

- PartyMember.java
- Warrior.java
- Mage.java

This means that you **do not have to write or submit these files**. The instructions for what these files include have been outlined below, since they will be used in other classes you are writing. It will be helpful to understand what the purpose of each class and its methods are, but you do not have to implement them.

- **Note:** If you are using an IDE, calling a method from one of the files above may give you red squiggly line errors. But since you've been given the .class file, compiling your files should still work. You can either 1) ignore the red squiggly lines or 2) write basic headers for the class and methods but still use the .class files provided.

You will need to complete and turn in 3 classes: InvalidPartyMemberException.java, QuestNotFoundException.java, and Party.java.

Notes:

1. All variables should be inaccessible from other classes and require an instance to be accessed through, unless specified otherwise.
2. All methods should be accessible from everywhere and must require an instance to be accessed through, unless specified otherwise.
3. Use constructor chaining and reuse methods whenever possible! Ensure that your constructor chaining helps you maximize code reuse!
4. Reuse code when possible and helper methods are optional.
5. Make sure to add Javadoc comments to your methods and classes!

PartyMember.java

This class represents a party member in *Computers & Compilers*, and it is an `abstract` class.

Variables:

- `characterName` – A `String` representing the name of the party member

- `health` – An `int` representing the amount of health points this party member has, which cannot exceed 100
- `baseAttack` – An `int` representing the amount of attack power this party member has

Constructors:

- A constructor that takes in `characterName`, `health`, and `baseAttack`.
  - If the passed value for `characterName` is an empty string or is null, throw an `IllegalArgumentException`.
  - If the passed in value for `health` is negative or exceeds the upper bound, throw an `IllegalArgumentException`.
  - If the passed in value for `baseAttack` is negative or is greater than double the `health` value, throw an `IllegalArgumentException`.

Methods:

- `toString()`
  - Should properly override `Object`'s `toString` method and return:
    "`Name: <characterName>, Health: <health>, Base Attack: <baseAttack>`"
- `equals()`
  - Should properly override `Object`'s `equals` method
  - Two `PartyMember` objects are equal if they have the same `characterName`, `health`, and `baseAttack`.
- `questLevel()`
  - This method will be used in calculating a party's overall ability to conquer a given quest.
  - Calculate the party member's questLevel as the average of their `health` and `baseAttack` using integer division. Return this result.

Warrior.java

This class describes a warrior and is a concrete implementation of `PartyMember`.

Variables:

- `weapon` – A `String` representing the weapon this warrior uses
- `armorClass` – An `int` representing how good their armor is, should be non-negative and no more than 10

Constructors:

- A constructor that takes in `characterName`, `health`, `baseAttack`, `weapon` and `armorClass`.
  - If the value passed for `weapon` is an empty string or is null, throw an `IllegalArgumentException`.
  - If the value passed for `armorClass` is invalid, throw an `IllegalArgumentException`.
  - If the values passed in for `characterName`, `health`, and `baseAttack` are invalid, throw an `IllegalArgumentException` as outlined by `PartyMember`.

Methods:

- `toString()`
  - Should properly override the parent class's `toString` method and return:

"`Class: Warrior, Name: <characterName>, Health: <health>, Base Attack: <baseAttack>, Weapon: <weapon>, Armor Class: <armorClass>`"

- `equals()`
  - Should properly override the parent class's `equals` method.
  - Two `Warrior` objects are equal if they have the same `characterName`, `health`, `baseAttack`, `weapon`, and `armorClass`.
- `questLevel()`
  - Should properly override the parent class's `questLevel` method.
  - Calculate the warrior's questLevel as the sum of:
    - the warrior's `armorClass`
    - and the average of their `health` and `baseAttack` as an integer.
  - Additionally, if a warrior has a "`mace`" weapon (case-sensitive), increment their questLevel by 2.

Mage.java

This class represents a mage and is a concrete implementation of `PartyMember`.

