

Analyse et Conception Formelles

Lesson 5

Crash Course on Scala

Bibliography

- *Programming in Scala*, M. Odersky, L. Spoon, B. Venners. Artima. <http://www.artima.com/pins1ed/index.html>
- *An Overview of the Scala Programming Language*, M. Odersky & al. <http://www.scala-lang.org/docu/files/ScalaOverview.pdf>
- *Scala web site*. <http://www.scala-lang.org>

Acknowledgements

- Many thanks to J. Noyé and J. Richard-Foy for providing material, answering questions and for fruitful discussions.

Scala in a nutshell

- “Scalable language”: small scripts to architecture of systems
- Designed by Martin Odersky at EPFL
 - ▶ Programming language expert
 - ▶ One of the designers of the Java compiler
- Pure object model: *only objects and method calls* (\neq Java)
- With functional programming: higher-order, pattern-matching, ...
- Fully interoperable with Java (in both directions)
- Concise smart syntax (\neq Java)
- A compiler and a read-eval-print loop integrated into the IDE



Scala worksheets!!

Outline

- 1 Basics
 - Base types and type inference
 - Control : if and match - case
 - Loops (for) and structures: Lists, Tuples, Maps
- 2 Functions
 - Basic functions
 - Anonymous, Higher order functions and Partial application
- 3 Object Model
 - Class definition and constructors
 - Method/operator/function definition, overriding and implicit defs
 - Traits and polymorphism
 - Singleton Objects
 - Case classes and pattern-matching
- 4 Interactions with Java
 - Interoperability between Java and Scala
- 5 Isabelle/HOL export in Scala

Outline

- 1 Basics
 - Base types and type inference
 - Control : if and match - case
 - Loops (for) and structures: Lists, Tuples, Maps
- 2 Functions
 - Basic functions
 - Anonymous, Higher order functions and Partial application
- 3 Object Model
 - Class definition and constructors
 - Method/operator/function definition, overriding and implicit defs
 - Traits and polymorphism
 - Singleton Objects
 - Case classes and pattern-matching
- 4 Interactions with Java
 - Interoperability between Java and Scala
- 5 Isabelle/HOL export in Scala

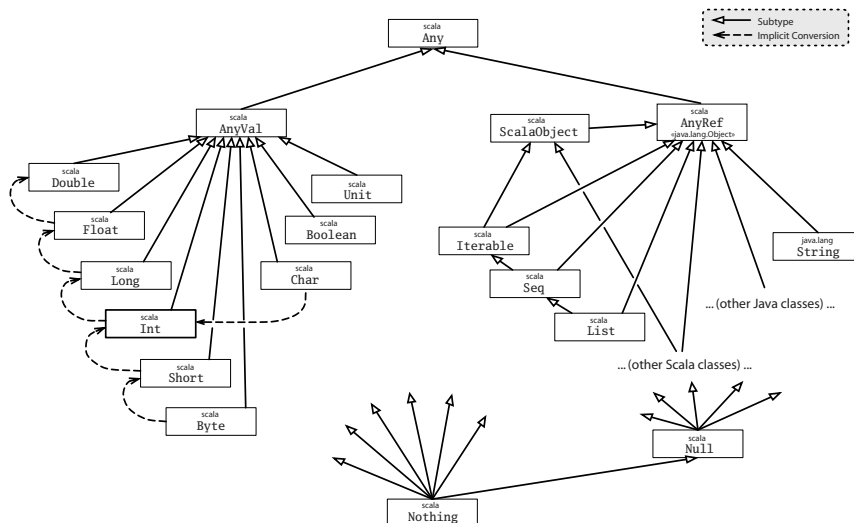
Base types and type annotations

- `1: Int, "toto": String, 'a': Char, (): Unit`
- Every data is an object, including base types!
e.g. `1` is an object and `Int` is its class
- Every access/operation on an object is a method call!
e.g. `1 + 2` executes: `1.+(2)` (`o.x(y)` is equivalent to `o x y`)

