OOP Advanced Exam – Cresla

You've probably heard about Elon Musk and his company Tesla, the idea of which is to find innovative ways of providing energy. Elon got pretty popular around his company, and a certain person heard about it. Meet Melon Usk – the guy who got inspired by Elon Musk, and created the company **Cresla**.

Overview

Cresla is a company which provides energy through cryogen reactors. The company needs a software to maintain its system and that is why Melon hired you. Melon tried to write some software himself, but couldn't manage much, so he will just let you build the rest. You must however use his code, that's one requirement.

Structure

The structure of the software circles around **Reactors** and **Modules**.

Reactors

The **Reactors** are initialized with an **id** (**int**).
There are generally 2 types of **Reactors**.

Cryo Reactor

The **CryoReactor** is initialized with an additional property:

- **cryoProductionIndex** – an **integer**.

Heat Reactor

The **HeatReactor** is initialized with an additional property:

- **heatReductionIndex** – an **integer**.

Modules

The **Modules** are initialized with an **id** (**int**).
There are generally 3 types of **Modules**.

Cryogen Rod

The **CryogenRod** is initialized with an additional property:

- **energyOutput** – an **integer**.

Heat Processor

The **HeatProcessor** is initialized with an additional property:

- **heatAbsorbing** – an **integer**.

Cooldown System

The **CooldownSystem** is initialized with an additional property:

- **heatAbsorbing** – an **integer**.

The **CryogenRod** is an **Energy module**, because they have **energyOutput**.

The **CooldownSystem** and the **HeatProcessor** are **Absorber modules**, because they have **heatAbsorbing**.

Module Container

The **`ModuleContainer`** is given to you in the skeleton. You can check more info about it in the **Skeleton** section.

Functionality

The functionality of the software involves adding **`Reactors`**, adding **`Modules`** to the Reactors, and so on. As you see the **`Reactors`** and **`Modules`** are the main entities of the program. As such, they have an **`id`**. That id is universal, for all **`Reactors`** and **`Modules`**. When you create a **`Reactor`** or a **`Module`** you **give** it an **id**. When you create another one, you give him the **next id** in order (**previous** + **1**). The **`id`**s start from **1**.

In **some** of the **commands**, you'll receive **`ids`** which may refer to a **`Reactor`** and a **`Module`**. You must check what is the object at that **`id`**, and process the command depending on the result.

Each Reactor has a **Module Container**, in which it **stores** its **Modules**.
The business logic of the program involves: calculating energy output, inspecting reactors and modules, adding more modules and reactors.

Check below, each section, and the functionality it describes.

Reactors

The Reactors are actually the ones that provide power. Initially they produce **NO power**, because they have no modules. Upon adding a module, it is added to the **`ModuleContainer`** of the Reactor.

When the **Module count** of a particular Reactor becomes **equal** to the **`ModuleContainer`**'s **`moduleCapacity`**, the **first Module entered**, is **removed**, so that there can be place for the new one.

A Reactor has an **energy output** – equal to the **sum** of the **`energyOutputs`** of all its **Energy Modules**.
A Reactor also has **heat absorbing** – equal to the **sum** of the **`heatAbsorbings`** of all its **Absorbing Modules**.

If the Reactor is a **`CryoReactor`**, you must **multiply** its **energy output** by the **`cryoProductionIndex`**.
If the Reactor is a **`HeatReactor`**, you must **add** the **`heatReduction`** to the **heat absorbing**.

If the Reactor's **energy output** is **greater** than the **heat absorbing**, the Reactor **overheats**. If that happens, it's **energy output** should be **presented** as **0**.

Modules

The **`Modules`** have no business logic around themselves. They are just **data models**.

Commands

There are several commands which are given from the user input, in order to control the program.
Here you can see how they are formed.

The **parameters** will be given in the **EXACT ORDER**, as the one **specified below**.
You can see the exact input format in the **Input section**.

Each command will **generate an output result**, which you must **print**.
You can see the exact output format in the **Output section**.

Reactor Command

Parameters – **type** (string), **additionalParameter** (int), **moduleCapacity** (int).

Creates a **`Reactor`** of the **given type**, with the **next id**.
The type will either be "**`Cryo`**" or "**`Heat`**".
Depending on the **type**, the **`additionalParameter`** will be set to either **`cryoProductionIndex`** or **`heatAbsorbing`**.

