**Data Structures Advanced with Java Exam - 16 May 2020**

This document defines the examination problems for ["Data Structures – Fundamentals (Java)" course @ Software University](https://softuni.bg/trainings/2812/data-structures-fundamentals-with-java-march-2020).

Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2440/Data-Structures-Advanced-with-Java-Exam-16-May-2020).

Write Java code for solving the tasks on the following pages. Code should compile under the Java 8 and above standards you can write and locally test your solution with the Java 13 standard, however **Judge will run the submission with Java 10 JRE**. Avoid submissions with **features included after Java 10** release doing **otherwise** will result in **compile time error**.

Some **tests may be provided** within the skeleton – use those for local **testing and debugging**, however there **is no guarantee that there are no hidden tests added inside Judge**.

Please follow the exact instructions on uploading the solutions for each task. Submit as **.zip archive** the files contained inside **"...\src\main\java"** folder this should work for all tasks regardless of current DS implementation.

In order for the solution to compile the tests **successfully** the project **must** have **single** **Main.java** file containing single **public static void main(String[] args)** method even empty one within the **Main class**.

You have to **study** the provided **skeleton**. The code is **separated** inside **different** **packages**, for you tasks you should be writing code **mainly** **inside** **the "core" package**.

There **are** **few** **entities** **inside** the **project** you are **allowed** to **add** code to those, also you have to study the classes provided. There is no restriction to the code added inside all the classes provided however adding new public method to interfaces will not work inside the tests since those methods will never be called.

There are two types of problems you have to solve two problems where you will need correct implementation and no performance will be measured that is for problems **1. Hero Database** and **2. Galaxy each problem gives 100 points.** The other two problems 1.5 and 2.5 they are performance tests to the previous two problems each one gives 50 points so total of 100 for the both, at the end the max points you can get are 300.

# Hero Database

You are given a skeleton with a class **HeroDatabaseImpl** that implements the **HeroDatabase interface.**

This database works with **Hero** entities, all heroes are identified by their **unique** **names** (there will **not** **be** **two** heroes with the same names). Implements all the operations from the **interface**:

* **void addHero(Hero hero)** – **adds** hero to the database if there **is** **hero** **with** **the** **same** **name** added before throw IllegalArgumentException()
* **boolean contains(Hero hero)** – **returns** whether the hero **is** **present** inside the database **or** **not**
* **int size() –** returns the **number** of heroes stored
* **Hero getHero(String name) –** **returns** the **hero** by the **name** specified, if there is **no** such hero throw IllegalArgumentException()
* **Hero remove(String name)** – **removes** the hero by the name specified and returns it – if such hero exists, **otherwise** throw IllegalArgumentException()
* **Iterable<Hero> removeAllByType(HeroType type)** – removes all the hero with the type specified and returns them if there are no heroes of this type return an empty collection. There is **no** **requirement** for any ordering.
* **void levelUp(String name)** – **increases** the level of the hero with the name specified by **one**, If the hero is **not** **present** throw IllegalArgumentException()
* **void rename(String oldName, String newName)** – this method should **change** the hero's name equal to the **oldName** with the **newName**, however if there is hero with the newName or there is no hero with the oldName you should throw **IllegalArgumentException()**
* **Iterable<Hero> getAllByType(HeroType type)** – returns all the heroes with the **type** **specified** **ordered** by **name** if there are none return an empty collection
* **Iterable<Hero> getAllByLevel(int level)** – returns all the heroes with the **level** **specified** **ordered** by **name** if there are none return an empty collection
* **Iterable<Hero> getInPointsRange(int lowerBound, int upperBound)** – returns the **heroes** with **points** in the range specified the lower bound is **inclusive** the upper bound is **exclusive** the result is ordered by hero **points** **descending** then by **level** **ascending**. If there are none return an empty collection
* **Iterable<Hero> getAllOrderedByLevelDescendingThenByName()** – returns all heroes ordered by level **descending** then by name **ascending**. If there are none return an empty collection
  1. **Hero Database – Performance**

For this task you will only be required to submit the **code from the previous problem**. If you are having problem with this task you should **perform detailed algorithmic complexity analysis**, and try to **figure** **out** **weak** spots inside your implementation.

