The Scala Type System

Part 1, Generics and Variance

Agenda

1. Simple Types
2. Scala Inheritance
3. Generics
4. Upper Bounds (mention)
5. Variance Concepts
6. Covariance
7. Contravariance
8. Bounds in depth

Simple Types

```scala
abstract class Food { val name: String }

abstract class Fruit extends Food

case class Banana(name: String) extends Fruit
case class Apple(name: String) extends Fruit
```

```scala
abstract class Cereal extends Food

case class Granola(name: String) extends Cereal
case class Muesli(name: String) extends Cereal
```

```scala
val fuji = Apple("Fuji")

val alpen = Muesli("Alpen")
```

Scala Inheritance

```
def eat(f: Food): String = s"${f.name} eaten"

scala> eat(fuji)
res0: String = Fuji eaten
scala> eat(alpen)
res1: String = Alpen eaten
```

```
case class Bowl(food: Food) {
  override def toString = s"A bowl of yummy ${food.name}s"
  def contents = food
}

scala> val fruitBowl = Bowl(fuji)
fruitBowl: Bowl = A bowl of yummy Fujis
scala> val cerealBowl = Bowl(alpen)
cerealBowl: Bowl = A bowl of yummy Alpens
```

```
scala> fruitBowl.contents
res2: Food = Apple(Fuji)
scala> cerealBowl.contents
res3: Food = Muesli(Alpen)
```

Enter Generics

```scala
case class Bowl[F](contents: F) {
  override def toString: String = s"A yummy bowl of ${contents}s"
}
```

```scala
val appleBowl = Bowl(fuji)
val muesliBowl = Bowl(alpen)
```

```
scala> appleBowl.contents
res5: Apple = "food - Fuji"

scala> muesliBowl.contents
res6: Muesli = "food - Alpen"
```

```
case class Bowl[F](contents: F) {
  override def toString: String = s"A yummy bowl of ${contents.name}s"
}

Main.scala:24: value name is not a member of type parameter F
  override def toString: String = s"A yummy bowl of ${contents.name}s"
                                                               ^
```

Upper Bounds (mention)

```
abstract class Animal {
  val name: String
  override def toString: String = s"Animal - $name"
}

case class Dog(name: String) extends Animal

val dottie = Dog("Dottie")

val dogBowl = Bowl(dottie)
dogBowl: Bowl[Dog] = "A yummy bowl of Animal - Dotties"
```

Not what we want, so:

```
case class FoodBowl[F <: Food](contents: F) {
  override def toString: String = s"A yummy bowl of ${contents.name}s"
}
```

Now we can use `.name`

Also

can no longer serve my dog as food

```
val dogBowl = FoodBowl(dottie)

Main.scala:27: inferred type arguments [Dog] do not conform to
  method apply's type parameter bounds [F <: Food]
FoodBowl(dottie)
^
Main.scala:27: type mismatch;
 found   : Dog
 required: F
FoodBowl(dottie)
         ^
```

Dottie is pleased by the power of the scala type System

Variance

```scala
def serveToFruitEater(fruitBowl: FoodBowl[Fruit]) =
  s"mmmm, those ${fruitBowl.contents.name}s were very good"
```

```scala
val fruitBowl = FoodBowl[Fruit](fuji)
val cerealBowl = FoodBowl[Cereal](alpen)
```

```scala
serveToFruitEater(fruitBowl)
res2_2: String = "mmmm, those Fujis were very good"
```

```scala
serveToFruitEater(cerealBowl)
Main.scala:27: type mismatch;
 found    : FoodBowl[Cereal]
 required: FoodBowl[Fruit]
serveToFruitEater(cerealBowl)
```

So far, so good, but...