The **`moduleCapacity`** is set to the **`ModuleContainer`** of the Reactor.

Module Command

Parameters – **reactorId** (int), **type** (string), **additionalParameter** (int).

Creates a **`Module`** of the **given type** with the **next id** and **adds** it to the **`ModuleContainer`** of the **Reactor** with the **given reactorId**.

The **`type`** will either be "**`CryogenRod`**", "**`HeatProcessor`**" or "**`CoolingSystem`**".

Depending on the Module type, the **`additionalParameter`** will be set to a different property:

- If it's a **`CryogenRod`** the **additionalParameter** will be set ot the **`energyOutput`**.
- If it's a **`CooldownSystem`** the **additionalParameter** will be set ot the **`heatAbsorbing`**.
- If it's a **`HeatProcessor`** the **additionalParameter** will be set ot the **`heatAbsorbing`**.

Report Command

Parameters – **id** (int)

Brings report of the **Reactor** or the **Module** with the **given id**, providing **detailed information** about it.

Exit

Exits the program. Prints **detailed information** about the **whole** system.

Skeleton

In this section you will be given information about the Skeleton, or the code that has been given to you.

You are allowed to change the **internal** and **private logic** of the **classes** that have been given to you.
In other words, you can change the **body code** and the **definitions** of the **private members** in whatever way you like.

However. . .

You are **NOT ALLOWED** to **CHANGE** the **Interfaces** that have been provided by the **skeleton** in **ANY way**.
You are **NOT ALLOWED** to **ADD** more **PUBLIC LOGIC**, than the **one**, **provided** by the **Interfaces**, **ASIDE FROM** the **`toString()`** method.

Interfaces & Others

You will be given the **interfaces** for the **`Reactor`** and **`Module`** entities. You should use them when you are implementing your entities.

You will **also be given** an **interface** for the **`ModuleContainer`** class, but you will be given the **class itself** too.

Read the documentation of the interfaces to gain basic knowledge of the behavior they define.

ModuleContainer

The **`ModuleContainer`** contains 3 collections – 2 for the **Energy** and **Absorbing Modules**, and one to **follow the order of input** of the Modules.

The class exposes **2 methods** for adding Modules – one for the **Energy Modules** and one for the **Absorbing Modules**.

Before adding an element, the methods check if there is **capacity** for it. If there is **not enough capacity**, the **first element** entered is **removed**, to make space for the new one.

Input

The input consists of several commands which will be given in the format, specified below: :

- `Reactor {reactorType} {additionalParameter} {moduleCapacity}`
- `Module {reactorId} {type} {additionalParameter}`
- `Report {id}`
- `Exit`

Output

Each of the commands generates **output**. Here are the **output formats** of each command:

- **Reactor Command** – creates a **reactor** of the **given type**, with the **given id**. Prints the following result:

  `Created {type} Reactor – {id}`

- **Module Command** – adds a **Module** of the **given type**, with the **given id** to a **specified Reactor.**

  `Added {moduleType} - {moduleId} to Reactor - {reactorId}`

- **Report command** – provides **detailed information** about a **Reactor** or a **Module**, in one of the following formats:

Reactor	Module
{reactorType} – {reactorId} Energy Output: {energyOutput} Heat Absorbing: {heatAbsorbing} Modules: {moduleCount}	{moduleType} Module – {moduleId} {additionalParam}: {additionalParamValue}

  Because of the fact, that the Module is not particular, the **`additionalParameter`** should either be "**Energy Output**" or "**Heat Absorbing**".

- **Exit command** – Terminates the program; **prints** detailed statistics about the whole system. The format, in which the statistics should be printed is:

```
Cryo Reactors: {cryoReactorsCount}
Heat Reactors: {heatReactorsCount}
Energy Modules: {energyModulesCount}
Absorbing Modules: {absorbingModulesCount}
Total Energy Output: {totalEnergyOutput}
Total Heat Absorbing: {totalHeatAbsorbing}
```

  - **`Energy Modules`** and **`Absorbing Modules`** are all Modules that were registered in the system… Regardless of that whether they were removed in the process, you **print them**.
  - The **`totalEnergyOutput`** and **`totalHeatAbsoring`** are the **SUMS** of the **corresponding stats** of **all Reactors**.

