Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

October 8, 2019

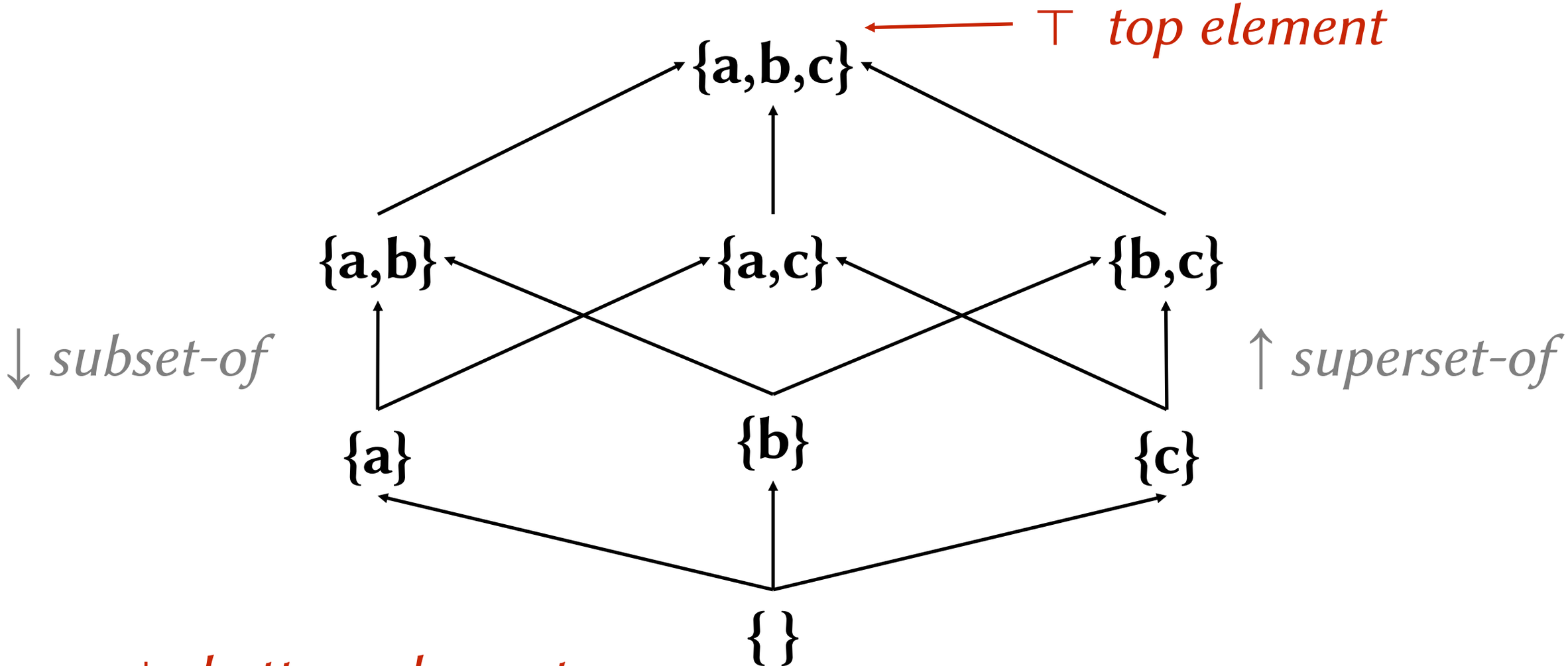
Scala Type Hierarchy

Type Hierarchies

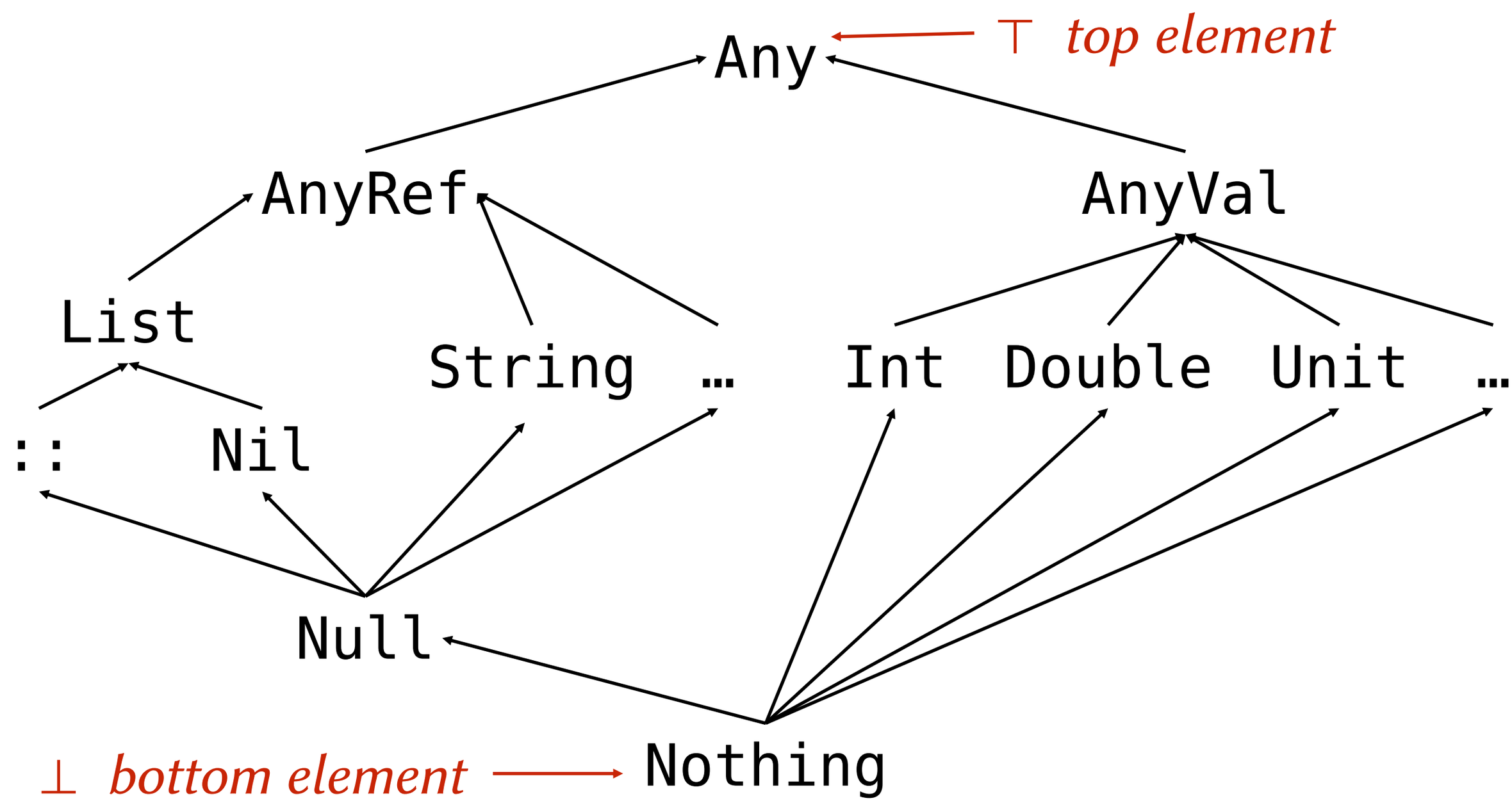
Inheritance (subclass / superclass relationships) form a *complete lattice* in the Scala type system:

- Each pair of classes has exactly one:
  - *Least upper-bound*
  - *Greatest lower-bound*
- The same applies to all value types

