17 Multiple Inheritance and ADT Extensions

We looked last time at inheritance and delegation as two ways to reuse implementation code. We also finished with an implementation of measurable lists, that uses delegation to extend a standard implementation for lists. We will revisit measurable lists here. First, we will rely on the following implementation of lists, obtained using the Specification design pattern.

```
public abstract class List<A> {

  public static <B> List<B> empty () {
    return new EmptyList<B>();
  }
  public static <B> List<B> cons (B i, List<B> l) {
    return new ConsList<B>(i,l);
  }

  public abstract boolean isEmpty ();
  public abstract A first ();
  public abstract List<A> rest ();
}


class EmptyList<A> extends List<A> {

  public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }
  public A first () {
    throw new Error ("first() on an empty list");
  }
  public List<A> rest () {
    throw new Error ("rest() on an empty list");
  }
}
```

```
class ConsList<A> extends List<A> {
  private A first;
  private List<A> rest;

  public ConsList (A f, List<A> r) {
    first = f;
    rest = r;
  }

  public boolean isEmpty () {
    return false;
  }
  public A first () {
     return first;
  }
  public List<A> rest () {
    return rest;
  }
}
```

17.1 Multiple Inheritance

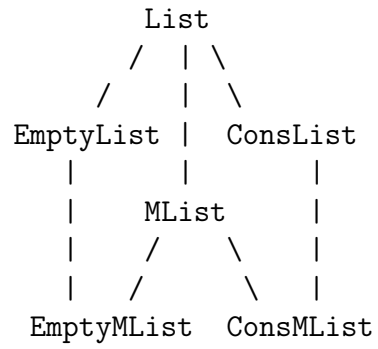
Recall measurable lists, with the following signature and specification for the `MList<A>` ADT:

```
 CREATORS:       static <B> MList<B> empty ()
                 static <B> MList<B> cons (B, MList<B>)
 OPERATIONS:     boolean isEmpty ()
                 A first ()
                 MList<A> rest ()
                 int length ()
 SPECIFICATIONS:
  empty().isEmpty() = true
  cons(i,s).isEmpty() = false
  cons(i,s).first() = i
  cons(i,s).rest() = s
  empty().length() = 0
  cons(i,s).length() = 1 + s.length()
```

As we said last time, there is no problem whatsoever applying the Specification Design Pattern to the measurable list ADT. Last time, we saw that we could extend the list ADT to obtain an implementation of the measurable list ADT.

That extension was rather ad hoc, however. In particular, it doesn't follow the Specification design pattern. Suppose we wanted to implement the measurable list ADT following the Specification design pattern (so that we have a base class `MList<A>` and two concrete subclasses `EmptyMList<A>` and `ConsMList<A>`) and still reuse code from the implementation of lists.

What would the structure of the classes looks like though? It makes sense to want to have `MList` be a subclass of `List`, because after all, any operation that works on lists should work perfectly fine on `MList`s. But there is more. 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) would look 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, so that 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.

If we *had* multiple inheritance, then we could write something like this to extend the `List` implementation given earlier:

```
public abstract class MList<A> extends List<A> {

  public static <B> MList<B> empty () {
    return new EmptyMList<B>();
```

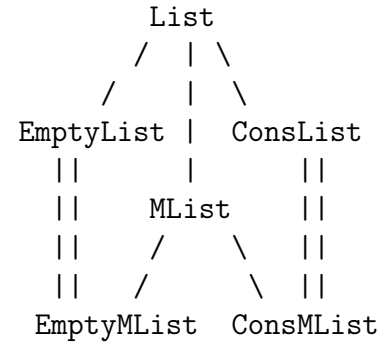
```
    }
    public static <B> MList<B> cons (B i, MList<B> l) {
        return new ConsMList<B>(i,l);
    }

    public abstract boolean isEmpty ();
    public abstract A first ();
    public abstract MList<A> rest ();
    public abstract int length ();
}


class EmptyMList<A> extends EmptyList<A>, MList<A> {
    public EmptyMList () {}

    public MList<A> rest () {
      return (MList<A>) super.rest();

    public int length () {
        return 0;
    }
}


class ConsMList<A> extends ConsMList<A>, MList<A> {
    private A first;
    private MList<A> rest;

    public ConsMList (A f, MList<A> r) {
        first =f ;
        rest = r;
    }

    public MList<A> rest () {
        return (MList<A>) super.rest();
    }

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

This would be ideal.[1] The only thing we need to write, really, is what makes measurable lists different than normal lists. In particular, we only need to create an abstract base class for `MList<A>` really only containing the creators, and two concrete subclasses `EmptyMList<A>` and `ConsMList<A>`, which only really include the methods that differ from those in `EmptyList<A>` and `ConsList<A>`. (Here I'm assuming that methods are inherited in the order in which classes are described after `extends`.)

The above cannot be written in Java because we do not have multiple inheritance, but we'll see three ways by which we can approximate the above.

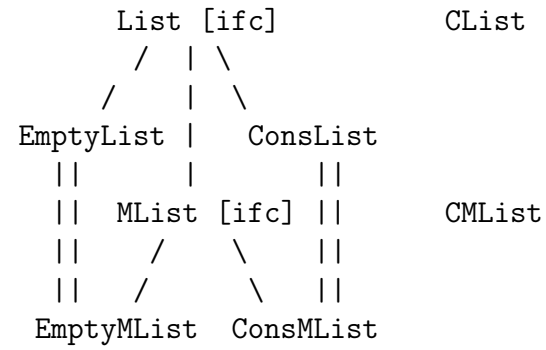
17.2 ADT Extension via Inheritance

Consider the hierarchy for lists and measurable lists. 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 if we want to do things this way, then `MList` must be an interface. Because an interface in Java cannot subclass an actual class, but can only subclass another interface, this means that `List` also needs to be an interface. (So we need to change the implementation of lists.) An additional problem is that `List` and `MList`, as per the Specification Design Pattern, should contain static methods corresponding to the creators. We cannot put them in interfaces, because interfaces contain no code. 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:

[1] Technically, we could also move the `rest()` method in the base class `MList<A>` and it would be available in the subclasses by inheritance, but that's minor.



That's a mess. It's implementable, but it's a mess. Here is the resulting implementation, first the modified `List` implementation:

