Scala Overview

where objects and functions meet.

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Scala: Scalable Language
History & Motivations

The design of Scala starts in 2001 at École Polytechnique Fédérale (EPFL) of Lausanne by Martin Odersky

- the first working release is out at the end of 2003;
- last stable release is 2.13.4 (Nov. 2020).

It runs on the JVM and interoperates with the Java libraries.

Scalable language

- succinct, elegant and flexible syntax (50%-75% of code reduction);
- interactive interpreter and
- support for embedded domain specific languages

Scala merges object-oriented and functional programming.

Scala is statically typed, it supports

- abstract and path-dependent types;
- generic classes and polymorphic methods;
- (a limited form of) type inference.

Scala: Scalable Language
My First Scala Program: A Special Form of HelloWorld

```
class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}
val up = new Upper
Console.println(up.upper("A", "First", "Scala", "Program"))
```

- parametric types
- (anonymous) functions are first order citizens

Interpreted as a script

```
[15:38]cazzola@surtur:~/lp/scala>scala upper.scala
ArrayBuffer(A, FIRST, SCALA, PROGRAM)
```

Or into an interactive section

```
[15:39]cazzola@surtur:~/lp/scala>scala
Welcome to Scala version 2.13.2 (OpenJDK 64-Bit Server VM, Java 11.0.9).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load upper.scala
Loading upper.scala...
defined class Upper
up: Upper = Upper@6d69c9a2
ArrayBuffer(A, FIRST, SCALA, PROGRAM)
```

Scala: Scalable Language
My First Scala Program: A Special Form of HelloWorld (Cont'd)

```
object Upper {
  def upper(strings: String*) = strings.map(_.toUpperCase())
}
println(Upper.upper("A", "First", "Scala", "Program"))
```

- the keyword `object` introduces a class with a single instance;
- don't exist static methods but methods of singleton objects;
- `_` as a wildcard.

```
[15:39]cazzola@surtur:~/lp/scala>scala
Welcome to Scala version 2.13.2 (OpenJDK 64-Bit Server VM, Java 11.0.9).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load upper.scala
Loading upper2.scala...
defined module Upper
ArrayBuffer(A, FIRST, SCALA, PROGRAM)
```

Scala: Scalable Language
My First Scala Program: A Special Form of HelloWorld (Cont'd)


```
object Upper {
  def main(args: Array[String]) = {
    args.map(_.toUpperCase()).foreach(printf("%s ",_))
    println("")
  }
}
```

- main as a method of a singleton object;
- two independent uses of the _ wildcard.

Compiled to Bytecode

```
[16:19]cazzola@surtur:~/lp/scala>scalac upper3.scala
[16:20]cazzola@surtur:~/lp/scala>ls
Upper$.class  upper3.scala  Upper.class
[16:20]cazzola@surtur:~/lp/scala>scala Upper hello world\!\!\!
HELLO WORLD!!!
```

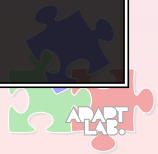
Note

- to use scalac the code to compile has to be legit scala code, i.e., all the code should be in a class or object definition
- this constraint is not enforced scala

Scala: Scalable Language
Types

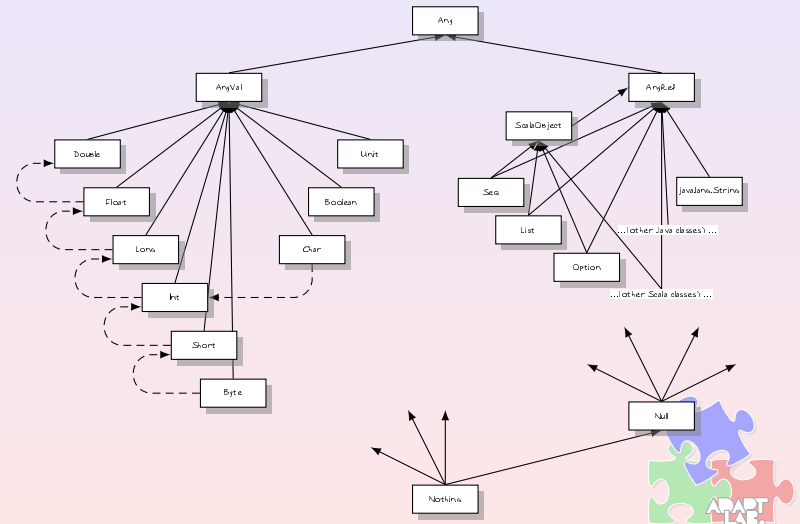

```
class Rational(n: Int, d: Int) extends AnyRef {
  val num = n
  val den = d
  def this(n: Int) = this(n,1)
  def + (that: Rational): Rational =
    new Rational(num*that.den + that.num*den, den*that.den)
  def + (i: Int): Rational = new Rational(num+i*den, den)
  override def toString = "" + num + "/" + den
}
```

```
[16:38]cazzola@surtur:~/lp/scala>scala
scala> :load rational.scala
Loading rational.scala...
defined class Rational

scala> val r1 = new Rational(1)
r1: Rational = 1/1

scala> val r2 = new Rational(2,3)
r2: Rational = 2/3

scala> r1+r2
res2: Rational = 5/3

scala> r1.+(r2)
res3: Rational = 5/3
```