Invariance

```
def serveToFoodEater(foodBowl: FoodBowl[Food]) =
  s"mmmm, I really liked that ${foodBowl.contents.name}"
```

```
serveToFoodEater(fruitBowl)
Main.scala:27: type mismatch;
 found   : FoodBowl[Fruit]
 required: FoodBowl[Food]
Note: Fruit <: Food, but class FoodBowl is invariant in type F.
You may wish to define F as +F instead. (SLS 4.5)
serveToFoodEater(fruitBowl)
```

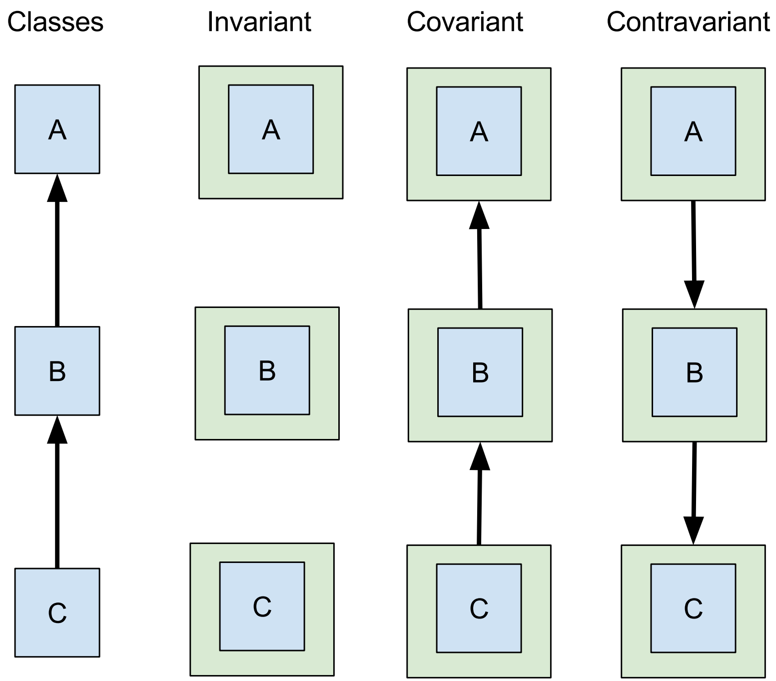
```
val fruitBowl = FoodBowl(fuji)
def serveToFruitEater(fruitBowl: FoodBowl[Fruit]) =
  s"mmmm, those ${fruitBowl.contents.name}s were very good"

serveToFruitEater(fruitBowl)
Main.scala:28: type mismatch;
 found   : FoodBowl[Apple]
 required: FoodBowl[Fruit]
Note: Apple <: Fruit, but class FoodBowl is invariant in type F.
You may wish to define F as +F instead. (SLS 4.5)
serveToFruitEater(fruitBowl)
```

Variance types

- What we want is for `FoodBowl[Apple]` to be a sub-type of `FoodBowl[Fruit]` *and* `FoodBowl[Food]`
- This relationship is known as variance and is demarked by **+** and **-** on the generics
- [+F] means that the type parameter is co-variant
- [-F] means that the type parameter is contra-variant
- These modifiers are only used in the definition of a generic class or trait
- If a type parameter has neither + or -, it is considered *invariant*

Different kinds of Variance



Covariance

```
case class FoodBowl[+F <: Food](contents: F) {
  override def toString: String = s"A yummy bowl of ${contents.name}s"
}
```

note the +F

```
def serveToFoodEater(foodBowl: FoodBowl[Food]) =
  s"mmmm, I really liked that ${foodBowl.contents.name}"
```

```
serveToFoodEater(FoodBowl[Fruit](fuji))
res4_0: String = "mmmm, I really liked that Fuji"
serveToFoodEater(FoodBowl(alpen))
res4_1: String = "mmmm, I really liked that Alpen"
```

