Functional programming with Scala

Basics

Djaouida ZAOUCHE, Romain DUJOL

CY TECH – ING2

2020 – 2021

Introduction

In the beginning. . .

[...] Functional code is characterised by one thing: the absence of side-effects. It doesn't rely on data outside the current function, and it doesn't change data that exists outside the current function. Every other " functional " thing can be derived from this property. Use it as a guide rope as you learn.

[...] La programmation fonctionnelle se caractérise par une chose : l'absence d'effet de bord. Elle ne s'appuie pas sur des données hors de la fonction courante et elle ne change pas de donnée existant hors de la fonction courante. Toute autre caractéristique «fonctionnelle » peut être déduite de cette propriété. Utilisez cela comme fil conducteur lors de votre apprentissage.

Mary Rose COOK, *A practical introduction to functional programming*

Functional paradigm $\equiv$ no side-effect

Functional paradigm $\equiv$ no side-effect

Variable

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

Instruction

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Only interest of an instruction : the side-effect it produces !

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Only interest of an instruction : the side-effect it produces !

Procedure

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Only interest of an instruction : the side-effect it produces !

~~Procedure~~ $\longrightarrow$ **Pure function**

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Only interest of an instruction : the side-effect it produces !

~~Procedure~~ $\longrightarrow$ **Pure function**

Only interest of a procedure : the side-effect it produces !

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Only interest of an instruction : the side-effect it produces !

~~Procedure~~ $\longrightarrow$ **Pure function**

Only interest of a procedure : the side-effect it produces !

Loop

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Only interest of an instruction : the side-effect it produces !

~~Procedure~~ $\longrightarrow$ **Pure function**

Only interest of a procedure : the side-effect it produces !

~~Loop~~ $\longrightarrow$ **Recursivity**

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Only interest of an instruction : the side-effect it produces !

~~Procedure~~ $\longrightarrow$ **Pure function**

Only interest of a procedure : the side-effect it produces !

~~Loop~~ $\longrightarrow$ **Recursivity**

Loop $\equiv$ Instructions with some variable modification

Functional paradigm $\equiv$ no side-effect

~~Variable~~ $\longrightarrow$ **Constant**

Variable modification IS a side-effect

~~Instruction~~ $\longrightarrow$ (Typed) **expression**

Only interest of an instruction : the side-effect it produces !

~~Procedure~~ $\longrightarrow$ **Pure function**

Only interest of a procedure : the side-effect it produces !

~~Loop~~ $\longrightarrow$ **Recursivity**

Loop $\equiv$ Instructions with some variable modification

+ Any function is a (typed) **expression**.

Introduction

History

Functional programming languages

History

- 1958 : LISP (*LISt Processor*)
  - 1970 : Scheme
- 1973 : ML (*Meta Language*)
  - 1985 : CAML
    - 1996 : OCAML
  - 1990 : Standard ML
- 1990 : Haskell
- 2003-2004 : **Scala**
- 2005 : F# (based on C#)

Imperative programming languages

Historique

- 1972 : C
- 1983 : Turbo Pascal
- 1985 : C++ (object oriented)
  - **2011 : C++ includes anonymous functions**
- 1995 : Java (object oriented)
  - **2014 : Java 8 includes anonymous functions**
- 2000 : C# (object oriented)
- 2011 : Kotlin (object oriented with functional elements)

Other languages start to give it in functional features

- Anmonymous functions $\equiv$ functions defined on-the-fly
- Functions as parameters $\Longleftrightarrow$ Processings as parameters
  $\Rightarrow$ Code factorization improved

Less elements so less expressive. . . isn't it ?

Less elements so less expressive. . . isn't it ?

Well, not at all !

- 1930-1936, Alonzo CHURCH : λ-calculus
  Representation of functional programming languages
- 1936-1937, Alan TURING : machines de TURING
  Representation of current computers
- 1936-1937, Alan TURING :

  Both representations are equivalent.

Less elements so less expressive... isn't it ?

Well, not at all !

- 1930-1936, Alonzo CHURCH : λ-calculus
  Representation of functional programming languages
- 1936-1937, Alan TURING : machines de TURING
  Representation of current computers
- 1936-1937, Alan TURING :
  Both representations are equivalent.

Advantages

- less elements $\Rightarrow$ simpler, more intuitive
- easier to analyse, simpler unit tests
- closer to mathematical description

Less elements so less expressive... isn't it ?

Well, not at all !

- 1930-1936, Alonzo CHURCH : λ-calculus
  Representation of functional programming languages
- 1936-1937, Alan TURING : machines de TURING
  Representation of current computers
- 1936-1937, Alan TURING :

  Both representations are equivalent.

Advantages

- less elements $\Rightarrow$ simpler, more intuitive
- easier to analyse, simpler unit tests
- closer to mathematical description

Drawbacks

- needs more resources (in progress)
- limit of applicability (I/O)

And you can make big money !

What languages are associated with the highest salaries worldwide?

1. **Scala** : 76 000 $ (Global), 150 000 $ (USA)
2. ...

Source : *StackOverflow* 2020 Developer Survey
(Global median over all respondants)

And you can make big money !