For this problem it is important that other operations are **implemented** **correctly** according to the specific problems: **add**, **size**, **remove**, **get** etc…

You can submit code to this problem **without full coverage** from the previous problem, **not all test cases** will be considered only the **general** **behaviour** will be considered important, **edge** **cases** will mostly be ignored such as throwing exceptions etc…

## Galaxy

You are given a skeleton with a class **MilkyWay** that implements the **Galaxy interface.**

This class stores two types of entities **Stars** and **Planets**, those entities are identified by their **unique** **ids** (there will **not** **be** **two** stars and planets with the same ids). Implements all the operations from the **interface**:

* **void add(Star star) –** adds **star** to the **galaxy** if the star is **previously** **stored** throw IllegalArgumentException()
* **void add(Planet planet, Star star) –** adds **planet** to the **galaxy**. The planets are **orbiting** stars so you have to store the planet in some sort of **relation** to its star, it is like we know the star and we discover new planet so we add the planet as another planetary object. There are a couple tricky things to be considered if the **star** is **not** previously added to the galaxy **or** the **planet** is **already** added throw IllegalArgumentException()
* **boolean contains(Planet planet)** – returns weather the **planet** is **stored** inside the galaxy or **not**
* **boolean contains(Star star)** – returns weather the **star** is **stored** inside the galaxy or **not**
* **Star getStar(int id)** – returns the **star** with the **id** specified if there is no such star throw IllegalArgumentException()
* **Planet getPlanet(int id)** – returns the **planet** with the **id** specified if there is no such star throw IllegalArgumentException()
* **Star removeStar(int id)** – removes the **star** and its **planets** form the galaxy and **returns** **only** the **star** if there is no such star throw IllegalArgumentException()
* **Planet removePlanet(int id)** – removes the **planet** form the galaxy and **returns** **it** if there is no such planet throw IllegalArgumentException()
* **int countObjects()** – returns the number of **stars** **and** **planets** stored
* **Iterable<Planet> getPlanetsByStar(Star star)** – returns the **planets** **orbiting** the star specified **ordered** by **distance** from the star and **then** by **mass**, if there are none planets return an empty collection
* **Iterable<Star> getStars()** – returns the stars **ordered** by **luminosity** **descending** if there are **none** return an empty collection
* **Iterable<Star> getStarsInOrderOfDiscovery()** – returns all the **stars** in **order** of **discovery**/**insertion** if there are none return an empty collection
* **Iterable<Planet> getAllPlanetsByMass()** – returns all the **planets** **ordered** by **mass** in **descending** order if there are none return an empty collection
* **Iterable<Planet> getAllPlanetsByDistanceFromStar(Star star)** – returns the **planets** orbiting the star specified **ordered** **only** by the **distance** from the star
* **Map<Star, Set<Planet>> getStarsAndPlanetsByOrderOfStarDiscoveryAndPlanetDistanceFromStarThenByPlanetMass()** – returns a **map** with **stars** as a **keys** and **set** of **planets** orbiting where stars are ordered in order of **discovery**/**insertion** and the planets are **ordered** by **distance** from the star and **then** by **mass**
  1. **Galaxy – Performance**

For this task you will only be required to submit the **code from the previous problem**. If you are having problem with this task you should **perform detailed algorithmic complexity analysis**, and try to **figure** **out** **weak** spots inside your implementation.

For this problem it is important that other operations are **implemented** **correctly** according to the specific problems: **add**, **size**, **remove**, **get** etc…

You can submit code to this problem **without full coverage** from the previous problem, **not all test cases** will be considered only the **general** **behaviour** will be considered important, **edge** **cases** will mostly be ignored such as throwing exceptions etc…

“A good bookshop is just a genteel Black Hole that knows how to read.”― Terry Pratchett, Guards! Guards!