20 Subclassing and Mutation

Suppose we have a class `A` that subclasses a class `B` (which I will write `A` $\leq$ `B`). Should we consider a collection of `A`s to be a subclass of a collection of `B`s — 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 an operation that can happily work with a collection of `B`s, then giving it a collection of `A`s should work just as well, because every `A` is a `B` due to subclassing. Unfortunately, in the presence of mutation, things are not that simple, and that simple intuition sometimes fail. Let's look at (the skeleton of) two examples, one that matches our intuition and one that doesn't.

Let's work with the points and colored points introduced in previous lectures. We know that `CPoint` $\leq$ `Point`. Assume we have an operation `showCollection()` that takes a collection of `Point`s and can display them on the screen. (I don't what exactly – nor do I especially care what the collection is.) Here's the operation:

```
public static void showCollection (Collection-of-Points coll) {
  // show the points in coll, somehow:
  //   take each point in the collection,
  //   extract its xPos() and yPos(),
  //   and plot it or something
}
```

If we create a collection of colored points, it should be possible to pass it to `showCollection()`:

```
Collection-of-CPoints c = ...   // some code to create a collection of
                                // colored points
showCollection(c);
// here, for fun, let's just print the color of the points in c
// for each colored point in c, extract its color() and print it
```

That should work perfectly fine, right? Right. If you work through the execution by hand, you see that you create a collection of colored point `c` — the dynamic type of `c` is a collection of `CPoint`s — you pass it to `showCollection()`, which looks at every point in the collection, extract its `xPos()` and `yPos()` — which we know is fine because the dynamic type of each element in the collection is a `CPoint`, which does have a `xPos()` and `yPos()` method — then the function returns, and the collection `c` is looked at again, this time we exact the

`color()` out of every element — which is fine because the run-time type of each element of the collection is `CPoint` and therefore has a `color()` method — and everything executes without a hitch.

Consider the following slight variant, though, when the collection is mutable. The function `showCollection()` now does something additional: it mutates the collection.

```
public static void showCollection (Collection-of-Points coll) {
  // show the points in coll, somehow:
  //   take each point in the collection,
  //   extract its xPos() and yPos(),
  //   and plot it or something

  // then do something sneaky
  first-element-of(coll) = Point.create(0,0);
}
```

This is pseudo-code for updating the first element of the collection to be a newly-created `Point`. Note that this should work because the type of `coll` is a collection of `Points` — replace the first `Point` by another `Point` should be fine. But put that new function in conjunction with the old code:

```
Collection-of-CPoints c = ...   // some code to create a collection of
                                // colored points
showCollection(c);
// here, for fun, let's just print the color of the points in c
// for each colored point in c, extract its color() and print it
```

and everything fails. Let's run this one through, keeping track of the run-time type of everything we have. Again, we create a collection of colored point `c` — the dynamic type of `c` is a collection of `CPoints` — you pass it to `showCollection()`, which looks at every point in the collection, extract its `xPos()` and `yPos()` — which we know is fine because the dynamic type of each element in the collection is a `CPoint`, which does have a `xPos()` and `yPos()` method — then *it changes the first element of the collection to be a* ***`Point`*** — so now the first element of the collection has run-time type `Point`, while everything else in the collection has run-time type `CPoint` — then the function returns, and the collection `c` is looked at again, this time we exact the `color()` out of every element — which fails on the first element because it has run-time type `Point`, so does not provide a `color()` method. Method not found. Ouch.

The above examples look very similar, and are undistinguishable if you just look at the type of the data and the functions. The crux: should we allow the call to `showCollection()` when the value supplies is a collection of `CPoints` while the function expects a collection of

Points? One example shows that sometimes treating a collection of CPoints as a collection of Points is fine, other times, it's not.

So: you're designing a language. You have that A $\leq$ B. Do you consider a collection of As to be a subclass of a collection of Bs— thereby accepting the code above — or do not consider a collection of As to be a subclass of a collection of Bs? You have to make a choice. There are really only three choices possible:

(1) Allow a collection of As to be a subclass of a collection of Bs, but do runtime checks to prevent potential problems like the one above.

(2) Disallow a collection of As to be a subclass of a collection of Bs.

(3) Sometimes allow a collection of As to be a subclass of a collection of Bs, and sometimes disallow it, depending on properties of the collection.

Java, it turns out, uses both (1) and (2), depending on the kind of collection you have. Roughly: arrays are handled using (1), while polymorphic collections (using generics) are handled using (2). Let's investigate.

Recall what I pointed out earlier in the course, that the main property enforced by the Java type system is safety, defined as follows: *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 Subclassing for Arrays in Java

Arrays are treated specially in Java. The type system uses the following rule to determine subclassing for array types: whenever S is a subclass of T, then S[] is a subclass of T[].

Let's look at the two examples above, specialized to use arrays as the collections.

