Scala : un langage de programmation multi-paradigmes
cours 1 : généralités et programmation objet

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

DataCloud – Master 2 SAR 2021/2022

Présentation

Scala = Scalable Language

Bref historique

- 2001 : Début de développement par Martin Odersky (EPFL, Lausane)
- 2003 : Premier enseignement EPFL
- 2004 : Version Scala 1.0
- 2006 : Version Scala 2.0
  - compilateur scalac écrit en scala
- Buzz 2009 : Utilisation par Twitter
- Buzz 2013 : Apache Spark
- 2016 : Scala 2.12
- sept. 2019 : Scala 2.13

Principales caractéristiques

- Langage multi-paradigmes :
  - orienté objet hérité de **Java**
    - ⇒ héritage, polymorphisme, encapsulation
    - ⇒ Java est verbeux
  - programmation fonctionnelle héritée de **Haskell**
    - ⇒ grande expressivité en peu de lignes
    - ⇒ les traitements (fonctions) deviennent des variables
    - ⇒ notion forte de pureté et d'immutabilité
    - ⇒ complexe à appréhender
- Compilé en ByteCode :
  - ⇒ interopérable avec Java
  - ⇒ s'exécute dans une JVM (portabilité)
- Inférence de type : le compilateur peut deviner le type d'une expression
  - ⇒ programmation moins lourde
  - ⇒ syntaxe du programme allégée
- Typage statique : toutes les variables ont un type
  - ⇒ programmation sûre

Les mots clés du langage Scala (vs. ceux de Java)

Communs avec Java 8

- Avec sémantique totalement équivalente

```
import package                  while for  do if else   return
try    finally catch throw      super this new          abstract
true   false   null             extends                 class
```

- Avec sémantique quasi-équivalente

```
private       protected       final                     case
```

Propres à Scala

```
def      val      var      with     match    override   implicit
object   type     yield    sealed   forSome  lazy       trait
<:       <%       >:       _        :        =          =>
<-       #        @
```

Propres à java (pour info) : `boolean float double byte short int long char void switch default continue break throws interface enum implements instanceof public static transient strictfp synchronized volatile assert native goto const`

Hello World

Le HelloWorld Java

```
public class Hello{
 public static void main(String args[]){
  System.out.println("Hello world");
 }
}
```

Le HelloWorld Scala

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

Le HelloWorld Scala V2

```
object Hello2 extends App{
  println("Hello world")
}
```

Compilation et exécution

Compilation

```
$ scalac Hello.scala
$ ls
Hello.scala  Hello.class
```

Exécution

- Soit à partir d'un fichier compilé

  ```
  $ scala Hello
  ```

- Soit directement à partir du fichier source

  ```
  $ scala Hello.scala
  ```

- Soit en interpréteur interactif

  ```
  $ scala
  Welcome to Scala version 2.11.6 (OpenJDK 64-Bit Server VM, Java 1.8.0_91).
  Type in expressions to have them evaluated.
  scala>
  ```

Alternative plus complète : **S**cala **B**uild **T**ool (SBT)

Règles de base

Point d'entrée d'une application scala

- soit une fonction main dans un `object`

```
def main(args: Array[String]){...}
```

- soit un `object` héritant de `App`

```
object MyAppScala extends App {
        this.args //accès aux arguments
        ...
}
```

Règles de casse

- Sensible à la casse : MOT $\neq$ mot
- Les noms de type doivent conventionnellement commencer par une majuscule
- Les noms de variable, de paramètre ou d'attribut doivent conventionnellement commencer par une minuscule

Les Commentaires

2 types de commentaires

- Sur une ligne avec //

```
//un commentaire sur une seule ligne
```

- Sur plusieurs lignes avec /* et */

```
/* Un commentaire
sur plusieurs lignes
*/
```

Les variables non-mutables

Définition

Une variable non mutable est une variable constante. Elle n'est pas accessible en écriture sauf à sa déclaration pour son initialisation.

Syntaxe

```
val <nomVariable> : <typeVariable> = <valeurExpressionInitiale>
```

Exemple

```
scala> val i : Int = 42
i : Int = 42
scala> i = 43
<console>:8: error: reassignment to val
       i1 = 43
          ^
```

Équivalent java

```
final int i = 42;
```

Les variables mutables

Définition

Une variable mutable est une variable dont la valeur peut changer au cours de l'exécution du programme.

