

Submission (Preliminary: 5/12, Final: 12/12)

The mandatory exercises for **Code Submission** are **1, 3** (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 1 (Party Composition)**. A Flipgrid invitation link will be posted on Moodle.



Figure 1: This lab shouldn't be that hard...
(*Neon Genesis Evangelion* © Gainax and Tatsunoko)

Exercise 1 – ! 🎥 Party Composition

[#ExceptionHandling](#) [#CheckedException](#)

In Final Fantasy, your **Character** has a specific role: *Damage*, *Tank* or *Healer*.

These roles become important when grouping multiple Characters into a so-called **Party**, which has certain limits:

- 1° A **Party** can contain up to **4** Characters.
- 2° The allowed **Party** composition is as follows:
 - 1 Character of role *Healer*
 - 1 Character of role *Tank*
 - 2 Characters of role *Damage*

Update the existing **Party** class by implementing its **add** method while respecting the previously indicated requirements.

Afterwards, implement a sample main method in the **Launcher** class which reads input from the console. Importantly, include proper **exception handling** for the various aspects of your program.

- 1° Initialize two empty **Party** instances.

2° The program will continuously ask for input from the user:

- '0' - Exit the program
- '1' - Create a new character
 - Ask for the **Character's name**
 - It can't be empty.
 - Its length can't exceed 20 characters.
 - Ask for the **Character's role**
 - 'DAMAGE'
 - 'TANK'
 - 'HEALER'

Note: **CharacterRole** is an Enum structure, so you will need to convert the user's String input of the role name somehow...

- Ask for the **Party** to which the **Character** should be added ('0' or '1')
- '2' - Choose a **Party** to view. (print the characters contained in the **Party** to console)
 - Ask for the **Party** to show: '0' or '1'

Your exception handling should cover every aspect of your program, i.e. throw an exception for each of the following aspects AND catch them accordingly:

- Invalid **Character name** (empty / too long)
- Input type mismatch, e.g. you expect an int as input but the user enters a String
- Invalid *role* name ('DAMAGE' or 'HEALER' or 'TANK')
- Invalid index for the **Party** (0 or 1)
- Trying to add a **Character** to an already full **Party**
- Trying to add a **Character** with a *Role* which is invalid for the given **Party** (e.g. if the **Party** already contains a *Healer*, do not allow the inclusion of a second *Healer*) - remember the **Party** composition rules indicated above
- Trying to add a **Character** to a **Party** when it is already in that **Party**

(Tip) 'throws' keyword: When indicating a method's signature, e.g. its return type, parameters and so on, you can also indicate if the method can throw specific Exception(s) by using the keyword 'throws'.

For example: `'public void delete(int index) throws IndexOutOfBoundsException, EmptyListException {...}'`.

Also, remember that a constructor is just a special method - meaning it too can throw Exception(s) if you so wish. ;)

Remember, do not perform any sanity checking or return a status code from your methods: rely exclusively on the **'try - catch'** block and **throwing / catching Exceptions**. You are allowed to implement your own custom **Exception** classes for certain aspects of your program, though you can also rely on existing **Exception** classes in some cases, e.g. **InputMismatchException**.

Also, parts of the code are already provided (Character, CharacterRole, Party, Launcher), but you may (and should) edit any part of these classes, e.g. the constructor.

You can view a full sample output given in the included file 'SampleOutput.txt' to get an idea of how the program should work.



Figure 2: Eva Unit-01
(*Neon Genesis Evangelion* © Gainax and Tatsunoko)

Exercise 2 – Evangelion

#Generics

In the anime/manga series *Neon Genesis Evangelion*, humans use humanoid-like robots called Evas to defend their planet from the danger of so-called Angels.

To function, an **Eva Unit** requires a **Pilot**, which can either be a **Human Pilot** or an automated **Dummy Pilot**. However, **by design they only ever accept one type** of **Pilot**, e.g. Eva Unit-01 can only be piloted by a human.

Bounded Type parameter:

To solve this exercise, you may have to use a bounded-type parameter for your Eva Unit's class declaration. Bounded-type parameter is a concept related to Generics which you can review [here](#) along with an example.

Hint: Your class declaration may look like 'public class EvaUnit<T extends Pilot>'

- **Pilots** have a *name* and a *compatibility* number ranging from 0 to 100 (included).
 - **Dummy Pilots** have a less stable *compatibility*, meaning whatever their intended *compatibility* value might be, it can randomly be reduced by an amount ranging from 0 to 5.
- **Human Pilots** have a *sanity* level which always starts at a maximum of 100.

