

# Programming in Scala

## Lecture Three

---

Angelo Corsaro

17 September 2020

Chief Technology Officer, ADLINK Technology Inc.

# Table of contents i

1. Logistics
2. Reviewing Fold
3. More on Scala Classes
4. Algebraic Data Types
5. Case Classes and  
Pattern Matching

## Table of contents ii

- 6. For Expression Revisited
- 7. Type Parameterization
- 8. Monads: A Gentle Introduction
- 9. Parallel and Concurrent Computations
- 10. Homeworks

## Reviewing Fold

---

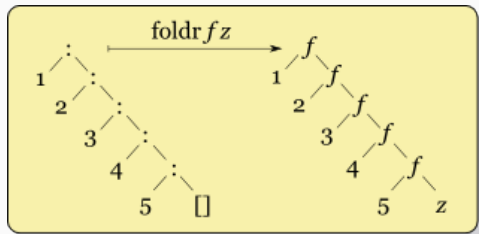
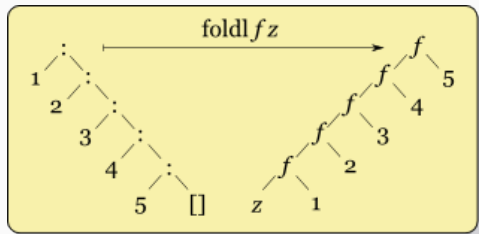
# Folding

In functional programming, **fold**, refers to a family of *higher-order functions* that analyze a recursive data structure and through use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value.

Folds can be regarded as consistently replacing the structural components of a data structure with functions and values.

Two variants of **fold** exist, namely **foldl** and **foldr**.

# Visualizing folds



# Folding Lists

The functions we just wrote for lists can all be expressed in terms of fold operator.

---

```
1  def sum(xs: List[Int]): Int = xs.foldRight(0)(_ + _)
2
3  def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)
4
5  def reversel(ys: List[Int]) = ys.foldLeft(List[Int]())((xs: List[Int], x: Int) => x :: xs)
6
7  def reverser(ys: List[Int]) = ys.foldRight(List[Int]())((x: Int, xs: List[Int]) => xs :: List(x) )
```

---

# foldr and foldl

---

```
1  def reverse(xs: List[Int]): List[Int] = xs match {
2      case List() => List()
3      case y::ys => reverse(ys) ::: List(y)
4  }
5  val xs = List(1,2,3,4,5)
6  val ys = reverse(xs)
7
8  val zs = ys.foldLeft(List[Int]())((xs: List[Int], x: Int) => x :: xs)
9
10 val ks = zs.foldRight(List[Int]())((x: Int, xs: List[Int]) => xs ::: List(x) )
```

---



## More on Scala Classes

---

# Scala Classes Refresher

In the first lecture we saw the basic mechanism provided by Scala to declare Classes. Back then we had seen *closures* without realizing it, thus let's take a fresh look at some of the examples:

---

```
1 class Complex(val re: Double, val im: Double) {  
2   override def toString = s"$re+i$im"  
3 }  
4  
5 class Rational(a: Int, b: Int) {  
6   assert(b != 0)  
7   val num: Int = a / gcd(a, b)  
8   val den: Int = b / gcd(a, b)  
9  
10  def this(n: Int) = this(n, 1)  
11 }
```

---

# Abstract Classes

In Scala the **abstract** modifier is used to denote an abstract class. In other terms, a class that cannot be instantiated.

An abstract class *may have* abstract members that don't have an implementation (definition).

---

```
1 package edu.esiee.scala18.ex1
2
3 abstract class Shape {
4     def scale(a: Float): Shape
5
6     // Parameter-less methods: Two takes:
7     def show(): Unit
8
9     def area: Float
10    def perimeter: Float
11
12 }
```

---

# Parameter-less Methods

Parameter-less methods can be declared with or without parenthesis.

The convention is that parenthesis are used for methods that cause side-effects – like show above.

Parenthesis are not used to methods that compute properties. This way there **uniform access** between attributes and methods.