What languages are associated with the highest salaries worldwide?

1. **Scala** : 76 000 $ (Global), 150 000 $ (USA)
2. ...

Source : *StackOverflow* 2020 Developer Survey
(Global median over all respondants)

Why such a renewed interest ?

Classical progamming languages (C++, Java, C#) make it hard to address new needs in applications (cf. *Reactive Manifesto*) :

- highly availability and scalability ;
- *concurrent* and distributed applications ;
- interfaces for both end-users and other applications.

Introduction

Recap

Pure function

Pure function $\equiv$ function with no side-effect

$\Rightarrow$ **input only** parameters

$\Rightarrow$ result **depending only on function parameters**

$\rightsquigarrow$ *Similar to a function in the mathematical sense*
(if a type is associated to its set of values)

Pure function

Pure function ≡ function with no side-effect

⇒ **input only** parameters

⇒ result **depending only on function parameters**

⇝ *Similar to a function in the mathematical sense*
(if a type is associated to its set of values)

Such a function is a typed constant

The function type is given by :

- the types of its parameters ;
- and the type of its result.

Recursive function

Recursive function
Function calling itself

Base case
To terminate, a recursive function must contain at least one **base case**, i.e. a non-recursive evaluation.

Tail-recursive function
Recursive function such that any recursive call is a direct call of the function itself

A tail-recursive function needs less memory than a non tail-recursive one.

Recursive function

Factorial (non tail-recursive)

Function factorial(n : integer) : integer
  If $n \leq 0$ **then**
    Return 1
  Else
    Return $n \times \text{factorial}(n-1)$
  EndIf
EndFunction

Recursive function

Factorial (non tail-recursive)

Function factorial(n : integer) : integer
  If $n \leq 0$ **then**
    Return 1
  Else
    Return $n \times$ factorial$(n-1)$
  EndIf
EndFunction

Factorial (tail-recursive)

Function factorialTR(n : integer, a : integer) : integer
  If $n = 0$ **then**
    Return a
  Else
    Return factorialTR$(n-1, n \times a)$
  EndIf
EndFunction
Function factorial(n : integer) : integer
  Return factorialTR$(n, 1)$
EndFunction

Static vs dynamic type checking

Définition (Static type checking)

Types are bound to **variables**.

Static vs dynamic type checking

Définition (Static type checking)

Types are bound to **variables**.

Exemples

Java, C++, **Scala**

Static vs dynamic type checking

Définition (Static type checking)

Types are bound to **variables**.

Exemples

Java, C++, **Scala**

Définition (Dynamic type checking)

Types are bound to **values of variables**.

Static vs dynamic type checking

Définition (Static type checking)

Types are bound to **variables**.

Exemples

Java, C++, **Scala**

Définition (Dynamic type checking)

Types are bound to **values of variables**.

Exemples

Javascript, Python, Ruby, Groovy

Scala

Scala

Description

Identity sheet

Created in 2003 by Martin ODERSKY

- Professor at École Polytechnique Fédérale de Lausanne
- Co-designed Java Generics (for Java 5)
- Original author of current `javac` reference compiler

Apache 2.0 license

- Current version : **2.13.3** (June 25, 2020)
- Version 3 expected for end of 2020 : available for test under "Dotty" code name

Features

- Functional **AND** object-oriented language
- Strong static type checking
- UTF-8 encoding by default
- JVM language (hence generates Java bytecode)

Foundation of various frameworks

Actor/message-based development

- Akka

Web development

- Play!
- Lift
- Scalatra

Testing

- ScalaTest
- ScalaCheck
- Specs2

Distributed computation

- Spark

Scala

Environment

Console

Call : `scala` (with no parameter)

```
$ scala
Welcome to Scala 2.11.12 (OpenJDK 64-Bit Server VM, Java 11.0.8).
Type in expressions for evaluation. Or try :help.

scala> 1 + 1
res0: Int = 2

scala> "42"
res1: String = 42
```

Option `-Dscala.color` enables syntax highlighting on results.

Special commands

- `:help`
- `:load` *file* load contents of *file* as Scala code
- `:quit` (ou Ctrl + D) : quits console

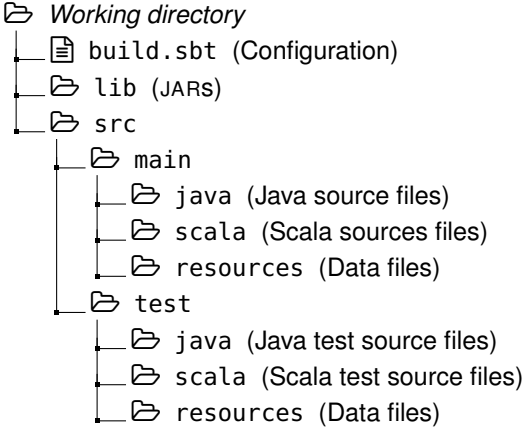
sbt (*Simple Build Tool*)

Features

- Minimal configuration for simple projects
- Support for Scala (and tests)
- Tasks written in Scala DSL
- Dependency support via Ivy
- Interactive mode (console) is available

