@AndrzejWasowski
Andrzej Wąsowski

# Advanced Programming

## Algebraic Data Types

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

- Algebraic Data Types
- Functional lists and trees
- Fold functions
- Style guidelines for functional programming in ADPRO
- In the next episode ...

# AGENDA

# Algebraic Data Types (ADTs)

**Def. Algebraic Data Type**

A type generated by one or more constructors, each taking zero or more arguments.

The sets of objects generated by each constructor are **summed** (unioned), each constructor can be seen as a representation of a Cartesian **product** (tuple) of its arguments; thus the name **algebraic**.

**Example: immutable lists**

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head :A, tail :List[A]) extends List[A]
```

**sealed**: extensible in the same file only

`Nothing`: **subtype of any type**

**Example: operations on lists**

```scala
object List {
  def sum(ints :List[Int]) :Int =
    ints match { case Nil => 0
                 case Cons(x,xs) => x + sum(xs) }
  def apply[A](as :A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

**companion object** of `List[+A]`

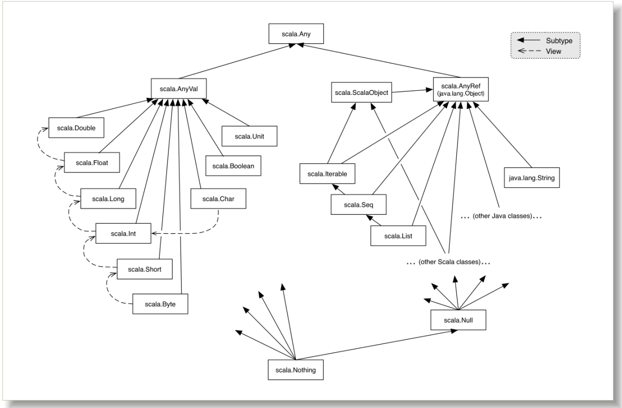
**pattern matching** uses case class constructors

**overload function application** for the object

**variadic function**

# Lists are covariant

## All share the same tail!




For any type `A` we have that

`Nil <: List[Nothing] <: List[A]`

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head :A, tail :List[A]) extends List[A]
```

# Another Poll: How is your recursion?

```scala
def f (a :List[Int]) :Int = a match {
  case Nil => 0
  case h::t => h + f(t)
}
```

What is the result of `f (List(42,-1,1,-1,1,-1)` ?

# Function Values

- In functional programing **functions are values**
- Functions can be **passed to other functions**, composed, etc.
- Functions operating on function values are **higher order** (HOFs)

```scala
def map (a :List[Int]) (f :Int => Int) :List[Int] =
  a match { case Nil      => Nil
            case h::tail => f(h)::map (tail) (f) }
```

A functional (pure) example

```scala
val mixed = List(-1, 2, -3, 4)
map (mixed) (abs _)
```

```scala
map (mixed) ((factorial _) compose (abs _))
```

see method abs as a function value

alternatively type it explicitly:

`(abs :Int => Int)`

An imperative (impure) example

```scala
val mixed = Array (-1, 2, -3, 4)
for (i <- 0 until mixed.length)
  mixed(i) = abs (mixed(i))
```

```scala
val mixed1 = Array (-1, 2, -3, 4)
for (i <- 0 until mixed1.length)
  mixed1(i) = factorial(abs(mixed1(i)))
```

# Parametric Polymorphism

**Monomorphic** functions operate on fixed types:

A monomorphic map in Scala

```
def map (a :List[Int]) (f :Int => Int) :List[Int] =
  a match { case Nil      => Nil
            case h::tail => f(h)::map (tail) (f) }
```

There is nothing specific here regarding Int.

A polymorphic map in Scala

```
def map[A,B] (a :List[A]) (f :A => B) :List[B] =
  a match { case Nil      => Nil
            case h::tail => f(h)::map (tail) (f) }
```

An example of use (type parameters are inferred):

