Gregor Raýman

Brief introduction to functional programming in Scala

Bratislava, 16 Oct 2017

CLOUDFARMS

What is a function?

We programmers often use the terms **function** and **procedure** as synonyms.

Procedure → Subroutine is a sequence of program instructions that perform a specific task, packaged as a unit.

Function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output.

WIKIPEDIA
The Free Encyclopedia

CLOUDFARMS

What is Scala?

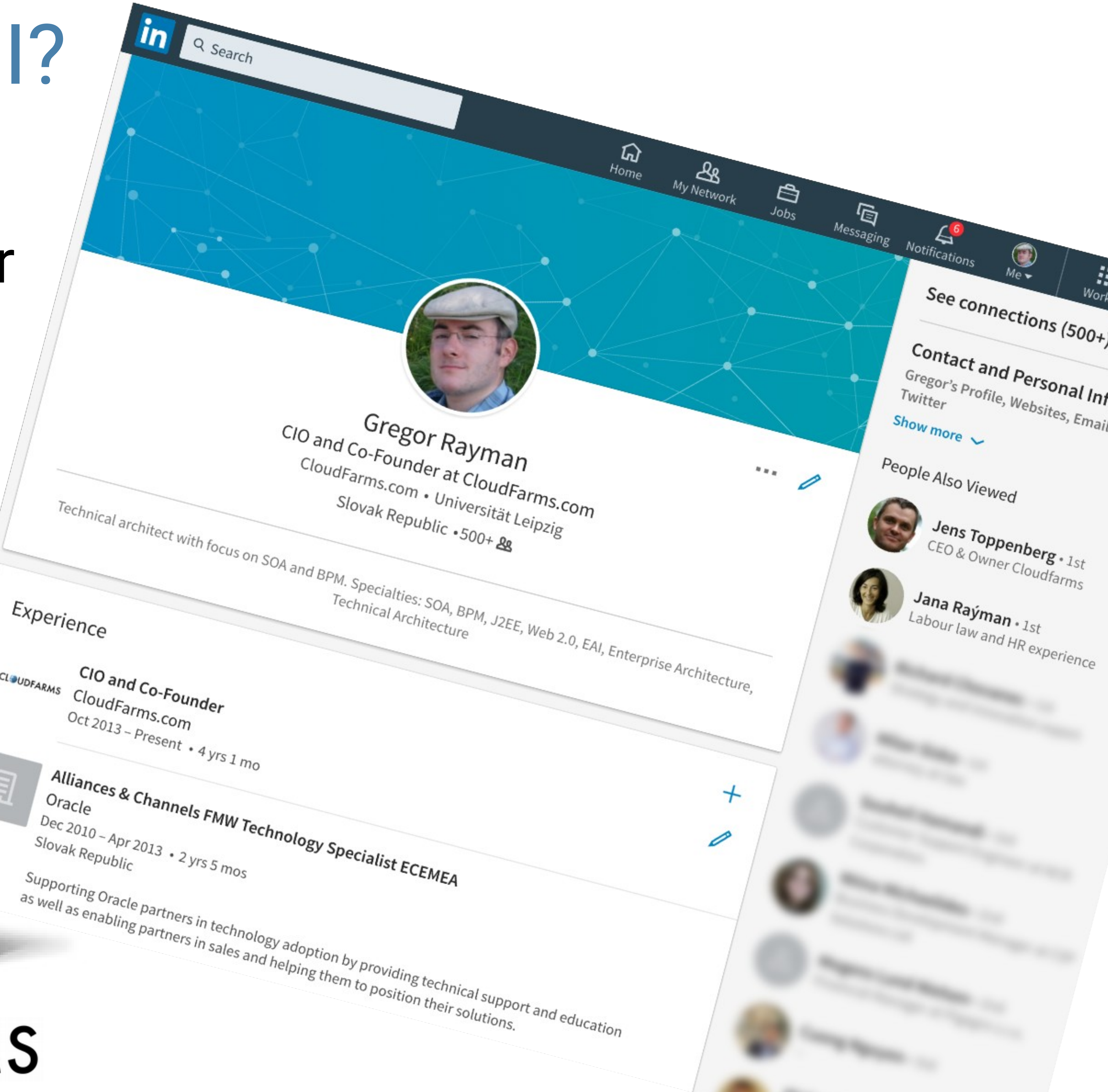
Scala

- is an object-oriented programming language
- is a functional programming language
- is a statically and strongly typed programming language
- is a scalable language
- compiles to JVM, JavaScript and native code*
- Has a lot of syntactic sugar

Who am I?

Programmer

Scala user



Search

Home

My Network

Jobs

Messaging

Notifications

Me

Gregor Rayman

CIO and Co-Founder at CloudFarms.com

CloudFarms.com • Universität Leipzig

Slovak Republic • 500+

Technical architect with focus on SOA and BPM. Specialties: SOA, BPM, J2EE, Web 2.0, EAI, Enterprise Architecture, Technical Architecture

Experience

CIO and Co-Founder

CloudFarms.com

Oct 2013 – Present • 4 yrs 1 mo

Alliances & Channels FMW Technology Specialist ECEMEA

Oracle

Dec 2010 – Apr 2013 • 2 yrs 5 mos

Slovak Republic

Supporting Oracle partners in technology adoption by providing technical support and education as well as enabling partners in sales and helping them to position their solutions.

