16 Multiple Inheritance and Extending ADTs

We looked last time at inheritance and delegation as two ways to reuse implementation code, and finished with a bit of a puzzle about how to reuse code in the context of the design pattern to derive code from a specification, something that I'll call the Specification Design Pattern from now on, because it's starting to be a mouthful.

Recall the specification for the List ADT, specialized to integers:

```
static List empty ()
static List cons (int, List)
boolean isEmpty ()
int first ()
List rest ()
String toString ()

empty().isEmpty() = true
cons(i,s).isEmpty() = false
cons(i,s).first() = i
cons(i,s).rest() = s
empty().toString() = "()"
cons(i,s).toString() = "(" + i + " . " + s.toString() + ")"
```

I've added `toString()` to give us an interesting operation for later.

16.1 A Detour: Inheritance in the Specification Design Pattern

Suppose we add in a new operation, `kind`, that returns a string with the kind of data structure at hand:

```
String kind ()
```

with specification:

```
empty().kind() = "list"
cons(i,s).kind() = "list"
```

(I'm not claiming this is an interesting operation, but it illustrates what I want to show.) You can now pretty much predict that the design pattern will yield in terms of implementation: the abstract class `List` will have an abstract method `kind()`, and the concrete subclasses `EmptyList` and `ConsList` will each implement a method `kind()`, each returning the string `"list"`. That's wasteful: two methods `kind()` with exactly the same code in both subclasses. We can avoid this code duplication by moving the definition of method `kind()` into the abstract class `List`, and rely on inheritance to make that method available in the subclasses:

```
public abstract class List {

  public static List empty () { return new EmptyList(); }

  public static List cons (int i, List l) { return new ConsList(i,l); }

  public abstract boolean isEmpty ();
  public abstract int first ();
  public abstract List rest ();

  public String kind () { return "list"; }
}


class EmptyList extends List {

    public EmptyList () {}

    public boolean isEmpty () { return true; }

    public int first () { throw new Error ("first() on an empty list"); }

    public List rest () { throw new Error ("rest() on an empty list"); }

    public String toString () { return "()"; }
}


class ConsList extends List {

    private int first;
    private List rest;

    public ConsList (int f, List r) {
```

```
    first = f;
    rest = r;
  }

  public boolean isEmpty () { return false; }

  public int first () { return first; }

  public List rest () { return rest; }

  public String toString () { return "(" + first + " . " + rest.toString() + ")"; }
}
```

Thus, for instance:

```
List l = List.cons(1,List.cons(2,List.empty()));
System.out.println("Kind of l is = " + l.kind());
```

with the expected output:

```
Kind of l is = list
```

Got that? Remember the lesson: we can sometimes use inheritance to "optimize" the resulting implementation, in the sense of reducing the amount of duplicated code.

16.2 Multiple Inheritance

Recall the measurable lists we talked about last time, with the following signature and specification. (Forget about the `kind()` operation for now.)

```
static MList empty ()
static MList cons (int, MList)
boolean isEmpty ()
int first ()
MList rest ()
String toString ()
int length ()

empty().isEmpty() = true
cons(i,s).isEmpty() = false
cons(i,s).first() = i
```

```
cons(i,s).rest() = s
empty().toString() = "()"
cons(i,s).toString() = "(" + i + " . " + s.toString() + ")"
empty().length() = 0
cons(i,s).length() = 1 + s.length()
```

(Eventually, we'll want to make the `length()` operation be efficient, that is, constant time, without having to walk down the whole list. For now, let's not worry about that part.)

There is no problem whatsoever applying the Specification Design Pattern to the measurable list ADT:

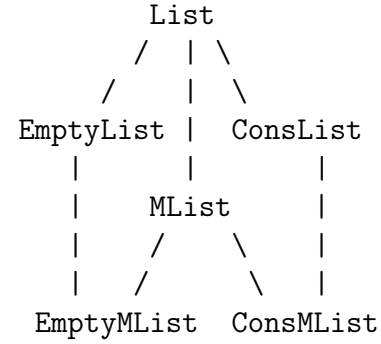
```
public abstract class MList {

  public static MList empty () { return new EmptyMList(); }

  public static MList cons (int i, MList l) { return new ConsMList(i,l); }

  public abstract boolean isEmpty ();
  public abstract int first ();
  public abstract MList rest ();
  public abstract String toString ();
  public abstract int length ();
}


class EmptyMList extends MList {

    public EmptyMList () {}

    public boolean isEmpty () { return true; }

    public int first () { throw new Error ("first() on an empty list"); }

    public MList rest () { throw new Error ("rest() on an empty list"); }

    public String toString () { return "()"; }

    public int length () { return 0; }
}


class ConsMList extends MList {
```

```
  private int first;
  private MList rest;

  public ConsMList (int f, MList r) {
    first = f;
    rest = r;
  }

  public boolean isEmpty () { return false; }

  public int first () { return first; }

  public MList rest () { return rest; }

  public String toString () { return "(" + first + " . " + rest.toString() + ")"; }

  public int length () { return 1+rest.length(); }
}
```

It makes sense to want to have an `MList` be a subclass of a `List`, because after all, any operation that works on lists should work perfectly fine on `MList`s. Now, it is easy to make `MList` a subclass of `List`, by simply defining it so:

```
public abstract class MList extends List {
  ...
```

Note that the implementation of `MList` duplicates a lot of code already appearing in the `List` class. In particular, much of the code in `EmptyMList` duplicates code in `EmptyList`, and similarly for `ConsMList` duplicating code in `ConsList`. Is there a way we can inherit from `EmptyList` in `EmptyMList` and from `ConsList` in `ConsMList`? Let's draw a picture — the resulting subclassing hierarchy (because inheritance in Java can only be done by also subclassing) 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 a problem for subclassing, we saw that. 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 (generally 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.)

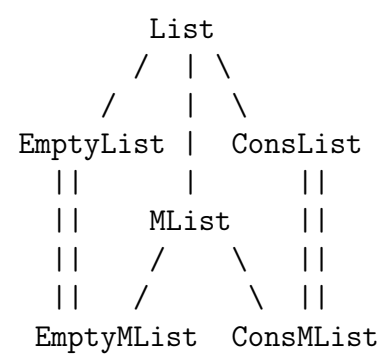
```
class A {
  public int foo () { return 1; }
}

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) 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.

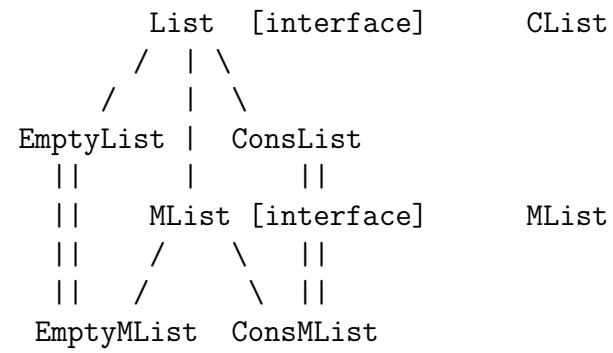
Java and many other languages take a different approach: forbid multiple inheritance altogether, you cannot inherit from more than one superclass. Then there is no problem with

determining where to look for methods if they are not in the current class: look in the (unique) superclass. This is why the `extends` keyword in Java, which expresses 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 lists and measurable lists, then. We argued above that we wanted `EmptyMList` to subclass and inherit from `EmptyList`, and for `ConsMList` to subclass and also inherit from `ConsList`. Inheritance relations are expressed with double lines in the the diagrams:



This means, in particular, that `MList` must be an interface. Because an interface cannot (in Java) subclass an actual class, but can only subclass another interface, this means that `List` also needs to be an interface. That's not necessarily a problem, except for the fact that `List` and `MList`, as per the Specification Design Pattern, 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 `CList` and `CMList`. (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:



That's a mess. It's implementable, but it's a mess. And it's a pain to use, you have to use both `CList` to create lists, and the lists created are of type `List`. Yuck. There are other ways of cleaning up that problem, but they're equally ugly.

There are other problems. Recall what we saw in §16.1, that we could use inheritance to reduce code duplication for some methods in `List` (say, with a method such as `kind()`). Well, we cannot use that here, because `List` is now an interface, and therefore cannot define methods. So we have to chose: do we use inheritance to reduce code duplication in the implementation of an ADT, or do we use inheritance to reduce code duplication for extensions of an ADT? We can't have it both ways.

Exercise: *Implement lists and measurable lists respecting the subclassing and inheritance relations in the above diagram.*

16.3 Delegation for ADTs Extension

So inheritance for avoiding code duplication when extending an ADT is not great — the resulting subclassing hierarchy is a pain to work with, and we cannot reuse as much code as we would like. This is where delegation comes in handy. Note that we want to reuse code from `EmptyList` in `EmptyMList` and from `ConsList` in `ConsMList`, but we don't actually care about them being subclasses of the others. After all, we never write code that expects to receive a `EmptyList` or a `ConsList`, we only every write code that expects a `List`, which will happily accept a `MList`, that is either an `EmptyMList` or a `ConsMList`.

So let's implement `MList` by having the subclasses `EmptyMList` and `ConsMList` delegate to an underlying list, allowing us to reuse the methods defined in `List`.