Syntaxe

```
var <nomVariable> : <typeVariable> = <valeurExpressionInitiale>
```

Exemple

```
scala> var i : Int = 42
i : Int = 42
scala> i = 43
i: Int = 43                ^
```

Équivalent java

```
int i = 42;
i=43;
```

Initialisation obligatoire à la déclaration

Les types de variables

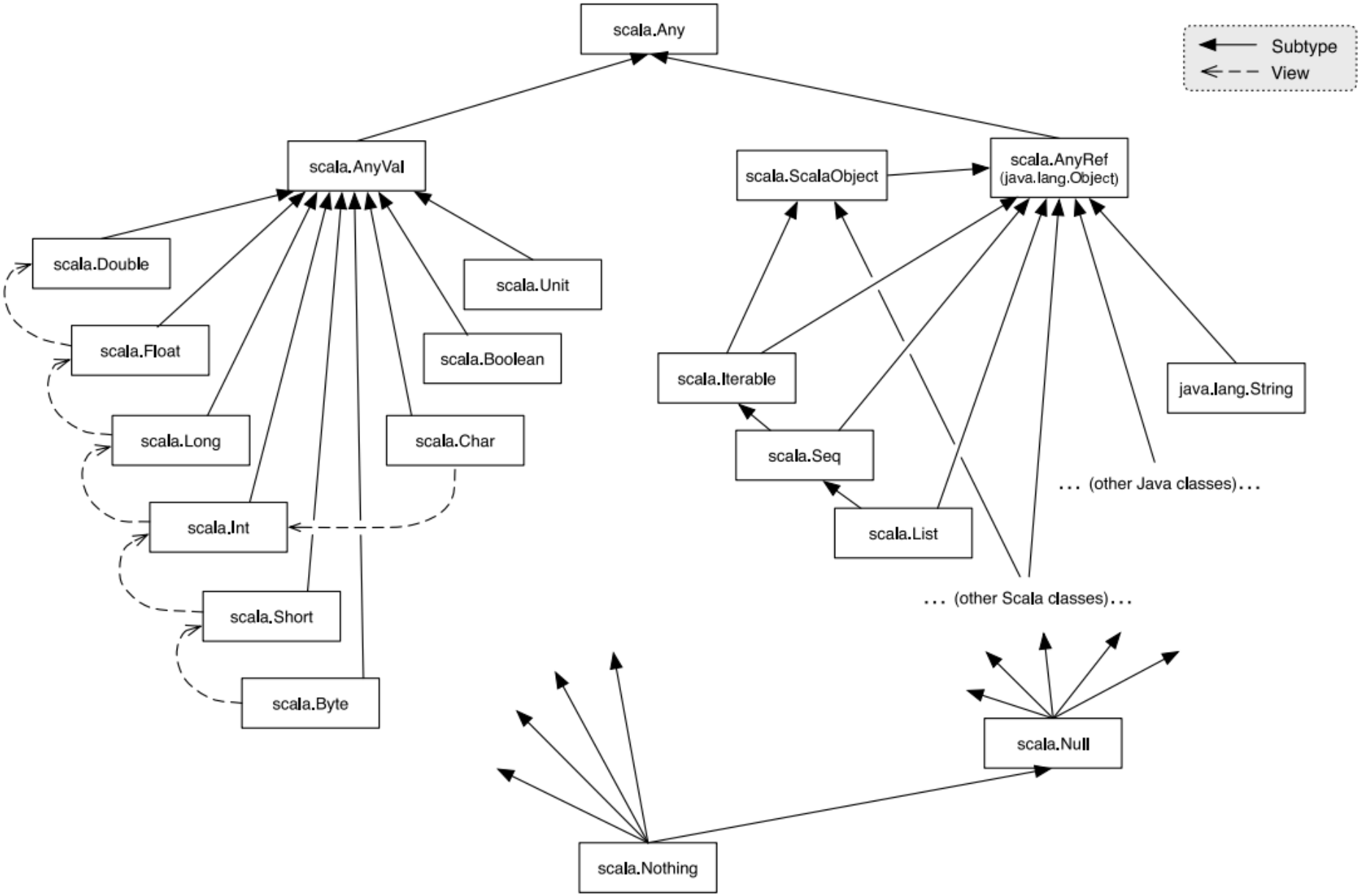
Les types primitifs

Pas en scala : toute variable référence un objet qui instancie une classe ! ! ! ! !

