# Assignment 4. 2020 Fall of CS102A

> Designer: ZHU Yueming
> Document: HUANG Yansong
> Javadoc: WU Yechang
> Local Judge: WANG Zhiyuan
> Official Judge: LIU Xukun
> Test: JIANG CHUAN

## General Description

In this assignment, you are asked for designing a system that allow users to do operations below:

- To define different courses with different properties.
- To storage and modify the information of different types of students.
- Set the dependency between courses.
- Select courses for a certain student or batch select courses for certain students.
- Manage student groups through university structure.
- Avoid illegal operations, etc.

Before starting this task, please understand the definition and the usage of class, abstract class, interface, inherit and polymorphism.

## Coding Requirements

You need to define or modify following classes and methods to achieve the functions before. We will judge your code by JUnit.

**Hint**:

- Do not modify or remove any methods or fields that have been already defined.
- You can add other methods or attributes that you think are necessary (Like getting and setting methods).

### Enum `CourseType`

`CourseType` is an `enum` that in order to distinguish different types of course. There are **three** possible value of this `enum`.

- `ARTS`, actual value 0, for art classes.
- `SCIENCE`, actual value 1, for science classes.
- `GENERAL`, actual value 2, for general classes.

### Class `Course`

`Course` is a class that in order to instantiate objects which storages information of different classes.

### Property `Course.courseNumber`

```
private String courseNumber;
```

**Private** property with type `String`, representing the number of the course.

This string will only contain one character in usage, but you shall not use `char` or `Character` to replace `String`.

### Property `Course.courseType`

```
private CourseType courseType;
```

**Private** property with type `CourseType`, representing the general type of the course.

As mentioned before, this property has three possible values: `ARTS`, `SCIENCE` and `GENERAL`.

### Property `Course.credit`

```
private int credit;
```

**Private** property with type `int`, representing the credit you can get after learning the course.

The value of this property would be small, you needn't worry about overflowing.

### Method `Course.toString`

```
@Override
public String toString() {
    return "Course{" +
        "courseNumber='" + courseNumber + '\'' +
        ", courseType=" + courseType +
        ", credit=" + credit +
        '}';
}
```

**Public** override method to convert the information of the course to a string object.

## Abstract Class `Student`

`Student` is an **abstract** class, which means it is the *super* class of all concrete student classes.

### Property `Student.number`

```
private int number;
```

**Private** property with type `int`, representing the ID number of the student.

The value of this property would be small, you needn't worry about overflowing.

### Property `Student.college`

```java
private int college;
```

**Private** property with type `int`, representing the residential college the student belongs to. There would be some operation that pick courses by colleges in later requirements.

The value of this property would be small, you needn't worry about overflowing.

### Method `Student.toString`

```java
@Override
public String toString() {
    return String.format("%d-%d", this.number, this.college);
}
```

**Public** override method to convert the information of the student to a string object.

### Method `Student.checkGraduate`

```java
public abstract boolean checkGraduate();
```

**Abstract public** method to check if the student can graduate. This method should be implemented by concrete classes in different ways. You can get the constraints that graduation should fit during chapter "Class `ArtsStudent` and `ScienceStudent`".

## Class `ArtsStudent` and `ScienceStudent`

`ArtsStudent` and `ScienceStudent` are concrete classes that inherit `Student` as *super* class. All the student objects should be instantiated to `ArtsStudent` or `ScienceStudent`.

### Method `ArtsStudent.checkGraduate`

```java
@Override
public boolean checkGraduate() {
    // TODO by yourself.
}
```

**Public** method to implement the abstract method in class `student`. Return whether the student could graduate.

**Constraints** that the art student should fit if he or she can graduate:

- Earn at least 8 credits in art courses.
- Earn at least 4 credits in general courses.
- Earn at least 4 credits in science courses.

### Method `ArtsStudent.toString`

```java
@Override
public String toString() {
    return String.format("%s-%s-course %d", super.toString(), "ARTS", [how many
courses selected]);
}
```

**Public** override method to convert the information of the art student to a string object.

The last format `%d` represents how many courses the current student selected.