See connections (500+)

Contact and Personal Inf

Gregor's Profile, Websites, Email

Twitter

Show more

People Also Viewed

Jens Toppenberg • 1st

CEO & Owner Cloudfarms

Jana Rayman • 1st

Labour law and HR experience

CLOUDFARMS

A bit of Scala syntax - variables

keyword name type (optional) initial value

```
var a: Int = 42
```

immutable

```
val b      = 24
```

```
a = a + b
```

STOP

```
b = a - b
```

```
a = a - b
println(a, b)
```

```
a = "hello"
```

STOP Wrong type!

CLOUDFARMS

A bit of Scala syntax - functions

keyword name parameter list(s) (optional) return type (optional)

```
def max(a: Int, b: Int): Int =
  if (a > b)
    a
  else
    b
```

function's body / expression

- Note that the **`if`**-statement is an expression that returns a value. That is why we don't need a **`return`** statement. (Scala has it, don't use it)
- If the function consist of only one expression, we don't need the parentheses

A bit of Scala syntax - functions

```
def gcd(a: Int, b:  Int) = {
  var x = b
  var y = a
  while (x ≠ 0) {
    val rest = y % x
    y = x
    x = rest
  }
  y.abs
}
```

- `while` is not an expression
- A block of multiple expressions enclosed in curly braces is itself an expression. The resulting value is the value of the last one. (here the value of y)

Look, a method call on a primitive integer. In Scala everything is a an object and so it has methods.

Even the operators are methods.

`1+2` is the same as `1.+(2)`

CLOUDFARMS

A bit of Scala syntax - functions

Inferred by the compiler

```
def gcd(a: Int, b:  Int) : Int  = {
  var x : Int  = b
  var y = a
  while (x ≠ 0) {
    val rest  : Int  = y % x
    y = x
    x = rest
  }
  y.abs
}
```

- **`while`** is not an expression
- A block of multiple expressions enclosed in curly braces is itself an expression. The resulting value is the value of the last one. (here the value of y)

CLOUDFARMS

A bit of Scala syntax - functions

Inferred by the compiler

```
def gcd(a: Int, b:  Int) : Int  = {
  var x : Int  = b
  var y = a
  while (x ≠ 0) {
    val rest  : Int  = y % x
    y = x
    x = rest
  }
  y.abs
}
```

- **while** is not an expression
- A block of multiple expressions enclosed in curly braces is itself an expression. The resulting value is the value of the last one. (here the value of y)

```
def gcd2(a: Int, b: Int): Int =
  if (a == 0) b.abs
  else gcd2(b % a, a)
```

- The result type of a recursive function has to be explicitly specified

CLOUDFARMS

A bit of Scala syntax - classes

Read only property

Parameter of the constructor

```
class Person(val name: String, aSurname: String) extends Mammal {
  private var surnameNow = aSurname
  private var spouse: Person = _
  def surname = surnameNow

  println(s"Person $name $surname was born")

  def marry(p: Person, takeSurname: Boolean): Unit = {
    if (this == p) throw new Exception("Cannot marry myself!")
    if (spouse != null) throw new Exception("Cannot marry twice!")
    println(s"$name $surname married ${p.name} ${p.surname}")
    if (takeSurname) surnameNow = p.surname
    spouse = p
    if (p.spouse != this) p.marry(this, !takeSurname)
  }
}
```

Member variable

Public member method (note, no parameter list)

Constructor code

Another public method

DO NOT PROGRAM THIS WAY PLEASE!

CLOUDFARMS

Substitution principle

```
val x = 7
```

```
val y: Int = 2 * x
val z: Int = x + x
```

```
var c = 0
def x:Int = {
  c = c + 1
  c
}
```

Do y and z contain the same value?
Is z odd or even?

CLOUDFARMS

Pure functions

A pure function:

- for the same input always returns the same value*
- the only effect it has is returning the result value. So no side effects.

```
def gcd(a: Int, b:  Int) = {
  var x = b
  var y = a
  while (x ≠ 0) {
    val rest = y % x
    y = x
    x = rest
  }
  y.abs
}
```