```
public abstract class MList extends List {

   public static MList empty () { return new EmptyMList(); }

   public static MList cons (int i, MList l) { return new ConsMList(i,l); }

   public abstract boolean isEmpty ();
   public abstract int first ();
   public abstract MList rest ();
   public abstract String toString ();
   public abstract int length ();
}


class EmptyMList extends MList {

   private List del;
```

```
    public EmptyMList () { this.del = List.empty(); }

    public boolean isEmpty () { return del.isEmpty(); }

    public int first () { return del.first(); }

    public MList rest () { return (MList) del.rest(); }

    public String toString () { return "()"; }

    public int length () { return 0; }
}


class ConsMList extends MList {

    private List del;

    public ConsMList (int v, MList l) { this.del = List.cons(v,l); }

    public boolean isEmpty () { return del.isEmpty(); }

    public int first () { return del.first(); }

    public MList rest () { return (MList) del.rest(); }

    public String toString () { return "(" + first() + " . " + rest().toString() + ")"; }

    public int length () { return 1+this.rest().length(); }
}
```

That works. We get to reuse code from `List`, although it's not so obvious here because the classes are so small, and we don't have to worry about how `List`s are actually implemented.

We can somewhat better though. Note that each of the concrete subclasses has a `del` field, and the methods `isEmpty()`, `first()`, and `rest()` in the concrete subclasses look the same. So we can use the trick we used in §16.1 to move that field and those methods into the abstract base class `MList`, and providing them in the subclasses via inheritance.

```
public abstract class MList extends List {

    protected List del;
```

```
    public static MList empty () { return new EmptyMList(); }

    public static MList cons (int i, MList l) { return new ConsMList(i,l); }

    public boolean isEmpty () { return del.isEmpty(); }

    public int first () { return del.first(); }

    public MList rest () { return (MList) del.rest(); }

    public abstract String toString ();
    public abstract int length ();
}


class EmptyMList extends MList {

    public EmptyMList () { this.del = List.empty(); }

    public String toString () { return "()"; }

    public int length () { return 0; }
}


class ConsMList extends MList {

    public ConsMList (int v, MList l) { this.del = List.cons(v,l); }

    public String toString () { return "(" + first() + " . " + rest().toString() + ")"; }

    public int length () { return 1+this.rest().length(); }
}
```

And that's much much better — now the concrete subclasses really only include the bits that make measurable lists different from normal lists.

Finally, let's worry about making `length()` more efficient, following what we said at the end of last lecture, recording the length in a field and setting that field appropriately at construction time. There is no "pattern" for getting this to work in general — this requires understanding the encoding and how we can get something better out of it.

```
public abstract class MList extends List {

   protected List del;
   protected int length;

   public static MList empty () { return new EmptyMList(); }

   public static MList cons (int i, MList l) { return new ConsMList(i,l); }

   public boolean isEmpty () { return del.isEmpty(); }

   public int first () { return del.first(); }

   public MList rest () { return (MList) del.rest(); }

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

   public abstract String toString ();
}


class EmptyMList extends MList {

   public EmptyMList () {
     this.del = List.empty();
     this.length = 0;
   }

   public String toString () { return "()"; }
}


class ConsMList extends MList {

   public ConsMList (int v, MList l) {
     this.del = List.cons(v,l);
     this.length = 1 + l.length();
   }

   public String toString () { return "(" + first() + " . " + rest().toString() + ")"; }
}
```

Note that the length of a list is accumulated in a field `length` at construction time, meaning that when we ask for the length of a list, we can simply look it up in the field instead of computing it from scratch every time.