Les types pré-définis en Scala

- `scala.Any` : le type racine. Possèdent deux sous classes directes :
  - `scala.AnyVal` : représente les types-valeur (`Int`, `Double`, `Long` ...)
    $\Rightarrow$ correspondent au types primitifs de Java
    $\Rightarrow$ conversion possible entre ces types
  - `scala.AnyRef` : représente tout autre type, super classe indirecte par défaut (représente le type `java.lang.Object`)
- `scala.Null` :le sous-type par défaut des `scala.AnyRef` (représente le `null` de java)
- `scala.Nothing` : le sous-types par défaut de tout type

Hiérarchie de type Scala



Inférence de type

Définition

Capacité du compilateur à deviner le type d'une expression sans que le programmeur n'est à l'expliciter.

Exemples

```
scala> val i = 3
i: Int = 3

scala> val s = "Toto"
s: String = Toto

scala> val c = 'c'
c: Char = c

scala> val b = true
b: Boolean = true
```

```
scala> val l = 3L
l: Long = 3

scala> val f = 3.5f
f: Float = 3.5

scala> val d = 2.3
d: Double = 2.3

scala> val u = ()
u: Unit = ()
```

Les blocs d'instructions

Définition

Un bloc est une expression contenant un ensemble d'expressions

Caractéristiques

- Délimité par des accolades
- Évaluer un bloc revient à évaluer sa dernière expression
- Toute variable déclarée à l'intérieur du bloc n'existe qu'à l'intérieur de celui-ci

```
scala> {
| 3 + 6
| var varBloc: String = "toto"
| 'A'
| }
res0: Char = A
```

Les opérateurs

Différences avec Java

- Tout opérateur est un nom de méthode d'instance
  $\Rightarrow$ Les opérateurs ne sont plus natifs et peuvent être redéfinis
- Opérateurs unaires = méthodes d'instances commençant par `unary_`

Spécificité par Type

Type d'opérateur	**Méthodes**	**Définis dans**
Arithmétiques	`% * + - /` `unary_+ unary_-`	`Double, Float` `Int, Short, Long, Byte, Char`
Comparaison sup/inf	`> >= < <=`	`Double, Float` `Int, Short, Long, Byte, Char`
Test Égalité/différence	`== !=`	`Any`
Bit à bit	`^ | & >> >>> << unary_~`	`Int, Short, Long, Byte, Char`
Booléen	`&& unary_! ||`	`Boolean`

Attention : `+= -= *= ++ --` **ne sont pas définis par défaut**

Exercice

```
val u  ={
        val i = 2
        val j: Int = {
                        val k = 4
                        k * k
                        }
        i + k
        }
```

Que vaut u, i j et k ? Erreur car `k` a été déclaré dans le dernier bloc imbriqué et n'est pas défini dans l'expression `i+k`.

Exercice

```
val u  ={
        val i = 2
        val k = 5
        val j: Int = {
                     val k = 4
                     k * k
                     }
        i + k
        }
```

Que vaut u, i j et k ? `k=4` (deuxième) `k=5` (premier) `j=16` `i=2` `u=7`

Les structures de contrôles conditionnelles

if-elseif-else

- Exécuter ou non un bloc de code en fonction de la valeur d'une expression booléenne

```
if (<exprBooleenne1>) EXPRESSION1
 else if (<exprBooleenne2>) EXPRESSION2
 else EXPRESSION3
```

L'instruction `switch` n'est pas reconnue en Scala

Les structures de contrôles itératives (boucles)

Boucle TantQue

Exécuter un bloc de code au moins 0 fois tant qu'une condition est vraie

```
while (<exprBooleenne>) {
        ...//code a faire tant que exprBooleenne est vraie
}
```

