18.04.2020
SCALA LOVE CONFERENCE

The trouble with subtyping

An introduction to type bounds and variance

Daniel Westheide

Twitter: @kaffeecoder

INNOQ

About me

- senior consultant at INNOQ
- co-organizer of ScalaBridge Berlin
- I like writing about Scala

- **Scala from Scratch: Exploration**:
  - Ebook: https://leanpub.com/scala-from-scratch-exploration/
  - Hardcover: https://www.blurb.com/b/9959223-scala-from-scratch-exploration

- **Scala from Scratch: Understanding:**
  - Ebook with discount for Scala Love attendees: http://leanpub.com/scala-from-scratch-understanding/c/scalalove2020

Scala is a hybrid language

Subclassing

```
class A {
  def magic(x: Int): Int = x * x
}

class B extends A
```

- class-oriented
- subclassing is
  - code sharing
  - nominal subtyping: B is-an A

In this talk...

- learn about type bounds and variance in Scala
- demystifying covariant and contravariant positions
- strategies for avoiding the complexity of variance
- any changes in Scala 3?

Upper type bounds

Modelling caffeinated beverages

```
abstract class CaffeinatedBeverage {
  def caffeineContent: Int
}

final case class FilterCoffee(override val caffeineContent: Int, region: String)
    extends CaffeinatedBeverage

final case class BlackTea(override val caffeineContent: Int)
    extends CaffeinatedBeverage

final case class CuteMate(override val caffeineContent: Int)
    extends CaffeinatedBeverage
```

Choosing a beverage

```
object CaffeinatedBeverage {
  def choose(x: CaffeinatedBeverage, y: CaffeinatedBeverage): CaffeinatedBeverage =
    if (x.caffeineContent >= y.caffeineContent) x
    else y
}
```

- choose the beverage with the highest caffeine content
- we lose precision in the return type
- we can mix different subtypes of `CaffeinatedBeverage`

Parametric polymorphism?

```
object CaffeinatedBeverage {
  def choose[A](x: A, y: A): A =
    if (x.caffeineContent >= y.caffeineContent) x
    else y
}
```

- `choose` should abstract over the type of beverage
- doesn't compile
- `choose` implementation makes certain assumptions about A

Upper type bounds

```
object CaffeinatedBeverage {
  def choose[A <: CaffeinatedBeverage](x: A, y: A): A =
    if (x.caffeineContent >= y.caffeineContent) x
    else y
}
```

- Adds a constraint to the type parameter A
- The assumptions needed to implement choose are satisfied

Upper type bounds in action (1)

```
scala> val guji = FilterCoffee(69, "Ethiopia")
guji: FilterCoffee = FilterCoffee(69,Ethiopia)

scala> val blueBatak = FilterCoffee(75, "Indonesia")
blueBatak: FilterCoffee = FilterCoffee(75,Indonesia)

scala> val chosen = CaffeinatedBeverage.choose(guji, blueBatak)
chosen: FilterCoffee = FilterCoffee(75,Indonesia)
```

Upper type bounds in action (2)

```
scala> val guji = FilterCoffee(69, "Ethiopia")
guji: FilterCoffee = FilterCoffee(69,Ethiopia)

scala> val mate = CuteMate(95)
mate: CuteMate = CuteMate(95)

scala> val chosen: FilterCoffee = CaffeinatedBeverage.choose(guji, mate)
                                                             ^

       error: type mismatch;
        found   : CuteMate
        required: FilterCoffee

scala> val chosen = CaffeinatedBeverage.choose(guji, mate)
chosen: Product with CaffeinatedBeverage with java.io.Serializable = CuteMate(95)
```

Covariance

Modelling caffeine sources

```
abstract class CaffeineSource[A <: CaffeinatedBeverage] {
  def pull(): A
}

class CuteMateSource extends CaffeineSource[CuteMate] {
  override def pull(): CuteMate = CuteMate(85)
}

class FilterCoffeeSource extends CaffeineSource[FilterCoffee] {
  override def pull(): FilterCoffee = FilterCoffee(69, "Ethiopia")
}
```

Using caffeine sources

- example: *Agile Fragile*, a consulting company
- not picky
- they can turn caffeine from any source into code