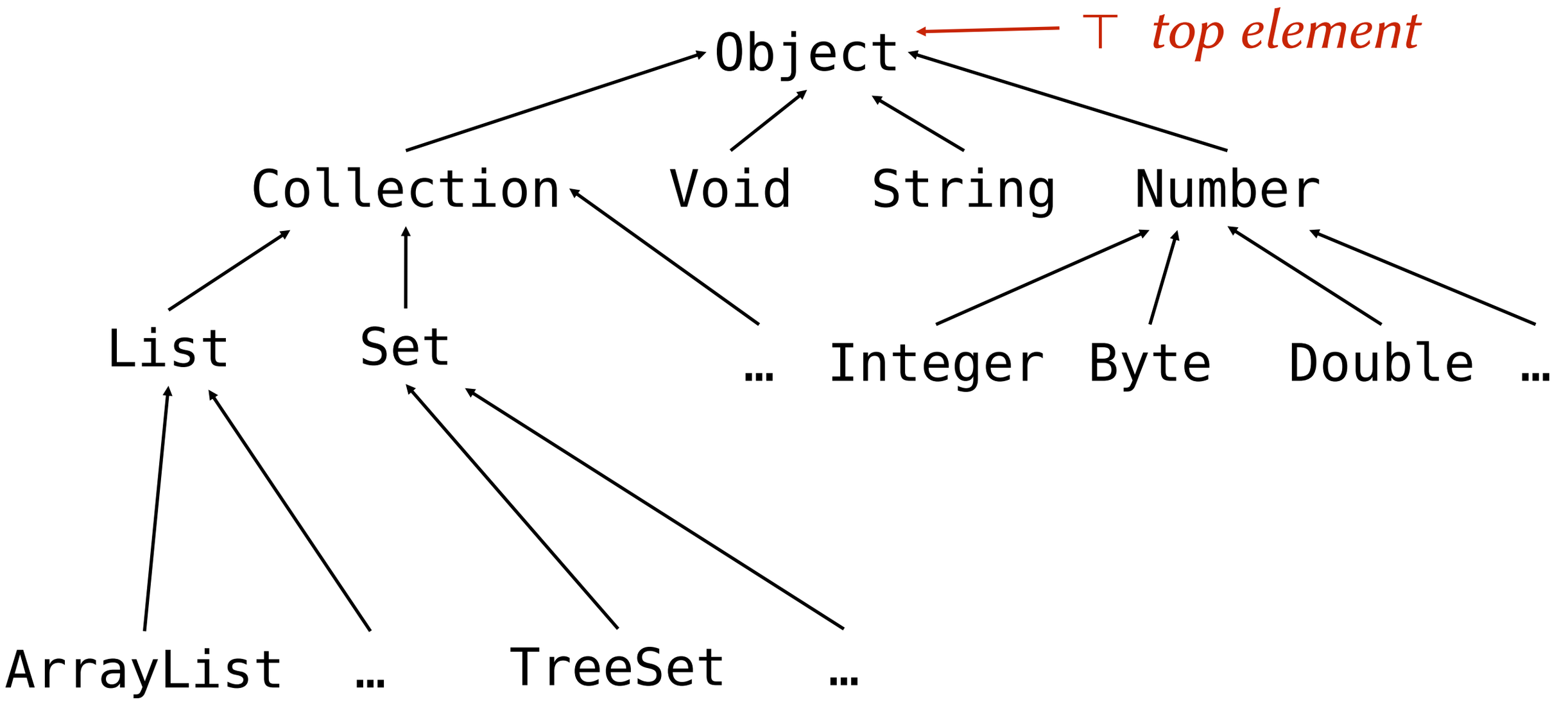
Hasse Diagrams



Scala Type Lattice



Java Type Semi-Lattice



no bottom element

Variance of Parametric Types

Covariance

In general, we say that a parametric type C is covariant with respect to its type parameter S if:

$$S <: T$$

implies

```
C[S] <: C[T]
```

Contravariance

In general, we say that a parametric type C is contravariant with respect to its type parameter S if:

S <: T

implies

C[T] <: C[S]

Invariance

In general, we say that a parametric type C is invariant with respect to its type parameter S if:

```
S <: T
```

implies neither

```
C[S] <: C[T]
```

nor

```
C[T] <: C[S]
```

Syntax for Variance

Syntactically:

- Covariant type parameter declarations are annotated with a *plus* sign.

- Contravariant type parameter declarations are annotated with a *minus* sign.

- Invariant type parameter declarations are not annotated with an extra symbol.

```
case class X[+A,-B,C]
```

