**Java OOP Basics Exam – The Expanse**

**Overview**

You have a task to write a software program, modelling a space colony.

**The Problem**

You need to provide a working model of a space colony. You should be able to add or remove colonists and eventually edit existing ones. Colonists live separated into families. Whole families can also be removed. Each colonist will have traits that let you calculate his potential. All individual potentials are summed up to the potential of the entire colony.

**Task I: Structure**

The main structure of the program should include the following elements:

**Colony**

Creating a colony requires: **maxFamilyCount** (**int**), **maxFamilyCapacity** (**int**).

Max family count specifies the number of families that can inhabit a colony simultaneously.

Max family capacity specifies the total colonists that can be in a single family.

**Family**

Creating a family requires: **id (String)**

Colonists are separated into groups called families. When a colonist is created, he or she is given a family id. There may be many families at the same time, but no more than the **maxFamilyCount**. Families are created or removed dynamically. If a colonist with a family id that does not exists is created, the family should be created too. If the last colonist from an existing family is removed, the family should be removed too.

Colonists are added only if the family id that he is given has spare slots (e.g. family size is less than **maxFamilyCapacity**) or if there is no such family yet, but there are spare family slots (e.g. **maxFamilyCount** has not yet been reached).

**Colonist**

Colonists can be different types that have different characteristics. All colonists have: **id** (**String**), **familyId** (**String**), **talent (int)**, **age (int)**.

Every colonist has a potential (**int getPotential()**) which is calculated by the sum of his talent plus bonuses given for his **class** and **age**.

**Soldier**

All soldiers get a class bonus +3 and age bonus +3 regardless of their age.

**Engineer**

All engineers get class bonus +3. Additionally, if an engineer's age is above 30, they get further +2.

Engineers can be of two types:

**SoftwareEngineer**

Gets a flat bonus +2, regardless of age or talent.

**HardwareEngineer**

Gets a bonus +2 if age is less than 18 (exclusive).

**Medic**

All medics get bonus +2.

Medics have one additional (last) property: **sign (String)**.

And can be one of two distinctive types:

**GeneralPractitioner**

All GPs older than 15 get a bonus of +1.

If a GP is born under the sign of the "caring" he gets a bonus of +1. If his sign is the "careless" he gets a penalty -2.

**Surgeon**

Surgeons get age bonus +2 only if they are between 25 and 35 years old (exclusive interval).

Further they get a bonus +3 if born under the sign of the "precise". Nevertheless, if a surgeon is born under the sign of the "butcher" he gets a penalty -3.

**Colony Methods**

Provide a public method that returns all colonists from a given family (sorted by id):

* **List<Colonist> getColonistsByFamilyId(String familyId)**

For this task you **don’t have** to provide a correct result, only correct design.

**Colonist Methods**

Provide a public method that returns a colonist's potential:

* **int getPotential()**

For this task you **don’t have** to provide a correct result, only correct design. Be cautious of using if-else or switch statements. Utilize **inheritance**, **polymorphism** and **abstract classes/methods** to achieve good design.

**Constructors**

Implement all **class constructors**, with the **parameters** in the **EXACT** **given order** and the **EXACT given types**. Zero tests will help you with this.

**Getters**

Provide public getter for every property passed through the constructor. The naming convention that you must follow is **getPropertyName (camelCase)**.

**Example: public int getAge();**

**Judge**

Don't forget to include a **Main class** with a **main method** in your project when submitting your solution in judge. You are allowed to create and use new methods and classes in your solution as long as they conform to good OOP practices.

**Task II: Business Logic**

The business logic of the program should be concentrated around several commands.

**The Colony Class**

Add these **methods** to the Colony class:

* **void addColonist(Colonist colonist)**
* **void removeColonist(String familyId, String memberId)**
* **void removeFamily(String id)**
* **List<Colonist> getColonistsByFamilyId(String familyId)**
* **void grow(int years)**
* **int getPotential()**
* **String getCapacity()**

**The Colonist Class**

Implement and provide correct return values for colonist methods:

* **int getPotential()**
* **void grow(int years)**

**Commands**

Here are the **input commands** you need to accept from the **user input**.

* **insert {class} {colonistId} {familyId} {talent} {age} {sign?}**
* **Adds** a colonist of the **given class**, with the **given** **id**, and the **given** **stats**.
* Talent and age are of type **int**, everything else will be a **String**
* **Sign** is provided only for eligible classes (GeneralPractitioner and Surgeon)
* Possible classes:
* SoftwareEngineer
* HardwareEngineer
* Soldier
* GeneralPractitioner
* Surgeon
* If the family max count is reached print "**colony is full**"
* If the family is full print "**family is full**"
* **remove {modificator} {familyId} {colonistId?}**
* **Removes** a family or a colonist from a given family.
* **Modificator** can be one of:
* **family**
* **colonist**
* **Colonist id** is provided only if the modificator is "colonist".
* **grow {years}**
* The colony ages with the years provided. All colonists age accordingly.
* **Years** is of type **int**.
* **potential**
* **Prints** the total potential of the colony in format:
* "potential: {totalPotential}"
* **capacity**
* **Prints** information about colony capacity in format **(families are sorted by id):**
* "families: {familyCount}/{maxFamilyCount}"
* "-{familyId}: {colonistCount}/{familyCapacity}"
* **family {familyId}**
* Prints information about a family in format **(colonists are sorted by id)**:
* "{familyId}:"
* "-{colonistId}: {potential}"
* …
* "-{colonistId}: {potential}"
* If the family does not exist print:
* "family does not exist"

**Task III: I / O (Input / Output)**

**Input**

* The input will come in the form of commands, in the format specified above.
* The input sequence ends when you receive the command "**end**".

**Output**

* The output should be in the format described above.

**Constrains**

* **Talent** will be in **range [0, 10]**.
* **Age** will be in **range [1, 100]**.
* **All strings** in the input may consist of **any ASCII character**, except **SPACE** so that the input is easily processed.
* There will be **NO invalid** input lines
* Note that throughout the program, you are working **only** with **integers**.
* Each **mathematical** or **logical action** performed on **numeric data**, should be performed between **integers**.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2 1  insert Surgeon Gosho S6 7 25 precise  insert SoftwareEngineer Stefcho S2 8 17  insert HardwareEngineer Mimi S2 8 32  potential  capacity  end | family is full  potential: 25  families: 2/2  -S2: 1/1  -S6: 1/1 |
| 2 1  insert Surgeon Gosho S6 7 25 precise  insert SoftwareEngineer Stefcho S2 8 17  remove colonist S2 Stefcho  potential  capacity  end | potential: 12  families: 1/2  -S6: 1/1 |
| 1 2  insert Surgeon Gosho S1 7 25 noEffectSign  insert SoftwareEngineer Stefcho S1 8 17  remove family S1  potential  capacity  end | potential: 0  families: 0/1 |
| 1 1  insert GeneralPractitioner Stefcho S1 6 12 caring  potential  grow 6  potential  family S1  end | potential: 9  potential: 10  S1:  -Stefcho: 10 |