# Extending a Class

The *Shape* abstract type had defined some of the characteristics shared by some geometrical shapes.

A concrete shape can be defined by extending the abstract Shape class as follows:

---

```
1 package edu.esiee.scala18.ex1
2
3 class Circle(val radius: Float) extends Shape {
4
5     def scale(a: Float): Shape = new Circle(a * radius)
6
7     def show(): Unit = print(s"Circle($radius)")
8
9     def area: Float = Math.PI.toFloat*radius*radius
10    def perimeter: Float = 2*Math.PI.toFloat*radius
11 }
```

---

## Methods or Vals?

The **area** and **perimeter** have been defined as parenthesis less methods.

As our shape are immutable, a valid question is if we could define them with **vals** and compute them only once, as opposed to every-time the method is called.

The good news is that Scala supports the concept of *abstract val*, let's see how to leverage them.

# Abstract val

Along with abstract methods, an abstract class can also have abstract vals.

These are values which are declared but not defined.

If we were to leverage abstract values our Shape class could be declared as follows:

---

```
1 package edu.esiee.scala18.ex2
2
3 abstract class Shape {
4     def scale(a: Float): Shape
5
6     def show(): Unit
7
8     val area: Float
9     val perimeter: Float
10
11 }
```

---

# Implementing abstract val

Implementing an abstract val is as simple as defining it as show in the code example below:

---

```
1 package edu.esiee.scala18.ex2
2
3 class Circle(val radius: Float) extends Shape {
4
5     def scale(a: Float): Shape = new Circle(a * radius)
6
7     def show(): Unit = print(s"Circle($radius)")
8
9     val area: Float = Math.PI.toFloat*radius*radius
10    val perimeter: Float = 2*Math.PI.toFloat*radius
11 }
```

---



# Observation

Now, as we are using *val* the shape area is computed only once, at the time of initialization. This is an improvement compared from the previous example. Additionally, the client code cannot tell whether we are implementing the area and the perimeter with a *val* or with a parenthesis less method which is very good.

Yet... We are still spending the time even if the client code is never calling that operation. In an ideal world we would want to compute it once and only if needed... Can we do that?

# lazy val

Scala defines the **lazy** modifier to indicate a value that should be computed lazily only when accessed.

If we want to do so, the only thing we need to change is the definition of the val, as shown below:

---

```
1 package edu.esiee.scala18.ex3
2
3 import edu.esiee.scala18.ex2.Shape
4
5 class Circle(val radius: Float) extends Shape {
6     def scale(a: Float): Shape = new Circle(a * radius)
7
8     def show(): Unit = print(s"Circle($radius)")
9
10    lazy val area: Float = Math.PI.toFloat*radius*radius
11    lazy val perimeter: Float = 2*Math.PI.toFloat*radius
12 }
```

---

# Overriding

A subclass can override both *methods* as well as *val* defined in the parent class.

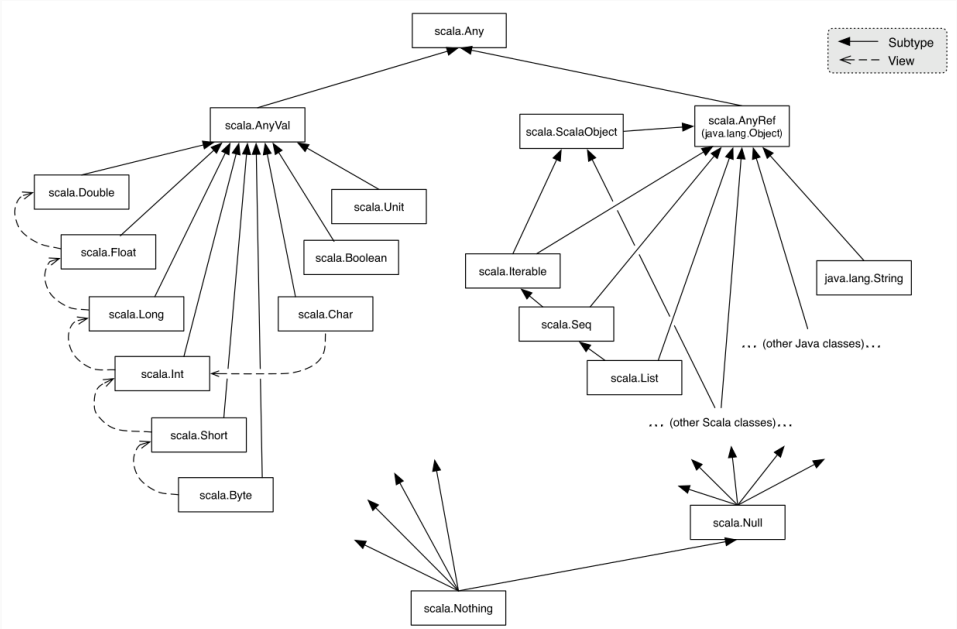
The super-class constructor can also be explicitly called as part of the extend declaration as shown in the example below:

---

```
1 package edu.esiee.scala18.ex3
2
3 class FastCircle(r: Float) extends Circle(r) {
4     override lazy val perimeter: Float = Math.PI.toFloat * (radius.toInt << 1)
5
6     override def show(): Unit = print(s"0($radius")
7 }
```

---

# Scala Type Hierarchy



# Algebraic Data Types

---

# Algebraic Data Type

In type theory and commonly in functional programming, an **algebraic data type** is a kind of composite type. In other term a type obtained by composing other types.

Depending on the composition operator we can have **product types** and **sum types**

# Product Types

**Product types** are algebraic data types in which the *algebra* is *product*

A **product type** is defined by the conjunction of two or more types, called fields. The set of all possible values of a product type is the set-theoretic product, i.e., the Cartesian product, of the sets of all possible values of its field types.

## Example

---

```
data Point = Point Double Double
```

---

In the example above the product type `Point` is defined by the Cartesian product of  $Int \times Int$

# Sum Types

The values of a sum type are typically grouped into several classes, called variants.

A value of a variant type is usually created with a quasi-functional entity called a constructor.

Each variant has its own constructor, which takes a specified number of arguments with specified types.



## Sum Types Cont.

The set of all possible values of a sum type is the set-theoretic sum, *i.e.*, the disjoint union, of the sets of all possible values of its variants.

Enumerated types are a special case of sum types in which the constructors take no arguments, as exactly one value is defined for each constructor.

Values of algebraic types are analyzed with pattern matching, which identifies a value by its constructor or field names and extracts the data it contains.

### Example

---

```
data Tree = Empty
          | Leaf Int
          | Node Tree Tree deriving (Show)

> let t = Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
> :t t
t :: Tree
```

---

# Case Classes and Pattern Matching

---

# Case Classes

Scala **case classes** provide a way to *mimic* algebraic data types as found in functional programming languages.

The Scala version of the *Tree* type we defined in Haskell is:

```
1  abstract class Tree
2  object Empty extends Tree
3  case class Leaf(v: Int) extends Tree
4  case class Node(l: Tree, r: Tree) extends Tree
5
6  val t = Node(Leaf(1), Node(Leaf(2), Leaf(3)))
```

If you noticed the *Tree* algebraic data type was declared through case classes that did not have named attributes. In other term, there were no **val** or **var**.

That does not mean that you cannot declare **val** or **var** as part of a case class. It has more to do with the fact that case classes are worked upon using pattern matching.

# Pattern Matching

Scala pattern matching has the following structure:

*selector* `match` *alternatives*

A pattern match includes a series of *alternatives*, each starting with the keyword **case**.

Each alternative includes a *pattern* and one or more expressions, which are evaluated only if the pattern matches.

Let's see what are the kinds of patterns supported by Scala.

# Wildcard Pattern

The *wildcard pattern*, denoted by `_`, matches anything – thus its appellation.

