Programming 1

Aryobarzan Atashpendar (aryobarzan.atashpendar@uni.lu)

Lab 9 – Inheritance, Generics, Enums, Records

Submission (Preliminary: 28/11, Final: 5/12)
The mandatory exercises for **Code Submission** are **1, 2** (marked with an ❗). All other exercises on this sheet are optional but still highly recommended! The **Explainer Video** 🎥 for this sheet must be realized on **Exercise 2 ("Recording game events")**. A Flipgrid invitation link will be posted on Moodle.

Before you start: the following exercise encompasses many of the concepts you have seen thus far into a single use case. However, don't be scared by the size - the exercise guides you step-wise as to what you need to implement.

Figure 1: Good luck! ;) (© Square Enix)

Exercise 1 – ❗ Inventory

#Polymorphism #Enum #EnumConstant #GenericType #GenericParameter

The developers of the video game series Final Fantasy are working on their next game and they require a flexible inventory.
The "inventory" is a well-known concept in video games where it acts as the storage of the player as it holds the items they gather throughout their adventures.

For this specific entry in the Final Fantasy series, a few requirements need to be satisfied for the inventory:

1° All **Item**s carry a *name*, a *value* denoting how much they cost, and a *rarity*. Item rarity indicates how unique a given item is. In the case of Final Fantasy, the rarity is indicated by the following colors:

- White, Pink, Green, Blue, Purple

2° Two **Item**s are considered equal if they have the same name and rarity.

3° There are three distinct types of items: **Trash** Items, **Gear** Items and **Consumable** Items.

- **Trash Items** have no extra properties and are just deemed filler items which do nothing.
- **Gear Items** make the player stronger, i.e. they hold certain statistics: *critical hit*, *determination* and *direct hit*. As an example, a "Helmet" gear item can hold the statistics 5/4/2, i.e. it increases the player's critical hit by 5, determination by 4 and direct hit by 2. These properties must be positive or 0.
- **Consumable Items** are usable items, e.g. a "potion" item can be drunk by the player. However, consumables have a limited number of *charges*, i.e. once the item has been used multiple items, the player will run out of *charges* and will no longer be able to use that item.

4° Your **Inventory** has a fixed *capacity*, i.e. a maximum number of items it can hold.

5° Implement the following methods for your **Inventory**:

- **boolean add(Item item)**: if the inventory is not full, add the item and return true. Otherwise, return false.

  Note: your inventory can hold duplicates, i.e. the same item can be added more than once to the inventory - choose the structure to hold your items appropriately based on this property.
- **int drop(Item item)**: Remove all instances of the given item from your inventory and return the number of items that were removed. (remember, duplicates of the same item are allowed)
- **boolean drop(int index)**: Remove the item at the indicated index if it exists and return true. Otherwise, return false.
- **void use(int index)**: An item can be used on its own, i.e. it has its own **use** method. However, an item can also be used through the inventory, in which case the behavior slightly changes: if a consumable item is used through the **Inventory**'s **use** method, it should automatically be removed from the inventory if it has no *charges* left after its usage.
- **boolean sell(int index)**: Return true if the item at the indicated index in your inventory can be sold and remove the item from your inventory.

  Note: only trash and consumable items can be sold, not gear items.

  Tip: remember the concept of "marker interfaces" and the method "instanceof". (Scrapbook)

6° **(Optional/Bonus)** Override the **compareTo** method for your **Item** class(es):

- **Item**s are normally compared to each other based on their *value*.
- **Gear Item**s are compared based on their statistics, i.e. if the sum of item A's *critical hit*, *determination* and *direct hit* are greater than item B's, item A is considered greater than B.

Tip: Before you can override the **compareTo** method, your class must implement the interface Comparable<YourClassName>

7° **(Optional/Bonus)** Your inventory can be sorted based on either the *name* of the items or their *value*, and you can also indicate whether the items should be sorted in ascending or descending order. Implement a corresponding method for the Inventory.

Tip: Look up the method sort for Lists, reverse for Collections and the method comparing for the Comparator class.

8° Override the **toString** method for your classes and add appropriate console messages for your various methods so they match the sample outputs given below.

