Multiple Inheritance and Interfaces

In Lecture 8, we saw a recipe for deriving an implementation from an ADT specification. I pointed out a few problems with that approach in Lecture 9:

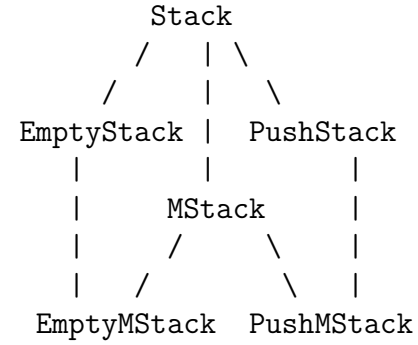
(1) Namespace pollution

(2) Extensibility

We talked about namespace pollution then, let's talk about extensibility now.

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. We saw that Java doesn't like non-tree hierarchies, and that it forces us to use interfaces.

Let's see why. Non-tree hierarchies are not an issue for subclassing. The problem is that *Java conflates subclassing and inheritance*. Subclassing allows you to reuse code on the client side, while inheritance allows you to reuse code on the implementation side. In other words,

inheritance is an implementation technique for subclassing that lets us reuse code. In Java, the way to define subclasses is to extend from a superclass using the `extends` keyword, and this extension not only defines a subclass, but also allows inheritance from the superclass. There is no nice way to just say "subclass" without allowing the possibility of inheriting in Java.

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 { }
```

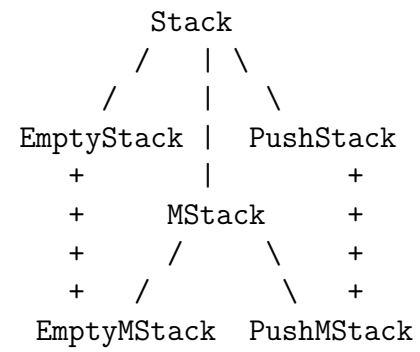
Class `B` inherits method `foo` from `A`, while `C` overwrites `A`'s `foo` method with its own. Now, suppose we have `d` an object of class `D`, and suppose that we invoke `d.foo()`. What do we get as a result. Because `D` does not define `foo`, we must look for it in its superclasses from which it inherits. 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.

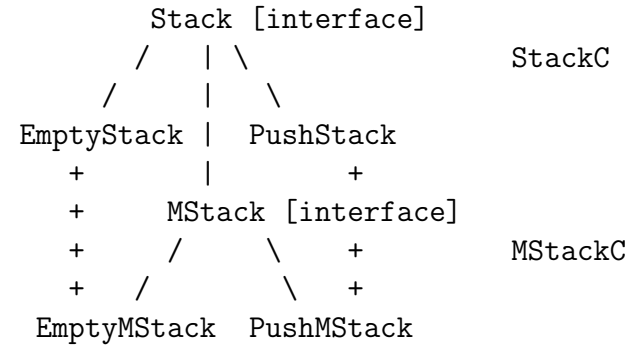
Because of this, the `extends` keyword in Java, which captures inheritance, can only be used to subclass a single superclass. If you want to subclass other classes as well, those have

to be interfaces. Interfaces are not a problem for inheritance, because they do not allow inheritance: interface contain no code, so there is no code to inherit.

Therefore, when you have a non-tree hierarchy, you need to first identify which subclassing relations between the class you want to rely on inheritance. This choice will force other classes to be interfaces. Consider the hierarchy for stacks and measurable stacks, then. We argued above that we wanted `EmptyMStack` to subclass and also inherit from `EmptyStack`, and for `PushMStack` to subclass and also inherit from `EmptyMStack`. These are represented in the diagram below using `+`:



This means, in particular, that `MStack` must be an interface. Because an interface cannot (in Java) subclass an actual class, but can only subclass another interface. This means that `Stack` also needs to be an interface. Not a problem, except for the fact that `Stack` and `MStack`, as per the recipe, should contain static methods corresponding to the creators. We cannot put them in interfaces. So where do they go? The best way to get around the problem is simply to define two new classes that implement only the creators. Let's call them `StackC` and `MStackC`. (I have no great suggestion for naming these classes.) Interestingly, if you think about it, these do not actually need to be in any relation, subclassing or otherwise, with the other classes. The picture we want, then, given the constraints that Java imposes, is the following:



Let's implement exactly that.

First, let's define the interfaces. These are just the signatures, minus the creators.

```
public interface Stack {
    public boolean isEmpty ();
    public Integer top ();
    public Stack pop ();
    public String toString ();
}
```

```
public interface MStack extends Stack {
    public boolean isEmpty ();
    public Integer top ();
    public MStack pop ();
    public String toString ();
    public int length ();
}
```

Then, the implementation of stacks, following a modified version of the recipe. (Exercise: can you make precise the modified recipe I am using here?):

```
public abstract class StackC {

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

    public static Stack push (Stack s, Integer i) {
      return new PushStack (s,i);
    }
}


// concrete class for empty stacks

class EmptyStack implements Stack {
    public EmptyStack () { }

    public boolean isEmpty () { return true; }

    public Integer top () {
        throw new IllegalArgumentException ("EmptyStack.top()");
    }
```

```
    public Stack pop () {
        throw new IllegalArgumentException ("EmptyStack.pop()");
    }

    public String toString () {
        return "<bottom of stack>";
    }
}


// concrete class for nonempty stacks

class PushStack implements Stack {
    private Integer topVal;
    private Stack rest;

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

    public boolean isEmpty () { return false; }

    public Integer top () { return this.topVal; }

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

    public String toString () { return this.top() + " " + this.pop(); }
}
```

Finally, the implementation of measurable stacks, following the modified recipe, and inheriting from the corresponding concrete classes in the stack implementation:

```
public abstract class MStackC {

    public static MStack empty () {
        return new EmptyMStack ();
    }

    public static MStack push (MStack s, Integer i) {
        return new PushMStack (s,i);
```

```
    }
}


// concrete class for empty stacks

class EmptyMStack extends EmptyStack implements MStack {
    public EmptyMStack () { }

    public int length () { return 0; }

    public MStack pop () { return (MStack) super.pop(); }
}


// concrete class for nonempty stacks

class PushMStack extends PushStack implements MStack {
    private int itemsCount;

    public PushMStack (MStack s, Integer v) {
        super(s,v);
        itemsCount = s.length() + 1;
    }

    public int length () { return itemsCount; }

    public MStack pop () { return (MStack) super.pop(); }
}
```