```
public interface List<A> {
  public boolean isEmpty ();
  public A first ();
  public List<A> rest ();
}


class EmptyList<A> implements List<A> {
  public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }
  public A first () {
    throw new Error ("first() on an empty list");
  }
  public List<A> rest () {
    throw new Error ("rest() on an empty list");
  }
}


class ConsList<A> implements List<A> {
  private A firstElement;
  private List<A> restElements;

  public ConsList (A f, List<A> r) {
    firstElement = f;
    restElements = r;
```

```
  }

  public boolean isEmpty () {
    return false;
  }
  public A first () {
      return firstElement;
  }
  public List<A> rest () {
    return restElements;
  }
}
```

with associated creators:

```
public abstract class CList {

  public static <B> List<B> empty () {
    return new EmptyList<B>();
  }
  public static <B> List<B> cons (B i, List<B> l) {
    return new ConsList<B>(i,l);
  }
}
```

and the implementation of `MList`:

```
public interface MList<A> extends List<A> {
  public boolean isEmpty ();
  public A first ();
  public MList<A> rest ();
  public int length ();
}


class EmptyMList<A> extends EmptyList<A> implements MList<A> {
    public EmptyMList () {}

    public MList<A> rest () {
        return (MList<A>) super.rest();
    }
    public int length () {
```

```
        return 0;
    }
}


class ConsMList<A> extends ConsList<A> implements MList<A> {
    private A first;
    private MList<A> rest;

    public ConsMList (A val, MList<A> l) {
        super(val,l);
        first = val;
        rest = l;
    }

    public MList<A> rest () {
        return (MList<A>) super.rest();
    }
    public int length () {
        return 1 + rest.length();
    }
}
```

and its associated creators:

```
public abstract class CMList {

  public static <B> MList<B> empty () {
    return new EmptyMList<B>();
  }
  public static <B> MList<B> cons (B i, MList<B> l) {
    return new ConsMList<B>(i,l);
  }
}
```

Not only does this require us to modify the implementation of `List`, which sometimes is not possible because you do not have the source code available, it is also a pain to use, since now you have to use `CList` to create lists, while 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. In some ADT implementations, it makes sense to use implement methods in the abstract base class if they occur as is in all (or most) of the concrete subclasses.

For instance, the `equals()` method of an ADT is often defined in the abstract base class, and inherited in the concrete subclasses. Similarly, we can have helper methods defined in the base class if they are useful in the concrete subclasses. This can be useful to avoid code duplication *within* an implementation of an ADT. (We will see an example of just that below in §17.4.) Well, we cannot do that here, because `MList` 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.

17.3 ADT Extension via Delegation

Here is an alternative implementation, that does not require us to change `List`, and that does not require us to have a different class implementing the creators of the ADT. We will keep `MList<A>` an abstract class, meaning that `EmptyMList<A>` and `ConsMList<A>` will extend `MList<A>`, meaning they will not be able to inherit from the concrete subclasses of `List<A>`. So instead of using inheritance, we will use delegation.

We use the original implementation of `List<A>` I gave at the beginning of this lecture (the one obtained from the Specification design pattern), and implement `MList<A>` using the Specification design pattern as well, except that concrete subclasses use delegation to create an instance of `List` to which to delegate most of the methods except for the one that need to be defined for measurable lists. Here is the code:

```
public abstract class MList<A> extends List<A> {

  public static <B> MList<B> empty () {
    return new EmptyMList<B>();
  }
  public static <B> MList<B> cons (B i, MList<B> l) {
    return new ConsMList<B>(i,l);
  }

  public abstract boolean isEmpty ();
  public abstract A first ();
  public abstract MList<A> rest ();
  public abstract int length ();
}


class EmptyMList<A> extends MList<A> {
    private List<A> del;   // delegate

    public EmptyMList () {
```

```
        del = List.empty();
    }

    public boolean isEmpty() {
        return del.isEmpty();
    }
    public A first () {
        return del.first();
    }
    public MList<A> rest () {
        return (MList<A>) del.rest();
    }
    public int length () {
        return 0;
    }
}


class ConsMList<A> extends MList<A> {
    private List<A> del;    // delegate
    private A first;
    private MList<A> rest;

    public ConsMList (A f, MList<A> r) {
        del = List.cons(f,r);
        first = f;
        rest = r;
    }

    public boolean isEmpty() {
        return del.isEmpty();
    }
    public A first () {
        return del.first();
    }
    public MList<A> rest () {
        return (MList<A>) del.rest();
    }
    public int length () {
        return 1 + rest.length();
    }
}
```

Notice what we do: the abstract base class is standard, and the concrete subclasses as well, except each defines a field containing the delegate, instanciates that delegate in the constructor, and delegates most of the operations to that delegate, adjusting the type when needed (`rest()`), while defining the measurable-list-specific methods.

That's pretty good, although it is still more code than the "ideal" implementation in §17.1. But it doesn't require us to change `List<A>`, it doesn't require us to put the creators in a different class, and it is still very much a uniform design pattern that we can apply more or less blindly.

17.4 ADT Extension via Delegation and Inheritance

In the last section, we managed to not duplicate any code from `List<A>` in `MList<A>`, thereby achieving a good amount of code reuse. (Were `List<A>` a larger ADT, we would get even more code reuse out of it.) There is still, however, a lot of code duplication between the concrete subclasses of `MList<A>`. In particular, the code that does the delegation looks exactly the same in `EmptyMList<A>` and in `ConsMList<A>`: each of the concrete subclasses has a `del` field, and the methods `isEmpty()`, `first()`, and `rest()` in the concrete subclasses look the same. Can we effect some code reuse there?

It turns out yes, by using the fact that the concrete subclasses of `MList<A>` can inherit from `MList<A>`. We can move the `del` field and the methods `isEmpty()`, `first()` and `rest()` into the abstract base class `MList<A>`, and provide them in the subclasses via inheritance. The only thing we need to do in the concrete subclasses is create the appropriate delegate. (That could also be put in the creators. It's a minor variant.)

Here is the resulting code:

```
public abstract class MList<A> extends List<A> {
   protected List<A> del;    // delegate

   public static <B> MList<B> empty () {
      return new EmptyMList<B>();
   }
   public static <B> MList<B> cons (B i, MList<B> l) {
      return new ConsMList<B>(i,l);
   }

   public boolean isEmpty () {
      return del.isEmpty();
   }
   public A first () {
      return del.first();
   }
```

```
    public MList<A> rest () {
        return (MList<A>) del.rest();
    }

    public abstract int length ();
}


class EmptyMList<A> extends MList<A> {
    public EmptyMList () {
        del = List.empty();
    }

    public int length () {
        return 0;
    }
}


class ConsMList<A> extends MList<A> {
    private A first;
    private MList<A> rest;

    public ConsMList (A f, MList<A> r) {
        del = List.cons(f,r);
        first =f ;
        rest = r;
    }

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

And that's pretty much the best I can do. It is as close to the "ideal" implementation from §17.1 as I can get. The only difference is that there is some additional code in the abstract base class to delegate the methods already defined in the `List<A>` class, and the setting of the delegate in the constructor of each of the concrete subclasses. Aside from that, the code contains only what's new in measurable lists.

17.5 Measurable Lists as an Augmented Data Structure

Let me close by illustrating a feature that has nothing to do with inheritance or delegation, but is something nice that I can illustrate simply with measurable lists. It has to do with efficiency of operations.

The naive implementation of a length method by simply counting how many elements are in the list can be inefficient if the list is large. There is no way to make the "count how many elements are in the list" algorithm more efficient, but there is a way to implement measurable list to make the `length()` method more efficient: keep a count of the current list size alongside the list content, and simply increment the count upon a `cons()`. The `length()` method now simply returns the current list size, a constant-time field lookup operation. This is an example of an *augmented data structure*, a data structure augmented with information that make some operations more efficient.

There is no design pattern specifically for writing augmented data structures. It is usually done more or less on a case-by-case basis, although general lessons usually emerge. Here is how you can get this to work for measurable lists.

```
public abstract class MList<A> extends List<A> {
   protected List<A> del;       // delegate
   protected int length;

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

   public static <B> MList<B> cons (B i, MList<B> l) {
      return new ConsMList<B>(i,l);
   }

   public boolean isEmpty () {
      return del.isEmpty();
   }
   public A first () {
      return del.first();
   }
   public MList<A> rest () {
      return (MList<A>) del.rest();
   }
   public int length () {
      return this.length;
   }
}
```

```
class EmptyMList<A> extends MList<A> {
   public EmptyMList<A> () {
     del = List.empty();
     length = 0;
   }
}


class ConsMList<A> extends MList<A> {
   private A first;
   private MList<A> rest;
   public ConsMList (A f, MList<A> r) {
     del = List.cons(v,l);
     length = 1 + l.length();
     first = f;
     rest = r;
   }
}
```

(Clearly, I could get rid of the fields in `ConsMList<A>` since they are not needed, but if measurable lists had other operations that needed them, they'd be there.)

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.