Variables:

- `spellAttack` – An `int` representing the amount of magic damage the mage can do. Must be greater than the Mage's baseAttack but less than double their baseAttack.
- `spellSlots` – An `int` representing how many spells they can cast. Must be greater than 0 and less than or equal to half their health.

Constructors:

- A constructor that takes in `characterName`, `health`, `baseAttack`, `spellAttack` and `spellSlots`.
  - If the value passed for `spellAttack` is invalid, throw an `IllegalArgumentException`.
  - If the value passed for `spellSlots` is invalid, throw an `IllegalArgumentException`.
  - If the values passed in for `characterName`, `health`, and `baseAttack` are invalid, throw an `IllegalArgumentException` as outlined by `PartyMember`.

Methods:

- `toString()`
  - Should properly override the parent class's `toString` method and return:
    "`Class: Mage, Name: <characterName>, Health: <health>, Base Attack: <baseAttack>, Spell Attack: <spellAttack>, Spell Slots: <spellSlots>`"
- `equals()`
  - Should properly override the parent class's `equals` method.
  - Two `Mage` objects are equal if they have the same `characterName`, `health`, `baseAttack`, `spellAttack`, and `spellSlots`.
- `questLevel()`
  - Should properly override the parent class's `questLevel` method.
  - Calculate the mage's questLevel as the sum of:

- the product of `spellSlots` and `spellAttack`
- and the average of their `health` and `baseAttack` as an integer.

InvalidPartyMemberException.java

This class describes an **unchecked** exception.

Constructors:

- A constructor that takes in a `String` representing the exception's message.
- A no-args constructor that has the default message "`Invalid party member!`"
- **Hint:** How can we use the super class's constructor here?

QuestNotFoundException.java

This class describes a **checked** exception.

Constructors:

- A constructor that takes in a `String` representing the exception's message.
- A no-args constructor that has the default message "`Selected Quest Not Found`"
- **Hint:** How can we use the super class's constructor here?

Party.java

This class will hold various `public static` methods that will let you read and write to the database.

Methods:

- `recruitParty()`
  - Takes in a `String` object representing the file name to read from.
  - Throws a `FileNotFoundException` if the passed in file is null or doesn't exist.
  - Returns an `ArrayList` of `PartyMember` objects.
  - Each line of the file will contain information about different types of party members.
    - File line tokens:

      ```
      Class: Warrior, Name: <characterName>, Health:
      <health>, Base Attack: <baseAttack>, Weapon:
      <weapon>, Armor Class: <armorClass>
      ```

      or

      ```
      Class: Mage, Name: <characterName>, Health:
      <health>, Base Attack: <baseAttack>, Spell
      Attack: <spellAttack>, Spell Slots: <spellSlots>
      ```

    - Iterate through the file and create the appropriate `PartyMember` objects for each line.
      - **Hint #1:** Try completing the processInfo method below first!
      - **Hint #2:** Don't process empty lines!
    - Add each `PartyMember` object to the `ArrayList`.
    - **Note: You can assume the Strings for class types, names, and weapons will all be ONE word. This is useful to consider when splitting each line up.**
- `processInfo()`

  - This is a private helper method that will be used to process a line of text from a CSV into a `Warrior` or `Mage` object.
  - Takes in a `String` representing the line of info to process.
  - Returns a `PartyMember` object that is either a `Warrior` or `Mage`.
  - If the type of character class is not `Warrior` or `Mage`, throw an `InvalidPartyMemberException`.
  - For instance, the `String`

    ```
    Class: Warrior, Name: <characterName>, Health:
    <health>, Base Attack: <baseAttack>, Weapon: <weapon>,
    Armor Class: <armorClass>
    ```

    would be converted to a `Warrior` object with attributes `characterName`, `health`, `baseAttack`, `weapon`, and `armorClass`.
  - You may **NOT** use a second Scanner object to process this line, as it will break the autograder.
    - **(Important) Hint #1:** The `String.split()` method can be used to split a String with a given delimiter into an array of tokens. Think about what sequence of characters you can split the line apart with to separate the pairs of information. Also, think about what sequence of characters you can split the pairs apart with to process the value.
    - **Hint #2:** `Integer.parseInt()` will be useful.

