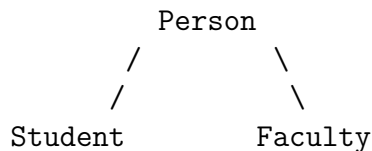


Subclassing from Multiple Classes, and Interfaces

Consider the following subclassing hierarchy:



Corresponding to the ADTs:

PERSON:

```
public static Person create (String, int);  
public String getName ();  
public String getNUId ();
```

STUDENT:

```
public static Student create (String, int)  
public String getName ();  
public String getNUId ();  
public void registerForCourse (String);
```

FACULTY:

```
public static Faculty create (String, int)  
public String getName ();  
public String getNUId ();  
public void offerCourse (String);
```

The specifications for the above are the obvious ones, such as `Person.create(n, id).getName() = n`. Note that we cannot give an algebraic specification for `registerForCourse` and for `offerCourse`, because they are really executed for their side-effects, and not for the value they return.

Consider their implementation, again, in the most obvious way, and such that they respect the desired subclassing hierarchy:

```
public class Person {
```

```

private String name;
private String nuid;

// again, we'll see this later
protected Person () { }

private Person (String n, String id) {
    name = n;
    nuid = id;
}

public static Person create (String n, String id) {
    return new Person(n,id);
}

public String getName () { return this.name; }

public String getNUId () { return this.nuid; }
}

public class Student extends Person {
    private String name;
    private String nuid;

    private Student (String n, String id) {
        name = n;
        nuid = id;
    }

    public static Student create (String n, String id) {
        return new Student(n,id);
    }

    public String getName () { return this.name; }

    public String getNUId () { return this.nuid; }

    public void registerForClass (String class) { ...whatever... }
}

```

```

public class Faculty extends Person {
    private String name;
    private String nuid;

    private Faculty (String n, String id) {
        name = n;
        nuid = id;
    }

    public static Faculty create (String n, String id) {
        return new Faculty(n,id);
    }

    public String getName () { return this.name; }

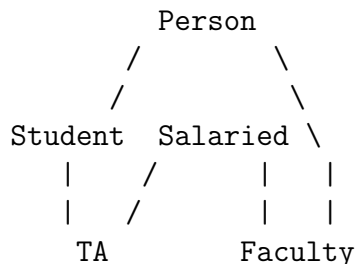
    public String getNUid () { return this.nuid; }

    public void offerClass (String class) { ...whatever... }
}

```

No problem whatsoever here. (Although there is a lot of code repetition. We'll see later how to get rid of it, and at what price.)

What I want to consider now is a slightly more involved class hierarchy. Suppose that we wanted to capture the fact that some of the persons in the hierarchy are salaried, that is, get a stipend from the university. Persons with salaries have a method `getSalary()` that returns the salary. We can capture this using a class `Salaried`, and a class subclasses `Salaried` when the class represents persons that are salaried. Faculty are salaried, but students are not. Just to make the example more interesting, suppose that we have a subclass of `Student` called `TA`, which are in fact salaried, and therefore are also a subclass of `Salaried`. Then this is the hierarchy we get:



This class hierarchy is not a tree. Yet, it is a very natural example of a subclassing hierarchy—it is a subclassing hierarchy we can well imagine occurring in practice.

There is no a priori reason why dag hierarchies are a problem. After all, subclassing is just a relationship between classes that indicates, roughly, a subclass must implement all the methods that a class makes available. (This ensures that when we pass a subclass to a method expecting a class, we don't run into problem invoking a method that is not defined.) Clearly, we can have **TA** implement all the methods of **Salaried** and all the methods of **Student**, and if we could just tell Java that **TA** is a subclass of both **Salaried** and **Student**, everything would work nicely. So in a language with only subclassing, we could implement the above class hierarchy by simply defining **TA** to “extend” both **Salaried** and **Student**. Many languages let you do the above cleanly.

In Java, however, we have a problem. A class cannot extend more than a single class. (This is because, as we shall see, Java conflates subclassing and inheritance—while subclassing from multiple classes is not a problem, inheriting from multiple classes is a headache.) That sucks, because as the **Salaried** example above was meant to illustrate, there are natural hierarchies that are not tree shaped. And restricting you to subclassing hierarchies that are not tree shaped limits what you can program naturally.

Fortunately, Java gives you a way out. You can actually subclass from multiple classes, but all but at most one of them *must* be a fully abstract class. These fully abstract classes are called *interfaces*.

An interface is defined as follows:

```
public interface Salaried {
    public int getSalary ();
}
```

Note, once again: only method signatures, no actual method implementation. And while I have annotated the methods as `public`, they cannot be but `public`. The annotation is somewhat redundant. (I like redundancy; I like to be explicitly reminded that my interface methods are `public` when I look at the code.)

To subclass from an interface, instead of using `extends`, we use `implements`. And as we shall see, this is pure subclassing.

Returning to the **Salaried** example, here is an implementation of the above hierarchy.

```
public class Person {
    private String name;
    private String nuid;

    protected Person () { }

    private Person (String n, String id) {
        name = n;
        nuid = id;
    }
}
```

```

}

public static Person create (String n, String id) {
    return new Person(n,id);
}

public String getName () { return this.name; }

public String getNUId () { return this.nuid; }
}

public class Student extends Person {
    private String name;
    private String nuid;

    protected Student () { }

    private Student (String n, String id) {
        name = n;
        nuid = id;
    }

    public static Student create (String n, String id) {
        return new Student(n,id);
    }

    public String getName () { return this.name; }

    public String getNUId () { return this.nuid; }

    public void registerForClass (String class) { ...whatever... }
}

public class TA extends Student implements Salaried {
    private String name;
    private String nuid;
    private int salary;

    private TA (String n, String id) {
        name = n;

```

```

    nuid = id;
}

public static TA create (String n, String id) {
    return new TA(n,id);
}

public String getName () { return this.name; }

public String getNUId () { return this.nuid; }

public void registerForClass (String class) { ...whatever... }

public int getSalary () { return this.salary; }
}

public class Faculty extends Person implements Salaried {
    private String name;
    private String nuid;
    private int salary;

    private Faculty (String n, String id) {
        name = n;
        nuid = id;
    }

    public static Faculty create (String n, String id) {
        return new Faculty(n,id);
    }

    public String getName () { return this.name; }

    public String getNUId () { return this.nuid; }

    public void offerClass (String class) { ...whatever... }

    public int getSalary () { return this.salary; }
}

```

We can really think of an interface as a fully abstract class. This means, in particular, that subclassing and hence subtyping works as expected. If we have an object a of type A and A

implements B, then we can consider a as an object of type B as well. In particular, we can pass a to a method that expects a B, or store it in a variable of type B.

This is of course just subclassing, and therefore allows us to reuse client code. In particular, it is possible to write a method that expects several objects of class `Salaried`, perhaps to compute their average salary, and we can of course pass to it any object of any class that implements `Salaried`.