Covariance

```
def serveToFruitEater(foodBowl: FoodBowl[Fruit]) =
  s"Nice fruity ${foodBowl.contents.name}"

serveToFruitEater(FoodBowl[Fruit](fuji))
res5_1: String = "Nice fruity Fuji"
serveToFruitEater(FoodBowl(fuji))
res5_2: String = "Nice fruity Fuji"
```

```
serveToFruitEater(FoodBowl(alpen))
Main.scala:29: type mismatch;
 found   : Muesli
 required: Fruit
serveToFruitEater(FoodBowl(alpen))
```

```
serveToFruitEater(FoodBowl[Food](fuji))
Main.scala:32: type mismatch;
 found   : FoodBowl[Food]
 required: FoodBowl[Fruit]
serveToFruitEater(FoodBowl[Food](fuji))
```

This is the difference between compile-time and run-time types

Contravariance!?

```
trait Food {
  val name: String
  override def toString = s"Yummy $name"
}
trait Fruit extends Food
case class Apple(val name: String) extends Fruit
case class Orange(val name: String) extends Fruit
```

```
trait Sink[T] {
  def send(item: T): String
}
object AppleSink extends Sink[Apple] {
  def send(item: Apple) = s"Coring and eating ${item.name}"
}
object OrangeSink extends Sink[Orange] {
  def send(item: Orange) = s"Juicing and drinking ${item.name}"
}
object FruitSink extends Sink[Fruit] {
  def send(item: Fruit) = s"Eating a healthy ${item.name}"
}
object AnySink extends Sink[Any] {
  def send(item: Any) = s"Sending ${item.toString}"
}
```

Contravariance!?

```scala
def sinkAnApple(sink: Sink[Apple]): String = {
  sink.send(Apple("Fuji"))
}
```

```
sinkAnApple(AppleSink)
res21: String = "Coring and eating Fuji"
```

```
sinkAnApple(OrangeSink)
Main.scala:27: type mismatch;
 found   : OrangeSink.type
 required: Sink[Apple]
sinkAnApple(OrangeSink)
```

```
sinkAnApple(FruitSink)
Main.scala:27: type mismatch;
 found   : FruitSink.type
 required: Sink[Apple]
Note: Fruit >: Apple (and FruitSink.type <: Sink[Fruit]),
  but trait Sink is invariant in type T.
You may wish to define T as -T instead. (SLS 4.5)
sinkAnApple(FruitSink)
```

Contravariance

```
trait Sink[-T] {
  def send(item: T): String
}
```

Note the `[-T]`

```
sinkAnApple(AppleSink)
res29: String = "Coring and eating Fuji"
```

```
sinkAnApple(FruitSink)
res30: String = "Eating a healthy Fuji"
```

And even..

```
sinkAnApple(AnySink)
res31: String = "Sending Yummy Fuji"
```

Notes so far

- Any supplied type T is invariant, covariant and contravariant to itself!
- Only type T is all three of these things to type T
- Any type T also satisfies lower and upper bounds to itself
- Class/Trait type parameters are the only place you can use variance modifiers
  - not in method type parameters
  - not in any usage of a type parameter
- We often say a class or trait is co- or contra-variant, but in fact type parameters, not the containing types have variance.
- And a class or trait can have both co- and contra-variant type parameters in its definition

Scala Function Variance

```scala
trait Description {
  val describe: String
}

case class Taste(describe: String) extends Description
case class Texture(describe: String) extends Description
```

Scala Function Variance

```
def describeAnApple(fn: Apple => Description) = fn(Apple("Fuji"))

val juicyFruit: Fruit => Taste =
  fruit => Taste(s"This ${fruit.name} is nice and juicy")

describeAnApple(juicyFruit)
res6: Description = Taste(This Fuji is nice and juicy)
```

Scala Function Variance

```scala
val bumpyOrange: Orange => Texture =
  orange => Texture(s"This ${orange.name} is bumpy")

def describeAFruit(fn: Fruit => Description) = fn(Apple("Fuji"))
```

```
describeAFruit(juicyFruit)
res19: Description = Taste(This Fuji is nice and juicy)

describeAFruit(bumpyOrange)
Main.scala:27: type mismatch;
 found   : Orange => Texture
 required: Fruit => Description
describeAFruit(bumpyOrange)
```