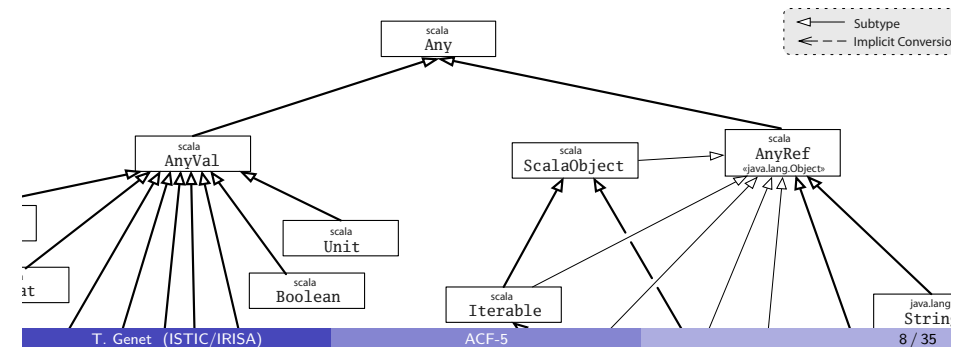
Exercise 1

Use the `max(Int)` method of class `Int` to compute the maximum of `1+2` and `4`.

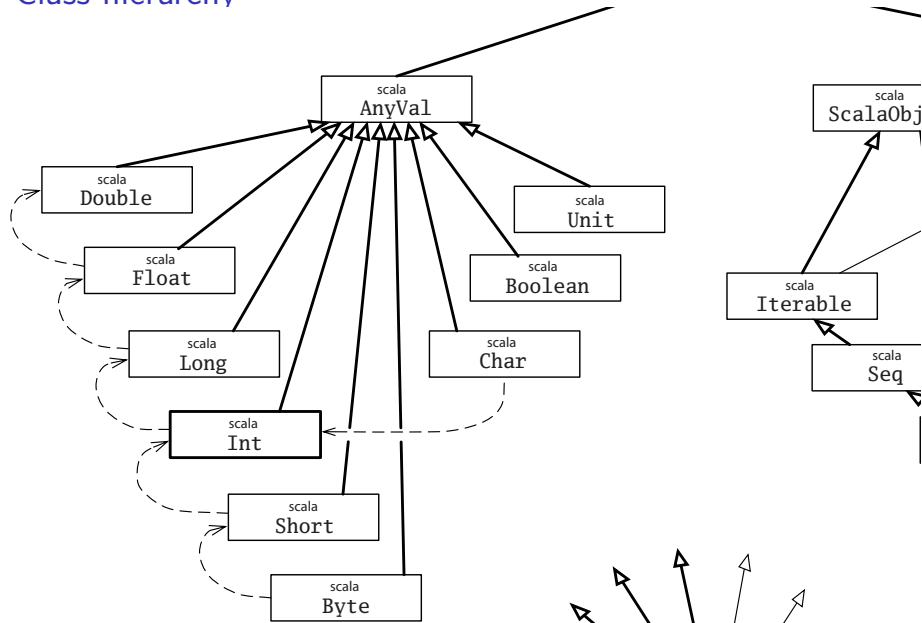
Class hierarchy



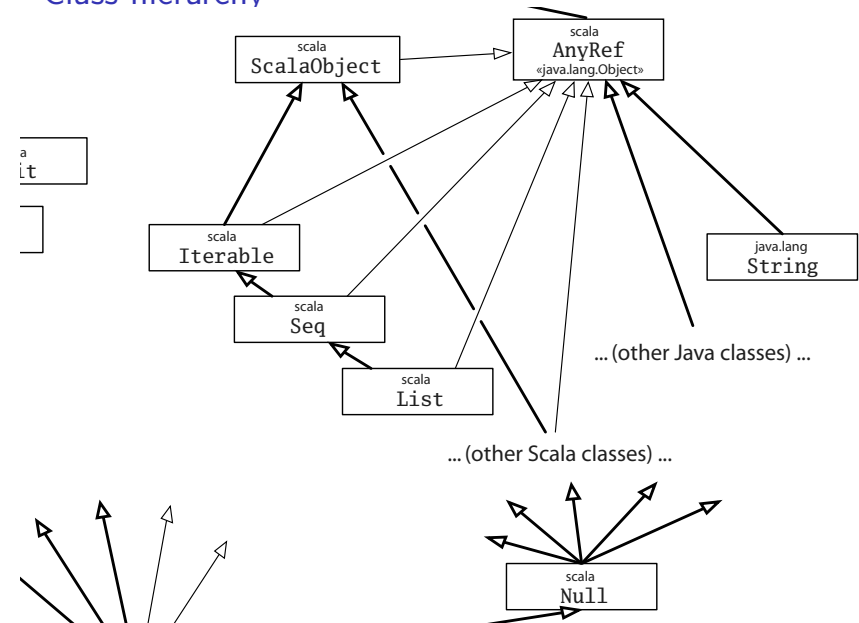
Class hierarchy



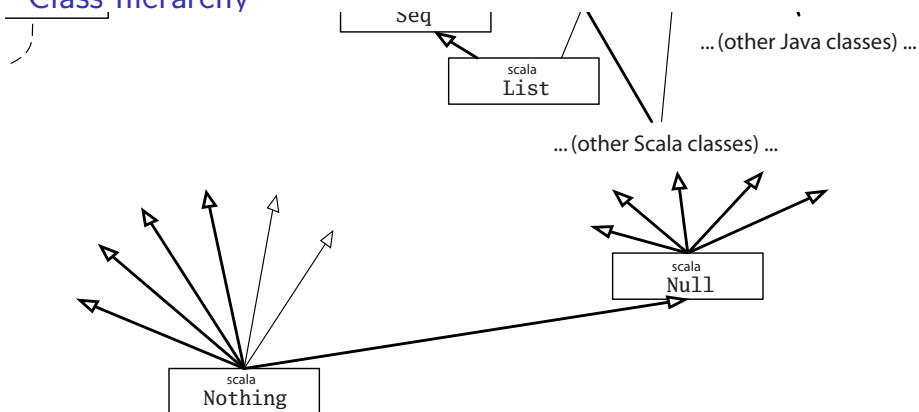
Class hierarchy



Class hierarchy



Class hierarchy



val and var

- `val` associates an object to an identifier and *cannot* be reassigned
- `var` associates an object to an identifier and *can* be reassigned
- Scala philosophy is to use `val` instead of `var` whenever possible
- Types are (generally) automatically inferred

```
scala> val x=1 // or val x:Int = 1
x: Int = 1
```

```
scala> x=2
<console>:8: error: reassignment to val
      x=2
      ^
```

```
scala> var y=1
y: Int = 1
```

```
scala> y=2
y: Int = 2
```

if expressions

- Syntax is similar to Java **if statements** ...
but that they are not **statements** but **typed expressions**
- `if (condition) e1 else e2`
Remark: the type of this expression is the supertype of e1 and e2
- `if (condition) e1 // else ()`
Remark: the type of this expression is the supertype of e1 and `Unit`

Exercise 2

What are the results and types of the corresponding `if` expressions:

- `if (1==2) 1 else 2`
- `if (1==2) 1 else "toto"`
- `if (1==1) 1`
- `if (1==1) println(1)`

match - case expressions

- Replaces (and **extends**) the usual switch - case construction
- The syntax is the following:

```
e match {
  case pattern1 => r1    //patterns can be constants
  case pattern2 => r2    //or terms with variables
  ...                    //or terms with holes: '_'
  case _ => rn
}
```
- Remark: the type of this expression is the supertype of r1, r2, ... rn

Example 1 (Match-case expressions)

```
x match {
  case "bonjour" => "hello"
  case "au revoir" => "goodbye"
  case _ => "don't know"
}
```

