



Subtyping and Generics

Principles of Functional Programming

Polymorphism

Two principal forms of polymorphism:

- ▶ subtyping
- ▶ generics

In this session we will look at their interactions.

Two main areas:

- ▶ bounds
- ▶ 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 would be the best type you can give to `assertAllPos`? Maybe:

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 would be the best type you can give to `assertAllPos`? Maybe:

```
def assertAllPos(s: IntSet): IntSet
```

In most situations this is fine, but can one be more precise?

Type Bounds

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:

It 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*, and
- ▶ $S >: T$ means: *S is a supertype of T*, or *T is a subtype of S*.

Lower Bounds

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`.

We will see in the next session examples where lower bounds are useful.

Mixed Bounds

Finally, it is also possible to mix a lower bound with an upper bound.

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]    ?
```


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.

Does covariance make sense for all types, not just for List?

Arrays

For perspective, let's look at arrays in Java (and C#).

Reminder:

- ▶ An array of T elements is written `T[]` in Java.
- ▶ In Scala we use parameterized type syntax `Array[T]` to refer to the same type.

Arrays in Java are covariant, so one would have:

```
NonEmpty[] <: IntSet[]
```

Array Typing Problem

But covariant array 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];
```

It looks like we assigned in the last line an Empty set to a variable of type NonEmpty!

What went wrong?

The Liskov Substitution Principle

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

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

[The actual definition Liskov used is a bit more formal. It says:

Let $q(x)$ be a property provable about objects x of type B. Then $q(y)$ should be provable for objects y of type A where $A \leq B$.

]

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 1
- ☐ A type error in line 2
- ☐ A type error in line 3
- ☐ A type error in line 4
- ☐ A program that compiles and throws an exception at run-time
- ☐ A program that compiles and runs without exception

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 1
- ☐ A type error in line 2
- ☐ A type error in line 3
- ☐ A type error in line 4
- ☐ A program that compiles and throws an exception at run-time
- ☐ A program that compiles and runs without exception