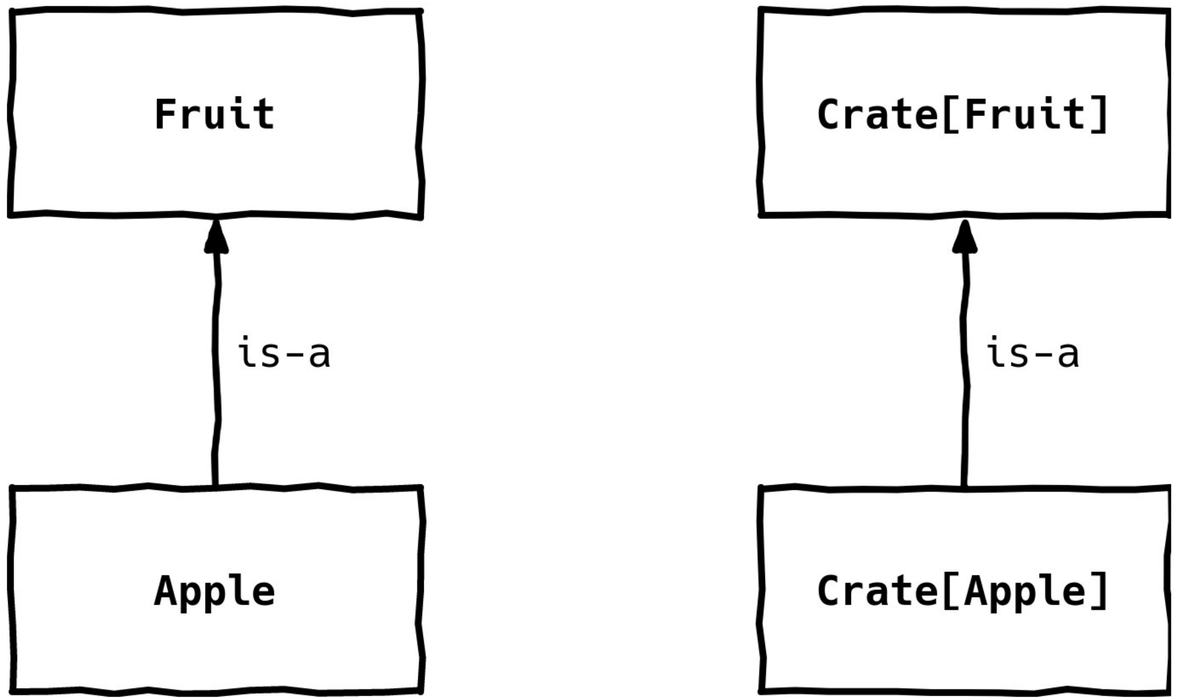
```
object AgileFragile {
  val caffeineSource: CaffeineSource[CaffeinatedBeverage] = ???
}
```

Oh no!

```
scala> val source: CaffeineSource[CaffeinatedBeverage] = new FilterCoffeeSource
                                                          ^
       error: type mismatch;
        found   : FilterCoffeeSource
        required: CaffeineSource[CaffeinatedBeverage]

Note: FilterCoffee <: CaffeinatedBeverage (and FilterCoffeeSource <:
CaffeineSource[FilterCoffee]), but class CaffeineSource is invariant in type A.
       You may wish to define A as +A instead.
```

A crate that is covariant in type A

`Crate[+A]`



A covariant caffeine source

```
abstract class CaffeineSource[+A <: CaffeinatedBeverage] {
  def pull(): A
}

scala> val source: CaffeineSource[CaffeinatedBeverage] = new FilterCoffeeSource
source: CaffeineSource[CaffeinatedBeverage] = FilterCoffeeSource@336a1b7d
```

Covariance in Scala collections

- **immutable** collection types are usually **covariant**
- examples:
  - `Seq[+A]`
  - `List[+A]`
  - `Option[+A]`

Contravariance

How others see programmers...

```
final case class Deliverable(description: String)

class Programmer[A <: CaffeinatedBeverage] {
  def transform(caffeine: A, feature: String): Deliverable = Deliverable(feature)
}
```