Create a sbt project

1. Create following directories :



2. Create file `build.sbt`.

sbt (*Simple Build Tool*)

Example for `build.sbt` file

```
organization := "fr.cyu"
name         := "HelloWorld"
version      := "1.0-SNAPSHOT"

// Add one dependency for tests (with '% "test"')
libraryDependencies += "junit" % "junit" % "4.8" % "test"

// Add various dependencies
libraryDependencies ++= Seq("org.scala-lang" % "scala-xml" % "2.11.0-M4",
                            "joda-time"      % "joda-time" % "2.9.5")
```

Subcommands

Any subcommand may be invoked :

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests
- `package` builds a JAR archive

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests
- `package` builds a JAR archive
- `run` *arguments* executes main class with *arguments* as parameters

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests
- `package` builds a JAR archive
- `run` *arguments* executes main class with *arguments* as parameters
- `help` prints the list of available subcommands

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests
- `package` builds a JAR archive
- `run` *arguments* executes main class with *arguments* as parameters
- `help` prints the list of available subcommands
- `help` *command* prints the help of *command*

Scala

Basic elements

Main class. . . rather main object

Same purpose as Java. . .

- Entry point for program with `main` method
- Arguments from call in console captured in a string array

Main class. . . rather main object

Same purpose as Java. . .

- Entry point for program with `main` method
- Arguments from call in console captured in a string array

. . . but some differences in defining it

- One instance $\Longrightarrow$ **`object`** declaration (instead of **`class`**)
- **`App`** trait defines `main` method :
  - no need to declare method if object extends **`App`** ;
  - **the whole object body becomes the `main` method body**.

(Some of these concepts will be addressed later.)

Main class. . . rather main object

Hello World

```
object HelloWorld
  {
    def main(args : Array[String])
      {
        println("Hello World !")
      }
  }
```

Main class. . . rather main object

Hello World

```
object HelloWorld
  {
    def main(args : Array[String])
      {
        println("Hello World !")
      }
  }
```

Hello World (extending **App**)

```
object HelloWorld extends App
  {
    println("Hello, world!")
  }
```

(Not recommended in Scala 3 for performance reasons)

Basic types

In Scala, **any basic type is a class**.

Basic types

In Scala, **any basic type is a class**.

Any basic type is a subclass of **`AnyVal`**.

Basic types

In Scala, **any basic type is a class**.

Any basic type is a subclass of **`AnyVal`**.

- Integer types : **`Byte`**, **`Short`**, **`Int`**, **`Long`**

Basic types

In Scala, **any basic type is a class**.

Any basic type is a subclass of **`AnyVal`**.

- Integer types : **`Byte`**, **`Short`**, **`Int`**, **`Long`**
- Floating types : **`Float`**, **`Double`**

Basic types

In Scala, **any basic type is a class**.

Any basic type is a subclass of **`AnyVal`**.

- Integer types : **`Byte`**, **`Short`**, **`Int`**, **`Long`**
- Floating types : **`Float`**, **`Double`**
- Other types : **`Boolean`**, **`Char`**

Basic types

In Scala, **any basic type is a class**.

Any basic type is a subclass of **`AnyVal`**.

- Integer types : **`Byte`**, **`Short`**, **`Int`**, **`Long`**
- Floating types : **`Float`**, **`Double`**
- Other types : **`Boolean`**, **`Char`**
- **`Unit`** $\equiv$ **`void`** in Java

Basic types

In Scala, **any basic type is a class**.

Any basic type is a subclass of **`AnyVal`**.

- Integer types : **`Byte`**, **`Short`**, **`Int`**, **`Long`**
- Floating types : **`Float`**, **`Double`**
- Other types : **`Boolean`**, **`Char`**
- **`Unit`** $\equiv$ **`void`** in Java
- **`Nothing`** : type with no value

Basic types

In Scala, **any basic type is a class**.

Any basic type is a subclass of **`AnyVal`**.

- Integer types : **`Byte`**, **`Short`**, **`Int`**, **`Long`**
- Floating types : **`Float`**, **`Double`**
- Other types : **`Boolean`**, **`Char`**
- **`Unit`** $\equiv$ **`void`** in Java
- **`Nothing`** : type with no value
- . . . (46 basic types)

Basic types

In Scala, **any basic type is a class**.

Any basic type is a subclass of **`AnyVal`**.