Constrains

- All **integers** in the input will be in **range [0, 1.000.000.000]**.
- All **input lines** will be **absolutely valid**.
- There will be **no** non-existent **ids** or **types** in the input.

Examples

Input	Output
Reactor Cryo 10 10 Reactor Cryo 2 15 Module 1 CryogenRod 100 Module 1 CryogenRod 100 Module 1 CryogenRod 100 Module 1 CryogenRod 100 Module 2 CryogenRod 100 Module 1 HeatProcessor 10000 Report 1 Exit	Created Cryo Reactor - 1 Created Cryo Reactor - 2 Added CryogenRod - 3 to Reactor - 1 Added CryogenRod - 4 to Reactor - 1 Added CryogenRod - 5 to Reactor - 1 Added CryogenRod - 6 to Reactor - 1 Added CryogenRod - 7 to Reactor - 2 Added HeatProcessor - 8 to Reactor - 1 CryoReactor - 1 Energy Output: 4000 Heat Absorbing: 10000 Modules: 5 Cryo Reactors: 2 Heat Reactors: 0 Energy Modules: 5 Absorbing Modules: 1 Total Energy Output: 4000 Total Heat Absorbing: 10000
Reactor Heat 250 10 Module 1 CryogenRod 140 Reactor Cryo 25 5 Module 1 CryogenRod 109 Module 3 CooldownSystem 10000 Module 3 CryogenRod 100 Module 3 CryogenRod 100 Module 3 CryogenRod 100 Module 3 CryogenRod 100 Report 1 Report 3 Module 3 HeatProcessor 20000 Report 3 Exit	Created Heat Reactor - 1 Added CryogenRod - 2 to Reactor - 1 Created Cryo Reactor - 3 Added CryogenRod - 4 to Reactor - 1 Added CooldownSystem - 5 to Reactor - 3 Added CryogenRod - 6 to Reactor - 3 Added CryogenRod - 7 to Reactor - 3 Added CryogenRod - 8 to Reactor - 3 Added CryogenRod - 9 to Reactor - 3 HeatReactor - 1 Energy Output: 249 Heat Absorbing: 250 Modules: 2 CryoReactor - 3 Energy Output: 10000 Heat Absorbing: 10000 Modules: 5 Added HeatProcessor - 10 to Reactor - 3 CryoReactor - 3 Energy Output: 10000 Heat Absorbing: 20000 Modules: 5 Cryo Reactors: 1 Heat Reactors: 1 Energy Modules: 6 Absorbing Modules: 2 Total Energy Output: 10249 Total Heat Absorbing: 20250

Tasks

Task 1: High Quality Structure

Refactor the given Skeleton code and use it.

Melon Usk tried to write some code before you, but he couldn't do much… But he somehow managed to write the **`ModuleContainer class`**. His work, however, is not that trustworthy, so you might have to give it an eye or two, for potential **functionality bugs** and things that **do NOT follow** the **good practices** of **Object-Oriented Programming**.

Refactor anything, which will **improve** the **code quality**, in your opinion. Be careful **NOT** to **break the code** or one of the **rules** specified in the **Skeleton section**.

High Quality Code.

Achieve good separation of concerns using abstractions and interfaces to decouple classes, while reusing code through inheritance and polymorphism. Your classes should have strong cohesion - have single responsibility and loose coupling - know about as few other classes as possible.

Reflection.

Since the **`ModuleContainer`** class does not reveal much, you will probably have to use a some **`reflection`** for the business logic of the **`Reactors`**.

For this task, submit only the "**`cresla`**" folder.

Task 2: Correct business logic.

The given code provides some functionality, but it does not cover the entire task. Implement the rest of the business logic, using the given code, and implement everything following the requirements specification. Check your solutions in the Judge system.

For this task, submit the whole "**`src`**" folder.

Task 3: Unit Testing.

Test the **`ModuleContainer class`**'s methods for potential bugs. Extensive testing might require you to have some of the core logic implemented, in order to cover all cases.

When testing, use **ONLY THE CLASSES, PROVIDED** by the **SKELETON**.

For this task submit the **folder** you have put your **tests** into.

NOTE: You are **NOT ALLOWED** to submit **non-test classes** for this task.