Mutable local variables are ok. This function is still pure

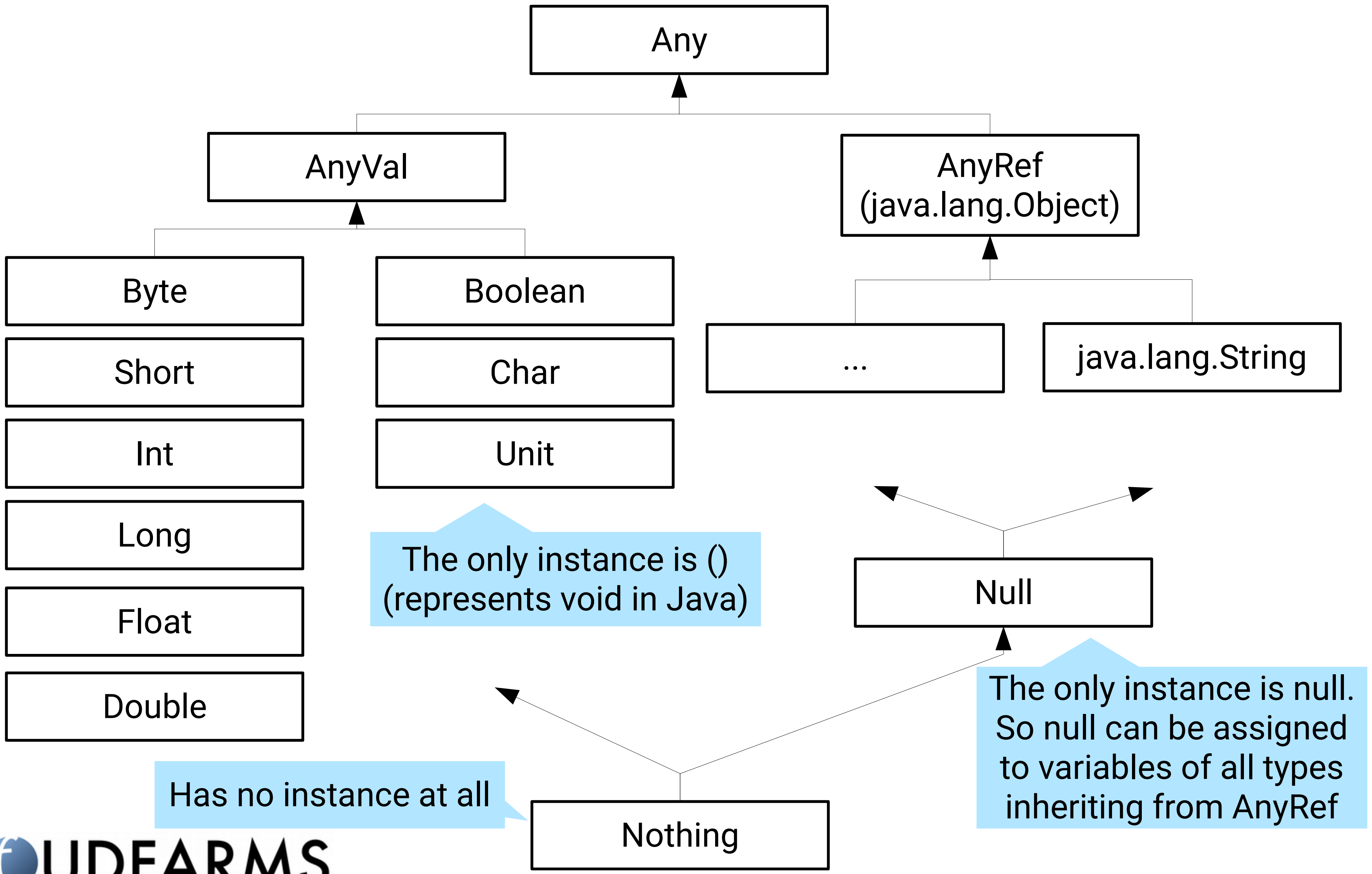
*) this also means that it always returns a value. So it must not throw an exception nor can it end in an endless loop

Benefits of purity and immutability

- Much simpler to reason about
- Easy to cache slowly computed functions
- Easier to use in multi-threaded environment
- Much simpler to reason about

CLOUDFARMS

Scala type hierarchy



CLOUDFARMS

Functions as first class objects

- In Scala functions are objects
- They themselves have types
- Can be assigned to variable
- Can be used as parameters of other functions
- Can be returned from functions

Anonymous function assigned to variables

```
val up = (x: String) ⇒ x.toUpperCase
val add = (a: Int, b: Int) ⇒ a + b
val plus: (Int, Int) ⇒ Int = add
```

The type of `add` and `plus` is `(Int, Int) ⇒ Int`

Functions have methods too

Explicitly named type of the parameter of the anonymous function

```
val withLen = (x: String) ⇒ x + x.length
val rev: (String ⇒ String) = _.reverse
```

Explicit type of the variable `rev`

No need to give the parameter a name (used only once)

Written as a method call

```
val withLenRev = withLen.andThen(rev)
val revWithLen = rev andThen withLen
```

Written as an operator

```
withLenRev("Scala") // returns 5alacS
revWithLen("Scala") // returns alacS5

revWithLen.apply("Scala")
```

Functions have a method called `apply`. Scala's syntactic sugar allows you to write just the parentheses.

CLOUDFARMS

Functions used as parameters

```
def doTwice(f: ⇒ Unit):Unit = {f; f}

doTwice { println("Hello world") }
```

I know this is an extremely simple example.

Have you noticed the curly braces instead of parentheses?
This is a nice syntactic sugar.
You can use `{x}` instead of `(x)` if the parameter list has one parameter.