- Integer types : **`Byte`**, **`Short`**, **`Int`**, **`Long`**
- Floating types : **`Float`**, **`Double`**
- Other types : **`Boolean`**, **`Char`**
- **`Unit`** $\equiv$ **`void`** in Java
- **`Nothing`** : type with no value
- ... (46 basic types)

Encapsulation of Java counterparts for namesakes

- Concerns integer, floating and other types

$\Longrightarrow$ All Java operations are available with some subtleties (in particular implicit conversions, becoming less implicit in Scala 3)

Basic types

Divide integers

```
scala> 1 / 2
res0: Int = 0
```

Basic types

Divide integers

```
scala> 1 / 2
res0: Int = 0

scala> (1 / 2).toDouble
res1: Double = 0.0
```

Basic types

Divide integers

```
scala> 1 / 2
res0: Int = 0

scala> (1 / 2).toDouble
res1: Double = 0.0

scala> 1.toDouble / 2
res2: Double = 0.5
```

Declaration of values and variables

Value versus variable

Declaration of values and variables

Value versus variable

- *Value* : **immutable** content, declared with `val`

Declaration of values and variables

Value versus variable

- *Value* : **immutable** content, declared with `val`
- *Variable* : mutable content, declared with `var`

Declaration of values and variables

Value versus variable

- *Value* : **immutable** content, declared with `val`
- *Variable* : mutable content, declared with `var`

Of course, **functional programming favors values.**

Declaration of values and variables

Value versus variable

- *Value* : **immutable** content, declared with **`val`**
- *Variable* : mutable content, declared with **`var`**

Of course, **functional programming favors values.**

Variable declaration in Scala

```
var i   : Int    = 7
var msg : String = "Hello world !"
```

Variable declaration in Java

```
int    i   = 7 ;
String msg = "Hello world !"
```

Value declaration in Scala

```
val i   : Int    = 7
val msg : String = "Hello world !"
```

Value declaration in Java

```
final int    i   = 7 ;
final String msg = "Hello world !"
```

Declaration of values and variables

Type may be infered (unlike Java)

Declaration of values and variables

Type may be infered (unlike Java)

- So type is not always required (although recommended) in a declaration.

```
scala> val answer : String = "42"
answer: String = 42
scala> val answer = "42"
answer: String = 42
```

Declaration of values and variables

Type may be infered (unlike Java)

- So type is not always required (although recommended) in a declaration.

```
scala> val answer : String = "42"
answer: String = 42
scala> val answer = "42"
answer: String = 42
```

- Static type checking prevents from assigning non-coherent type to value or variable.

```
scala> val answer : Int = "42"
<console>:11: error: type mismatch;
 found   : String("42")
 required: Int
       val answer : Int = "42"
                          ^
```

Standard evaluation

Block

- Sequence of instructions/expressions
- Result of block : result of last expression in block

Standard evaluation

Block

- Sequence of instructions/expressions
- Result of block : result of last expression in block

Declarations are done immediately

- Evaluation is performed as soon as it is declared
- Expression are evaluated when defined.

Standard evaluation

Fichier `Test.scala`

```
object Test extends App
  {
    var y = 90 ;
    val x =
      {
        println("Step 1 (x)") ; y = y + 1 ;
        println("Step 2 (x)") ; "#####" + y
      }
    println(x)
    println(x)

    val z =
      {
        println("Step 1 (z)") ; y = y + 1 ;
        println("Step 2 (z)") ; "#####" + y
      }
    println(z)
    println(z)
  }
```

Standard evaluation

Result of execution of `Test.scala`

```
scala> :load Test.scala
Loading Test.scala...
defined object Test
Step 1 (x)
Step 2 (x)
#####91
#####91
Step 1 (z)
Step 2 (z)
#####92
#####92
```

Lazy evaluation : `lazy val`

How lazy evaluation works

Lazy evaluation : `lazy val`

How lazy evaluation works

- Evaluation **at first actual use** outside declaration

Lazy evaluation : `lazy val`

How lazy evaluation works

- Evaluation **at first actual use** outside declaration
- Afterwards, stays unchanged (hey, it's still a `val` !)

Lazy evaluation : `lazy val`

How lazy evaluation works

- Evaluation **at first actual use** outside declaration
- Afterwards, stays unchanged (hey, it's still a `val` !)

`val` versus `lazy val`

Lazy evaluation : `lazy val`

How lazy evaluation works

- Evaluation **at first actual use** outside declaration
- Afterwards, stays unchanged (hey, it's still a `val` !)

`val` versus `lazy val`

```
scala> val x = {println("x is set") ; 3 + 3}
x is set
x: Int = 6
```

Lazy evaluation : `lazy val`

How lazy evaluation works

- Evaluation **at first actual use** outside declaration
- Afterwards, stays unchanged (hey, it's still a `val` !)

`val` versus `lazy val`

```
scala> val x = {println("x is set") ; 3 + 3}
x is set
x: Int = 6

scala> lazy val x = {println("x is set") ; 3 + 3}
x: Int = <lazy>
```

Lazy evaluation : `lazy val`

How lazy evaluation works

- Evaluation **at first actual use** outside declaration
- Afterwards, stays unchanged (hey, it's still a `val` !)

`val` versus `lazy val`

```
scala> val x = {println("x is set") ; 3 + 3}
x is set
x: Int = 6

scala> lazy val x = {println("x is set") ; 3 + 3}
x: Int = <lazy>

scala> x + 3
x is set
res0: Int = 9
```

Lazy evaluation : `lazy val`

How lazy evaluation works

- Evaluation **at first actual use** outside declaration
- Afterwards, stays unchanged (hey, it's still a `val` !)

