# Exercise: Hash Tables Sets and Maps

This document defines the lab for ["Data Structures – Advanced (Java)" course @ Software University](https://softuni.bg/trainings/4276/data-structures-advanced-with-java-september-2023). Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2429/06-Hash-Tables-Sets-and-Maps-Exercise).

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 the Java 10** release doing **otherwise** will result in a **compile time error**.

Any code files that are part of the task are provided as **Skeleton**. In the beginning, import the project skeleton, do not change any of the interfaces or classes provided. You are free to add additional logic in the form of methods in both interfaces and implementations you are not allowed to delete or remove any of the code provided. Do not change the names of the files as they are part of the test logic. **Do not change the packages** or move any of the files provided inside the skeleton if you have to add a new file add it in the same package of usage.

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 for 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 the current DS implementation.

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

Some of the problems will have simple **Benchmark** **tests** inside the skeleton. You can try to run those with **different** **values** and **different** **implementations** in order to **observe** behavior. However, **keep** in mind that the result comes **only as numbers** and this data may be **misleading** in some situations. Also, the tests are not started from the command prompt which may **influence** the **accuracy** of the results. Those tests are only added as an **example** of **different** **data** **structures** **performance** on their **common** operations.

## Royale Arena

You have been invited by the blue king himself. He is in a huge need for an application to help him fight in the glorious Royale Arena. As one of the top players in the arena, he has a reputation to maintain, therefore he cannot afford to lose. The application must be able to keep and query all the cards that exist within the game. Each card has its own specifics. The trick is that the intervals between the battles are so small, that the application must be fast in order to help the king.

**Battlecard** will hold:

* int id – unique card id
* CardType type – enumeration of battlecard
* String name – the name of the card
* double damage – the damage to the card
* double swag – the swag of the card

You need to support the following operations (and they should be **fast**):

* add() – Add a battle card to the arena. You will need to implement the contains() methods as well.
* contains(Battlecard) – checks if a given battlecard is present in the arena.
* count() – returns the number of cards in the arena
* changeCardType(id, type) – changes the status of the battlecard with the given id or throws IllegalArgumentException if no such card exists.
* getById(id) – return the card with the given id. If such a card doesn't exist, throw UnsupportedOperationException.
* removeById(id) – Removes the card with the given id, otherwise throws UnsupportedOperationException
* getByCardType(type) – Returns the cards with the given card type ordered by damage descending then by id. If there are no cards with the given type, throw UnsupportedOperationException
* getByTypeAndDamageRangeOrderedByDamageThenById(type, lo, hi) – returns all cards with particular card type ordered by damage descending (exclusive), then by id ascending. If there are no such types throw UnsupportedOperationException
* getByCardTypeAndMaximumDamage(type, damage) – returns all cards with a given type and damage less or equal to a maximum allowed amount ordered by damage descending, then by id. Throws an UnsupportedOperationException if such cards are not found.
* getByNameOrderedBySwagDescending(name) – search for all cards with a specific name and return them ordered by swag descending then by id. If there are no such cards throw UnsupportedOperationException
* getByNameAndSwagRange(name, lo, hi) – returns all cards with given name and swag between lo (inclusive) and hi (exclusive) ordered by swag descending then by id. If there are no such cards throw UnsupportedOperationException.
* getAllByNameAndSwag() – Returns the most swaggish card for each name. Returns an empty collection if none are found
* findFirstLeastSwag(n) – Returns the first n cards with the least swag. If there are two identical swags, order by id. If the argument passed exceeds the count of the arena, throw **UnsupportedOperationException**
* **getAllInSwagRange(lo, hi)** – returns all cards within a range ordered by swag(the range is inclusive). Returns an empty collection if no such cards were found.
* **iterator()** – Iterate the arena by insertion order

**Restrictions:**

* You are **not allowed to change the interface**.
* You can add to the Battlecard class but don't remove anything.
* You can edit the RoyaleArena class if it implements the IArena interface.

## Word Cruncher

Write a program that receives some **strings** and **forms another** string that is required. On the **first line,** you will be given **all of the strings** separated by **comma and space**. On the next line, you will be given the **string** that needs to be **formed from the given strings**. For more clarification see the examples below.

### Input

* On the first line you will receive the **strings** (separated by a comma and space **", "**).
* On the next line you will receive the **target string.**

### Output

* Print each of the found ways to form the required string as shown in the examples.

### Constraints

* There might be **repeating elements** in the input.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| text, me, so, do, m, ran  somerandomtext | so me ran do m text |
| this, th, is, Word, cruncher, cr, h, unch, c, r, un, ch, er  Wordcruncher | Word c r un ch er  Word c r unch er  Word cr un c h er  Word cr un ch er  Word cr unch er  Word cruncher |

***"****It's vital to remember who you really are. It's very important. It isn't a good idea to rely on other people or things to do it for you, you see. They always get it wrong.****"*** *– Terry Pratchett, "Sourcery"*