CLOUDFARMS

Functions used as parameters

Two parameter lists

```
def doWith(c: Closeable)(f: Closeable ⇒ Unit): Unit = {
  try {
    f(c)
  } finally {
    c.close()
  }
}
```

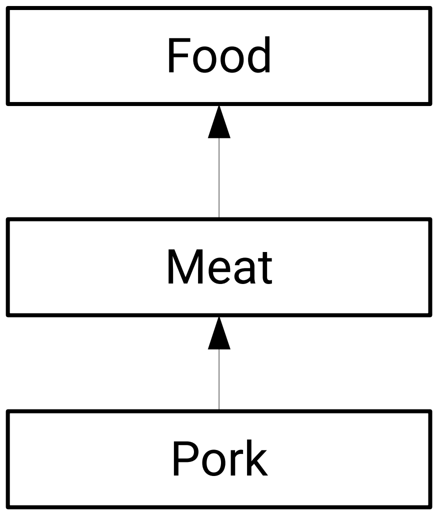
Now you can do the following. It looks like we have extended the syntax of Scala, doesn't it?

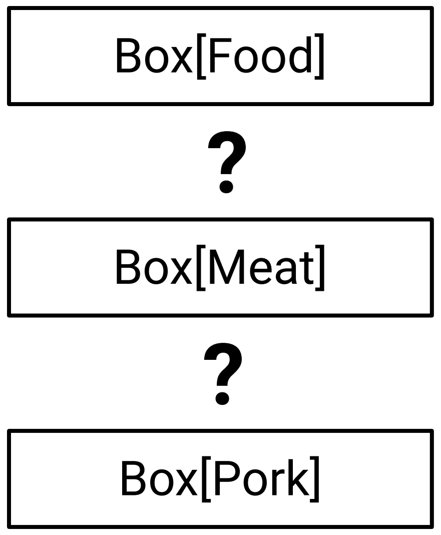
```
doWith(new FileInputStream("hello.txt")) { stream ⇒
     ...
}
```

Generics and variance

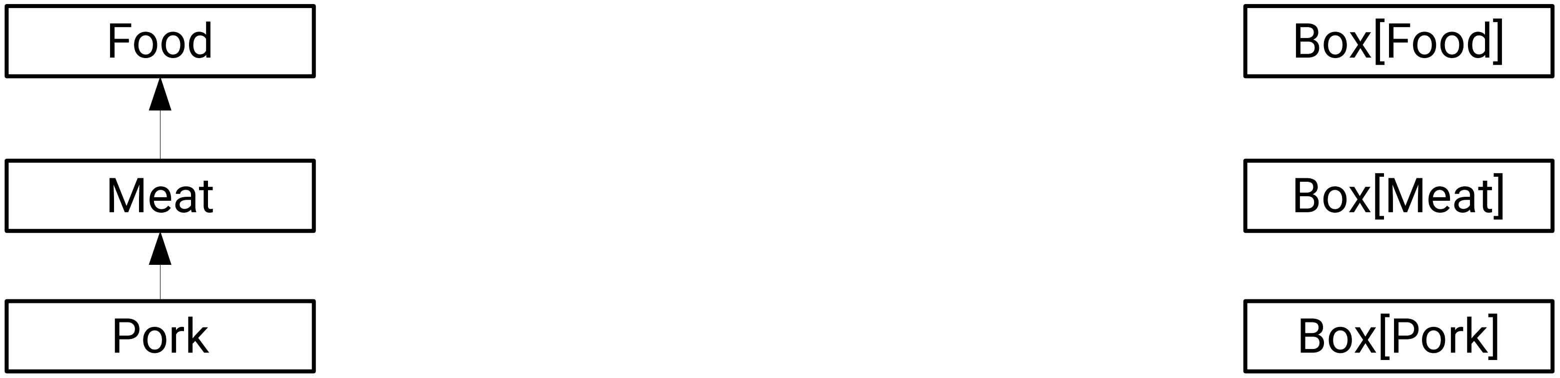
Invariant type parameter

```
class Box[A] {
  private var content: A = _
  def put(a: A): Unit = content = a
  def get: A = content
}
```





Generics - Invariant



```
def examineBoxedMeat(box: Box[Meat]): Unit = {
  val meat:Meat = box.get
  val meat2:Meat = markInLab(meat)
  box.put(meat2)
}
```

Box[Pork] is ok here. Box[Food] not

Box[Food] is ok here. Box[Pork] not

```
val mealBox: Box[Food]
val porkBox: Box[Pork]

examineBoxedMeat( ... )
```