```
map[Int,String] (mixed_list) { _.toString } compose
  (factorial _) compose (abs _))
```

- A **polymorphic** function operates on values of (m)any types (some restriction possible in Scala)
- A polymorphic **type constructor** defines a parameterized family of types
- Don't confuse with OO-polymorphism AKA "**dynamic dispatch**" (dependent on the inheritance hierarchy)

# HOFs in Scala Standard Library

## Methods of class `List[A]`, operate on `this` list, type `A` is bound in the class

`map[B](f: A =>B): List[B]`

Translates `this` list of `A`s into a list of `B`s using `f` to convert the values

`filter(p: A =>Boolean): List[A]`

Compute a sublist of `this` by selecting the elements satisfying the predicate `p`

`flatMap[B](f: A =>List[B]): List[B]` *type slightly simplified

Builds a new list by applying `f` to elements of `this`, concatenating results.

`take(n: Int): List[A]`

Selects first n elements.

`takeWhile(p: A =>Boolean): List[A]`

Takes longest prefix of elements that satisfy a predicate.

`forall(p: A =>Boolean): Boolean`

Tests whether a predicate holds for all elements of this sequence.

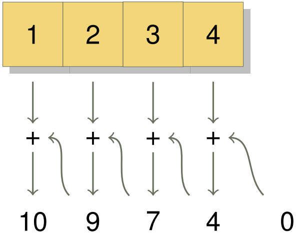
`exists(p: A =>Boolean): Boolean`

Tests whether a predicate holds for some of the elements of this sequence.

More at `http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List`

# [Right]Folding: Functional Loops

**Compute a sum of list's elements**




What characterizes similar computations?

- An **input list** `l = List(1,2,3,4)`
- An **initial value** `z = 0`
- A **binary operation** `f :Int => Int = _ + _`
- An **iteration algorithm** (folding)

```
def foldRight[A,B] (f : (A,B) => B) (z :B) (l :List[A]) :B =
  l match {
    case Cons(x,xs) => f(x, foldRight (f) (z) (xs))
    case Nil => z
  }
val l1 = List (1,2,3,4,5,6)
val sum     = foldRight[Int,Int] (_+_) (0) (l1)
val product = foldRight[Int,Int] (_*_) (1) (l1)
def map[A,B] (f :A=>B) (l: List[A])=
  foldRight[A,List[B]] ((x, z) => Cons(f(x),z)) (Nil) (l)
```

**Many HOFs can be implemented as special cases of folding**

# Preferred Programming Style in ADPRO

Always choose the best possible style for an exercise and your abilities

| Condemned (fail) → | Forgivable (medium grade*) → | Enlightened (top grade) |
|---|---|---|
| variables < | | < values |
| assignments < | | < value bindings |
| return statement < | | < expression value |
| Any/Object type < | | < parametric polymorphism |
| loops < | tail recursion* < simple recursion < folds* | < compose dedicated HOFs |
| | if conditions < pattern matching* | < use dedicated API |
| exceptions < | | < Option or Either monad |

* unless asked for explicitly, or really important for memory use.

# Scala: Summary

- **Basics** (objects, modules, functions, expressions, values, variables, operator overloading, infix methods, interpolated strings.)
- **Pure functions** (referential transparency, side effects)
- **Loops and recursion** (tail recursion)
- **Functions as values** (higher-order functions)
- **Parametric polymorphism** (monomorphic functions, dynamic and static dispatch)
- **Standard HOFs** in Scala's library
- **Anonymous functions** (currying, partial function application)
- **Traits** (fat interfaces, multiple inheritance, mixins)
- **Algebraic Data Types** (pattern matching, case classes)
- **Folding**

# In the next episode ...

- Variance of type parameters
- Basics of functional design: exceptions vs values, partial functions, the Option data type, exception oriented API of Option, for comprehensions, Either
- Experience your first computation in a Monad (but we will not call it so yet)
- The reading should be relatively easy, so you should really try it!