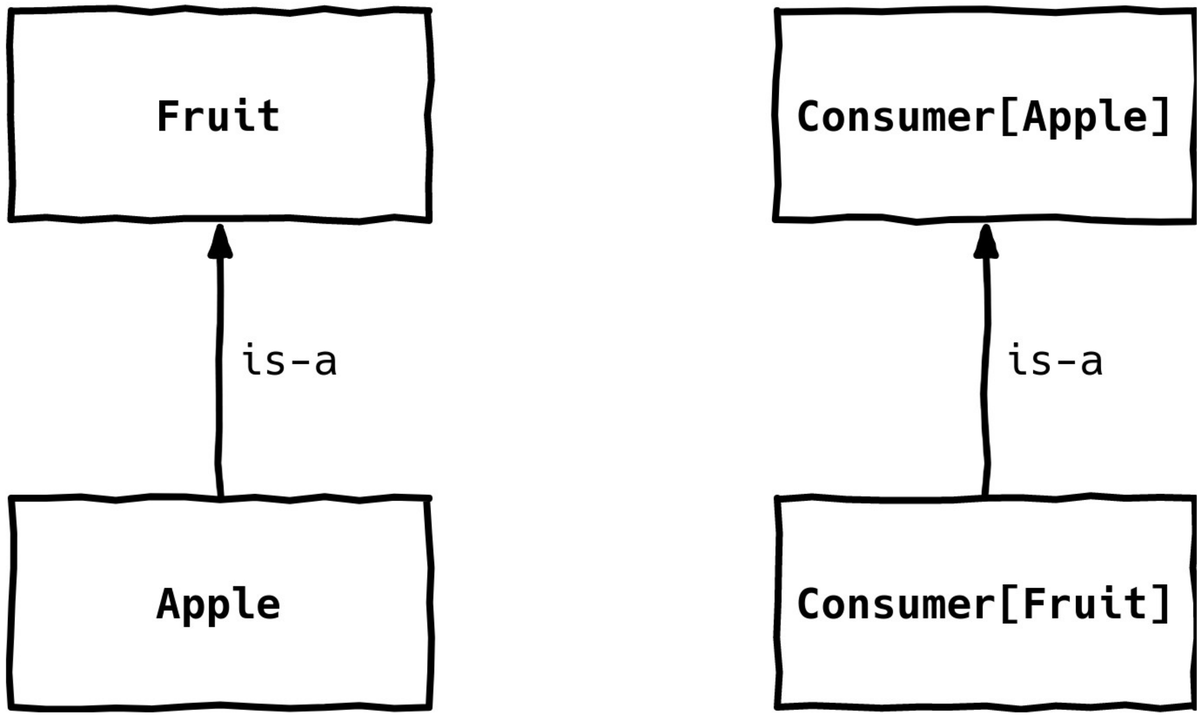
How they deliver at Startupr

```
object Startupr {
  def deliver(
      feature: String,
      programmer: Programmer[CuteMate],
      caffeineSource: CaffeineSource[CuteMate]
  ): Deliverable = programmer.transform(caffeineSource.pull(), feature)

  val cto                                          = new Programmer[CaffeinatedBeverage]
  val caffeineSource: CaffeineSource[CuteMate] = new CuteMateSource

  def main(args: Array[String]): Unit =
    deliver("emojis", cto, caffeineSource)
}
```

- Does not compile!
- expected `Programmer[CuteMate]`, found `Programmer[CaffeinatedBeverage]`

A consumer that is contravariant in type A

Consumer[-A]



A contravariant programmer

```
class Programmer[-A <: CaffeinatedBeverage] {
  def transform(caffeine: A, feature: String): Deliverable =
    Deliverable(feature)
}

scala> val cto = new Programmer[CaffeinatedBeverage]
cto: Programmer[CaffeinatedBeverage] = Programmer@76e78d0

scala> val resource: Programmer[CuteMate] = cto
resource: Programmer[CuteMate] = Programmer@76e78d0
```

Covariant and contravariant positions

Covariant positions

```
scala> val pullFilterCoffee = () => FilterCoffee(69, "Ethopia")
pullFilterCoffee: () => FilterCoffee = $
$Lambda$5188/0x00000008019ff440@50695810

scala> val pullBeverage: () => CaffeinatedBeverage = pullFilterCoffee
pullBeverage: () => CaffeinatedBeverage = $
$Lambda$5188/0x00000008019ff440@50695810
```

- `FilterCoffee` *is-a* `CaffeinatedBeverage`
- a function returning `FilterCoffee` *is-a* function returning `CaffeinatedBeverage`

Covariant return types

```
trait Function0[+R] {
  def apply(): R
}
```

- Scala has covariant return types (just like Java)
- example: `Function0` is covariant in its return type R
- the same principle applies to methods

Covariance: The rules of the game

- If a class or trait is covariant in a type parameter A, it can only be used in **covariant positions**:
  - as a return type of a method
  - as a type of an immutable field
  - as a lower type bound for the type of a method parameter

Contravariant positions

```
def hasMoreCaffeineContent(
      x: CaffeinatedBeverage,
      y: CaffeinatedBeverage
  ): Boolean = x.caffeineContent > y.caffeineContent

val filterCoffees: List[FilterCoffee] = List(
    FilterCoffee(69, "Ethiopia"),
    FilterCoffee(75, "Indonesia")
  )

val sortedCoffees = filterCoffees.sortWith(hasMoreCaffeineContent)
```

- `FilterCoffee` *is-a* `CaffeinatedBeverage`
- a function expecting `CaffeinatedBeverage` *is-a* function expecting `FilterCoffee`

Contravariant input types

```
trait Function1[-T1, +R] {
  def apply(v1: T1): R
}
```

- Scala functions are contravariant in their input types
- the same principle applies to methods

Contravariance: The rules of the game

- If a class or trait is contravariant in a type parameter A, it can only be used in **contravariant position**
- it can only occur as a type of a method parameter

Invariance

A mutable caffeine source?

```
abstract class CaffeineSource[+A <: CaffeinatedBeverage] {
  def pull(): A
  def refill(a: A): Unit = ()
}
```

