Java generics are invariant

The Java language decrees so. Hence the following code now fails to type-check.

```
ArrayList<String> s = new ArrayList(10);[3]
ArrayList<Object> o;
o = s;                       // fails to type-check
o.set(5,"OK so far");        // type-checks OK
o.set(4, new Integer(42));   // type-checks OK
```

So generics are safer than arrays. But covariance and contravariance can be useful.

- What if I have an immutable array, so that writes to it are banned by the type checker, then surely it's OK for it to be covariant?

[2]Legal note: it doesn't matter here, but to exactly match the previous array-using code I should populate the ArrayList with 10 NULLs. Real code would of course populate both arrays and ArrayLists with non-NULL values.

Java variance specifications

In Java we can have safe co-variant generics using syntax like:

```
ArrayList<String> s = new ArrayList(10);
ArrayList<? extends Object> o;
o = s;                 // now type checks again
```

But what about reading and writing to `o`?

```
s.set(2,"Hello");
System.out.println((String)o.get(2)+"World"); //fine
o.set(4,"seems OK");      //faulted at compile-time
```

The trade is that the covariant `ArrayList o` cannot have its elements written to, in exchange for covariance.

- Java has *use-site variance* specifications: we can declare variance at every use of a generic.
- By contrast Scala has *declaration-site variance* which many find simpler (see later).