`val` versus `lazy val`

```
scala> val x = {println("x is set") ; 3 + 3}
x is set
x: Int = 6

scala> lazy val x = {println("x is set") ; 3 + 3}
x: Int = <lazy>

scala> x + 3
x is set
res0: Int = 9

scala> x
res1: Int = 6
```

Lazy evaluation : `lazy val`

Evaluation at first actual use

```
scala> lazy val x = {println("x is set") ; 3 + 3}
x: Int = <lazy>
```

Lazy evaluation : `lazy val`

Evaluation at first actual use

```
scala> lazy val x = {println("x is set") ; 3 + 3}
x: Int = <lazy>

scala> if (false) x else 0
res0: Int = 0
```

Lazy evaluation : `lazy val`

Evaluation at first actual use

```
scala> lazy val x = {println("x is set") ; 3 + 3}
x: Int = <lazy>

scala> if (false) x else 0
res0: Int = 0

scala> if (true) x else 0
x is set
res1: Int = 6
```

$\Longrightarrow$ *Dynamic* evaluation (although type checking remain static)

Everything is an expression. . . hence is typed

Some basic examples

```
scala> val e1 = 1
e1: Int = 1

scala> val e2 = 2
e2: Int = 2

scala> e1 + e2
res0: Int = 3

scala> e1 == e2
res1: Boolean = false

scala> val e0 = println(e1)
1
e0: Unit = ()
```

Everything is an expression. . . hence is typed

Expression `if (cond) expr1 else expr2`

- Condition `cond` must be within parentheses
- Type of result : least common supertype of `expr1` and `expr2`

```
scala> if (e1 == e2) e1 else e2
res2: Int = 2
```

Everything is an expression. . . hence is typed

Expression `if (cond) expr1 else expr2`

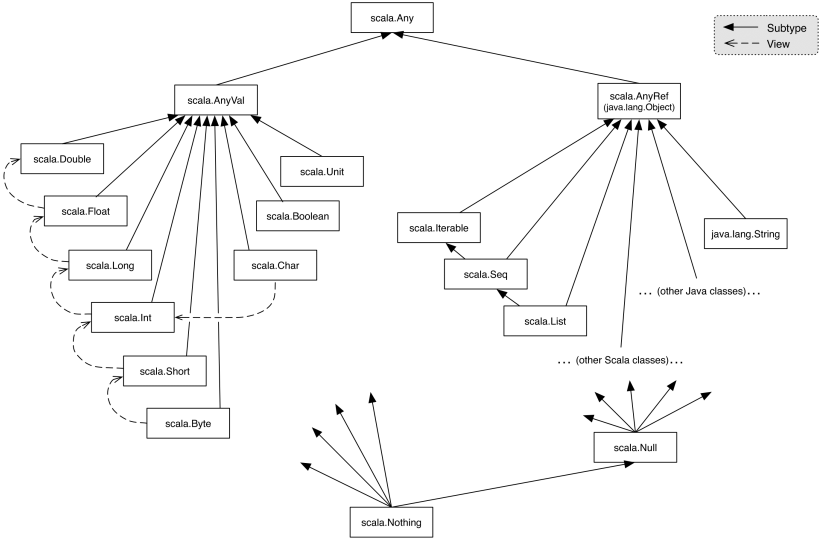
- Condition `cond` must be within parentheses
- Type of result : least common supertype of `expr1` and `expr2`

```
scala> if (e1 == e2) e1 else e2
res2: Int = 2
```