9° Test your implementation with a main method where you experiment with gear items, consumable items, printing the inventory after adding/dropping items, etc. (see below for sample outputs)

Design your code based on efficiency, by keeping in mind that in the future, new item types might be introduced other than gear and consumable items, which may also exhibit different behavior when used. Also, make sure all your variables are always private and only allow specific access based on getter and setter methods. And don't forget sanity checking wherever needed! :)

If you are completely lost, here is a sequence of words which indicate what you **could** be using for your implementation: Inheritance, Polymorphism, Generics, Abstract Type, Enum, List, equals and hashCode, Comparable, Marker Interface and instanceof

Sample output when printing the Inventory:

Inventory: (2)
{0} [PURPLE] Better Helmet (250 Gold) - 10 Critical Hit / 6 Determination / 8 Direct Hit
{1} [WHITE] Mana Potion (35 Gold) - 2 charges

More sample output: (dropping, adding, using, selling)

Dropped 2 instances of item: [BLUE] Helmet (150 Gold) - 5 Critical Hit / 3 Determination / 4 Direct Hit

Added item to inventory: [BLUE] Helmet (150 Gold) - 5 Critical Hit / 3 Determination / 4 Direct Hit

Used the consumable item Health Potion. (1 charges left)
Used the consumable item Health Potion. (0 charges left)
The consumable item Health Potion has no charges left.
The gear item Helmet can not be used.

Used item has no charges left and has been dropped: [WHITE] Mana Potion (35 Gold) - 0 charges

The following item cannot be sold: [PURPLE] Better Helmet (250 Gold) - 10 Critical Hit / 6 Determination / 8 Direct Hit
Item sold: [WHITE] Health Potion (50 Gold) - 0 charges
Item sold: [WHITE] Garbage (1 Gold)

Exercise 2 – ❗🎥 Recording game events #Inheritance #Interface

For online games, it is important to keep a history of various events that occur when people play their game. This stems from the fact that their actions are always transmitted to a server, meaning the developers need an exact record of events that they can store.

To that end, the developers of Final Fantasy want to add event logging to the usage of actions by player characters:

1° Implement a class **Character** which holds a *name*, a *maximum health* counter and a *current health* counter. This represents a player's character.

2° Design class(es) for the various **Action**s that a player character can perform: (note that every action has a *user*, i.e. the character executing it)

- "**Jump**" action.
- "**Ability**" action which has a *name*.
- "**Targetable**" ability action which can additionally hold the *target* character that the action should be performed on, and a *value* indicating the power of the ability (can be either positive or negative).
  Furthermore, a player can also perform certain "targetable" abilities on their own character rather than strictly on others.

3° An action can be **execute**d, though its behavior differs depending on the type of the action. Most importantly, **Targetable** abilities require a *target* character to be executed on. Make sure to appropriately modify the *health* of the target when a targetable ability is used on them!

4° Design a unified class for creating logs, storing them and printing them all to the console. Note that the game only ever needs a **single** instance of this logger...

5° Record two types of events using your previously designed logger:

a) Record whenever an action is used, including the date, the character using it and possibly the target of that action. Also record whether the action could be successfully executed or not, e.g. if the target for a targetable ability action was not specified, record that the action failed.

b) Record whenever a character kills another character and whenever a character kills themself.

Note: the developer should not have to manually check whenever an event has occurred to log it. Instead, have actions create a record whenever they are executed.

6° Write a sample main method testing your implementation with every type of action and at least 2 characters, with the logs being printed to the console.

Exercise 3 – Green apples and red apples

Final Fantasy games have this concept of the player evolving in their role: for example, if they were initially a Mage, they can become stronger through their adventures and turn into a specialized Red Mage.

Naturally, the previous developers opted for inheritance, where **Mage** is the base class and class **Red Mage** inherits from it. The problem is that they want the game to recognize that an instance of **Mage** and an instance of **Red Mage** should be considered equal when comparing them with the **equals** method that they have overriden in the base class **Mage**.

However, when they compare two such instances, the **equals** method always returns false, even though the comparison should only check if the field *name* present in the base class **Mage** is the same for both objects. Additional fields introduced in the subclass **Red Mage** are ignored for the equals comparison and are not deemed important.

Can you find the problem in the **equals** implementation of class **Mage** and possibly solve it? Explain!
To reiterate, your solution should lead to an instance of **Mage** and an instance of **Red Mage** being considered equal when they have the same value for the *name* field.

Figure 2: You've made it! (... right?) (© Square Enix)