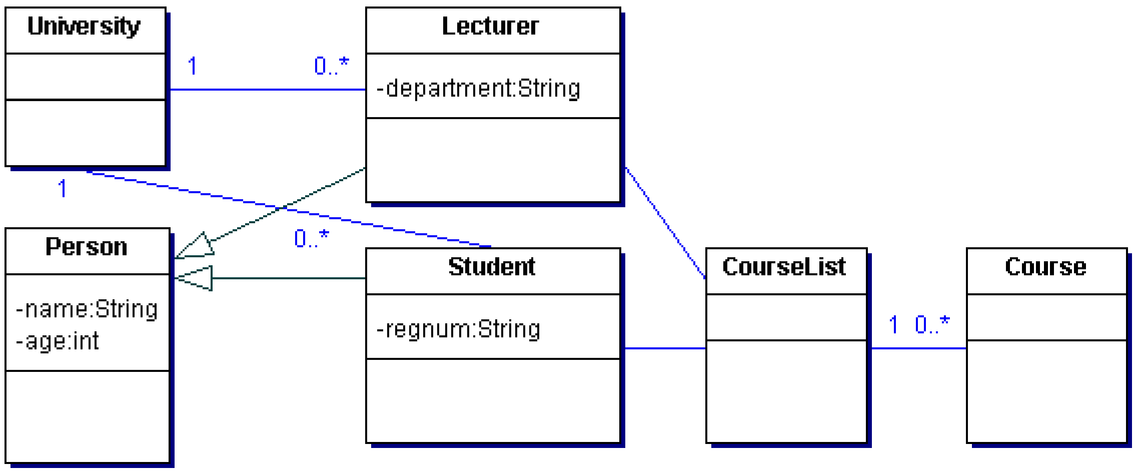
University of Stirling
Computing Science and Mathematics

CSC9P5 Software Engineering I　　　Autumn 2018

Tutorial 1　Sample answers



[As an initial design, this class diagram has discussable points, such as:

- Should name and age be protected, not private?
- Perhaps Person should be abstract? (Why?)
- We could have a PersonList class, or a LecturerList class, or a StudentList. (Why, where?)

]

1.
A possible chain of messages could be:
Boundary →(1) University →(2) Student →(3) CourseList →(4) Course
(and then results returned back along the chain)
Where message
(1) is "Does the student with this reg no take this course?"
(2) is "Does this student take this course?" (sent to the appropriate Student object)
(3) is "Is this course in the list?"
(4) is "Check if this is the course" (sent to courses in searching the list)

2.
An abstract class defines a class with a set of template properties and operations that will not be directly instantiated. When using an abstract class, you must derive a new class from the abstract class and implement any operations that have not been implemented. It is most useful when you need to define the standard behaviour of a basic object for which no ‘real’ object exists. An abstract class is most easily identified as a class which, for at least one of its operations, has no defined implementation. The implication from this is that you cannot directly instantiate an abstract class.

In the UML example, Person could be an abstract class because within the context of our model, there is "no such thing" as a Person – there are Students and Lecturers that are both types of Person.
In another example, Publication could be an abstract class for Books and Journals: you would not ask for a Publication but you might want to look for a Book or Journal which are types of Publication.

Also, a Java abstract class (and a Java interface) is useful to provide common functionalities to its clients, uniformly. For instance, you can think of some getDetails() method (defined abstractly in Person) that returns different views for the different subclasses of Person (with separate concrete implementations in the subclasses: say, the department of a Lecturer, and the registration number of a Student). We can then treat all Lecturer and Student objects uniformly: In this example, we might have a variable p of type Person (which will actually be referencing either a Lecturer or a Student object). Then p.getDetails() will call the relevant concrete method: dynamic binding will then select the appropriate method – see below.

3.
Although the age and name of a person are the same whether they are a student or a lecturer, it is likely that their other details will be different. We would like to return the relevant information for the student or the lecturer therefore we will want to override the implementation of getDetails to reflect this.

Dynamic binding (or late-binding) means that when given a reference to an object, the actual method called will be dependent upon the type of the reference.

```
Person p;
p = new Lecturer("Bob");
System.out.println(p.getDetails());

p = new Student("David");
System.out.println(p.getDetails());
```

It may also be the case that the actual value of p is not known until run-time, as in

```
Person p;
if (cond){
   p = new Lecturer("Bob");
}else{
   p = new Student("David");
}
System.out.println(p.getDetails());
```

where cond depends on a user choice.

4.One purpose of this point is to invite students to get acquainted with Java documentation and make an effort to use it.

```
public boolean contains(Object o)
```

Returns `true` if this vector contains the specified element o. More formally, returns `true` if and only if this vector contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`, otherwise false.

We should add an equals method that properly overrides the one inherited from Object, as that is what Vector will be trying to use. For example

```
public boolean equals(Object o) {
   return regnum.equals(((Student)o).regnum);
}
```

It is important to understand that we need to define it for `Object o.` (It has to do with dynamic binding and the choice of the most suitable/specific instance of the method …).

5.
Here we need to analyse the code in terms of *overloading*: System.out is a PrintStream, which has 10 overloaded alternative definitions for println. Assume x and y are int variables.

In `System.out.println("The sum of " + x + " and " + y + " is");` the parameter is an expression that builds a String, so the + is String concatenation, and the println called is the overloaded alternative expecting a String parameter. Example output:
```
The sum of 2 and 3 is
```

In `System.out.println(x+y);` the parameter is an expression that adds two ints, so the + is int addition, and the println called is the overloaded alternative expecting an int parameter. Example output: 5

We can make `System.out.println(someStudent);` display the student's name by adding
```
public String toString() {
   return name;
}
```