- `partyRoster()`
  - Takes in two parameters:
    - A `String` object representing the file name to write to. The original contents of the file, if it had already existed, should be preserved. The `String` is guaranteed to be non-null.
    - An `ArrayList` of `PartyMember` objects. The `ArrayList` as well as the elements of the `ArrayList` is guaranteed to be non-null.
  - Returns a `boolean` value representing whether the write was successful.
  - To implement this, you should follow a read-modify-write design pattern. This pattern can be used to change the contents of a file in the following three steps:
    - Reading data from a file into an intermediate structure
    - Performing any necessary modifications to that data in the intermediate structure
    - Writing the data in the intermediate structure back out to the same file.

    This allows us to use familiar tools like `Scanner` and `PrintWriter` to change the contents of a file without modifying the contents of the file directly (which is much more difficult).
  - This method should first read from a file which will have all its lines formatted as the one outlined in the `recruitParty` method
    - **Hint #1:** Do we have code which does this already?
    - **Hint #2:** In case we are thrown an exception in doing so, should we always stop the method and throw it again? Remember that while we could have started with the file previously existing, it could have also been the case that it didn't and we are creating a new file.
  - With the contents now in an easier to modify data structure (ex. an `ArrayList`), add the new elements you took in as arguments into the back of the original contents.

    - When writing these contents back to the file, iterate through the data structure and write each `PartyMember` object to its own line.
        - Each `PartyMember` should be written in the format outlined in the `recruitParty` method.
    - Make sure to catch any exceptions required of you! Upon catching, think about the best way to inform a user about what happened. This could be in the form of a print statement. Remember we only want to return false if **NO** writing occurred.
- `getQuest()`
    - Here is where we shall see if our assembled party is up to the test, by sending them out on a selected quest.
    - Takes in two parameters:
        - A `String` representing the name of the quest to select. This is guaranteed to be non-null.
        - An `ArrayList` of `PartyMember` objects. This is guaranteed to be non-null.
    - To implement this, you should follow a read-modify-write design pattern similar to the one explained above.
    - This method should read from a file "quest.csv" which will have all its lines formatted as: "`<questName>: <questLevel>, <reward>`"
        - **Hint:** The `questName` may or may not have spaces in it! Keep that in mind when trying to parse each line.
    - Throws a `FileNotFoundException` if there is no file named "quest.csv".
    - Returns a `boolean` that represents whether our party has succeeded.
    - Iterate through quest.csv:
        - Search for the inputted quest's `questName` while building an `ArrayList` of each quest's information (i.e. each line).
        - Get the corresponding `questLevel` of the named quest if it exists.
        - Compare the `questLevel` with the sum of all the questLevel attributes of the `PartyMembers` in the `ArrayList`.
            - **Hint:** Use `partyQuestLevel()` to help calculate the questLevel of the whole party.
            - If the `questLevel` is less than the party's total questLevel, print "`Success! Your party gained <reward> coins. This calls for a trip to the Tavern!`"
            - Otherwise, print "`Failure... Your party was defeated. Better Luck Next Time!`"
    - If the inputted `questName` was not found, throw a `QuestNotFoundException`. You may choose the message it contains.
    - If the Party was successful, remove that quest from the quest.csv and return true.
    - If the Party was **NOT** successful, do **NOT** modify the quest.csv and return false.
- `partyQuestLevel()`
    - This private helper method will return the total quest level of all the party members.
    - Takes in an `ArrayList` of `PartyMember` objects to calculate the total quest level of.
    - Returns an `int` that represents the total quest level.
        - Return –1 if the party is null or empty.
    - The total quest level of all the `PartyMembers` can be calculated by summing up each of the `PartyMember`'s questLevels, determined by their attributes and class.
- `main()`