(Immutable) Lists: `List[A]`

- List definition (with type inference)
`val l= List(1,2,3,4,5)`
- Adding an element to the head of a list
`val l1= 0::l`
- Adding an element to the queue of a list
`val l2= l1:+6`
- Concatenating lists
`val l3= l1++l2`
- Getting the element at a given position
`val x= l2(2)`
- Doing pattern-matching over lists

```
l2 match {
  case Nil => 0
  case e::_ => e
}
```

for loops

- `for (ident <- s) e`
Remark: s has to be a subtype of `Traversable`
(Arrays, Collections, Tables, Lists, Sets, Ranges, ...)
- Usual for-loops can be built using `.to(...)`
`"(1).to(5)"` \equiv `"1 to 5"` results in `Range(1, 2, 3, 4, 5)`

Exercise 3

Given `val lb=List(1,2,3,4,5)` and using `for`, build the list of squares of `lb`.

Exercise 4

Using `for` and `println` build a usual 10×10 multiplication table.

(Immutable) Tuples : (A,B,C,...)

- Tuple definition (with type inference)
`scala> val t= (1,"toto",18.3)`
`t: (Int, String, Double) = (1,toto,18.3)`
- Tuple getters: `t._1`, `t._2`, etc.
- ... or with `match - case`:

```
t match { case (2,"toto",-) => "found!"
          case (_,x,-) => x
        }
```

The above expression evaluates in "toto"

(Immutable) maps : Map[A,B]

- Map definition (with type inference)
`val m= Map('C' -> "Carbon", 'H' -> "Hydrogen")`
Remark: inferred type of `m` is `Map[Char,String]`
- Finding the element associated to a key in a map, with default value
`m.getOrElse('K', "Unknown")`
- Adding an association in a map
`val m1= m+('O' -> "Oxygen")`
- A `Map[A,B]` can be traversed (using `for`) as a `Collection` of pairs of type `Tuple[A,B]`, e.g. `for((k,v) <- m) { ... }`

Exercise 5

Print all the keys of map `m1`

Outline

- 1 Basics
 - Base types and type inference
 - Control : if and match - case
 - Loops (for) and structures: Lists, Tuples, Maps
- 2 Functions
 - Basic functions
 - Anonymous, Higher order functions and Partial application
- 3 Object Model
 - Class definition and constructors
 - Method/operator/function definition, overriding and implicit defs
 - Traits and polymorphism
 - Singleton Objects
 - Case classes and pattern-matching
- 4 Interactions with Java
 - Interoperability between Java and Scala
- 5 Isabelle/HOL export in Scala

Basic functions

- `def f (arg1: Type1, ..., argn: Typen): Typef = { e }`
Remark 1: type of `e` (the type of the last expression of `e`) is `Typef`
Remark 2: `Typef` can be inferred for *non recursive functions*
Remark 3: The type of `f` is : `(Type1,...,Typen) Typef`

Example 2

```
def plus(x:Int,y:Int):Int={
  println("Sum of "+x+" and "+y+" is equal to "+(x+y))
  x+y // no return keyword
} // the result of the function is the last expression
```

Exercise 6

Using a map, define a phone book and the functions
`addName(name:String,tel:String)`, `getTel(name:String):String`,
`getUserList>List[String]` and `getTelList>List[String]`.

Anonymous functions and Higher-order functions

- The anonymous Scala function adding one to x is:
`((x:Int) => x + 1)`
Remark: it is written $(\lambda x. x + 1)$ in Isabelle/HOL
- A higher order function takes a function as a parameter
e.g. method/function `map` called on a `List[A]` takes a function $(A \Rightarrow B)$ and results in a `List[B]`

```
scala> val l=List(1,2,3)
l: List[Int] = List(1, 2, 3)

scala> l.map ((x:Int) => x+1)
res1: List[Int] = List(2, 3, 4)
```

Exercise 7

Using `map` and the `capitalize` method of the class `String`, define the `capUserList` function returning the list of capitalized user names.