### Property `ScienceStudent.generalWeight`

```
private double generalWeight;
```

**Private** property with type `double`, representing the weight of general classes when calculating the total credits.

The value of this property is guaranteed to be in the open interval $(0,1)$.

### Property `ScienceStudent.artsWeight`

```
private double artsWeight;
```

**Private** property with type `double`, representing the weight of art classes when calculating the total credits.

The value of this property is guaranteed to be in the open interval $(0,1)$.

### Method `ScienceStudent.checkGraduate`

```
@Override
public boolean checkGraduate() {
    // TODO by yourself.
}
```

**Public** method to implement the abstract method in class `student`. Return whether the student could graduate.

**Constraints** that the science student should fit if he or she can graduate:

- Earn at least 10 credits after calculating the total credit by weights.
- Has picked at least 4 courses.

The weight of science courses is $1.0$ when calculating the final credit.

### Method `ScienceStudent.toString`

```
@Override
public String toString() {
    return String.format("%s-%s-course %d", super.toString(), "SCIENCE", [how
many courses selected]);
}
```

**Public** override method to convert the information of the science student to a string object.

The last format `%d` represents how many courses the current student selected.

## Interface `University`

`University` is an interface that contains a series of methods about operations on courses and students. To view the detail information of the requirements, please check the java document on sakai system.

## Class `ConcreteUniversity`

`ConcreteUniversity` is a concrete class which implements the interface `University`. In this class, you should implement all the methods that has been declared in the interface `University`. In addition, there will be a member field you should add in this class: `ConcreteUniversity.students`.

### Property `ConcreteUniversity.students`

```java
private List<Student> students;
```

**Private** property with type `List<Student>`, representing all the students registered in the university.

You shall **NOT** change the type of this property to `ArrayList` or something else, it must be a `List` object.

### Inner Class `ConcreteUniversity.Relation`

**This inner class in class `ConcretrUniversity` is NOT a mandatory requirement to your programming**, but if you need to record relationships between students and courses, maybe you can add such type of class. Whether to add the inner class is determined by your design.

### Method `ConcreteUniversity.showArtsANDScienceCount`

This Method will return a String value with the number of arts students and science students. The return value must be: "ARTS-*arts_num*-SCIENCE-*science_num*"

For example: If there are 7 arts students and 11 science students in the university, the value you should return is **"ARTS-7-SCIENCE-11"**.

# Testing Supplements

Although your code will be judged by JUnit on online judging platform, which means you do not need to create the `main` method, but you might want to judge the correctness of your code offline. In order to help you do the test, here are some basic test data. To use these data as input, you should create `main` method and get the input by using `Scanner`.

**Caution: The separator in all of the following sample inputs has only ONE space character.**

## Test Case of Course

```
E 1 2
F 1 4
G 2 2
C 2 3
D 2 3
A 1 3
B 0 2
```

For each line, the data follows the sequence "`CourseNumber CourseType Credit`", each line can be read in as a whole `String` object and fit in all rules declared below .

Not only in offline test cases, but also online JUnit test cases will follow the rules below:

- `CourseNumber` is guaranteed to be **only one letter**, and in all the statements each `CourseNumber` will only be provided once but without specific order.
- `CourseType` only has **three possible values**: 0 for `ARTS`, 1 for `SCIENCE` and 2 for `GENERAL`.
- `Credit` is a **positive** integer.

You can check the result after input in your ways.

## Test Case of Students

```
4 1
5 2 0.3 0.8
9 1 0.2 0.9
1 1
2 2 0.5 0.5
3 3 0.5 0.5
```

For each line, if there are **two** numbers, the data follows the sequence "`number college`", which represents an art student; if there are **four** numbers, the data follows the sequence "`number college generalWeight artsWeignt`", which represents a science student.

Not only in offline test cases, but also online JUnit test cases will follow the rules below:

- `number` is an integer, and in all the statements each `number` will only be provided once but without specific order.
- `college` is an integer, represents the ID number of the residential college this student belonging to. Many students can have the same college number.
- `generalWeight` and `artsWeight` are two double value in open interval $(0,1)$.