## Example

```
1  def isEmpty(xs: List[Int]) = xs match {  
2      case List() => return true  
3      case _ => return false  
4  }  
5  
6  def isSingleElementList(xs: List[Int]) = xs match {  
7      case List(_) => true  
8      case _ => false  
9  }
```

# Constant Pattern

A constant pattern matches only itself. Any literal, such as 18, 42, "Ciao", true can be used as a constant.

Any **val** or **singleton object** can also be used as a constant.

## Example

```
1  def toString(d: Int) = d match {  
2      case 0 => "Zero"  
3      case 1 => "Uno"  
4      // ...  
5      case 9 => "Nove"  
6  }  
7
```

# Variable Pattern

A variable pattern matches any object, just like a wildcard. But unlike a wildcard, Scala binds the variable to whatever the object is.

You can then use this variable to act on the object further.

## Example

```
1  def checkZero(d: Int) =  
2      d match {  
3          0 => "zero"  
4          v => v + " is not zero"  
5      }
```



# Constructor Pattern

A constructor pattern looks like `Node(Leaf(1), Node(Leaf(2), _))` . It consists of a name (`Tree`) and then a number of patterns within parentheses.

Assuming the name designates a case class, such a pattern means to first check that the object is a member of the named case class, and then to check that the constructor parameters of the object match the extra patterns supplied.

## Example

```
1 def triangleTree(t: Tree) =  
2   t match {  
3     case Node(Leaf(_), Leaf(_)) => True  
4     case _ => False  
5   }
```

Please notice that Sequence and Tuple patterns are just a special case of constructor patterns.

## Exercise

Define the following functions for our Tree type:

1. `height` which computes the tree height.
2. `sum` which is the function that computes the sum of all the elements of the tree.
3. `fold` which is a function that applies a generic binary operator to the tree, where the zero is used when for empty nodes.

# Typed Pattern

Typed pattern are used to test and cast types.

## Example

```
1  def generalSize(x: Any) = x match {  
2    case s: String => s.length  
3    case m: Map[_ , _] => m.size  
4    case _ => -1  
5  }
```

# Pattern Guards

A pattern guard comes after a pattern and starts with an **if**. The guard can be an arbitrary boolean expression, which typically refers to variables in the pattern.

If a pattern guard is present, the match succeeds only if the guard evaluates to true.

## Example

```
1 // match only positive integers
2 case n: Int if 0 < n => ...
3
4 // match only strings starting with the letter 'a'
5 case s: String if s(0) == 'a' => ...
```

## Other use of Patterns

Patterns can be used also in assignments, such as in:

```
1  val (a, b, c) = tripletFun(something)
```

Patterns can also be used in for expressions:

```
1  for ((key, value) <- store)
2    print("The key = " + key + "has value = " + value)
```

# For Expression Revisited

---

# For Expression

The general form of a **for** expression is:

**for** ( *seq* ) **yield** *expr*

Here, *seq* is a sequence of *generators*, *definitions*, and *filters*, with semicolons between successive elements.

# Generator, Definition and Filter

A **generator** is of the form:

$$pat \leftarrow expr$$

A **definition** is of the form:

$$pat = expr$$

A **filter** is of the form:

$$if \quad expr$$



# for expression: Examples

## Example

```
1  for (x <- List(1, 2); y <- List("one", "two")) yield (x, y)
2
3  for ( i <- 1 to 10; j <- i to 10) yield (i,j)
4
5  val names = List("Morpheus", "Neo", "Trinity", "Tank", "Dozer")
6
7  for (name <- names; if name.length == 3) yield name
8
9  val xxs = List(List(1,2,3,4), List(6, 8, 10, 12), List(14, 16, 18))
10
11 for (xs <- xxs; e <- xs; if (e % 2 == 0)) yield e
12
13 for (xs <- xxs; if (xs.isEmpty == false); h = xs.head) yield h
```

# Type Parameterization

---

# Type Constructors in Scala

Beside first order types, Scala supports Type Constructors, – a kind of higher order types.

In the case of Scala, a parametric class is an n-ary type operator taking as argument one or more types, and returning another type.