- Create three `Warrior` objects and three `Mage` objects
- Write these objects into a file called "TestParty.csv"
- Create another `PartyMember` object and add it to "TestParty.csv".
- Read this csv file using `recruitParty` and print each object to a new line
- Search for a quest and put your party up against it; let's see how well they stack up.
- These tests and the ones on Gradescope are by no means comprehensive, so be sure to create your own.
- Make sure to handle the exceptions dealt to the main method!

Checkstyle

You must run Checkstyle on your submission (to learn more about Checkstyle, check out cs1331-style-guide.pdf under the Checkstyle Resources module on Canvas). **The Checkstyle cap for this assignment is 35 points.** This means there is a maximum point deduction of 35. If you don't have Checkstyle yet, download it from Canvas → Modules → Checkstyle Resources → checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. 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 number of points we would take off (limited by the Checkstyle cap). In future assignments 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
```

For additional help with Checkstyle see the CS 1331 Style Guide.

Turn-In Procedure

Submission

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

- `InvalidPartyMemberException.java`
- `QuestNotFoundException.java`
- `Party.java`

Make sure you see the message stating the assignment was submitted successfully. From this point, Gradescope will run a basic autograder on your submission as discussed in the next section. **Any autograder tests are provided as a courtesy to help "sanity check" your work and you may not see all the test cases used to grade your work.** You are responsible for thoroughly testing your submission on

your own to ensure you have fulfilled the requirements of this assignment. If you have questions about the requirements given, reach out to a TA or Professor via the class forum for clarification.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the assignment. We will only grade your latest submission. **Be sure to submit every file each time you resubmit**.

Gradescope Autograder

If an autograder is enabled for this assignment, you may be able to see the results of a few basic test cases on your code. Typically, tests will correspond 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. **We reserve the right to hide any or all test cases, so you should make sure to test your code thoroughly against the assignment's requirements.**

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.

Burden of Testing

You are responsible for thoroughly testing your submission against the written requirements to ensure you have fulfilled the requirements of this assignment.

Be **very careful** to note the way in which text output is formatted and spelled. Minor discrepancies could result in failed autograder cases.

If you have questions about the requirements given, reach out to a TA or Professor via the class forum for clarification.

Allowed Imports

To prevent trivialization of the assignment, you are only allowed to import the following packages:

- `java.util.ArrayList`
- `java.util.Scanner`
- `java.io.File`
- `java.io.FileNotFoundException`
- `java.io.IOException`
- `java.io.PrintWriter`

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our autograder. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`
- `System.arraycopy`

Collaboration

Only discussion of the assignment at a conceptual high level is allowed. You can discuss course concepts and assignments broadly; that is, at a conceptual level to increase your understanding. If you find yourself dropping to a level where specific Java code is being discussed, that is going too far. Those discussions should be reserved for the instructor and TAs. To be clear, you should never exchange code related to an assignment with anyone other than the instructor and TAs.

The only code you may share are test cases written in a Driver class. If you choose to share your Driver class, they should be posted to the assignment discussion thread on the course discussion forum. We encourage you to write test cases and share them with your classmates, but we will not verify their correctness (i.e., use them at your own risk).

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 Ed Discussion for a note containing all official clarifications and sample outputs.**

It is expected that everyone will follow the Student-Faculty Expectations document and the Student Code of Conduct. The professor expects a **positive, respectful, and engaged academic environment** inside the classroom, outside the classroom, in all electronic communications, on all file submissions, and on any document submitted throughout the duration of the course. **No inappropriate language is to be used, and any assignment deemed by the professor to contain inappropriate offensive language or threats will get a zero.** You are to use professionalism in your work. Violations of this conduct policy will be turned over to the Office of Student Integrity for misconduct.