CLOUDFARMS

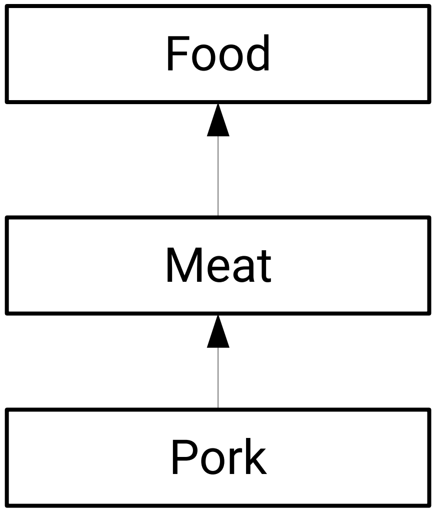
Variance - Covariant

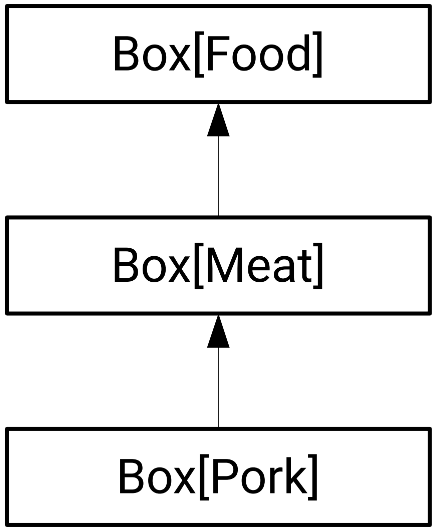
Covariant

```
class Box[+A](content: A) {
  def get: A = content
}

def examineBoxedMeat(box: Box[Meat]): Boolean = {
  val meat:Meat = box.get
  val result:Boolean = sendToLab(meat)
  result
}
```

Note that Box[+A] can only return A. It cannot accept it as parameters.





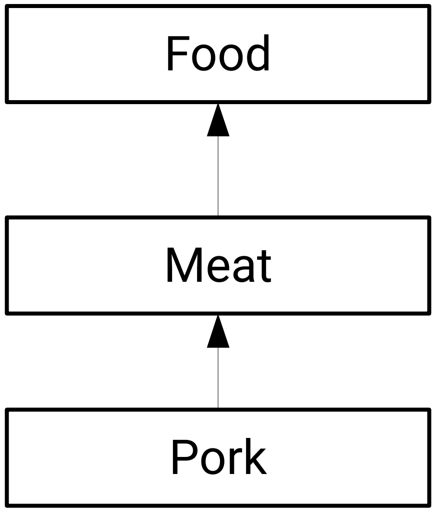
Variance - Contravariant

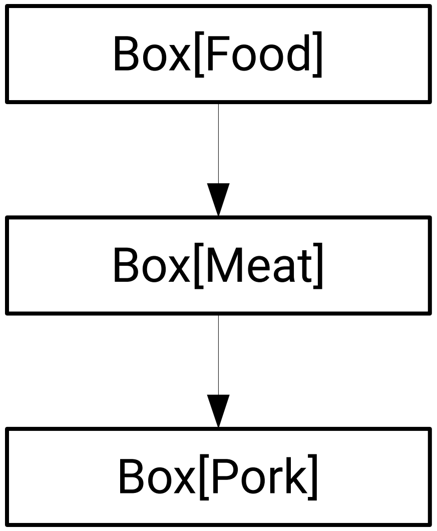
Contravariant

```
class Bin[-A] {
  def getdispose(a: A):Unit = ???
}

def cleanTheFridge(bin: Bin[Meat]): Unit = {
  val rottenMeat:Meat = getOldMeat()
  bin.dispose(rottenMeat)
}
```

Note that `Bin[-A]` methods cannot return A.
It can take A as parameters.





Collections

Covariant collections can only be immutable. Let's define our own:

```
trait Collection[+A] {
  def isEmpty:  Boolean
  def first: A
  def rest: Collection[A]
}
```

A **`trait`** is similar to Java's **`interface`** , it is abstract, defines capability of its instances. A **`class`** can implement (can inherit from) multiple **`trait`**s.

However, **`trait`**s can implement methods and they can also have member variables.

The simplest collections ever

A singleton instance

Is an empty collection of every type.
There is no Person in, no Bear in, no Integer in.

```
object Empty extends Collection[Nothing] {
  override def isEmpty = true
  override def first = throw new NotImplementedError
  override def rest = ???
}
```

This is a real Scala function

```
class One[+A](a:A) extends Collection[A] {
  override def isEmpty = false
  override def first = a
  override def rest = Empty
}
```

Is this useful?

CLOUDFARMS

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

Sir Charles Antony Richard Hoare

CLOUDFARMS

Option

All directly inheriting implementations must be in this same file

```
sealed abstract class Option[+A] {
  val isEmpty: Boolean
  def get: A
}
```

Look, an abstract val

```
object None extends Option[Nothing] {
  override val isEmpty = true
  override def get = ???
}
```

What is this?

```
case class Some[+A](it: A) extends Option[A] {
  override val isEmpty = false
  override val get = it
}
```

Look, a val overrides a def

The type `Option` allows us to explicitly on the type level to define, whether a variable, a parameter or a return value can be without a value.

