**Data Structures Advanced with Java Exam - 23 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/2448/Data-Structures-Advanced-with-Java-Retake-Exam-23-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. Book Land** and **2. Race Manager each problem gives 100 points.** The other two problems 1.5 and 2.5 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 is at total of 300.

# Book Land

You are given a skeleton with a class **BookLand** that implements the **BookStore interface.**

This class works with **Book** entities, all Books are identified by their **unique** **keys** (there will **not** **be** **two** books with the same keys). The key for a book is created by combining the author string and the title string where the separator is single symbol **"\_"**.

For example if we have author **"Terry Pratchett"** and title **"Mort"** the result **key** will be **"Terry Pratchett\_Mort",** you can see the code inside the Book class.

Implements all the operations from the **interface**:

* **void add(Book book)** – adds new book to the store if the book is **not** **present** **otherwise** throw IllegalArgumentException
* **boolean contains(Book book)** – returns **whether** the book is present inside the store or not true or false
* **int size()** – returns the count of books stored
* **Book getBook(String key)** – **gets** the book stored without removing it with the **key** **value** if present **otherwise** throws IllegalArgumentException
* **Book remove(String key)** – **removes** the book stored and **returns** **it** with the key value if present **otherwise** throws IllegalArgumentException
* **Collection<Book> removeSold()** – removes the book which are **sold** and returns them. No order is required for this operation, the books should be removed. If there are **no sold** books return an **empty** collection
* **void sellBook(String key)** – finds the book with the key and **sets** it's sold field to **true** if present **otherwise** throws IllegalArgumentException
* **void replace(Book oldBook, Book newBook) – replaces** the **old** book with the **new** one inside the store. If the **old** book is **not** present or the **new** book **is** present throw UnsupportedOperationException
* **Collection<Book> getAllByAuthor(String author)** – **returns** all the books with the **author** specified **sorted** **by** **price** ascending and then **by** **title** ascending, if there is **no** books with this author return an **empty** collection
* **Collection<Book> getAllByTitle(String title)** – **returns** all the books with the **title (there can be many books with the same title)** specified **sorted** **by** **price** ascending and then **by** **title** ascending, if there is **no** books with this author return an **empty** collection
* **Collection<Book> getByPriceRange(int lowerBoundCents, int upperBoundCents)** – **returns** all the books inside the given **price** **range** the **first** **parameter** is **inclusive** the **second** one is **exclusive**, the books are then ordered by price descending. If there are no books in that range return an **empty** collection
* **Collection<Book> getAllOrderedByAuthorThenByTitle() – returns** all the **books** **ordered** by **author** ascending and then **by** **title** ascending if there are no books return an **empty** collection
  1. **Book Land – 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…

## Race Manager

You are given a skeleton with a class **RaceManager** that implements the **Race interface.**

This class stores Cars, those entities are identified by their **unique** **ids** (there will **not** **be** **two** cars with the same ids). Implements all the operations from the **interface**:

* **void add(Car car)** – **adds** the car to the **manager** if the car is **previously** added throw **UnsupportedOperationException**
* **boolean contains(Car car)** – **returns** **weather** the car is **present** inside the manager or **no** true or false
* **Car getCar(long id)** – **returns** the **car** with that id if **present** **otherwise** throw UnsupportedOperationException
* **Car removeCar(long id)** – **removes and returns** the **car** with that id if **present** **otherwise** throw UnsupportedOperationException
* **int size()** – returns the **count** of cars stored inside the manager
* **Collection<Car> getCarsByBestLapTime()** – **returns** all the **cars** **ordered** by **best lap** time ascending if there are no cars stored return an **empty** collection
* **Collection<Car> getCarsByHorsePower()** – **returns** all the **cars** **ordered** by **horse power** descending if there are no cars stored return an **empty** collection
* **Collection<Car> getCarsByLapsCount()** – **returns** all the **cars** **ordered** by **laps count** descending if there are no cars stored return an **empty** collection
* **Collection<Car> getCarsJoinedAfter(Date date)** – **returns** all the **cars joined after the data specified (you can use the date.after(dateJoined) method inside the Date class)** **ordered** by date ascending if there are no cars stored return an **empty** collection
* **Collection<Car> getCarsJoinedBefore(Date date)** – **returns** all the **cars joined before the data specified (you can use the date.before(dateJoined) method inside the Date class)** **ordered** by date ascending if there are no cars stored return an **empty** collection
* **Collection<Car> getAllOrdered() – returns** **all** the cars stored **ordered** by **date** ascending and then by **id** ascending if there are **no** cars stored return an **empty** collection
  1. **Race Manager – 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…

“You are not controlling the storm, and you are not lost in it. You are the storm.” ― Sam Harris, Free Will