**Data Structures Fundamentals with Java Retake Exam - 11 April 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/2332/Data-Structures-Fundamentals-with-Java-Retake-Exam-11-Apr-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.

# Shop

You are given a skeleton with a class **OnlineShop** that implements the **Shop interface.**

This class provides **storage** behavior for **Orders** objects inside simulated shop. The orders are **stored** in **sequential** order **from** **first** **to** **last**. There is no **guarantee** that the **ids** will be **unique**, however this **should** **not** be a matter of **consideration** when solving the problems.

The **HardwareOrder** class have two main properties:

* **int id –** the identification field
* **String description** – this is simply the description of the order

The following methods inside the **OnlineShop** are **not** **implemented** you have to **implement** **them**:

* **void add(Order order)** – **adds** the order at the **next** **free** **position** inside the storage, this should also **increase** the size.
* **Order get(int index)** – gets and returns **(without removing)** the order **located** at specific **index**, this means you have to build **index** based storage to handle that. **Validate** the index if the parameter is **not valid** (goes out of range) throw **IndexOutOfBoundsException**
* **int indexOf(Order order)** – returns **the** **zero** **based** **index** location of the order (**use the id for comparison**). If the order **is not stored** **return** **-1**
* **Boolean contains(Order order)** – returns **whether** the **order** is **stored** or **not** (**use the id for comparison**). Return **true** or **false** value
* **Boolean remove(Order order)** – returns **whether** the **order** is **stored and removed** or **not** (**use the id for comparison**). Return **true** or **false** value
* **Boolean remove(int id)** – returns **whether** the **order** is **stored and removed or not** based on the order **id**. Return **true** or **false** value
* **void set(int index, Order order)** – **sets** the order parameter at the **index** provided if the index **is** **valid,** if notthrow **IndexOutOfBoundsException**
* **void set(Order oldOrder, Order newOrder)** – sets the **newOrder** parameter at the position of the **oldOrder** if the position is **invalid** (oldOreder is not stored) throw **IndexOutOfBoundsException**
* **void clear()** – **removes** all the **orders** from the shop and **resets** the size to **zero**
* **Order[] toArray()** – **returns** an **array** with the **orders** currently stored, if there **are no orders** return an array of **length zero**
* **void swap(Order first, Order second)** – **swaps** the **two** **elements** if both elements are stored. If **one** or **both** orders are not inside the shop – **throw** **IllegalArgumentException**
* **List<Order> toList()** – returns a **List<Order>** with the orders **currently stored**, if there **are no** orders return an **empty List<Order>**
* **void reverse()** – **reverses** the **order** of the elements stored the **first** **one** **e. g.** **at index 0** will become the last one at **index** **shop size – 1** etc..
* **void insert(int index, Order order)** – adds the order **at the index specified** if the index is **outside** of the **bounds** **or the shop is empty** throw **IndexOutOfBoundsException**
* **Boolean isEmpty()** – returns **whether** the **shop** is empty or **not** (true/false)
* **int size()** – returns the **count** of orders **stored**

1. **File Manager**

You are given a skeleton with a class **FileExplorer** that implements the **FileManager interface.**

For this problem you have to **implement** **something like windows** file systemthat works with **SampleFile** those files have **three** main properties:

* **int number – non**-**unique** identification field (you can have **multiple** **files** with the same **name(number)** in **windows** **storage** inside **different** **folders**). There are **no test cases** with **duplicate** **file** **numbers** inside the **same file or directory**, however there may be some duplicates in **different** directories. This should be used more as an **introduction** to the upcoming problem. You **don’t have to think** of some **complicated** **handling** of those situations. If you have to find or do something else with file identified by its **number** you do it with the first one **located** when **traversing** in **order** **of** **levels** (remember how we did **level** traversal or **sibling** traversal)
* **String text** – this is simply the file data
* **List<File> children** – this field holds **collection** **of files** a file can be **directory** or any **other** **file** (directory **has** children while others **don't**). For this problem you can assume all the files are actually directories.

The file manager should be used in a similar way as the windows explorer. The same way we have hierarchy inside windows **"C:\Program Files"** etc… where **C** – is the root of the system.