## Example

The list time we have seen this far, is a type constructor declared as:

```
class List[+T]
```

Thus *List[Int]* and *List[String]* are two instances of the *List[+T]* type constructor.

# Monads: A Gentle Introduction

---

# Monad: Basic Idea

Monads can be thought of as composable computation descriptions.

The essence of monad is the separation of composition timeline from the composed computation's execution timeline, as well as the ability of computation to implicitly carry extra data, as pertaining to the computation itself, in addition to its one (hence the name) output, that it will produce when run (or queried, or called upon).

This lends monads to supplementing pure calculations with features like I/O, common environment, updatable state, etc.

## Monad: Basic Idea / Cont.

Each monad, or computation type, provides means, subject to Monad Laws, to:

- create a description of a computation that will produce (a.k.a. "return") a value, and
- combine (a.k.a. "bind") a computation description with a *reaction* to it, – a pure function that is set to receive a computation-produced value (when and if that happens) and return another computation description.

Reactions are thus computation description constructors. A monad might also define additional primitives to provide access to and/or enable manipulation of data it implicitly carries, specific to its nature; cause some specific side-effects; etc..

# Monad Class and Monad Laws

Monads can be viewed as a standard programming interface to various data or control structures, which is captured by the Monad class. All common monads are members of it:

---

```
class Monad m where
  (>=>=)  :: m a -> (a -> m b) -> m b
  (>=>)   :: m a -> m b          -> m b
  return :: a                   -> m a
  fail   :: String              -> m a
```

---

In addition to implementing the class functions, all instances of Monad should obey the following equations, or Monad Laws:

---

```
return a >=>= k           = k a
m      >=>= return        = m
m      >=>= (\x -> k x >=>= h) = (m >=>= k) >=>= h
```

---

# Maybe Monad

The Haskell Maybe type is defined as:

---

```
data Maybe a = Just a | Nothing
  deriving (Eq, Ord)
```

---

It is an instance of the Monad and Functor classes.



# Maybe Monad Implementation

Let's see how the Monad operations are defined for the Maybe type:

---

```
return :: a -> Maybe a
```

```
return x = Just x
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
(>>=) Nothing _ = Nothing
```

```
(>>=) (Just x) f = f x
```

```
(>>) :: Maybe a -> Maybe b -> Maybe b
```

```
(>>) x y = x >>= (\_ -> y)
```

---

# Verifying the First Monad Laws for Maybe

The first law states that:

$$\text{return } a \gg= k = k \ a$$

Let's play the substitution game:

```
return a >>= k
Just a >>= k    ;; by definition of return
k a             ;; by definition of >>=
                ;; C.V.D.
```

# Verifying the Second Monad Laws for Maybe

The second law states that:

---

```
m >>= return = m
```

---

Let's play the substitution game:

```
Just x >>= return      ;; by definition of >>=
return x = Just x      ;; by definition of return
                        ;; C.V.D.
```

```
Nothing >>= return = Nothing ;; by definition of >>=
                        ;; C.V.D.
```

# Verifying the Third Monad Laws for Maybe

The third law states that:

---

$$m \gg= (\backslash x \rightarrow k \ x \gg= h) \quad = \quad (m \gg= k) \gg= h$$

---

Let's play the substitution game:

---

```
Just x  >>= (\y -> k y >>= h)  ;; by definition of >>=
k x >>= h                        ;; adding parenthesis
(k x) >>= h                      ;; by definition of >>=
(Just x >>= k) >>= h              ;; C.V.D.
```

---

Thus we have proved that the Maybe type satisfies the three key monads rule. There are types you are already familiar with that satisfy the monadic laws, such as lists.

# Monads in Scala

Scala does not define a `Monad` higher-order type, but monadic type (as well as some time that are not strictly monads), provide the key monadic operations.

These operations, however are named differently.

# Option Type in Scala

The **Option** type is Scala's equivalent to Haskell's *Maybe* Monad. Looking at it is a good way of learning about monadic computations in Scala.