## Prerequisite Rules Definition

```
A
C
F
H
I
M
N
A 1 B
C 1 B
C D 1 E
F 1 E
A 1 J
```

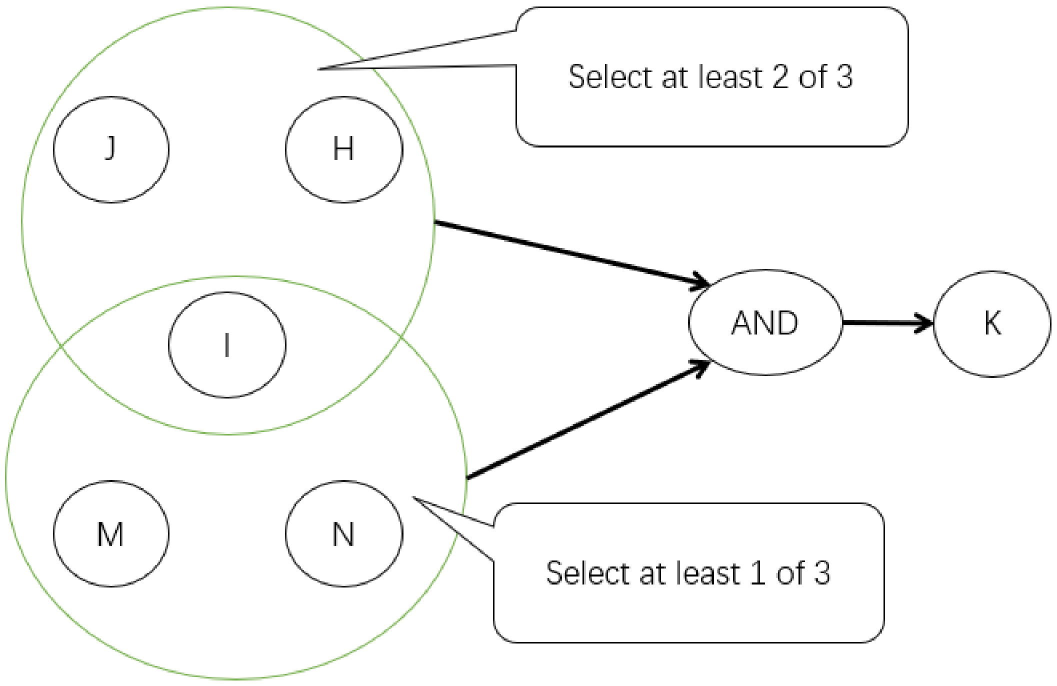
```
H I J 2 K
I M N 1 K
```

This test case is for operations that after the definition of courses and students, generally in university testing.

Each line is a declaration that describing a course and its prerequisite requirements:

- If there is only one letter, that means the course has no prerequisite requirements. The only letter represents the number of this course.
- If there is not only one letter, the format follows the rules below:
    - The first is **at least** one letter, representing the prerequisite requirement courses of this course.
    - The second is an integer, representing the minimum of courses should have been selected before select this course.
    - The third is a letter, representing the name of this course.

For example, in this test case, if a student want to select the course `K`, he or she should select at least two courses among `H`, `I` and `J`, and beside this, he or she should also select at least one course among `I`, `M` and `N`.



## Other Rules of Test Data:

- **In Prerequisite Rules Definition**: During judging, we guarantee that every course name appeared while course modifying **must be defined** during the course definition process, and every course name appeared as a prerequisite requirement **must be declared** already.
- **Parameters about course selection**: During judging, we guarantee that all parameters as course name that passing in all methods about course selection must be declared already.
- **Parameters about course selection**: There might be some repeated course selection operations, like someone has already selected course `A`, then if he or she selects course `A` again, the relative method should return `false` and **terminate such illegal operation, do not make one student select the same course again**.

## Submission of Assignment

1. You should submit all the source code files (with an extension **".java"**).
2. You need submit at least those java files: `ArtsStudent` , `ScienceStudent` , `Student`, `Course`, `CourseType` , `University`, `ConcreteUniversity`
3. You should submit all source code **directly** into your OJ system, **do not compress** them into one folder.
4. **No Chinese characters** are allowed to appear in your code.
5. **No package** included.
6. The output must strictly follow the description and the sample in this document and the , and you can `LocalJudgeA4Test.java` only get points for each task only when you pass the test case.