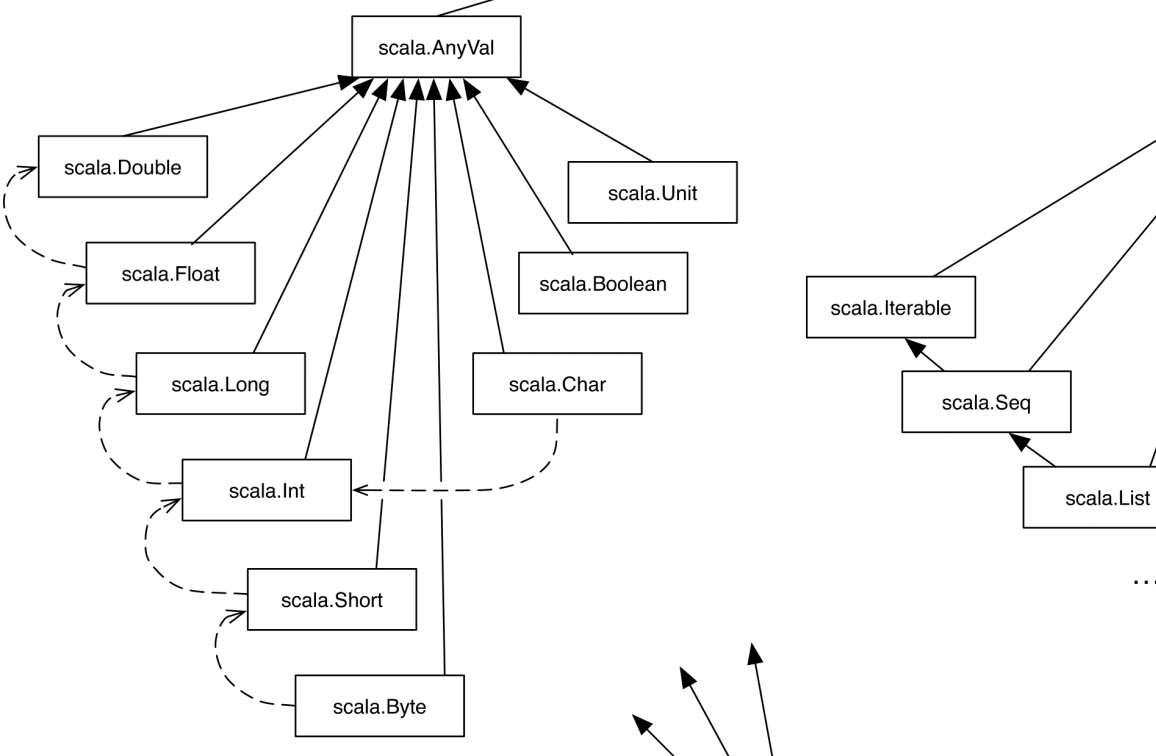
Advanced example

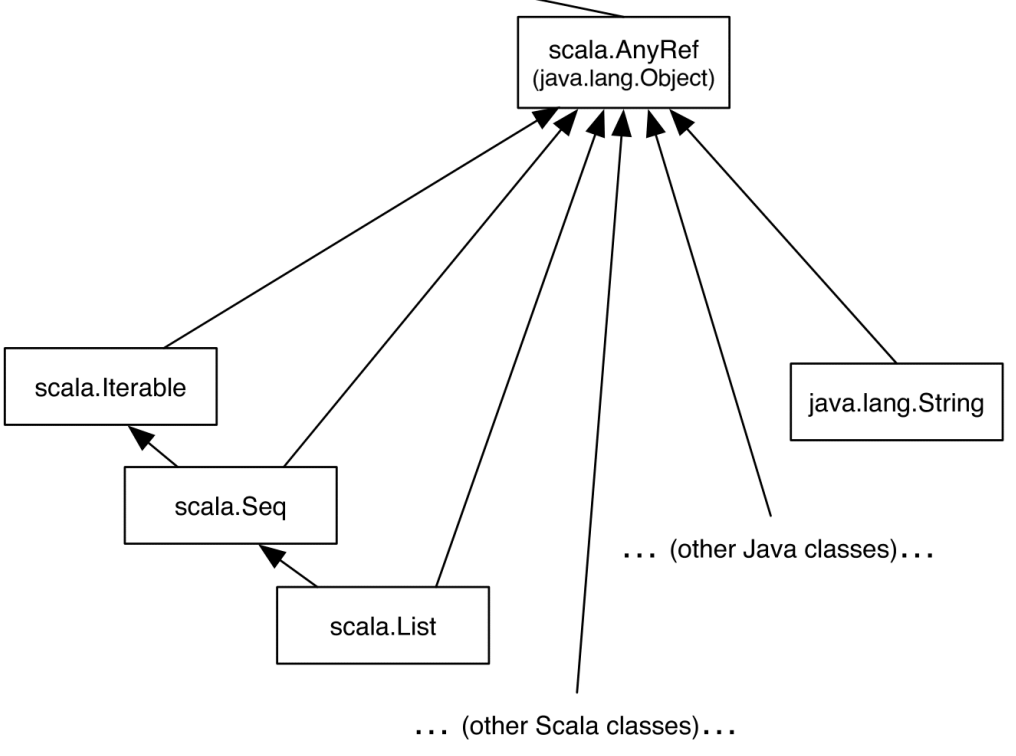
```
scala> val e =
       {
         val cmPerInch = 2.54
         15 * cmPerInch
       }
e: Double = 38.1

scala> cmPerInch
<console>:12: error: not found: value cmPerInch
       cmPerInch
       ^
```

