Assignment 1: The Basics of Object Orientation in Java

Pre-Assignment Notes

The **late penalty** is -10% per day late. After two days, late submissions won't be accepted.

This is an **individual assignment**, meaning you're not supposed to share or copy your solution/code from other students or the internet. There's still the class's Piazza if you want to ask questions to the group, or you can ask during lecture, send an email to the class's help account, or attend an office hour on Discord.

Note you can use any code shared by the instructor or anything from class or the tutorial videos linked on the course website. You *can* get outside help from guides and other people, just try to make sure they're teaching you how to solve the assignment, not writing the code for you. You won't learn anything that way, and they can't write your exam or midterm for you. If in doubt about whether something is plagiarism or not, don't be afraid to reach out and ask!

1. Setting Up Java

One of the goals of this assignment is to shake out any problems with our coding setups. As such, it's highly recommended you use the **IntelliJ IDE**, which provides a fairly standardized Java coding environment that cuts down on the risk that your code won't run for the TAs.

Install IntelliJ Community Edition (don't worry, it's free) :
https://www.jetbrains.com/idea/download/

Also be sure you're using Java 14 under File -> Project Structure, you may need to download a new SDK in order to have it available! 15 *should* be compatible as well, though testing if that's true is one of the purposes of a good technical shakedown.

2. Assignment Outline

For this assignment, we'll be taking on the role of a video game programmer working as part of a team. You've been tasked with implementing part of the designer's vision for how the game should be structured, which might be a little tricky because the designer is not a programmer themselves.

As such, they've described below a number of structures and actions they want the game system to be capable of, many of which will be used by your fellow team members in their own tasks. It's up to you how best to structure your Java code – while certain functions, variables, and relationships are heavily hinted at in this outline, part of the challenge is recognizing where they apply.

Your submitted code will not just be marked on whether it compiles, runs, and works, but also on whether its design follows the principles of object-oriented design, and makes best use of the data structures and algorithms we've learned about so far. Coding style, brevity, clarity, and documentation are all also valid grading criteria!

2.1 The Different Classes Of Actor

Your game will include a number of characters – though to avoid confusion with the 'char' data type, let's call them **Actors**. Your main responsibility is setting up the data structures that other developers will use when filling up the game world with interesting people for the player to meet and battle against. As such, you'll need to create a number of classes and functions to implement these Actors according to the designer's wishes.

All Actors must belong to one (and ony one) of these four types:
-The **Player Actor** is the character in the game controlled by the player. They can fight some Actors and talk to some others.
-**Friendly Actors** are allied characters controlled by the computer. They can talk with the player or with neutrals, and will fight alongside the player against monsters.
-**Monster Actors** are enemy characters controlled by the computer. They will not speak, but will attack the player and their allies.
-**Neutral Actors** are background characters controlled by the computer. They can also talk with the player and neutrals, but they don't do any fighting.

All Actors will have
-A *name*, which is a String used to identify them. Actors should have a placeholder name by default based on their type.
-A *level*, which is an integer that tracks how powerful they are. By default, all Actors start at level 1.
-A level up function that increases the Actor's level by one.
-A function to announce themselves, which involves printing a message with their name and level to the screen. This message should be different depending on their type.

Those Actors who can fight should also have a method for Fight, which accepts another Actor as a target. This function compares the levels of the Actor and their target, and then posts a message to the screen declaring a winner (be sure to mention both their names!). If a Player or a Friendly Actor wins, they should gain a level. Note that Friendly and Player Actors only fight Monsters, while Monsters only fight Friendly and Player Actors, and nobody fights Neutrals. If an Actor receives an invalid target, it should say so in a message!

Those Actors who can speak, meanwhile, should have a Conversation function that takes in another Actor as a subject. The conversation involves printing a different message to the screen depending on the type of Actor that is speaking and the type that they are speaking to, and the printed message should include both of their names.

There are also a few small differences to the common methods based on the Actor's type:
-Whenever a Player Actor levels up, a message should also be printed informing the player that their Actor has just gained a level, printing the new total.
-Monster Actors have a second way to announce themselves that accepts another Actor as a target. If their target's level is lower than theirs, then the message is a challenge, while if it is equal or higher then the message is much less confident.

2.2 An Array Of Cast Members

The designer would also like to store an array containing all of the Actors defined so far. This object would act as the game's **Cast**[1], giving your team somewhere to keep all of the Actors they develop. The Cast should therefore have a method that can accept a new Actor, even if that involves increasing the size of the array to make room for them. The designer has asked that the Actors be stored alphabetically according to their name, so new Actors shouldn't be added on to the end of the array, but rather inserted among the other Actors.

The designer has also declared a rule that no two Actors should have the same name. Therefore, if someone tries to add an Actor with the same name as one already stored in the Cast, that should be treated as a user error and the new Actor should not be added.

This allows us to include a method for removing Actors from the Cast by using their name. The array of Actors should shrink to match the reduced size while also staying in alphabetical order. If no Actor by that name can be found to be removed, a message saying as much should be printed to the screen.

Finally, the Cast should have two methods for printing the names and levels of stored Actors to the screen. One should print all of them, in order, while the other should print the top five Actors ranked according to their level.

2.3 Demo

Since the designer wouldn't be able to tell what you've implemented by looking at the code, you need to create a demonstration. You need not create an entire game, don't worry, just a simple text-based command-line demo that will exercise your different functions and objects to persuade the designer that you've followed their instructions. Include a Main class that has a “public static void main (String args[])” function that does all of its printing to the terminal, which we'll be able to run using IntelliJ's 'Run' command.

During this demo, you should create a Cast, and then load it with one Player Actor and three instances of each other kind of Actor (set their names and levels as you see fit, but a variety would help).

After that, some things that the designer wants to see in your demo include:
-Have the Cast announce themselves in alphabetical order.
-Print the top five highest-leveled Actors in order.
-Have an Actor of each type attempt to converse with and/or fight an Actor of every other type.
-Add a new Actor to the Cast with a name that's alphabetically in the middle of the rest.
-Try adding a new Actor to the Cast with the name of an existing Actor.
-Remove an Actor from the Cast using their name.
-Try removing an Actor from the Cast using a name that isn't in the Cast.
-Have an Actor win enough fights to increase their level such that they increase their ranking in the top five by a spot, then print the top five again.
-Have the Cast announce themselves in alphabetical order again to close.

---

1 Cast like a movie's cast, not the concept of Casting we discussed in class – cursed ambiguous English language!

...And that's all you need to do! This assignment is primarily concerned with getting you up to speed with the basics and to figure out the workflow that we'll use for the rest of the course, so don't worry if it seemed a little straightforward this time.

3. Deliverables

All assignment submissions will be done through CourSys at https://courses.cs.sfu.ca/. For this assignment, that should only include:

1. A .zip archive of your code, organized into a single Java project. To ensure compatibility, it's recommended to use IntelliJ while developing your code, then using the export function under File -> Export -> Project To .zip File.

Be sure you tidy up your code before submitting! Check the Java code style guide posted on the course website, and do your best to provide helpful comments.