CLOUDFARMS

Note: This is not the actual Scala implementation

Case classes – a bit of Scala sugar

A case class in Scala is a normal Scala class with a lot of helpful functionality automatically provided by the compiler

- All constructor parameters become read-only properties
- Automatic `toString` and `equals` implementations
- Generated `copy` methods
- Generated companion object* with useful methods

```scala
case class Point(x: Int, y: Int) {
   def moveByX(dx: Int)
     = copy(x = x + dx)
}

val p0 = Point(0,2)
val p10 = p0.moveByX(10)
p10.toString // returns "Point(10, 2)"
```

*Scala does not know static methods. But it knows singleton objects. An object with the same name as a class is called a companion object. For case classes a companion object is automatically generated. It contains an factory method for creating instances of the case class:

```scala
object Point {
  def apply(x: Int, y:  Int)
    = new Point(x, y)
   …
}
```

CLOUDFARMS

Switch and If on steroids. Pattern matching

```
def describe(it: Any): String = {
  it match {
    case 0 ⇒ "zero"
    case 1 ⇒ "one"
    case x: String ⇒ x
    case Point(0, 0) ⇒ "origin"
    case Point(0, y) if y > 0 ⇒ s"$y up on the x axis"
    case Point(x, y) ⇒ s"[$x,$y]"
    case _ ⇒ it.toString
  }
}
```

These patterns match only a single value

Type based pattern

Type based patterns, checking properties of the case class and binding them the local variables.

The “catch all” pattern.
If no pattern matches, a run-time exception is thrown.
The compiler can actually check, whether the patterns are exhaustive. Sealed classes are needed for this functionality.

CLOUDFARMS

Options instead of null

```
def organizeLecture(
  room: Room,
  projector: Projector,
  speaker: Person,
  interpreter: Person
): Lecture
```

Do we need an interpreter? Do we need a projector? Does the method signature tell us? Will we be able to organize the lecture?

```
val lecture = organizeLecture(
  Room("C"), null, Person("Gregor"), null)

lecture.sendInvitations()
```

Really? No projector needed?

Will the function never return null?

CLOUDFARMS

Options instead of null

```
def organizeLecture(
  room: Room,
  maybeProjector: Option[Projector],
  speaker: Person,
  maybeInterpreter: Option[Person]
): Option[Lecture]
```

This is clearly an allowed value

```
organizeLecture(Room("C"), None, Person("Gregor"), None)
match {
  case Some(lecture) ⇒ lecture.sendInvitations()
  case None ⇒ // do nothing
}
```

This will be called only when the function returns `Some[Lecture]`

No need for a default case, it can only be `Some` or `None`

More than one element - Lists

```scala
class Cons[+A](
  override val first: A,
  override val rest: Collection[A]
) extends Collection[A] {
  override val isEmpty = false
}
```

Now we can have collections with as many elements as we want. Here a list of 3 elements:
`val list123 = Cons(1, Cons(2, Cons(3, Empty)))`

Note the list is constructed from the end. We start with the `Empty` collection and then add the elements to the head of the list.
Luckily, Scala has an implementation with more functionality an a much nicer syntax.
In Scala's collection library our `first` is called `head`, `rest` is called `tail` and `Empty` is `Nil`.

Scala Lists

You have already seen that in Scala a method with one parameter can be written as an operator. So

`1 + 2` is the same as `1.+(2)` and `f andThen g` is the same as `f.andThen(g)`

However, when the operator ends with a semicolon, it is bound to the right operand. So

`a +: b` is the same as `b.+:(a)` and `a :: b` is the same as `b.::(a)`

These four lists are equal

```
List(1, 2, 3)
1 :: 2 :: 3 :: Nil
Nil.::(3).::(2).::(1)
::(3, ::(2, ::(1,Nil)))
```

Can you guess, what `::` means here?

Working with lists – Summing up

Lists can also be used in pattern matching:

```scala
def sum(xs: List[Int]): Int = xs match {
  case Nil ⇒ 0
  case h :: t ⇒ h + sum(t)
}
```

Danger. Stack overflow possible

```scala
@tailrec
def sumWithAcc(acc: Int, xs: List[Int]): Int = xs match
{
  case Nil ⇒ acc
  case h :: t ⇒ sumWithAcc(acc + h, t)
}
```

```scala
def sumWithLoop(xs: List[Int]): Int = {
  var acc = 0
  var rest = xs
  while (rest.nonEmpty) {
    acc += rest.head
    rest = rest.tail
  }
  acc
}
```

All these variants loop over the list

CLOUDFARMS

Working with Lists - Transformation

```scala
def map[A,B](as: List[A])(f: A ⇒ B): List[B] =
as match {
   case Nil ⇒ Nil
   case h :: t ⇒ f(h) :: map(t)(f)
}
```

We know and use the internal structure of the list to “loop” over its elements

What will the following code return?

Look, no loop visible here

```scala
map(List("one","two","three")) { _.length }
```

Exercise: Implement a function that filters a list and returns only element for which another functions returns true. What will the type of the function be?

Note: Scala’s Lists have the methods map, filter etc...

CLOUDFARMS

Useful methods on collections

```
trait C[A] {
  def map[B](f: A ⇒ B): C[B]
  def flatMap[B](f: A ⇒ C[B]): C[B]
  def filter(p: A ⇒ Boolean): C[A]
  def exists(p: A ⇒ Boolean): Boolean
  def forall(p: A ⇒ Boolean): Boolean
  def foreach(p: A ⇒ Unit): Unit
  def find(p: A ⇒ Unit): Option[A]
  def reduce(op: (A, A) ⇒ A): A
  def fold(z: A)(op: (A, A) ⇒ A): A
  def foldLeft[B](z: B)(op: (B, A) ⇒ A): B
  def foldRight[B](z: B)(op: (A, B) ⇒ A): B
  def collect[B](pf: PartialFunction[A, B]): C[B]
   ...
  def sum: A = reduce( _ + _ )
  def min: A = reduce( (x:A, y:A) ⇒ if (x < y) x else y )
}
```

Can we do that?
Where do the +, < come from?