Partial application

- The `'_'` symbol permits to *partially* apply a function
e.g. `getTel(_)` returns the function associated to `getTel`

Example 3 (Other examples of partial application)

```
(_:String).size    (_:Int) + (_:Int)    (_:String) == "toto"
```

Exercise 8

Using `map` and partial application on `capitalize`, redefine the function `capUserList`.

Exercise 9

Using the higher order function `filter` on Lists, define a function `above(n:String):List(String)` returning the list of users having a capitalized name greater to name `n`.

Outline

- 1 Basics
 - Base types and type inference
 - Control : if and match - case
 - Loops (for) and structures: Lists, Tuples, Maps
- 2 Functions
 - Basic functions
 - Anonymous, Higher order functions and Partial application
- 3 Object Model
 - Class definition and constructors
 - Method/operator/function definition, overriding and implicit defs
 - Traits and polymorphism
 - Singleton Objects
 - Case classes and pattern-matching
- 4 Interactions with Java
 - Interoperability between Java and Scala
- 5 Isabelle/HOL export in Scala

Class definition and constructors

- `class C(v1: type1, ..., vn:typen) { ... }`
the primary constructor
- e.g.

```
class Rational(n:Int,d:Int){
  val num=n          // can use var instead
  val den=d          // to have mutable objects
  def isNull:Boolean=(num==0)
}
```

- Objects instances can be created using `new`:
`val r1= new Rational(3,2)`

Exercise 10

Complete the `Rational` class with an `add(r:Rational):Rational` function.

Overriding, operator definitions and implicit conversions

- Overriding is explicit: `override def f(...)`

Exercise 11

Redefine the `toString` method of the `Rational` class.

- All operators `'+'`, `'*'`, `'=='`, `'>'`, ... can be used as function names
e.g. `def +(x:Int):Int= ...`

Remark: when *using* the operator recall that `x.+(y) ≡ x + y`

Exercise 12

Define the `'+'` and `'*'` operators for the class `Rational`.

- It is possible to define `implicit` (automatic) conversions between types
e.g. `implicit def bool2int(b:Boolean):Int= if b 1 else 0`

Exercise 13

Add an implicit conversion from `Int` to `Rational`.

Traits

- Traits stands for interfaces (as in Java)

```
trait IntQueue {  
  def get:Int  
  def put(x:Int)  
}
```

- The keyword `extends` defines trait implementation

```
class MyIntQueue extends IntQueue {  
  private var b= List[Int]()  
  def get= {val h=b(0); b=b.drop(1); h}  
  def put(x:Int)= {b=b:+x}  
}
```

Singleton objects

- Singleton objects are defined using the keyword `object`

```
trait IntQueue {  
  def get:Int  
  def put(x:Int)  
}
```

```
object InfiniteQueueOfOne extends IntQueue {  
  def get=1  
  def put(x:Int)={}  
}
```

- A singleton object does not need to be “created” by `new`

```
InfiniteQueueOfOne.put(10)  
InfiniteQueueOfOne.put(15)  
val x=InfiniteQueueOfOne.get
```

Type abstraction and Polymorphism

Parameterized function/class/trait can be defined using type parameters

```
trait Queue[T] { // more generic than IntQueue  
  def get:T  
  def push(x:T)  
}
```

```
class MyQueue[T] extends Queue[T] {  
  protected var b= List[T]()  
  
  def get={val h=b(0); b=b.drop(1); h}  
  def put(x:T)= {b=b:+x}  
}
```

```
def first[T1,T2](pair:(T1,T2)):T1=  
  pair match case (x,y) => x
```

Case classes

- Case classes provide a natural way to encode Algebraic Data Types
e.g. binary expressions built over rationals: $\frac{18}{27} + -(\frac{1}{2})$

```
trait Expr
case class BinExpr(o:String,l:Expr,r:Expr) extends Expr
case class Constant(r:Rational) extends Expr
case class Inv(e:Expr) extends Expr
```

- Instances of case classes are built without `new`
e.g. the object corresponding to $\frac{18}{27} + -(\frac{1}{2})$ is built using:
`BinExpr("+",Constant(new Rational(18,27)),
 Inv(Constant(new Rational(1,2))))`

Case classes and pattern-matching

```
trait Expr
case class BinExpr(o:String,l:Expr,r:Expr) extends Expr
case class Constant(r:Rational) extends Expr
case class Inv(e:Expr) extends Expr
```

- `match case` can directly inspect objects built with case classes

```
def getOperator(e:Expr):String= {
  e match {
    case BinExpr(o,_,_) => o
    case _ => "No operator"
  }
}
```

Exercise 14

Define an `eval(e:Expr):Rational` function computing the value of any expression.

