# 20 Subtyping and Mutation

Suppose we have a class `A` that subclasses a class `B` (which I will write `A` $\leq$ `B`). Should we consider an aggregate class over `A` to be a subclass of that aggregate class over `B` — more concretely, should `A[]` $\leq$ `B[]`, or `List<A>` $\leq$ `List<B>`, or `Option<A>` $\leq$ `Option<B>`? Intuitively, the answer should be yes. After all, if I write a method that can happily work with lists of `B`s, then giving it a list of `A`s should work just as well, because every `A` is a `B` due to subclassing. Unfortunately, in the presence of mutation, it's not that simple: it's sometimes false!

So how can a type system deal with possible subtyping between an aggregate over `A` and an aggregate over `B`? There are really three approaches:

(1) Allow the aggregate over `A` to be a subtype of the aggregate over `B`, but do runtime checks to prevent potential problems.

(2) Disallow the aggregate over `A` to be a subtype of the aggregate over `B`.

(3) Sometimes allow the aggregate over `A` to be a subtype of the aggregate over `B`, and sometimes disallow it.

Java, it turns out, has both (1) and (2). Let's investigate.

And recall what I pointed out earlier in the course, that the main property enforced by the Java type system is the following: *if a program type checks, then at no point during the execution of the program does the system attempt to invoke a method* `meth` *on an object that does not provide method* `meth`*.*

## 20.1 Subtyping for Arrays in Java

Arrays are treated specially in Java. The type system uses the following rule to determine subtyping of array types: whenever `S` is a subtype of `T`, then `S[]` is a subtype of `T[]`. Now, this means that the following code type checks, since `String` $\leq$ `Object` and thus `String[]` $\leq$ `Object[]`:

```
public class Test1 {
   public static void main (String[] argv) {
      Integer[] a = {1,2,3};
```

```
        show(a);
        System.out.println("First element * 2  = " + (2*a[0].intValue()));
    }

    public static void show (Object[] a) {
        for (Object i : a) {
            System.out.println(i.toString());
        }
    }
}
```

(Recall that the `Integer` class is just a wrapper around an integer, with a `intValue()` method to extract the underlying integer.) Of course, this code executes perfectly well: it is okay to pass the array of integers to `show`, because each of the `Integer` will have a `toString()` method, and execution proceeds without encountering an undefined method.

The problem is that the following code also type checks for the same reason, namely that `String[]` $\leq$ `Object[]`:

```
public class Test2 {
    public static void main (String[] argv) {
        Integer[] a = {1,2,3};
        show(a);
        update(a);
        System.out.println("First element * 2  = " + (2*a[0].intValue()));
    }

    public static void show (Object[] a) {
        for (Object i : a) {
            System.out.println(i.toString());
        }
    }

    public static void update (Object[] a) {
        a[0] = new Object();
    }
}
```

`Test2` is very similar to `Test1`, except that method `update` to which we pass the array of integers modifies the first element of the array, making it hold a new `Object`. First, make sure that you understand why the code above type checks: Because `Integer` $\leq$ `Object`, the type system lets you pass an `Integer[]` to a method expecting an `Object[]`. And because

the array `a` is an array of `Object`, the type system is quite happy to let you update the first element in the array into a different `Object`.

The problem is that passing an object (including an array) to a method in Java only passes a reference to that object. The object is not actually copied, as we saw when we saw the mutation model. So when method `update` updates the array through its argument `a`, it ends up modifying the underlying array `a` in method `main`. But that means that when we come back from the `update` method, array `a` is an array of integers where the first element of the array is not an integer any longer, but rather an instance of `Object`. And when we attempt to invoke method `intValue()` on that first element, Java would choke because that first element, being a plain `Object`, does not in fact implement the `intValue()` method! That contradicts the guarantee the type system is supposed to make. In other word, the type system messed up — it said something was okay when it wasn't.

Java trades off this inadequacy of the type system by doing a runtime check at the statement that causes the problem: the update `a[0] = new Object()`. Java catches the fact that you are attempting to modify an array by putting in an object that is not a subclass of the dynamic type of the data in the array, and throws an `ArrayStoreException`. Here, that's because we are trying to put an `Object` into an array originally created to hold `Integer`s.

