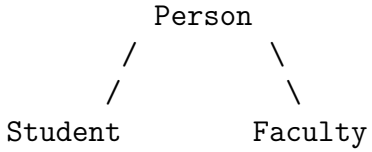
9 Subclassing from Multiple Classes

Consider the following subclassing hierarchy:



Corresponding to the ADTs:

```
PERSON:
  public static Person create (String, int);
  public String name ();
  public int NUid ();

STUDENT:
  public static Student create (String, int)
  public String name ();
  public int NUid ();
  public void registerForCourse (String);

FACULTY:
  public static Faculty create (String, int, int)
  public String name ();
  public int NUid ();
  public int salary ();
  public void newCourse (String);
```

To create a `Person`, we give it a name and an NU id, similarly for creating a `Student`. For a `Faculty`, we give it a name, an NU id, as well as a salary. The specifications for the above are the obvious ones, such as `Person.create(n,id).name() = n`. Note that we cannot give an algebraic specification for `registerForCourse` and for `newCourse`, because I haven't told you what they do, and frankly, we don't really care. I just wanted some methods that we can use to illustrate our concepts.

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

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

  // to allow subclassing
  protected Person () { }

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

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

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

  public int NUid () { return this.nuid; }
}
```

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

  // to allow subclassing
  protected Student () { }

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

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

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

  public int NUid () { return this.nuid; }
```

```
  public void registerForCourse (String course) { ...whatever... }
}
```

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

  // to allow subclassing
  protected Faculty () { }

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

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

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

  public int NUid () { return this.nuid; }

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

  public void newCourse (String course) { ...whatever... }
}
```

No problem whatsoever here. Although there is a lot of code repetition. We'll see in a later lecture how we can reduce it, and at what price.

Okay, now what? Let's define a function to compute the average of salaries of people with salaries. Right now, the only such is `Faculty`, and any possible subclasses, so the definition is pretty simple:

```
public static int averageSalary (Faculty p, Faculty q) {
  return (p.salary() + q.salary())/2;
}
```

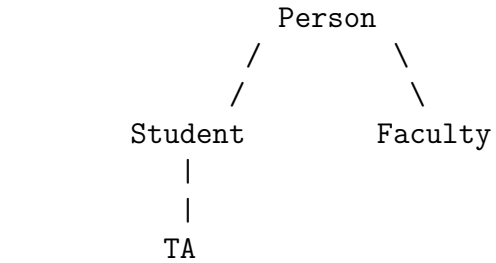
For simplicity, I'm only considering the average of two salaries. It's easy enough to take in more `Faculty` as arguments, or even to use an array of `Faculty`, but to keep our focus on

the subclassing part, let me make the function as simple as I can.

Let me add a new subclass to the subclassing hierarchy: a TA is a kind of student that also has a salary, with ADT:

```
TA:
  public static TA create (String, int, int)
  public String name ();
  public int NUid ();
  public int salary ();
  public void registerForCourse (String);
```

which gives us the following subclassing hierarchy:



and implementation:

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

  // to allow subclassing
  protected TA () { }

  private TA (String n, int id, int s) {
    name = n;
    nuid = id;
    salary = s;
  }

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

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

```
  public int NUid () { return this.nuid; }

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

  public void registerForCourse (String course) { ...whatever... }
}
```

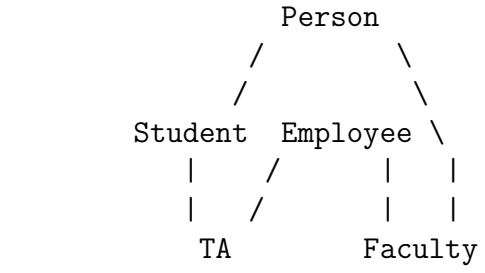
Now, what about computing the average of salaries of TAs? We cannot directly reuse `averageSalary()`, because while it can take as arguments `Faculty` and any of its subclasses, `TA` is not a subclass of `Faculty`. So we would need to write a new version of `averageSalary()` that works with `TA`, and that's sad because if you write that function, you see that aside from the types, the code is exactly the same.

Can we somehow reuse the `averageSalary()` function we wrote above? To do so, we would either need to make `TA` a subclass of `Faculty` instead of `Student` — which has its own problems: what do we do for method `newCourse()` in `TA`? It will have to throw an exception along the lines of "sorry, can't call this method, while it's defined, it's not really callable", which is annoying for several reasons, not the least of which it's just not right — or we need to define `averageSalary()` to take as arguments a class that is a superclass of both `TA` and `Faculty`. The only class we have in our hierarchy with that property is `Person`, but if we write:

```
public static int averageSalary (Person p, Person q) {
  return (p.salary() + q.salary())/2;
}
```

the type checker complains that `Person` doesn't necessarily have a method `salary()` defined, so it cannot guarantee the safety of this code, and so it rejects it. As it should — what would happen if we created two `Student`s and somehow called `averageSalary()` with those students? Those `Student`s don't have a `salary()` method defined.

What we need, really, is class that is a superclass only of `TA` and `Faculty`, and define `averageSalary()` to take arguments of that class. In other words, we want a subclassing hierarchy that looks like:



where `Employee` is a class with method `salary()` defined, so its subclasses must have that method defined as well. We could take `Employee` to be a subclass of `Person`, but I will choose not to for reasons that will become clear in a bit.

If we had such a class, then we could simply write

```
public static int averageSalary (Employee p, Employee q) {
  return (p.salary() + q.salary())/2;
}
```

What's so interesting about this kind of subclassing hierarchy is that it's not a tree hierarchy. In a tree hierarchy, every class has at most one superclass. Here, both `TA` and `Faculty` have two superclasses. This kind of hierarchy actually form a dag (a directed acyclic graph). There is no a priori reason why dag hierarchies are problematic. After all, subclassing is just a relationship between classes that indicates, roughly, a subclass must implement all the methods that its superclass makes available, thereby ensuring 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. The main reason why dag hierarchies are interesting is that they allow you to maximize the amount of client code reuse by letting you define classes that can play the role that `Employee` plays in the example above, that is to tailor the expected arguments of methods to allow for maximum reuse of the method.

Many languages let you implement dag subclassing hierarachies cleanly. Java, not so much.

9.1 Multiple Superclasses in Java

In Java, there is a slight problem with wanting to have a a class with more than one superclass. 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. More on this in a few lectures.) That sucks, because as the `Employee` example above was meant to illustrate, there are natural hierarchies that are not tree hierarchies, and restricting you to tree hierarchies limits the amount of client code reuse you can get.

Java does give you a way out. It *is* possible to subclass from multiple classes, but all but at most one of them *must* be a fully abstract class, that is, a class with no implementation for any of its methods, and without any fields. These fully abstract classes are called *interfaces*, and because they are abstract they cannot be instantiated.

This means, in particular, that if your subclassing hierarchy is not a tree hierarchy and you want to implement it in Java, you have to decide which of the classes in your hierarchy will be interfaces. Sometimes it's pretty clear, other times it's more difficult. In the above example, `Employee` is a reasonable choice of an interface, because we added it in for the express purpose of getting a type to give to the arguments of `averageSalary()`, and therefore we have no expectation of instantiating `Employee`.

An interface is defined as follows:

```
public interface Employee {
  public int salary ();
}
```

As I said: 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`. This defines a subclassing relation between the class being defined and the interface.

Returning to the `Employee` example, here is an implementation of the `TA` and `Faculty` classes, which both are subclasses of `Employee`:

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

  // to allow subclassing
  protected Faculty () { }

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

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

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

  public int NUid () { return this.nuid; }

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

  public void newCourse (String course) { ...whatever... }
}
```

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

  // to allow subclassing
  protected TA () { }

  private TA (String n, int id, int s) {
    name = n;
    nuid = id;
    salary = s;
  }

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

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

  public int NUid () { return this.nuid; }

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

  public void registerForCourse (String course) { ...whatever... }
}
```

And that's it.

An interface really is just a fully abstract class. This means, in particular, that subclassing 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`.

There is no limit to the number of interfaces a class can implement; they can be listed as `implements X, Y, Z`. Also note that you can implement an interface without actually extending another class.