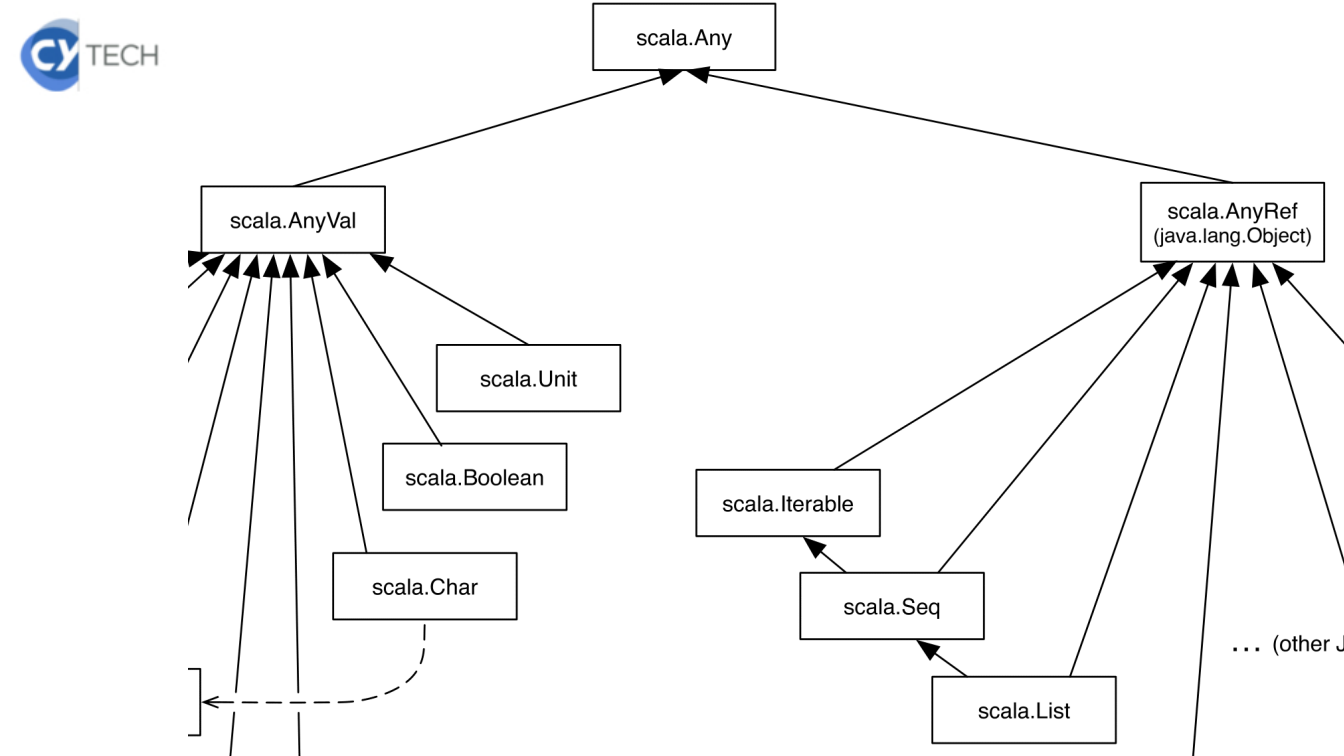
Scala class/type hierarchy

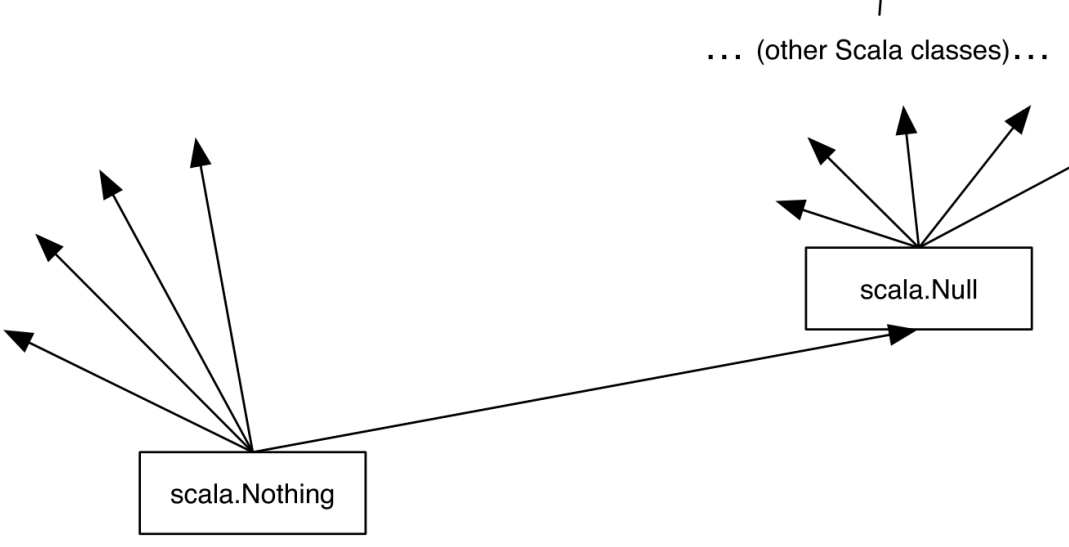


Source : scala-lang.org


scala.AnyVal
scala.Double
scala.Float
scala.Long
scala.Int
scala.Short
scala.Byte
scala.Char
scala.Boolean
scala.Unit
scala.Iterable
scala.Seq
scala.List


scala.AnyRef
(java.lang.Object)
scala.Iterable
scala.Seq
scala.List
java.lang.String
. . . (other Java classes) . . .
. . . (other Scala classes) . . .


scala.Any
scala.AnyVal
scala.AnyRef
(java.lang.Object)
scala.Unit
scala.Boolean
scala.Char
scala.Iterable
scala.Seq
scala.List
... (other J



Source : s

Scala class hierarchy

Supertypes

- **`AnyVal`** : supertype of **all basic types**
- **`AnyRef`** : supertype of all other classes, *including Java classes*
  Default supertype of any user-defined class
- **`Any`** : supertype of **any class** (hence of **`AnyVal`** and **`AnyRef`**)

Subtypes

- **`Nothing`** : subtype of **any class**
- **`Null`** : subtype of any non-basic class

Class **`String`**

Alias to Java class `java.lang.String`

Scala class hierarchy

Example

```
scala> val x : Any = "abc"
x: Any = abc
```

Scala class hierarchy

Example

```
scala> val x : Any = "abc"
x: Any = abc

scala> val y : String = x
x: Any = abc
<console>:12: error: type mismatch;
 found   : Any
 required: String
       val y : String = x
                        ^
```

Scala class hierarchy

Example

```
scala> val x : Any = "abc"
x: Any = abc

scala> val y : String = x
x: Any = abc
<console>:12: error: type mismatch;
 found   : Any
 required: String
       val y : String = x
                        ^
```

(Reverse way by switching types works, of course.)