On the other hand, an **Eva Unit** has the following properties:

- A *unit* number which is positive.
- A *pilot*.
- A *synchronization level* which can range from 0 to 100 (included).
- A *status* indicating whether the unit is on or off. (by default: off)

An **Eva Unit** can perform two types of actions:

1° **Start**: if the unit has a *pilot*, the **Eva Unit** will try to **synchronize** with the *pilot*.

- A *stability* is computed based on the difference between the **Eva Unit**'s *synchronization level* and the **Pilot**'s *compatibility*.
- A random number is rolled between 0 and 100 (included). If the roll is less than or equal to the *stability*, the **Eva Unit** starts. Otherwise, it fails to start.
 - **Human Pilots** have a special behavior when trying to start an **Eva Unit**, namely that their *sanity* drops by 5. If their *sanity* is 0, they can no longer perform any actions for their **Eva Unit**.

2° **Maneuver**: An **Eva Unit** can perform a maneuver when facing an **Obstacle**, which can either be an immobile **Object** or an **Angel**. The **Eva Unit** can only maneuver if it is started (on) and the *pilot* rolls a random number (0 to 100 included) higher than or equal to 50.

- **Obstacles** have a *mental toll* property, which is always 0 for **Object Obstacles**, but which can range from 0 to 100 for an **Angel Obstacle**.
- When a **Human Pilot** tries to dodge an **Obstacle**, i.e. roll a number from 0 to 100, their *sanity* will drop by an amount equal to the *mental toll* of the **Obstacle**. If their *sanity* is 0, they can not dodge, i.e. their roll will always be 0. Otherwise, their roll is always increased by a fixed value of 20, e.g. if they roll a 45, it will be increased to 65.
- **Dummy Pilots** do not have this advantage of an increased roll, though they also do not have a *sanity* level.

Write a sample launcher to test your implementation with different types of **Eva Units** piloted by **Human** and **Dummy Pilots**, who try to start their respective **Eva Units** and maneuver various **Obstacles** (**Object Obstacles** or **Angel Obstacles**).

Sample output:

```
Eva Unit-01 (50 sync) can't start: no pilot present.
Eva Unit-01 (50 sync) is now piloted by (Human) Shinji (50 compatibility) (100 sanity)
Start sequence for Eva Unit-01 (50 sync): Started!
Eva Unit-01 (50 sync) dodged Object Obstacle (0)
Eva Unit-01 (50 sync) failed to dodge Angel Obstacle (25)
Eva Unit-01 (50 sync) dodged Angel Obstacle (25)
Eva Unit-01 (50 sync) dodged Angel Obstacle (25)
(Human) Shinji (50 compatibility) (0 sanity) has lost their sanity.
Eva Unit-01 (50 sync) failed to dodge Angel Obstacle (25)
Eva Unit-02 (78 sync) is now piloted by (Dummy) Dummy (87 compatibility)
Start sequence for Eva Unit-02 (78 sync): Started!
Eva Unit-02 (78 sync) dodged Object Obstacle (0)
Eva Unit-02 (78 sync) dodged Angel Obstacle (25)
Eva Unit-03 (100 sync) is now piloted by (Human) Asuka (5 compatibility) (100 sanity)
Start sequence for Eva Unit-03 (100 sync): Failed!
```

Exercise 3 – ! Steam

#Inheritance #ExceptionHandling

The game store and client Steam allows **Users** to own **Software**, which they can install to their **Disk**.

Steam supports two types of **Software**, namely **Games** and **Tools**.

- A **Game** has a *license*, which is a unique ID. A **Game** can only be installed to a **Disk** if the **User** of that **Disk** owns the *license* for the **Game**. Throw a corresponding exception which has variables for the **Game** and **User**.
- A **Tool** can only be installed to a **Disk** which is not running the *MacOS* operating system. Throw a corresponding exception which has variables for the **Tool** and **Disk**.

Thus, your task is to create two new classes, **Game** and **Tool**, which both inherit from the provided class **Software**. Modify their behavior for the **install()** method according to the above requirements. Do not modify any of the provided files (**Software**, **Disk**, **User**, **InstallStatus**, **OperatingSystem**).

Test your implementation to make sure it works properly, though you don't need to provide a sample launcher for this exercise. (And check out the hint on the next page!)

Exceptions and inheritance:

In practice, whenever you want to define a custom exception class, you always inherit from the base class `Exception` provided by the standard Java library.

However, something you may not have known is that inheritance also works at the level of method signatures! This means that if you have a method whose signature is `"void myMethod() throws Exception"`, the method's body cannot only throw instances of the base class `Exception`, but also... ;)



Figure 3: You've made it again!
(*Final Fantasy* © Square Enix)