- This doesn't compile!
- The covariant type A occurs in contravariant position

A mutable caffeine source!

```
abstract class CaffeineSource[A <: CaffeinatedBeverage] {
  def pull(): A
  def refill(a: A): Unit = ()
}
```

- If a type parameter occurs in both covariant and contravariant positions, it must be invariant
- This means that mutable classes must be invariant in their type parameters

Motivating invariance

```
String[] strings = new String[] { "one", "two" };
Object[] objects = strings; // because arrays are covariant
objects[1] = 42; // java.lang.ArrayStoreException: java.lang.Integer
```

- Java arrays are covariant
- The compiler allows you to sneak values of the wrong type into an array
- Scala plays it safe: `Array[A]` is invariant

Lower type bounds

Prepending elements to a list

```
scala> val strings = "a" :: "b" :: "c" :: Nil
strings: List[String] = List(a, b, c)

sealed abstract class List[+A] {
 def ::(elem: A): List[A] = new ::(elem, this)
}
```

- This doesn't compile!
- `List` is covariant in A
- A occurs in contravariant position in the `::` method

Prepending with lower type bounds

```
sealed abstract class List[+A] {
  def ::[B >: A](elem: B): List[B] =  new ::(elem, this)
}

scala> val coffees = FilterCoffee(69, "Ethiopia") :: Nil
coffees: List[FilterCoffee] = List(FilterCoffee(69,Ethiopia))

scala> val beverages: List[CaffeinatedBeverage] = CuteMate(95) :: coffees
beverages: List[CaffeinatedBeverage] = List(CuteMate(95),
FilterCoffee(69,Ethiopia))
```

- We need to add a type parameter B to the prepend method
- The type B of prepended elements must be a super type of A
- A is no longer used in contravariant position
- The result is a `List[B]`

Avoidance strategies

Common subclassing use cases

- Modules
  - `trait UserRepository`
  - `class PostgresUserRepository extends UserRepository`

- Typeclass hierarchies
  - `trait Semigroup[A]`
  - `trait Monoid[A] extends Semigroup[A]`

- Algebraic data types

Algebraic data types

```
sealed abstract class User extends Product with Serializable
object User {
  final case class Authenticated(id: Long, name: String) extends User
  final case class Anonymous(sessionId: String) extends User
}
```

- Subclassing is an implementation detail of algebraic data types in Scala
- Other languages don't use subclassing for this
  - Haskell: data constructors
  - Rust: variants

Can’t we use an invariant List?

```
sealed trait LinkedList[A] {
  def ::(a: A): LinkedList[A] = LinkedList.::(a, this)
}
object LinkedList {
  final case class ::[A](head: A, tail: LinkedList[A])
      extends LinkedList[A]
  final case class Nil[A]() extends LinkedList[A]
}

scala> val users = User.Anonymous("1ABC") :: User.Authenticated(1, "hans") :: Nil()
                                  ^
       error: type mismatch;
        found   : User.Anonymous
        required: User.Authenticated
```

Hiding the subclasses

```
sealed abstract class User extends Product with Serializable
object User {
  final case class Authenticated(id: Long, name: String) extends User
  final case class Anonymous(sessionId: String) extends User

  def authenticated(id: Long, name: String): User =
    Authenticated(id, name)
  def anonymous(sessionId: String): User = Anonymous(sessionId)
}

scala> val users = User.anonymous("1ABC") :: User.authenticated(1, "hans") :: Nil()
users: LinkedList[User] = ::(Anonymous(1ABC),::(Authenticated(1,hans),Nil()))
```

A Scala 3 outlook

Algebraic data types in Scala 3

```
enum User {
  case Authenticated(id: Long, name: String)
  case Anonymous(sessionId: String)
}

scala> val users = User.Anonymous("1ABC") :: User.Authenticated(1, "hans") :: Nil()
val users: LinkedList[User] = ::(Anonymous(1ABC),::(Authenticated(1,hans),Nil()))
```

- Scala 3 enums generate subclasses
- The generated apply factory methods hide the subclass type
- Friendlier towards classes that are invariant in their type parameters

Thank you! Questions?

Daniel Westheide
daniel.westheide@innoq.com

Twitter: @kaffeecoder
Website: https://danielwestheide.com

innoQ Deutschland GmbH

Krischerstr. 100
40789 Monheim am Rhein
Germany
+49 2173 3366-0

Ohlauer Str. 43
10999 Berlin
Germany
+49 2173 3366-0

Ludwigstr. 180E
63067 Offenbach
Germany
+49 2173 3366-0

Kreuzstr. 16
80331 München
Germany
+49 2173 3366-0

Hermannstrasse 13
20095 Hamburg
Germany
+49 2173 3366-0

innoQ Schweiz GmbH

Gewerbestr. 11
CH-6330 Cham
Switzerland
+41 41 743 0116