Scala: Scalable Language
Types

Type Hierarchy



Scala: Scalable Language
Types

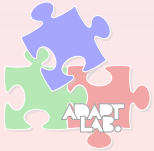
`Any` is the root of the whole hierarchy.

- `AnyRef` is the root for the reference classes (both Java and Scala classes) and coincides with `Object`;
- `AnyVal` is the root for all the basic types.

Two different "empty" values

- `Null` for all the reference types and it is instatiates by **`null`**;
- `Nothing` for all types and it can't be instantiated.

It can be used to define `Empty` as `List[Nothing]` for any `List[T]`.



Scala: Scalable Language
Pure Object-Oriented Paradigm

As in Smalltalk:

- everything is an object and any operation is a method

```
scala> 1.+(2)
res0: Double = 3.0
scala> 3.14.+(res0)
res3: Double = 6.140000000000001
```

Identifiers

- alfanumerics strings on a given set of characters
- `e_1 id e_2` is the short for `e_1.id(e_2)`

Immutable/mutable variables.

```
scala> val array: Array[String] = new Array(3)
array: Array[String] = Array(null, null, null)

scala> array = new Array(2)
<console>:6: error: reassignment to val
       array = new Array(2)

scala> array(0) = "Hello"
scala> array
res7: Array[String] = Array(Hello, null, null)

scala> var price: Double = 100
price: Double = 100.0
scala> price += price*.20
scala> price

res9: Double = 120.0
```


Scala: Scalable Language
Case Classes

```
abstract class Bool {
  def and(b: => Bool): Bool
  def or(b: => Bool): Bool
}
case object True extends Bool {
  def and(b: => Bool) = b
  def or(b: => Bool) = this
}
case object False extends Bool {
  def and(b: => Bool) = this
  def or(b: => Bool) = b
}
def bottom: () => Nothing = () => bottom()
```

```
scala> :load short-circuit.scala
Loading short-circuit.scala...
defined class Bool
defined module True
defined module False
bottom: () => Nothing

scala> True and bottom()
java.lang.StackOverflowError
scala> True or bottom()
res4: object True = True
```

bottom in questo caso non viene mai evaluato

Scala: Scalable Language
Option: None and Some instead of Null

Options are used to smoothly integrate functions and objects

```
val RegionCapitals = Map(
  "Val d'Aosta" -> "Aosta", "Piemonte" -> "Torino", "Liguria" -> "Genova",
  "Lombardia" -> "Milano", "Emilia Romagna" -> "Bologna" // ...
)

println( "Get the capital cities wrapped in Options:" )
println( "Liguria: " + RegionCapitals.get("Liguria") )
println( "Lombardia: " + RegionCapitals.get("Lombardia") )
println( "Padania: " + RegionCapitals.get("Padania") + "\n")
println( "Get the capital cities themselves out of the Options:" )
println( "Liguria: " + RegionCapitals.get("Liguria").get )
println( "Lombardia: " + RegionCapitals.get("Lombardia").getOrElse("Oops!") )
println( "Padania: " + RegionCapitals.get("Padania").getOrElse("Oops2!") )
```

```
[11:19]cazzola@surtur:~/lp/scala>scala option.scala
Get the capital cities wrapped in Options:
Liguria: Some(Genova)
Lombardia: Some(Milano)
Padania: None

Get the capital cities themselves out of the Options:
Liguria: Genova
Lombardia: Milano
Padania: Oops2!
```

```
def get[A,B](key: A): Option[B] = {
  if (contains(key)) new Some(getValue(key))
  else None
}
```


Scala: Scalable Language
Functions and Methods

Methods & functions

- functions are high-order;
- (parametric) polymorphism limited to methods

They look similar but are not

```
scala> val succfun = (x:Int) => x+1
succfun: Int => Int = $Lambda$1029/0x00000008405cb040@1bd8afc8

scala> def succmeth(x: Int) = x+1
succmeth: (x: Int)Int
```

- functions are values of a particular class with method `apply`
- they are similarly called: `succfun(2)` and `succmeth(2)` but the first is the short for `succfun.apply(2)`

