Subtyping and Generics

Polymorphism

- Two principal forms of polymorphism are subtyping and generics
- Their interactions consist in two main areas: bounds and variance

Type Bounds

- Consider the method ***assertAllPos*** which:
    - Takes an IntSet
    - Returns the IntSet itself if all its elements are positive
    - Throws an exception otherwise

  What is the best type you can give to assertAllPos?
- One might want to express that assertAllPos takes Empty sets to Empty sets and NonEmpty sets to NonEmpty sets
- A way to express this is: ***def assertAllPos[S <: IntSet](r: S): S = …***
- Here, **"<: IntSet"** is an upper bound of the type parameter S(this means that S can be instantiated only to types that conform to IntSet)
- Generally, the notation:
    - ***S <: T*** means: S is a subtype of T
    - ***S >: T*** means: S is a supertype of T
- You can also use a lower bound for a type variable
- Example***: [S >: NonEmpty]*** introduces a type parameter s that can range only over supertypes of NonEmpty. So S could be one of NonEmpty, IntSet, AnyRef or Any
- It is also possible to mix a lower bound with an upper bound, but the lower bound comes first
- For instance: ***[S >: NonEmpty <: IntSet]*** would restrict S any type on the interval between NonEmpty and IntSet

Covariance

- There's another interaction between subtyping and type parameters we need to consider: Given ***NonEmpty <: IntSet*** is ***List[NonEmpty] <: List[IntSet]*** ?
- Intuitively, this makes sense: a list of non-empty sets is a special case of a list of arbitrary sets
- We call types for which this relationship holds covariant because their subtyping relationship varies with the type parameter

Array Typing Problem

- Arrays in Java are covariant, but covariant typing causes problems
- To see why, consider the Java code below:

```
NonEmpty[] a = new NonEmpty[]{
    new NonEmpty(1, new Empty(), new Empty())
};
IntSet[] b = a;
b[0] = new Empty();
NonEmpty s = a[0];
```

- By the third line, we don't have a NonEmpty element in the array anymore; the element here is now empty
- That means that in the fourth line we assign an Empty element to a variable of type NonEmpty, which is a violation of type soundness
- So in the third line, you would get a runtime error similar to a class cast exception

The Liskov Substitution Principle

- The following principle, stated by Barbara Liskov, tells us when a type can be subtype of another:

  If A <: B, then everything one can do with a value of type B, one should also be able to do with a value of type A.

Exercise: *The problematic array example would be written as follows in Scala:*

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a
b(0) = Empty()
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe? – ***A type error in line two****, because Arrays in Scala are not covariant(otherwise, the Liskov principle would be violated)*