Nested Classes

Last time, we saw a recipe for deriving an implementation from an ADT specification.

There are two problems with that approach, however, one minor, one major:

(1) Namespace pollution

(2) Extensibility

Let's take care of the minor one first.

Consider the Stack class as derived from the recipe. It consists of a public abstract class `Stack`, and two subclasses, `EmptyStack` and `PushStack`. Now, because of the way Java works, either all of those classes must be made public, or, in the case where you decide to put all the classes in the same file, only `Stack` is made public, and the two subclasses are unannotated. This makes the two subclasses package-public: they are visible to other classes in the same package, but unavailable to classes outside the package. In other word, they are public for classes within the same package, but private for classes outside the package.

In practice, the only class we really want to be able to create instances of `EmptyStack` and `PushStack` is the `Stack` class. All stack object creation shold go through the `Stack` class static methods. In parts, this is because we rarely want to have objects specifically of class `EmptyStack` or `PushStack` about, but rather only want to have `Stack` objects. The static creators in the `Stack` class ensure this is the case.

So how do we prevent `EmptyStack` and `PushStack` from being available even from within the package. The answer, which may or may not be obvious, is to simply package up the two subclasses directly within the `Stack` class, and make them private so that they cannot be accessed from outside the class. Here is the code:

```
public abstract class Stack {

  public static Stack emptyStack () {
    return new EmptyStack ();
  }

  public static Stack push (Stack s, int i) {
    return new PushStack (s,i);
```

```
}

public abstract boolean isEmpty ();
public abstract int top ();
public abstract Stack pop ();


private static class EmptyStack extends Stack {

  public EmptyStack () { }

  public boolean isEmpty () {
    return true;
  }

  public int top () {
    throw new IllegalArgumentException ("Invoking top() on empty stack");
  }

  public Stack pop () {
    throw new IllegalArgumentException ("Invoking pop() on empty stack");
  }
}


private static class PushStack extends Stack {

  private int topVal;
  private Stack rest;

  public PushStack (Stack s, int v) {
    topVal = v;
    rest = s;
  }

  public boolean isEmpty () {
    return false;
  }

  public int top () {
    return this.topVal;
  }
```

```
    public Stack pop () {
      return this.rest;
    }
  }
}
```

These are examples of *nested classes*. In fact, these are *static* nested classes. A static nested class is just a class defined in its own file, except that it is defined within another class. (If a nested class T is defined as public within a class S, then class T can be accessed from outside S as $S.T$.)

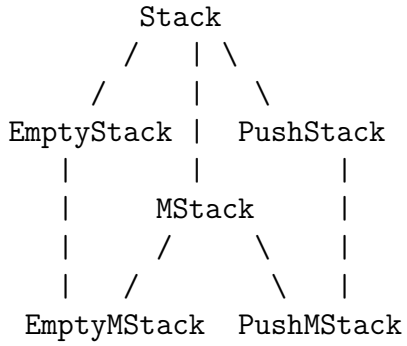
Why do we need the static qualifier? The static qualifier, as usual, indicates that the nested class is associated with the class definition itself. In particular, the nested class cannot access instance variables of the class containing it. This essentially says that the nested class is defined only once, when the containing class is itself defined. Note that this is a "containment" relation. Class S contains class definition T, just like it contains static methods and static fields.

(Without the static qualifier, a nested class (say T) is associated with instances of the class containing the definition (say S). Every instance of S "redefines" the nested class T. In particular, the nested class can refer to instance variables of S. When they are not static, nested classes are called *inner classes*. Inner classes are very powerful, and are related to closures, which can be used to do higher-order programming. We will not use inner classes much in this course, but it's good to know that they exist.)

Nesting classes takes care of the namespace pollution problem.

What about the other problem, extensibility? Recall the measurable stacks we talked about last time. There is no problem whatsoever applying the recipe to the measurable stack ADT. We just obtain an abstract class `MStack` with subclasses `EmptyMStack` and `PushMStack` (say). Just like before, we want `MStack` to be a subclass of `Stack`, which is easy to obtain: `public abstract class MStack extends Stack { ... }`.

