Part I. Basic intuitions about subtyping
trait Fruit
trait Plum extends Fruit
trait Apricot extends Fruit
trait Pluot extends Plum with Apricot
Fruit is-a-subtype-of Plum
Plum is-a-subtype-of Fruit
Plum is-a-subtype-of Fruit

Plum => Fruit
Plum => Fruit

def plumToFruit(p: Plum): Fruit
Plum is-a-subtype-of Fruit

def plumToFruit(p: Plum): Fruit

def eat(fruit: Fruit): Unit
Plum => Fruit

def plumToFruit(p: Plum): Fruit

def eat(fruit: Fruit): Unit

eat(plumToFruit(new Plum {}))
Plum => Fruit

def plumToFruit(p: Plum): Fruit

def eat(fruit: Fruit): Unit

eat(plumToFruit(new Plum {}))
eat(new Plum {})
The *is-a-subtype-of* arrow between types:

- represents an implicit conversion from subtype to supertype
- implies memory need not change to perform the conversion
- is reflexive and transitive
The *is-a-subtype-of* relation (<:) is...

```plaintext
Fruit
   /\   \\/
Plum Apricot
   \  /   \\/
    \ /     \\/
     Pluot
```

reflexive:

```
Apricot <: Apricot
```
The *is-a-subtype-of* relation (<:) is...

- Reflexive:
  - Apricot <: Apricot

- Transitive:
  - Pluot <: Plum and Plum <: Fruit, therefore Pluot <: Fruit
Can think of types as *sets of values*, which we call *inhabitants*

- **Unit**
  - ()

- **Boolean**
  - true, false

- **TrafficLight**
  - sealed trait TrafficLight
    - object Red extends TrafficLight
    - object Yellow extends TrafficLight
    - object Green extends TrafficLight

- **Int**
  - Int.MinValue, ... -1, 0, 1, 2, ... 
  - Int.MaxValue
**is-a-subtype-of**

Plum <: Fruit

**is-a-subset-of**

Plum ⊂ Fruit
Understanding Scala’s Type System

Plum

Plum is-a-subtype-of Fruit

Plum => Fruit

def plumToFruit(p: Plum): Fruit

def eat(fruit: Fruit): Unit

eat(plumToFruit(new Plum {}))

eat(new Plum {}})
Any: the set of all values

for any type T:

T <: Any  T ⊂ Any

Nothing <: T  ∅ ⊂ T

Nothing: the set of no values, a.k.a, the empty set (∅)
If you map types to inhabitant counts, `<:` maps to is-less-than-or-equal-to.

- Pluot `<:` Plum
- 3 <= 8
Scala’s type hierarchy is:

- a pre-order
- a category
- a lattice (in Scala 3)
A type “lattice” requires that for any two types there exists a unique least upper bound (or “join”) and a unique greatest lower bound (or “meet”).

```scala
val f = // “LUB” is Fruit
if (cond)
  new Plum {}
else
  new Apricot {}
```
Co- and Contravariance adds to the pre-order

trait Basket[+T]
trait Eater[-T]
trait Show[T]

Fruit
Basket[Fruit]
Eater[Plum]
Show[Plum]

Plum
Basket[Plum]
Eater[Fruit]
Show[Fruit]
Scala 2 Language Specification:
least upper bound and greatest lower bound
do not always exist

trait Basket[+T]
class FruitBasket extends Basket[Fruit]
class IntBasket extends Basket[Int]

val f = if (cond) new FruitBasket else new IntBasket

// LUB is limit of Basket[Any], Basket[Basket[Any]]
// Basket[Basket[Basket[Basket[Basket[...]]]], ...
Understanding Scala’s Type System

Union type:
- Plum | Apricot
- Plum
- Apricot
- Plum & Apricot
- Pluot
- Plum ∩ Apricot

Intersection type:
- Fruit
- Plum
- Apricot
- Pluot
- Plum U Apricot
- Plum ∪ Apricot

---

artima
Understanding Scala’s Type System

class Any

class AnyRef extends Any

trait Fruit extends AnyRef

trait Plum extends Fruit

the “exception” to the rule, because Nothing is not an “extends” subtype
Type parameters give abstraction on the inside...

class Basket[+A](a: A) {
  def map[B](f: A => B): Basket[B] = ???
}

Type parameters give abstraction on the inside...

class Basket[+A](a: A) {
  def map[B](f: A => B): Basket[B] = new Basket(f(a))
}
Type parameters give abstraction on the inside...

class Basket[+A](a: A) {
  def map[B](f: A => B): Basket[B] = new Basket(f(a))
}

...and information on the outside

val b: Basket[Pluot] = new Basket(new Pluot { })
Understanding Scala’s Type System

Part II (unconference)
- Abstract type (members)
- Refinement types
- Structural types
- Type projections
- Singleton types
- Path dependent types
- Higher-kindred types
We are:

- hiring ([careers@artima.com](mailto:careers@artima.com))
- available ([info@artima.com](mailto:info@artima.com))
- listening (brown bags)