The point remains: the type system does not fully do its job, and has to deleguate to the runtime system the responsability of ensuring that the problem above does not occur. And that's a problem — recall that lecture we had about why it was a good idea to report errors early, such as when the program is being compiled as opposed to when it executes?

That's approach (1), then, accept the subtyping between aggregates, that Java uses for arrays.

## 20.2 Subtyping for Generics in Java

The above examples use arrays. What about using a polymorphic class that is not predefined like arrays are? For example, the `List<A>` ADT that we've been using for the past weeks, augmented with both functional and mutable iterators. Here is `Test1`, rewritten with lists:

```
public class Test3 {

   public static void main (String[] argv) {
      List<Integer> a = List.empty();
      a = List.cons(1,List.cons(2,List.cons(3,a)));
      show(a);
      System.out.println("First element * 2  = " + (2*a.first().intValue()));
   }

   public static void show (List<Object> a) {
```

```
        for (Object i : a) {
            System.out.println(i.toString());
        }
    }
}
```

Trying to compile this program fails miserably: it does not type check. In fact, it is *not the case* that if `S` is a subtype of `T`, then `List<S>` is a subtype of `List<T>`. And that's the case for all uses of generics.[1] This seems counterintuitive, but it prevents us from writing code such as in `Test2` that updates an aggregate structure and forces us to do a runtime check and possibly throw an exception. Bottom line: we cannot write code such as that in `Test2` using generics.

Of course, we also cannot write code such as in `Test3`, which is a bit like throwing the baby out with the bathwater. Code such as that in `Test3` is actually quite useful, and works fine. It's only when we update an aggregate structure that problems occur. There are ways around that, usable in some situations, which reinstate some amount of subtyping. But we have to be explicit about where we want the subtyping. Consider the type for `show` in `Test3`. Suppose we wanted to be explicit about the kind of subtyping we allowed here. Roughly, we would like it to say that `show` accepts any list with some type of element `T` that is a subclass of `Object`. We don't care and don't know what that type of element `T` is, so we'll write it down as a question mark. We therefore get the code:

```
public class Test4 {

    public static void main (String[] argv) {
        List<Integer> a = List.empty();
        a = List.cons(1,List.cons(2,List.cons(3,a)));
        show(a);
        System.out.println("First element * 2  = " + (2*a.first().intValue()));
    }

    public static void show (List<? extends Object> a) {
        for (Object i : a) {
            System.out.println(i.toString());
        }
    }
}
```

[1] Why, one might ask? Wouldn't it have made more sense to make generics behave like arrays? Turns out that's because of the way that generics are implemented: parameters are erased and replaced by `Object` before execution, meaning that the system does not have the dynamic data required to do the kind of checking that occurs at updates in order to throw the exception we saw in `Test2`.

And this type checks perfectly okay, and executes perfectly okay. The subtyping rule for this kind of generics is as follows: if `S` is a subtype of `T`, then `List<S>` is a subtype of `List<? extends T>`. Wrap your head around this rule, and the above example.