First, some code showing that it ought to make sense to pass an array of CPoints to a function expecting an array of Points:

```
public class Test1 {

   public static void showCollection (Point[] coll) {
      System.out.println("In showCollection()");
      for (Point p : coll)
          System.out.println(" element = " + p.xPos() + "  " + p.yPos());
   }

   public static void main (String[] argv) {
```

```
        CPoint[] coll = new CPoint[2];

        coll[0] = CPoint.create(10,10,"red");
        coll[1] = CPoint.create(20,20,"blue");

        System.out.println("In main()");
        for (CPoint cp : coll)
            System.out.println(" element color = " + cp.color());

        showCollection(coll);

        System.out.println("In main()");
        for (CPoint cp : coll)
            System.out.println(" element color = " + cp.color());
    }
}
```

Of course, this code executes perfectly well: it is okay to pass the array of `CPoints` to `showCollection()`, because each `CPoint` has both a `xPos()` and `yPos()` method, and execution proceeds without encountering an undefined method:

```
In main()
 element color = red
 element color = blue
In showCollection()
 element = 10 10
 element = 20 20
In main()
 element color = red
 element color = blue
```

The problem is that, as we saw above, the following variant also type checks, and for the same reason:

```
public class Test2 {

    public static void showCollection (Point[] coll) {
        System.out.println("In showCollection()");
        for (Point p : coll)
            System.out.println(" element = " + p.xPos() + "  " + p.yPos());

        coll[0] = Point.create(0,0);
```

```
    }

    public static void main (String[] argv) {

        CPoint[] coll = new CPoint[2];

        coll[0] = CPoint.create(10,10,"red");
        coll[1] = CPoint.create(20,20,"blue");

        System.out.println("In main()");
        for (CPoint cp : coll)
            System.out.println(" element color = " + cp.color());

        showCollection(coll);

        System.out.println("In main()");
        for (CPoint cp : coll)
            System.out.println(" element color = " + cp.color());
    }
}
```

`Test2` is very similar to `Test1`, except that function `showCollection()` now modifies the first element of the array, making it hold a new `Point`. First, make sure that you understand why the code above type checks: Because `CPoint` $\leq$ `Point`, the type system lets you pass a `CPoint[]` to a method expecting a `Point[]`. And because the array `coll` in `showCollection()` is declared to be an array of `Point`s (that its compile-time type), the type system is quite happy to let you update the first element in the array into a different `Point`.

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 function `showCollection()` updates the array through its argument `coll`, it ends up modifying the underlying array `coll` in function `main()`. But that means that when we come back from the `showCollection()` function, array `coll` is an array of `CPoint`s where the first element of the array is not a `CPoint` any longer, but rather a `Point`. And when we attempt to invoke method `color()` on that first element, Java would choke because that first element, being a plain `Point`, does not in fact implement the `color()` 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 `coll[0] = Point.create(0,0)`. 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 a `Point` into an array with dynamic type `CPOint[]`:

```
In main()
 element color = red
 element color = blue
In showCollection()
 element = 10 10
 element = 20 20
Exception in thread "main" java.lang.ArrayStoreException: Point
      at Test2.showCollection(Test2.java:9)
      at Test2.main(Test2.java:23)
```

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 subclassing between collections, which Java uses for arrays.

20.2 Subclassing 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? To make the comparison with arrays, let's make `List` mutable by introducing a `mutateFirst()` method that lets you mutate the first element of a (nonempty) list. Here is the implementation:

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

public abstract class List<A> implements Iterable<A> { // we can
                                                       // iterate over
                                                       // instances of
                                                       // this class
  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 ();

  public abstract FuncIterator<A> funcIterator ();
  public Iterator<A> iterator () {
    return IteratorFromFuncIteratorAdapter.create(this.funcIterator());
  }

  public abstract void mutateFirst (A element);
}


class EmptyList<A> extends List<A> {

  public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }
  public A first () {
    throw new Error("EmptyList.first()");
  }
  public List<A> rest () {
    throw new Error("EmptyList.rest()");
  }
  public FuncIterator<A> funcIterator () {
    return new Iterator<A>();
  }
  public void mutateFirst (A element) {
    throw new Error("EmptyList.mutateFirst()");
  }

  private class Iterator<A> implements FuncIterator<A> {
    public Iterator () {}

    public boolean hasElement () {
      return false;
    }
    public A element () {
      throw new java.util.NoSuchElementException
                              ("EmptyList.Iterator.element()");
```

```
    }
    public FuncIterator<A> moveToNext () {
      throw new java.util.NoSuchElementException
                                ("EmptyList.Iterator.moveToNext()");
    }
  }
}


class ConsList<A> extends 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;
  }
  public FuncIterator<A> funcIterator () {
    return new Iterator<A>();
  }
  public void mutateFirst (A element) {
    firstElement = element;
  }

  private class Iterator<A> implements FuncIterator<A> {
    public Iterator () {}

    public boolean hasElement () {
      return true;
    }
    public A element () {
      return firstElement;
```

```
    }
    public FuncIterator<A> moveToNext () {
      return restElements.funcIterator();
    }
  }
}
```

Given this implementation of `List`, here is `Test1` using `List`:

```
public class Test3 {

  public static void showCollection (List<Point> coll) {

    System.out.println("In showCollection()");
    for (Point p : coll)
        System.out.println(" element = " + p.xPos() + "  " + p.yPos());
  }

  public static void main (String[] argv) {

      List<CPoint> coll = List.empty();

      coll = List.cons(CPoint.create(20,20,"blue"),coll);
      coll = List.cons(CPoint.create(10,10,"red"),coll);

      System.out.println("In main()");
      for (CPoint cp : coll)
        System.out.println(" element color = " + cp.color());

      showCollection(coll);

      System.out.println("In main()");
      for (CPoint cp : coll)
        System.out.println(" element color = " + cp.color());
  }
}
```

Trying to compile this program fails miserably: it does not type check — on my machine:

```
> javac Test3.java
Test3.java:22: showCollection(List<Point>) in Test3 cannot be applied
to (List<CPoint>)
```

```
        showCollection(coll);
        ^
1 error
```

This is because it is *not the case* that if `S` is a subclass of `T`, then `List<S>` is a subclass 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 a collection 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 very much like throwing the baby out with the bathwater, because code such as that in `Test3` is actually quite useful, and works fine. (See `Test1`, which is the same but for arrays.) It's only when we update a collection that problems occur. There are ways to restore some amount of subclassing and get `Test3` to compile, but there is no one-size-fits-all solution. The idea is to be explicit about where we want subclassing to occur. Consider the type for `showCollection()` in `Test3`. Suppose we wanted to be explicit about the kind of subclassing we allowed here. Roughly, we would like it to say that `showCollection` accepts any list with some type of element `T` that is a subclass of `Point`. 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 showCollection (List<? extends Point> coll) {

    System.out.println("In showCollection()");
    for (Point p : coll)
        System.out.println(" element = " + p.xPos() + "  " + p.yPos());
  }

  public static void main (String[] argv) {

     List<CPoint> coll = List.empty();

     coll = List.cons(CPoint.create(20,20,"blue"),coll);
     coll = List.cons(CPoint.create(10,10,"red"),coll);

     System.out.println("In main()");
     for (CPoint cp : coll)
```

[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`.

```
        System.out.println(" element color = " + cp.color());

    showCollection(coll);

    System.out.println("In main()");
    for (CPoint cp : coll)
        System.out.println(" element color = " + cp.color());
  }
}
```

And this type checks perfectly okay, and executes perfectly okay:

```
In main()
 element color = red
 element color = blue
In showCollection()
 element = 10 10
 element = 20 20
In main()
 element color = red
 element color = blue
```

The subclassing rule for this kind of generics is as follows: if `S` is a subclass of `T`, then `List<S>` is a subclass of `List<? extends T>`. Wrap your head around this rule, and the above example.

So, we can reinstate some form of subclassing for generics. Have we added too much? Can we write a version of `Test2` in this setting?

```
public class Test5 {

  public static void showCollection (List<? extends Point> coll) {

    System.out.println("In showCollection()");
    for (Point p : coll)
        System.out.println(" element = " + p.xPos() + "  " + p.yPos());

    coll.mutateFirst(Point.create(0,0));
  }

  public static void main (String[] argv) {

     List<CPoint> coll = List.empty();
```

```
        coll = List.cons(CPoint.create(20,20,"blue"),coll);
        coll = List.cons(CPoint.create(10,10,"red"),coll);

        System.out.println("In main()");
        for (CPoint cp : coll)
            System.out.println(" element color = " + cp.color());

        showCollection(coll);

        System.out.println("In main()");
        for (CPoint cp : coll)
            System.out.println(" element color = " + cp.color());
    }
}
```

Bang! Fails to type check. On my machine:

```
> javac Test5.java
Test5.java:10: mutateFirst(capture of ? extends Point) in
List<capture of ? extends Point> cannot be applied to (Point)
     coll.mutateFirst(Point.create(0,0));
          ^
1 error
```

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

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

So in order for the invocation of `mutateFirst()` to type check, it must be the case that `Point.create(0,0)` be an expression returning a value of type `T`, where `T` is an unknown type. Java cannot establish that `Point.create(0,0)` 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 subclassing 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 Subclassing by Distinguishing Mutable and Immutable Classes

There is a third approach to managing subclassing in the presence of mutation, one that Java does not implement. But the basic idea here is to sometimes allow subclassing, and sometimes not. If you look at all the examples above, all those that involve immutable classes have no problem with subclassing. The only examples where sometimes bad can occur is when we mutate a collection (`Test2`, or `Test5`).

So one approach, which some languages implement, is to allow subclassing when classes are immutable (so that immutable classes are treated like arrays in Java), and disallow subclassing 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 distinction is pretty much impossible to make in Java, at least for a comiler — we saw, for instance, that mutability was contagious, so that something may look immutable while still depending on something mutable, making the resulting class mutable as well. Languages such as ML or Haskell do make such a distinction possible. In those languages, 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 subclassing for collections is possible.