Can you guess what these methods do, just by looking at their types?

Note: This is just a simplification

CLOUDFARMS

map and flatMap

Task: Split a sentence (list of strings) to a list of characters codes

```
List("Hello","World") ⤳ List(72, 101, 108, 108, 111, 87, 111, 114, 108, 100)

def strToCharCodes(s: String) = s.toList.map(_.toInt)
def split1(ws: List[String]) = ws map strToCharCodes
```

List of lists

```
split1(List("Hello","World")) ⤳ List(List(72, 101, 108, 108, 111), List(87, 111, 114, 108, 100))

split1(List("Hello","World")).flatten ⤳ List(72, 101, 108, 108, 111, 87, 111, 114, 108, 100)
```

Unwraps the list of lists

```
List("Hello","World").map( w ⇒ w.map(c ⇒ c.toInt)).flatten
List("Hello", "World").flatMap(w ⇒ w.map(c ⇒ c.toInt))
```

CLOUDFARMS

For comprehensions

In Scala there are no **for**-cycles. **for** is just syntax sugar for `map`, `flatMap`, `withFilter` and `foreach`

```
List("Hello", "World").flatMap(w ⇒ w.map(c ⇒ c.toInt))
```

is the same as

```
for (
  w ← List("Hello", "World");
  c ← w
) yield c.toInt
```

CLOUDFARMS

For comprehensions

- Last ← before **`yield`** becomes `map`
- Last ← without **`yield`** becomes `foreach`
- Other ← become `flatMap`
- **`if`** becomes `withFilter`

This is not limited to collections. Any class that implements (some of) the methods, can be used in a for-comprehension.

`for` is just syntactic sugar

CLOUDFARMS

Options instead of null, again

```
def organizeLecture(
  room: Room,
  maybeProjector: Option[Projector],
  speaker: Person,
  maybeInterpreter: Option[Person]
): Option[Lecture]

for (lecture ← organizeLecture(
  Room("C"), None, Person("Gregor"), None)
) {
  lecture.sendInvitations()
}
```

This will be called only when the function returns `Some[Lecture]`. No need to write an empty clause when it returns `None`.

Useful methods (on collections?)

```
trait C[A] {
  def map[B](f: A ⇒ B): C[B]
  def flatMap[B](f: A ⇒ C[B]): C[B]

    ...
}
```

C stands for Context. The function `f` called by `map` does not need to know anything about the structure of C. The function used in `flatmap` knows about C, so that we cannot combine incompatible contexts.

```
for (
  person ← personContext;
  meal ← mealContext if !meal.meat || !person.vegetarian
) yield (person.name, meal.name)
```

This works if both `personContext` and `mealContext` are of the same kind (both Collection, or both Future, etc).

CLOUDFARMS

Using **for** for some random stuff :)

```scala
trait Rnd[+A] {
  def next(): A
}


object RndDouble extends Rnd[Double] {
  override def next(): Double = Math.random()
}

class RndInt(from: Int, to: Int) extends Rnd[Int] {
  override def next(): Int =
    from + ((to - from + 1) * RndDouble.next).floor.toInt
}

val tenGenerator = new RndInt(1, 10)
```

Does this function really needs to know that it deals with random numbers?

CLOUDFARMS

Using **for** for some random stuff :)

```
trait Rnd[+A] { self ⇒
  def next(): A
  final def map[B](f: A ⇒ B) = new Rnd[B] {
    override def next(): B = f(self.next())
  }
}

val tenGenerator = RndDouble map { n ⇒
  1 + (10 * n).floor.toInt
}
```

`self` is just an alias for **this**, because **this** in the `Rnd[B]` references the anonymous nested class

This function does not know that n is a random number

CLOUDFARMS

Using **for** for some random stuff :)

```scala
trait Rnd[+A] { self ⇒
  def next(): A
  final def map[B](f: A ⇒ B) = new Rnd[B] {
    override def next(): B = f(self.next())
  }
  final def flatMap[B](f: A ⇒  Rnd[B]) = new Rnd[B] {
    override def next(): B = f(self.next()).next()
  }
}

val moveGenerator = for (
  c ← tenGenerator;
  col = ('A' + c).toChar;
  row ← tenGenerator
) yield (col, row)

moveGenerator.next() // returns (F,2) or (B,4) or  ...
```

`flatMap` combines this Rnd with the one returned from f

Using **for** for some random stuff :)

We have created a generator of (Char, Int) pairs.
Can we also create a generator of Int sequences?

```
val seqGenerator = for (
  i ← 1 to 10;
  n ← tenGenerator
) yield n
```

Does not work! We cannot combine a Range with a Rnd this way

```
class IntSeqRnd(len:  Int) extends Rnd[Seq[Int]] {
  override def next() =
    for (i ← 1 to len) yield tenGenerator.next()
}
```

But this is not functional!

A pure function:

- for the same input always returns the same value
- the only effect it has is returning the result value. So no side effects.