So, we can reinstate some form of subtyping for generics. Have we added too much? Can we write a version of `Test2` in this setting? In order to write something like `Test2`, we need to be able to mutate lists. So let's modify our implementation of `List<A>` to make it mutable, by adding an operation `void mutate (A v)` that replaces the first element of a list by `v`:

```
import java.util.Iterator;
import java.lang.Iterable;

public abstract class ListMut<A> implements Iterable<A> {
   public static <A> ListMut<A> empty () {
      return new EmptyListMut<A>();
   }
   public static <A> ListMut<A> cons (A v, ListMut<A> l) {
      return new ConsListMut<A>(v,l);
   }

   public abstract boolean isEmpty ();
   public abstract A first ();
   public abstract void mutate (A val);
   public abstract ListMut<A> rest ();
   public abstract String toString ();

   public abstract FuncIterator<A> getFuncIterator ();

   public Iterator<A> iterator () {
      return IteratorAdapter.create(this.getFuncIterator());
   }
}


class EmptyListMut<A> extends ListMut<A> {
   public EmptyListMut () {}

   public boolean isEmpty () { return true; }

   public A first () { throw new RuntimeException("EmptyList.first()"); }

   public void mutate (A val) {
      throw new RuntimeException("EmptyList.mutate()");
```

```
    }

    public ListMut<A> rest () { throw new RuntimeException("EmptyList.rest()"); }

    public String toString () { return ""; }

    public FuncIterator<A> getFuncIterator () {
        return new EmptyFuncIterator<A>();
    }
}


class EmptyFuncIterator<A> implements FuncIterator<A> {
    public EmptyFuncIterator () {}

    public boolean hasElement () { return false; }

    public A current () {
        throw new java.util.NoSuchElementException("EmptyFuncIterator.current()");
    }

    public FuncIterator<A> advance () {
        throw new java.util.NoSuchElementException("EmptyFuncIterator.advance()");
    }
}


class ConsListMut<A> extends ListMut<A> {
    private A first;
    private ListMut<A> rest;

    public ConsListMut (A f, ListMut<A> r) {
        this.first = f;
        this.rest = r;
    }

    public boolean isEmpty () { return false; }

    public A first () { return this.first; }

    public void mutate (A val) { this.first = val; }
```

```
    public ListMut<A> rest () { return this.rest; }

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

    public FuncIterator<A> getFuncIterator () {
        return new ConsFuncIterator<A>(this.first(),
                                       this.rest().getFuncIterator());
    }
}


class ConsFuncIterator<A> implements FuncIterator<A> {
    private A current;
    private FuncIterator<A> rest;

    public ConsFuncIterator (A c, FuncIterator<A> r) {
        this.current = c;
        this.rest = r;
    }

    public boolean hasElement () { return true; }

    public A current () { return this.current; }

    public FuncIterator<A> advance () {
        return this.rest;
    }
}
```

Let's double-check that using mutable lists doesn't affect what we can type, that is, that `Test4` still compiles with mutable lists instead of immutable lists:

```
public class Test5 {

    public static void main (String[] argv) {
        ListMut<Integer> a = ListMut.empty();
        a = ListMut.cons(1,ListMut.cons(2,ListMut.cons(3,a)));
        show(a);
        System.out.println("First element * 2  = " + (2*a.first().intValue()));
    }

    public static void show (ListMut<? extends Object> a) {
```

```
        for (Object i : a) {
            System.out.println(i.toString());
        }
    }
}
```

No problem, this compiles and executes beautifully. So what about the equivalent of `Test2`, then?

```
public class Test6 {

    public static void main (String[] argv) {
        ListMut<Integer> a = ListMut.empty();
        a = ListMut.cons(1,ListMut.cons(2,ListMut.cons(3,a)));
        show(a);
        update(a);
        System.out.println("First element * 2  = " + (2*a.first().intValue()));
    }

    public static void show (ListMut<? extends Object> a) {
        for (Object i : a) {
            System.out.println(i.toString());
        }
    }

    public static void update (ListMut<? extends Object> a) {
        a.mutate(new Object());
    }
}
```

Bang! Fails to type check. The reason for the type-checking failure here is a bit subtle. Note that the type of `a`, as far as Java is concerned is `List<T>` for some unknown `T`. (That's what the `?` says.) Now, method `mutate()` in `ListMut<A>` has signature:

```
public void mutate (A element);
```

So in order for the invocation of `mutate()` to type check, it must be the case that `new Object()` be an expression returning a value of type `T`, where `T` is an unknown type. Java cannot establish that `new Object()` has type `T`, because, and that's the key, `T` is unknown!

Leaving aside the details, the main consequence of this is that the `<? extends T>` notation permits the use of subtyping in some instances, and disallows it in the cases where it could cause an exception.

So, generics in Java are typed using approach (2), and it works pretty well.

## 20.3 Subtyping by Distinguishing Mutable and Immutable Classes

There is a third approach to managing subtyping for aggregates, one that Java does not implement. But the basic idea here is to sometimes allow subtyping, and sometimes not. If you look at all the examples above, all those that involve immutable classes have no problem with subtyping. The only examples where sometimes bad can occur is when we mutate an aggregate structure (`Test2`, or `Test6`).

So one approach, which some languages implement, is to allow subtyping when classes are immutable (so that immutable classes are treated like arrays in Java), and disallow subtyping when classes are mutable (so that mutable classes are treated like generics in Java).

Of course, in order to do so, it must be possible for the programming language to distinguish mutable from immutable classes. That's what some languages do (ML, or Haskell, for instance): everything is immutable by default, and you have to explicitly define a piece of data to be mutable. In such languages, approach (3), a mix of allowing and disallowing subtyping for aggregate classes is possible.