Parametric polymorphism for methods

```
scala> def id[T](x:T) = x
id: [T](x: T)T

scala> id(3)
res7: Int = 3

scala> id("ciao")
res8: java.lang.String = ciao
```

Scala: Scalable Language
Comprehensions and Generators

Comprehensions are a mechanism
- to traverse a set of something;
- to "comprehend" what we find and
- computing something new from it

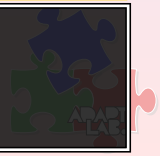
```
def sum_evens = (L:List[Int]) => {var sum=0; for (X <- L if X%2 == 0) sum += X; sum}
```

```
scala> :load sumevens.scala
sum_evens: (List[Int]) => Int

scala> sum_evens(List.range(1,1000))
res5: Int = 249500
```

Yielding
- to get a new collection from a comprehension

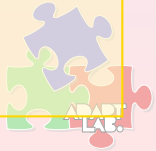
```
val is_prime = (X:Int) => {
  val divisors = (X:Int) =>
     for { Y <- List.range(2,math.sqrt(X).toInt) if (X % Y == 0)} yield Y
  divisors(X).length == 0
}
```

```
scala> :load is_prime.scala
is_prime: (Int) => Boolean = <function1>

scala> is_prime(100)
res0: Boolean = false

scala> is_prime(7)
res1: Boolean = true
```



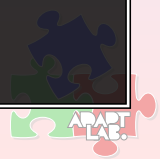
Scala: Scalable Language
Some (Known) Functions

```
def map[A,B](f: A=>B, list: List[A]): List[B] =
  list match {
    case Nil => Nil
    case hd::tl => f(hd)::map[A,B](f,tl)
}
def reduce[T](f:(T,T)=>T, list:List[T]):T = {
  def reduce2(acc:T, list:List[T]):T =
    list match {
      case Nil => acc
      case hd::tl => reduce2(f(acc,hd), tl)
    }
  reduce2(list.head, list.tail)
}
def exists[T](p: T=>Boolean, list:List[T]):Boolean = {
  var exists = false; for (elem <- list if p(elem)) exists = true; exists
}
def forall[T](p: T=>Boolean, list:List[T]):Boolean =
  reduce( (X:Boolean,Y:Boolean)=>X&&Y, map(p, list) )
def quicksort[T](lt: (T,T) => Boolean, list:List[T]): List[T] = {
  list match {
    case Nil => Nil
    case pivot::tl =>
      val (p1, p2) = tl.partition( (X:T) => lt(X, pivot) )
      quicksort(lt, p1) ::: (pivot::Nil) ::: quicksort(lt, p2)
  }
}
```



Scala: Scalable Language
Some (Known) Functions

```
scala> :load mylists.scala
map: [A,B](f: (A) => B,list: List[A])List[B]
reduce: [T](f: (T, T) => T,list: List[T])T
exists: [T](p: (T) => Boolean,list: List[T])Boolean
forall: [T](p: (T) => Boolean,list: List[T])Boolean
quicksort: [T](lt: (T, T) => Boolean,list: List[T])List[T]

scala> val is_even = (X:Int) => X%2==0
is_even: (Int) => Boolean = <function1>

scala> map( (X:Int) => math.sqrt(X), List.range(1,5))
res10: List[Double] = List(1.0, 1.4142135623730951, 1.7320508075688772, 2.0)

scala> exists(is_even, List.range(1,10))
res30: Boolean = true
scala> exists(is_even, List.range(1,10,2))
res31: Boolean = false

scala> reduce((X:Int,Y:Int)=>X+Y,List.range(1,1000))
res26: Int = 499500

scala> forall(is_even, List.range(1,10))
res33: Boolean = false
scala> forall(is_even, List.range(1,10,2))
res34: Boolean = false

scala> quicksort((X:Int,Y:Int) => X>Y, 1::2 :: 7 :: 25 :: 0 :: -3 ::Nil )
res40: List[Int] = List(25, 7, 2, 1, 0, -3)
scala> quicksort((X:Int,Y:Int) => X<Y, 1::2 :: 7 :: 25 :: 0 :: -3 :: Nil )
res41: List[Int] = List(-3, 0, 1, 2, 7, 25)
```



References

- Martin Odersky and Matthias Zenger.
  Scalable Component Abstractions.
  In Proceedings of OOPSLA'05, pages 41–57, San Diego, CA, USA, October 2005. ACM Press.
- Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
  Traits: Composable Units of Behaviour.
  In Proceedings of the ECOOP'03, LNCS 2743, pages 248–274, Darmstadt, Germany, July 2003. Springer.
- Venkat Subramaniam.
  Programming Scala.
  The Pragmatic Bookshelf, June 2009.
- Dean Wampler and Alex Payne.
  Programming Scala.
  O'Reilly, September 2009.