**Data Structures Fundamentals – Exam – 03.10.2020**

This document defines the examination for ["Data Structures – Fundamentals (C#)" course @ Software University](https://softuni.bg/trainings/3112/data-structures-fundamentals-with-csharp-september-2020).

Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2567/Data-Structures-Fundamentals-with-CSharp-Exam-03-Oct-2020).

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 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 tests logic. **Do not change the namespaces** or move any of the files provided inside the skeleton if you have to add new file add it in the same class library.

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 **the library** folder (everything **except** the content inside **bin** and **obj** folders) this should work for all tasks regardless of current DS implementation.

You have to **study** the provided **skeleton**. The code is **separated** inside **different** **libraries**.

There **are** **few** **entities** **inside** the **project** you are **allowed** to **add** code to those, for example **Equals()**, **GetHashCode(), CompareTo()** etc…

# Inventory

You are given a skeleton with a class **Inventory** that implements the **IHolder interface.** The methods **are not** **implemented** your task is to **implement them:**

* **int Capacity** – returns the **number of weapons** that are inside the inventory.
* **void Add(IWeapon weapon)** – **adds** an awesome **weapon** inside the inventory.
* **IEntity GetById(int id) –** returns the **weapon** by the **given id**, if the weapon **does NOT** exist return **null**.
* **bool Contains(IWeapon) –** returns **true** if the weapon is **stored** and **false otherwise.**
* **int Refill(IWeapon weapon int ammunition) – refills** the weapon with the **given ammunition**. If the weapon **does NOT** exist throw an **InvalidOperationException("Weapon does not exist in inventory!")**. Keep in mind that the ammunition of a weapon **should NOT overflow** the given **max capacity**. Return the **new ammunition**.
* **bool Fire(IWeapon weapon, int ammunition) – fires** a weapon from the inventory **with the given ammunition.** If the weapon **does NOT** exist throw an **InvalidOperationException("Weapon does not exist in inventory!")**. The method returns if firing is **possible**, e.g the **fire ammunition is NOT** **more than** the one **inside our weapon**. (*Don't forget to decrease the ammo if possible*)
* **IWeapon RemoveById(int id) – removes** the weapon with **the given id** from the inventory and **returns** it. If the **id** **is NOT valid** throw an **InvalidOperationException("Weapon does not exist in inventory!")**.
* **void Clear ()** **–** **removes all the weapons stored** inside the Inventory
* **List<IWeapon> RetrieveAll() –** returns **all the weapons** inside a List. If there are **none** – return an **empty list**.
* **void Swap(IWeapon first, IWeapon second) – finds** and **swaps the position** of the first weapon with the second weapon. If either of them **DO NOT** exist throw an **InvalidOperationException("Weapon does not exist in inventory!")**. Keep in mind that weapons can be swapped **only** if they are from the **same category**.
* **List<IWeapon> RetrieveInRange(Category lowerBound, Category upperBound) –** returnsall the weapons with given category **within the range (both are inclusive)** if none are found returnan **empty List**
* **void EmptyArsenal(Category category) –** empties **the ammunition** of all weapons from the **given category**.
* **int RemoveHeavy() –** removes **all heavy weapons** and returns their **count**.
* **IEnumerator<IWeapon> GetEnumerator() –** iterates through the collection **in insertion** order

1. **Performance Tests - Inventory**

For this task you will only be required to submit the **code from the Inventory 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.

1. **Legion System**

You are given a skeleton with a class **Legion** that implements the **IArmy interface.** The legion **stores** different types of enemies in **hierarchical** order. Each **enemy** has the following properties:

* **Int AttackSpeed**
* **Int Health**

The legion is **based around** the **attack speed** of its enemies (could be in **increasing** or **decreasing** order). Another thing to keep in mind is that it **CANNOT** contain enemies with the **same attack speed**. Your task is to implement the following methods:

* **int Size –** returnsthe **number** of enemies inside the legion system.
* **void Create(IEnemy enemy) – adds** an enemy inside the legion system. Should store **only unique enemies**, with a **different** attack speed.
* **IEnemy GetByAttackSpeed (int attackSpeed) –** returnsthe **enemy** with the **given attack speed**, otherwise returns **null**.
* **bool Contains(IEnemy enemy) –** returns if the enemy **exists** inside the legion**.**
* **IEnemy GetFastest() –** returns the **fastest enemy** with the **largest attack speed**. If the legion system is **empty** throw an **InvalidOperationException("Legion has no enemies!")**.
* **IEnemy GetSlowest() –** returns the **slowest enemy** with the **least attack speed**. If the legion system is **empty** throw an **InvalidOperationException("Legion has no enemies!")**.
* **IEnemy ShootFastest() – removes** the **fastest enemy** from the legion. If the legion system is **empty** throw an **InvalidOperationException("Legion has no enemies!")**.
* **IEnemy ShootSlowest() – removes** the **slowest enemy** from the legion. If the legion system is **empty** throw an **InvalidOperationException("Legion has no enemies!")**.
* **IEnemy[] GetOrderedByHealth()** – returns the **whole legion** as an **array** ordered by their **health** in **descending**. If the legion contains **no enemies** return an **empty array**.
* **List<IEnemy> GetFaster(int speed)** – retrieves **all enemies** that are **faster** than the **given attack speed** (not inclusive), otherwise returns an **empty collection**.
* **List<IEnemy> GetSlower(int speed)** – retrieves **all enemies** that are **slower** than the **given attack speed** (not inclusive), otherwise returns an **empty collection**.

1. **Performance Tests – Legion System**

For this task you will only be required to submit the **code from the Legion 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.