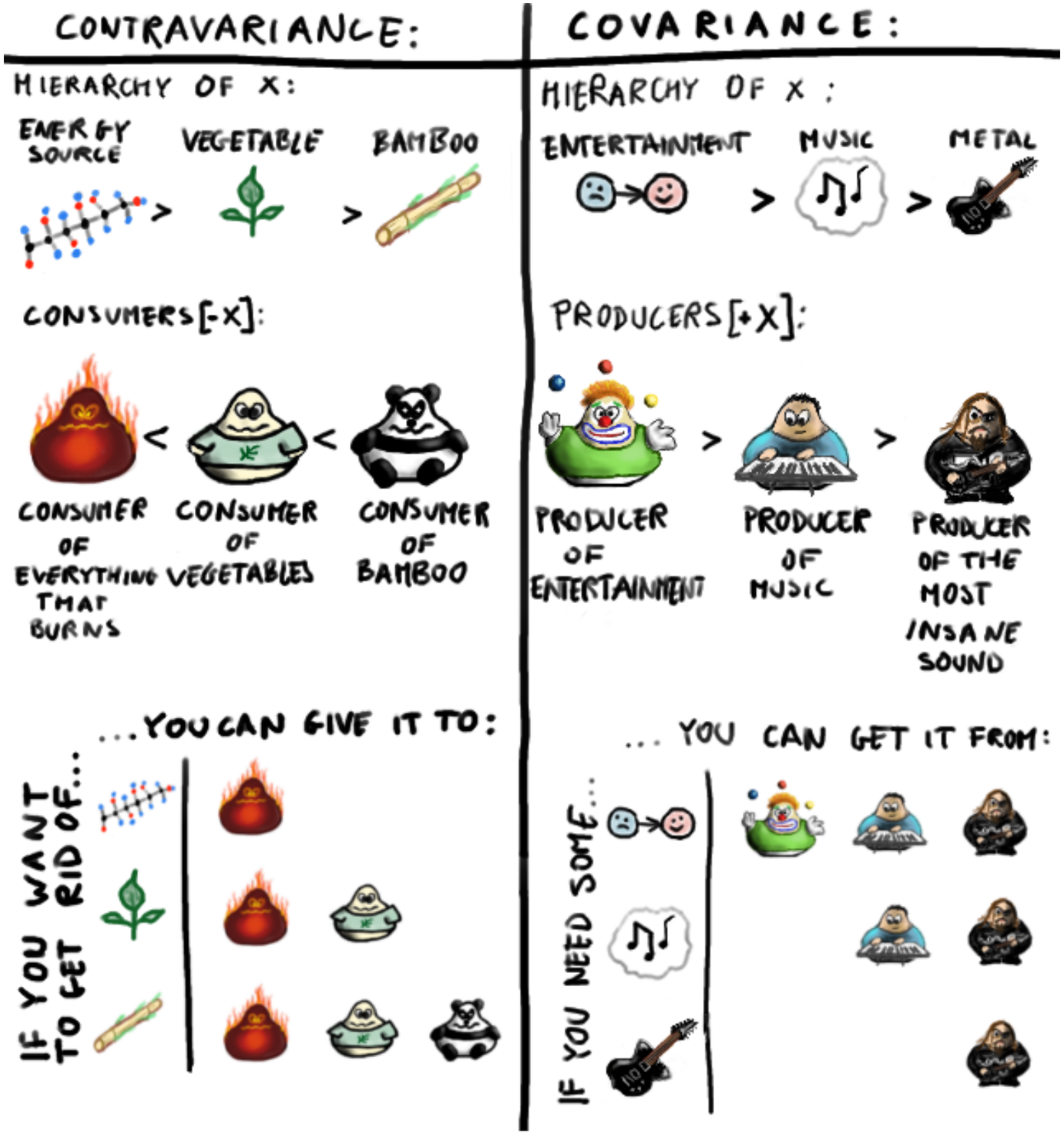


Image by Andrey Tyukin:
https://stackoverflow.com/a/19739576/1427124

See also: https://www.cs.rice.edu/~javaplt/nv4/scala-variance/

Generic Functions

Parametric Functions

- Just as we can add type parameters to a class definition, we can also add them to a function definition

- The type parameters are in scope in the header and body of the function

Map Revisited

```
abstract class List[+T] {
  …
  def map[U](f: T => U): List[U]
}
```

Is this occurrence of T acceptable?
Does this definition of `map` still work as expected given covariance-enabled subtyping?

We Consider Specific Instantiations

```
abstract class List[Any] {
  …
  def map[U](f: Any => U): List[U]
}
abstract class List[String] {
  …
  def map[U](f: String => U): List[U]
}
```

List[String] is an acceptable subtype of List[Any]
only if the function argument `f: Any ⇒ U`
works when passed to the `map` method of List[String].

We Consider Specific Instantiations

Scala-style hash-code

```
val xs: List[Any] =
  List[String]("a", "b", "c")

xs map[Int] { x: Any => x.## }
// ↦ List[Int](97, 98, 99)
```

List[String] is an acceptable subtype of List[Any]
only if the function argument `f: Any ⇒ U`
works when passed to the `map` *method of List[String].*

Generalizing Our Rules

- In our example, type parameter T occurs as the parameter of an arrow type:
  - `(String => U) >: (Any => U)`, provided:
    - `String <: Any`
    - `U <: U`
- Consistent with `List[String] <: List[Any]`

An Example of How We Might Use Contravariant Type Parameters

```
abstract class Function1[-S,+T] {
  def apply(x:S): T
}
```

Map Revisited

```
case object Empty extends List[Nothing] {
  …
  def map[U](f: Nothing => U) = Empty
}
```

Map Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  …
  def map[U](f: T => U) =
    Cons(f(head), tail.map(f))
}
```

Checking Variance

To Check Variance, We Annotate Each Type Location With A *Polarity*

- Recursively descend a class definition:
  - At top level, all locations are positive
  - Polarity is flipped at method parameter positions
  - Polarity is flipped at method type parameter positions
  - Polarity is flipped at arrow type parameter positions
  - Polarity is flipped at positions of contravariant type parameters

Annotating Polarity

```
abstract class List[+T] {
  def ++[S- >: T+](ys: List[S-]): List[S+]
  def map[U-](f: T+ => U-): List[U+]
}
```