Unit 21: Variance

After this unit, students should:

- understand the definition of the variance of types: covariant, contravariant, and invariant.
- be aware that the Java array is covariant and how it could lead to run-time errors that cannot be caught during compile time.

Both the methods `findLargest` and `contains` takes in an array of reference types as parameters:

```
1  // version 0.4
2  GetAreable findLargest(GetAreable[] array) {
3    double maxArea = 0;
4    GetAreable maxObj = null;
5    for (GetAreable curr : array) {
6      double area = curr.getArea();
7      if (area > maxArea) {
8        maxArea = area;
9        maxObj = curr;
10     }
11   }
12   return maxObj;
13 }
14
15 // version 0.1 (with polymorphism)
16 boolean contains(Object[] array, Object obj) {
17   for (Object curr : array) {
18     if (curr.equals(obj)) {
19       return true;
20     }
21   }
22   return false;
23 }
```

What are some possible arrays that we can pass into these methods? Let's try this:

```
1  Object[] objArray = new Object[] { new Integer(1), new Integer(2) };
2  Integer[] intArray = new Integer[] { new Integer(1), new Integer(2) };
3
4  contains(objArray, new Integer(1)); // ok
5  contains(intArray, new Integer(1)); // ok
```

Line 4 is not surprising since the type for `objArray` matches that of parameter `array`. Line 5, however, shows that it is possible to assign an instance with run-time type

`Integer[]` to a variable with compile-time type `Object[]`.

Variance of Types

So far, we have established the subtype relationship between classes and interfaces based on inheritance and implementation. The subtype relationship between *complex types* such as arrays, however, is not so trivial. Let's look at some definitions.

The *variance of types* refers to how the subtype relationship between complex types relates to the subtype relationship between components.

Let $C(S)$ corresponds to some complex type based on type S. An array of type S is an example of a complex type.

We say a complex type is:

- *covariant* if $S <: T$ implies $C(S) <: C(T)$
- *contravariant* if $S <: T$ implies $C(T) <: C(S)$
- *invariant* if it is neither covariant nor contravariant.

Java Array is Covariant

Arrays of reference types are covariant in Java[1]. This means that, if $S <: T$, then $S[] <: T[]$.

For example, because `Integer` <: `Object`, we have `Integer[]` <: `Object[]` and we can do the following:

```
Integer[] intArray;
Object[] objArray;
objArray = intArray; // ok
```

By making array covariant, however, Java opens up the possibility of run-time errors, even without typecasting!

Consider the following code:

```
Integer[] intArray = new Integer[2] {
  new Integer(10), new Integer(20)
};
Object[] objArray;
objArray = intArray;
objArray[0] = "Hello!"; // <- compiles!
```

On Line 5 above, we set `objArray` (with a compile-time type of `Object[]`) to refer to an object with a run-time type of `Integer[]`. This is allowed since the array is covariant.

On Line 6, we try to put a `String` object into the `Object` array. Since `String` <: `Object`, the compiler allows this. The compiler does not realize that at run-time, the `Object` array will refer to an array of `Integer`.

So we now have a perfectly compilable code, that will crash on us when it executes Line 6 -- only then would Java realize that we are trying to stuff a string into an array of integers!

This is an example of a type system rule that is unsafe. Since the array type is an essential part of the Java language, this rule cannot be changed without ruining existing code. We will see later how Java avoids this pitfall for other complex types (such as a list).

---

1. Arrays of primitive types are invariant. ↵