In this problem the root is the main file you can see inside the skeleton of **FileExplorer** class you have a single field – **File root**.That filed should be **instantiated** upon FileExplorer creation.

The number of the root **should always** be set to **1** –you can see how the tests work. The text of that object is not subjected to any testing so you can write whatever you want for example **"Root"** or just leave it empty or null (be careful with **null** if you want to access that).

**Hint:** think of some data structure which allows hierarchy and multiple children storage.

The following methods **are not** **implemented** your task is to **implement them:**

* **void addInDirectory(int directorNumber, File file)** – **adds** the file inside the directory/file with the **number** specified. If there is no file with that number throw **IllegalStateException.** The file should be added after all previously stored files.
* **File getRoot()** – returns the **root** of the **file system** in this case **always** the root with number **1**
* **File get(int number)** – returns the **file** **with** the **number** **provided** if there are multiple files with that number stored return the **first occurrence** in **level** **traversal** **order** if there is no such file throw **IllegalStateException**
* **Boolean deleteFile(File file)** – returns **whether** a file is **removed** or **not**. When removing file you remove **all** **its** **children** respectively
* **List<File> getFilesInPath(File path)** – returns **all the files** inside the path (directory/file) specified if there is no such file throw **IllegalStateException**. If there are no **files** inside that path return an **empty** List<File>
* **void move(File file, File destination)** – **moves** the file from its current directory to the **destination** directory, this means that **this file and all of its children** should be placed at the new destination after the operation completes. If you attempt to move the root file throw **IllegalStateException** the destination will **always** be valid. The file should be added after all previously stored files.
* **Boolean contains(File file)** – returns **whether** the file is inside the file system or **not**
* **List<File> getInDepth()** – returns **all the files** including the root in **depth order traversal** if there is only the root return only one **element inside the List<File>**
* **List<File> getInLevel**– returns **all the files** including the root in **level order traversal** if there is only the root return only one **element inside the List<File>**
* **void cut(int number)** – cuts a file with the number provided if there is no such file throw IllegalStateException. This means that the **file** is **removed** from the file **system** and is stored inside **buffer** that buffer will be used for **paste** operations
* **void paste(File destination)** – removes the latest cut from the memory buffer for **example**: if we cut files **2** and **3** the first paste operation will place **file 3** inside the **destination** and it will remove it from the **memory buffer**. If paste is called second time the next one in this case **2** will be placed inside the destination file. If the destination is not valid throw **IllegalStateException**. There will **never** be **paste** operations without **enough** **cut** operations before that so you **don't have to handle that**. Think about how you cut a document at one place and paste it somewhere else. The files should be added after all previously stored files.
* **Boolean isEmpty()** – returns **whether** there are files inside the file system **excluding** the **root**. The **root** **must always be non-null file.** So if there is only the root return false otherwise true.
* **String getAsString()** – **returns** the **tree** as **String**. This method is **implemented** inside the skeleton and is used for test cases, you can use it for **better** **visualization** of the tree. This method is **not** **called** inside the **performance** **tests** since String operations may reduce performance so you don’t have to think about it just use it as you find convenient.

## Performance Tests

For this task you will only be required to submit the **code from the previous two problems**. Some part of the tests will **test the performance of the first task**, the **other** one will tests the **second**. 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.

You can submit code to this problem **without full coverage** from the previous two problems, **not all test cases** will be considered only the **general** **behaviour** will be considered important, **edge** **cases** will mostly be ignored for this problem, however the performance will be measured for all operations.

**Hint**: you can submit to this problem only one of the solutions you don’t have to submit both implementations at the same time. This means that after you have finished the first task it is a good idea to submit the code here. After that try to submit only the second task, that way you can find out what is the test coverage for each task. And finally you can submit the both first and the second task. **There** **are** **more** tests on **the first problem** that the second in this exam.

**Good Luck!**

“The nitrogen in our DNA, the calcium in our teeth, the iron in our blood, the carbon in our apple pies was made in the interiors of collapsing stars. We are made of star stuff.” – Carl Sagan