`Math.random()` certainly is not a pure function. It does not always return the same result value (that would make it be quite pointless) and calling it changes some internal **state** of the pseudo-random generator, so it has side effects.

Can we have a pure function that can provide random numbers?

Modeling state functionally

- In object oriented programming state is modeled as objects. State changes are modeled as changing the data of object’s member variables.
- In (pure) functional programming, data is immutable.

```scala
class Person(n:String) {
  private var name = n
  def getName = name
  def setName(nn:String):Unit = {name = nn}
}

val p = new Person("Gregor")
p.setName("Greg")
p.getName
```

The state of the object p has been changed

```scala
case class Person(name: String)

val p = Person("Gregor")
val p1 = p.copy(name = "Greg")
```

The object p is not changed.
New state is in the new object p1

CLOUDFARMS

“Pure?” functional random numbers

```scala
class FunRandom extends Function0[(Double, FunRandom)] {
  private val n = Math.random()
  private lazy val next = new FunRandom
  def apply(): (Double, FunRandom) = (n, next)
}

val f0 = new FunRandom
val (n1a, f1a) = f0()
val (n1b, f1b) = f0()  // n1a == n1b
val (n2a, f2a) = f1a()
val (n2b, f2b) = f1b() // n2a == n2b
```

`FunRandom` always returns the same result. It returns a pair of a random number and another instance of `RunRandom`.

It is still not pure, because creating the new instance has side effects on the state of `Math.random`.

Is there a way? Can a real world program be functionally pure?

Useful programs have to interact with the outside world. They have to have have inputs, outputs. So totally pure programs are not really useful. But we can “push” the impure , state changing functionality to the borders of the programs. (To learn more about this, study the IO monad)

Note: Yes, it is possible to create a purely functional pseudo-random generator by keeping the state inside the function instances.

Type classes

Remember?

```
trait C[A] {
    ...
  def sum: A = reduce( _ + _ )
  def min: A = reduce( (x:A, y:A) ⇒ if (x < y) x else y )
}
```

Can we do that?
Where do the +, < come from?

Type classes

Let’s write a function that finds the smallest element. To be able to that, we need a decision function that tells which from two elements is smaller.

```
trait LessThan[-T] {
  def lt(a:T, b:T): Boolean
}

case class Person(name: String, age: Int, height: Int)


val ageLessThan: LessThan[Person] = new LessThan[Person] {
  override def lt(a: Person, b: Person) = a.age < b.age
}
```

A Person does not have any “natural” ordering

Using this, we will sort people by age

CLOUDFARMS

Type classes

```
val ageLessThan: LessThan[Person] = new LessThan[Person] {
  override def lt(a: Person, b: Person) = a.age < b.age
}

def least[T](a: T, b: T)(lessThan: LessThan[T]) =
  if (lessThan.lt(a, b)) a else b

Least(
  Person("Gregor", 47, 189),
  Person("Vincent", 7, 130)
)(ageLessThan)
```

The function `least` will be applicable to any type `T` for which we can provide an instance of `LessThan[T]`

Scala has a very powerful feature called implicit parameters. It instructs the compiler to automatically use implicit variables whenever we have not specified one explicitly.

Let's use this feature to simplify our code

Type classes

```
implicit val ageLessThan: LessThan[Person] =
  new LessThan[Person] {
    override def lt(a: Person, b: Person) = a.age < b.age
  }

def least[T](a: T, b: T)(implicit lessThan: LessThan[T]) =
  if (lessThan.lt(a, b)) a else b

Least(
  Person("Gregor", 47, 189),
  Person("Vincent", 7, 130)
)
```

No need to provide the parameter list explicitly

Note that this works only, when the implicit parameter can be selected unambiguously.

There is even more concise way to write the function `least[T]`

Type classes

This notation means the same as the previous one.
The function automatically gets another parameter list with an anonymous parameter of the type `LessThan[T]`
We say that `T` belongs to the type class `LessThan`.

```
def least[T:LessThan](a: T, b: T) = {
  val lessThan = implicitly[LessThan[T]]
  if (lessThan.lt(a, b)) a else b
}
```

To access the parameter by name,
we use the helper method *`implicitly`*

Let’s find the smallest element from more than two elements.

Type classes, Higher kinded types

This means, that C needs a parameter. C[T] is a type,
C is called a type constructor.

```
trait Reducer[-C[_]] {
  def reduce[T](c: C[T])(f: (T, T) ⇒ T):T
}

implicit val seqReducer: Reducer[TraversableOnce] = new Reducer[TraversableOnce] {
  override def reduce[T](c: TraversableOnce[T])(f: (T, T) ⇒ T) = c.reduce(f)
}

def min[C[_]:Reducer, T:LessThan](c:C[T]) = {
  val reducer = implicitly[Reducer[C]]
  val lessThan = implicitly[LessThan[T]]
  reducer.reduce(c)(least[T])
}

min(List(
  Person("Gregor", 47, 189),
  Person("Vincent", 7, 130),
  Person("Adam", 4, 101))
)
```

CLOUDFARMS

Questions?

cloudfarms.com and cloudfarms.online