Scala Function Variance

```
trait Function1[-T1, +R]
```

[Scaladoc for Function1]

[Scaladoc for Function2]

- Parameters In ==> Contravariant `[-T]`
- Parameters Out ==> Covariant `[+T]`
- No relationship ==> Invariant `[T]`

Variance Problems

```scala
trait CombineWith[T] {
  val item: T
  def combineWith(another: T): T
}

case class CombineWithInt(item: Int) extends CombineWith[Int] {
  def combineWith(another: Int) = item + another
}
```

```scala
val cwi: CombineWith[Int] = CombineWithInt(10)
cwi.combineWith(5)
res23: Int = 15
```

Variance Problems

If CombineWith was covariant:

```
val cwo: CombineWith[Any] = CombineWithInt(10)
cwo.combineWith("ten")
```

But if we try...

```
trait CombineWith[+T] {
  val item: T
  def combineWith(another: T): T
}
Main.scala:25: covariant type T occurs in
  contravariant position in type T of value another
  def combineWith(another: T): T
                  ^
```

This seems very limiting...

Bounds Revisited

```
val ints = List(1,2,3,4)
ints: List[Int] = List(1, 2, 3, 4)
```

```
val anyvals = true :: ints
anyvals: List[AnyVal] = List(true, 1, 2, 3, 4)
```

```
val anys = "hello" :: anyvals
anys: List[Any] = List(hello, true, 1, 2, 3, 4)
```

- Adding an item to a List will return a new List with a suitable type
- The type is the most specific supertype for all items in the new List
- This is called the Least Upper Bounds (or sometimes LUB)

Everybody LUB Now

```scala
case class MixedFoodBowl[+F <: Food](food1: F, food2: F) {
  override def toString: String = s"${food1.name} mixed with ${food2.name}"
}
```

```scala
case class FoodBowl[+F <: Food](food: F) {
  override def toString: String = s"A bowl of ${food.name}"
  def mix(other: F): MixedFoodBowl[F] = MixedFoodBowl(food, other)
}
Main.scala:28: covariant type F occurs in
  contravariant position in type F of value other
  def mix(other: F): MixedFoodBowl[F] = MixedFoodBowl(food, other)
```

```scala
case class FoodBowl[+F <: Food](food: F) {
  override def toString: String = s"A bowl of ${food.name}"
  def mix[M >: F <: Food](other: M) = MixedFoodBowl[M](food, other)
}
```

- Introduce a new type M with lower bounds F, upper bounds Food
- M is a super-type of F (including F itself)
- M is still a sub-type of Food

In Use

```
val apple = Apple("Fuji")
val banana = Banana("Chiquita")
val alpen = Muesli("Alpen")

FoodBowl(banana).mix(apple)
res17: MixedFoodBowl[Fruit] = Chiquita mixed with Fuji

FoodBowl(apple).mix(alpen)
res18: MixedFoodBowl[Food] = Fuji mixed with Alpen
```

- M is inferred by the compiler as the LUB of F and the type passed in
- Scala type inference works hand-in-hand with generics and bounds

  "Just add +/- until it compiles." -- Adriaan Moors

  "A correctly written library uses Variance and Bounds so that the library user does not need to (mostly)" -- Dick Wall

Final Notes

- Variance is used on type parameters in class and trait definitions only
- Bounds are used everywhere else they are needed
- For co-variance you need a method generic lower bounded by the class generic for incoming method parameters

```
def incoming[U >: T](item: U) = ...
```

- For contra-variance you need a method generic upper bounded by the class generic for return types

```
def outgoing[U <: T]: U = ...
```

- Any type T is co-variant, contra-variant and invariant to itself
- Any type T is also a lower bound and an upper bound of itself
- If you can leave things invariant, life is simpler
- But maybe not for your library users...