Scala class hierarchy

Example

```
scala> val x : Any = "abc"
x: Any = abc

scala> val y : String = x
x: Any = abc
<console>:12: error: type mismatch;
 found   : Any
 required: String
       val y : String = x
                        ^
```

(Reverse way by switching types works, of course.)

```
scala> if (true) 42 else "42"
res0: Any = 42
```

*(Type checking is **static**)*

Tuples

Tuple ≡ **immutable** record with numbered fields

- Syntax taken from mathematics
- Fixed number of components (maximum 22)
  (limitation removed in Scala 3)
- Types of components may be different

Comes in handy for making function return multiple values

Tuples

Tuple ≡ **immutable** record with numbered fields

- Syntax taken from mathematics
- Fixed number of components (maximum 22)
  (limitation removed in Scala 3)
- Types of components may be different

Comes in handy for making function return multiple values

Examples

```
scala> val t = ("The answer", 42, "Douglas Adams")
t: (String, Int, String) = (The answer,42,Douglas Adams)

scala> t._2
res0: Int = 42

scala> t._4
<console>:13: error: value _4 is not a member of (String, Int, Double)
       t._4
         ^
```

Option[T]

Aim of Option[T]

Provides a way to add a *“no value”* value to type T

`Option[T]`

Aim of `Option[T]`

Provides a way to add a *“no value”* value to type T

Two kinds of expressions

`Option[T]`

Aim of `Option[T]`

Provides a way to add a *“no value”* value to type T

Two kinds of expressions

- `Some(x)` where x has type T (*mere encapsulation of value x*)

`Option[T]`

Aim of `Option[T]`

Provides a way to add a *“no value”* value to type T

Two kinds of expressions

- `Some(x)` where x has type T (*mere encapsulation of value x*)
- `None` for the *“no value”* case

`Option[T]`

Aim of `Option[T]`

Provides a way to add a *"no value"* value to type T

Two kinds of expressions

- `Some(x)` where x has type T (*mere encapsulation of value x*)
- `None` for the *"no value"* case

Designed to replace exceptions

$\Rightarrow$ **Lack of result becomes a normal behaviour** (with `None`)

`Option[T]`

Aim of `Option[T]`

Provides a way to add a *"no value"* value to type T

Two kinds of expressions

- `Some(x)` where x has type T (*mere encapsulation of value x*)
- `None` for the *"no value"* case

Designed to replace exceptions

$\Rightarrow$ **Lack of result becomes a normal behaviour** (with `None`)

- Error handling made easier *(e.g. for generic functions)*

`Option[T]`

Aim of `Option[T]`

Provides a way to add a *"no value"* value to type T

Two kinds of expressions

- `Some(x)` where x has type T (*mere encapsulation of value x*)
- `None` for the *"no value"* case

Designed to replace exceptions

⇒ **Lack of result becomes a normal behaviour** (with `None`)

- Error handling made easier *(e.g. for generic functions)*
- Error may be transmitted to calling function

`Option[T]`

Aim of `Option[T]`

Provides a way to add a *"no value"* value to type T

Two kinds of expressions

- `Some(x)` where x has type T (*mere encapsulation of value x*)
- `None` for the *"no value"* case

Designed to replace exceptions

⇒ **Lack of result becomes a normal behaviour** (with `None`)

- Error handling made easier *(e.g. for generic functions)*
- Error may be transmitted to calling function
- *May be not enough to differentiate multiple errors*

Option[T]

Example : conversion **String** $\rightarrow$ **Int**

```
scala> def toInt(s : String) : Option[Int] =
         {
           try
             {
               Some(Integer.parseInt(s.trim))
             }
           catch
             {
               case e : Exception => None
             }
         }
toInt: (s: String)Option[Int]
```

Option[T]

Example : conversion **String** → **Int**

```
scala> def toInt(s : String) : Option[Int] =
         {
           try
             {
               Some(Integer.parseInt(s.trim))
             }
           catch
             {
               case e : Exception => None
             }
         }
toInt: (s: String)Option[Int]

scala> toInt("42")
res0: Option[Int] = Some(42)
```

Option[T]

Example : conversion **String** → **Int**

```
scala> def toInt(s : String) : Option[Int] =
         {
           try
             {
               Some(Integer.parseInt(s.trim))
             }
           catch
             {
               case e : Exception => None
             }
         }
toInt: (s: String)Option[Int]

scala> toInt("42")
res0: Option[Int] = Some(42)

scala> toInt("The answer")
res1: Option[Int] = None
```