However, applying the recipe naively duplicates a lot of the code already appearing in the `Stack` class. In particular, much of the code in `EmptyMStack` duplicates code in `EmptyStack`, and similarly for `PushMStack` duplicating code in `PushStack`. Ideally, we would like to inherit from `EmptyStack` in `EmptyMStack` and from `PushStack` in `PushMStack`. But if you think about it, the resulting hierarchy looks like this:

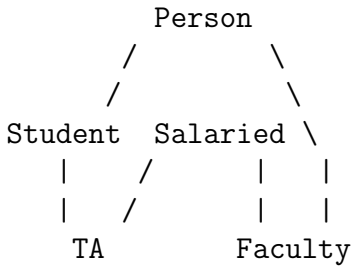


This hierarchy is not a tree, but a dag—a directed acyclic graph.

And that turns out to be a problem.

Interfaces

Let's look at a simpler example where the hierarchy is a dag. Recall the Person/Student/Faculty example of a few lectures ago. 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:



Again, this is not a tree. And it is still 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.

The problem is that Java conflates subclassing and inheritance. Recall that inheritance is an implementation technique for subclassing that lets us reuse code. In Java, the way to define subclasses is to inherit from a superclass, using the `extends` keyword. There is no real way to just say "subclass" without inheriting.

Why is this the problem? Because multiple inheritance—inheriting from multiple superclasses, is ambiguous. Consider the following classes `A`, `B`, `C`, `D`, defined in some hypothetical extension of Java with multiple inheritance. (I've elided the constructors of the classes, because I really care about the `foo` method anyways.)

```
class A {
  public int foo () { return 1; }
}

class B extends A { }

class C extends A {
  public int foo () { return 2; }
}

class D extends B,C { }
```

Class `B` inherits method `foo` from `A`, while `C` overwrites `A`'s `foo` method with its own. Now, suppose we have `d` an object of class `D`, and suppose that we invoke `d.foo()`. What do we get as a result. Because `D` does not define `foo`, we must look for it in its superclasses from which it inherits. But it inherits one `foo` method returning 1 from `B`, and one `foo` method returning 2 from `C`. Which one do we pick? There must be a way to choose one or the other. This is called the *diamond problem* (because the hierachy above looks like a diamond—well, a rhombus, which is a diamond if you squint real hard.) Different languages that support multiple inheritance have made different choices. The most natural is to simply look in the classes in the order in which they occur in the `extends` declaration. But that's a bit fragile, since a small change (flipping the order of superclasses) can make a big difference, and the small change can be hard to track down. There is also the problem of whether we look up in the hierarchy before looking right in the hierarchy. (We did not find `foo` in `B`; do we look for it in `A` before looking for it in `C`, or the other way around?) The point is, it becomes complicated very fast.

Java and many other languages take a different approach: forbid multiple inheritance altogether. You cannot inherit from more than one superclass. No problem with determining where to look for methods, then, if they are not in the current class—look in the (unique) superclass.

Unfortunately, as the Salaried example above was meant to illustrate, there are natural hierarchies that are not tree shaped. And having single inheritance keeps you from expressing those hierarchies, limiting what you can program.

To help, the Java designers give you a way out. They only allow you to inherit from a single superclass, but allow you to subclass from many—the only restriction is that you cannot inherit from those other superclasses. In fact, to enforce this, they require those superclasses to be "fully abstract"—they do not provide any method implementation, they only promise that subclasses will implement those methods. 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 note that this gives us pure subclassing; no methods are inherited. Implementing an interface is a promise that you will write the code for the methods described in the interface in the class.

Returning to the Salaried example, here is an implementation of the above hierarchy, including some of the code I gave a couple of lectures ago.

```
public class Person {
  private String name;
  private String nuid;
  public Person (String n, String id) {
    this.name = n;
    this.nuid = id;
  }
  public String getName () {
    return this.name;
  }
  public String getNUId () {
    return this.nuid;
  }
}

public class Student extends Person {
  public void registerForCourse (String course) {
      // some implementation for registration
  }
}
```

```
public class Faculty extends Person implements Salaried {
  public void makeAppointmentWithDean (String time) {
   // some implementation for appointment making
  }

  public int getSalary () {
    return 20000;
  }
}

public class TA extends Student implements Salaried {
  public int getSalary () {
    return 10000;
  }
}
```

Note that we have to implement the `Salaried` interface's method `getSalary` in all the classes that implement `Salaried`.