The *Option* type is defined as follows:

---

```
sealed abstract class Option[+A] extends Product with Serializable

case object None extends Option[Nothing] {
  // ...
}

final case class Some[+A](value: A) extends Option[A] {
  // ...
}
```

---

# Option Type in Scala

The monad operations provided are the following:

---

```
// this is the equivalent of Haskell's bind (>>=)
def flatMap[B](f: A => Option[B]): Option[B]
```

---

Notice that the equivalent of *return* is the constructor. That said, Scala does not provide the sequencing operator (*>>*). But as we've seen that can be easily defined from *bind*.

The Option type is technically also a functor (as lists also are), thus it also provides the *map* operation.

---

```
// this is the equivalent of Haskell's sequence           (>>)
def map[B](f: A => B): B
```

---

# Monadic Computation with the Option Type

---

```
def parseInt(s: String): Option[Int] = {  
  try {  
    Some(s.toInt)  
  } catch {  
    case e: java.lang.NumberFormatException => None  
  }  
}  
  
val x = parseInt("10")  
val y = parseInt("10")  
  
val z = x.flatMap(a => y.flatMap(b => Some(a+b)))  
  
// Or using for  
  
for (a <- x; b <- y) yield (a + b)
```

---



# Parallel and Concurrent Computations

---

# Motivation

As the number of cores on processors are constantly increasing while the clock frequencies are essentially stagnating, exploiting the additional computational power requires to leverage parallel and concurrent computations.

Scala provides some elegant mechanism for supporting both parallel as well as concurrent computations.

# Parallel Collections

Scala supports the following parallel collections:

- `ParArray`
- `ParVector`
- `mutable.ParHashMap`
- `mutable.ParHashSet`
- `immutable.ParHashMap`
- `immutable.ParHashSet`
- `ParRange`
- `ParTrieMap`

# Using Parallel Collections

Using parallel collections is rather trivial. A parallel collection can be created from a regular one by using the `.par` property.

## Example

---

```
val parArray = (1 to 10000).toArray.par
parArray.fold(0)(_ + _)
```

```
val lastNames = List("Smith", "Jones", "Frankenstein", "Bach", "Jackson", "Rodin").par
lastNames.map(_.toUpperCase)
```

---

Given this *out-of-order* semantics, also must be careful to perform only associative operations in order to avoid non-determinism.

# Concurrent Computation on the JVM

The traditional approach to concurrent computations on the JVM is to leverage threads and the built-in support for monitors, through the *synchronized* keyword, to avoid race conditions.

This concurrency model is based on the assumption that different threads of execution collaborate by mutating some shared state. This shared state has to be protected to avoid race conditions.

This programming model is quite error-prone and is often source of bugs that are very hard to debug due to the inherent non-determinism of concurrent computations.

Furthermore, lock-based synchronization can lead to deadlocks.

# Scala futures

Scala provide a mechanism for concurrent execution that makes it easier to work under the share nothing model and that support more importantly supports composability.

This mechanism is provided by Scala's futures.

Futures are monads, thus, you can operate and compose them as any other monad.

# Futures Example

Below is an example of computing and composing futures.

---

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

val f1 = Future {
  Thread.sleep(2)
  10
}

val f2 = Future {
  Thread.sleep(1)
  20
}

val c = for (a <- f1; b <- f2) yield a + b
c.onComplete(r => r.foreach(a => println(a)))
```

---

# Promises

The most general way of creating a future and is through *promises*.

The completion of *future* create from a promise is controlled by completing the *promise*.

## Example

---

```
def asynchComputation(): Future[Int] = {  
  import Converters._  
  val promise = Promise[Int]  
  new Thread {  
    val r = someComputeIntenseFun()  
    promise.success(r)  
  }.start()  
  return promise.future  
}
```

---



# Homeworks

---

From the **Programming in Scala** book you should read:

- Chapter 10
- Chapter 11
- Chapter 15
- Chapter 19 (Variance concepts already covered in lecture 1 and 2)
- Chapter 23
- Chapter 32