Outline

- 1 Basics
 - Base types and type inference
 - Control : if and match - case
 - Loops (for) and structures: Lists, Tuples, Maps
- 2 Functions
 - Basic functions
 - Anonymous, Higher order functions and Partial application
- 3 Object Model
 - Class definition and constructors
 - Method/operator/function definition, overriding and implicit defs
 - Traits and polymorphism
 - Singleton Objects
 - Case classes and pattern-matching
- 4 Interactions with Java
 - Interoperability between Java and Scala
- 5 Isabelle/HOL export in Scala

Interoperability between Java and Scala

- In Scala, it is possible to build objects from Java classes
e.g. `val txt:JTextArea=new JTextArea("")`
- And to define scala classes/objects implementing Java interfaces
e.g. `object Window extends JFrame`
- There exists conversions between Java and Scala data structures
`import scala.collection.JavaConverters._`

```
val l1:java.util.List[Int]= new java.util.ArrayList[Int]()
l1.add(1); l1.add(2); l1.add(3) // l1: java.util.List[Int]

val sb1= l1.asScala.toList // s1: List[Int]
val s1= sb1.asJava // s1: java.util.List[Int]
}
```

- Remark: it is also possible to use Scala classes and Object into Java

Outline

- 1 Basics
 - Base types and type inference
 - Control : if and match - case
 - Loops (for) and structures: Lists, Tuples, Maps
- 2 Functions
 - Basic functions
 - Anonymous, Higher order functions and Partial application
- 3 Object Model
 - Class definition and constructors
 - Method/operator/function definition, overriding and implicit defs
 - Traits and polymorphism
 - Singleton Objects
 - Case classes and pattern-matching
- 4 Interactions with Java
 - Interoperability between Java and Scala
- 5 Isabelle/HOL export in Scala

Isabelle/HOL exports Scala case classes and functions...

```
theory tp
[...]
```

```
datatype 'a tree= Leaf | Node "'a * 'a tree * 'a tree"
fun member:: "'a => 'a tree => bool"
where
"member _ Leaf = False" |
"member x (Node(y,l,r)) = (if x=y then True else ((member x l)
                                                    ∨ (member x r)))"
```

_____to Scala_____

```
object tp {
  abstract sealed class tree[+A] // similar to traits
  case object Leaf extends tree[Nothing]
  case class Node[A](a: (A, (tree[A], tree[A]))) extends tree[A]
  def member[A : HOL.equal](uu: A, x1: tree[A]): Boolean =
    (uu, x1) match {
      case (uu, Leaf) => false
      case (x, Node((y, (l, r)))) => (if (HOL.eq[A](x, y)) true
                                     else member[A](x, l) || member[A](x, r))
    }
}
```

... and some more cryptic code for Isabelle/HOL equality

```
object HOL {
  trait equal[A] {
    val 'HOL.equal': (A, A) => Boolean
  }

  def equal[A](a: A, b: A)(implicit A: equal[A]): Boolean =
    A.'HOL.equal'(a, b)

  def eq[A : equal](a: A, b: A): Boolean = equal[A](a, b)
}
```

To link Isabelle/HOL code and Scala code, it can be necessary to add:
(Demo tp.thy + CM5Integ)

```
implicit def equal_t[T]: HOL.equal[T] = new HOL.equal[T] {
  val 'HOL.equal' = (a: T, b: T) => a==b
}
```

Which defines `HOL.equal[T]` for all types `T` as the Scala equality `==`