Boucle Faire-TantQue

Exécuter un bloc de code au moins 1 fois tant qu'une condition est vraie

```
 do{
        ...//code a faire une fois a repeter
           //tant que exprBooleenne est vraie
}while(<exprBooleenne>)
```

Les structures de contrôles itératives (boucles)

Boucle Pour

Équivalent à la boucle for étendue de java mais syntaxe différente !

```
for (x <- COLLECTION) EXPRESSION
```

Cas particuliers

- Pour une sequence de valeurs entières

```
for (i <- Range.apply(MIN,MAX,STEP) ) { /*code pour valeur i*/}

/*ou bien */
for (i <-  MIN to MAX by STEP ) { /*code pour valeur i*/ }
```

- Pour une table associative

```
for ((k,v) <- my_map ) { /*code pour la cle k et la val v*/ }
```

Les débranchements de boucle

Les débranchements de boucle ne sont pas natifs au langage Scala

Équivalence en passant par des objets de type `Breaks`

```
val outer = new Breaks;
val inner = new Breaks;

outer.breakable {
  for( a <- 1 to 10){
    inner.breakable {
      for( b <- 1 to 20){
        if( b == 12 ){
          inner.break;
        }
      }
    } // inner breakable
  }
} // outer breakable
```

La définition de fonctions

Syntaxes

```
def functionName ([list of parameters]) : [returnType] = {
    function body
    return [expr]
}
```

ou bien plus généralement :

```
def functionName ([list of parameters]) : [returnType] = EXPRESSION
```

ou si la fonction n'a pas d'argument :

```
def functionName  : [returnType] = EXPRESSION
```

Particularité des fonctions en Scala

- Une fonction peut se définir à l'intérieure d'une autre fonction.
- le nom de la fonction peut avoir des caractères comme :

```
+ ++ ~ & - -- \ / ;
```

- Une fonction est abstraite si on omet l'opérateur = suivi du corps

La définition de fonctions : exemples

Syntaxe à la Java

```
def add (a:Int, b:Int):Int ={
    var sum:Int = 0
    sum = a + b
    return sum
}
```

Versions plus concise

```
def add(a:Int,b:Int)=a+b
```

Dans les deux cas, l'interpréteur scala affiche :

```
add: (a: Int, b:Int)Int
```

Les classes

Déclaration

```
class MaClass{
        //corps de la classe
}
```

Instanciation

```
scala> val a = new MaClass
a: MaClass = MaClass@311d617d
```

Le type d'évaluation de `new` est la référence de l'objet dans la JVM

Les constructeurs

Constructeur principal

- les paramètres sont définis au niveau de l'en-tête de la classe
- le corps est toute instruction en dehors d'une méthode.

```
class Point(a:Int, b:Int){//parametre constructeur principal
    ...//corps du constructeur si en dehors d'une méthode
}
```

Constructeurs secondaires

```
class Point(x:Int, y:Int){
    def this() = this(0,0)
    def this(x:Int) = this(x,0)
    ...
 }
```

Attention

Les secondaires doivent obligatoirement faire appel au principal

Les attributs

Caractéristiques

Toute déclaration de `val` ou `var` :

- Dans le corps d'une classe en dehors d'une méthode

```
class Point(xc:Int , yc:Int){
        var x: Int = xc
        var y: Int = yc
        ...
}
```

- Ou/et directement dans les paramètres du constructeur principal

```
class Point(var x: Int,  var y: Int){ ...}
```

Accéder aux attributs

Attributs Mutables

```
$ cat Point.scala
 class Point(var x: Int,
             var y: Int)

$ javap -p Point

Compiled from "Point.scala"
public class Point {
  private int x;
  private int y;
  public int x();
  public void x_$eq(int);
  public int y();
  public void y_$eq(int);
  public Point(int, int);
}
```

Attributs Non-Mutables

```
$ cat Point.scala
 class Point(val x: Int,
             val y: Int)

$ javap -p Point
Compiled from "Point.scala"
public class Point {
  private final int x;
  private final int y;
  public int x();
  public int y();
  public Point(int, int);
}
```

Les compilateur Scala créé des accesseurs par défaut

```
val p = new Point(3,4)
p.x= p.y  // comme si on avait fait à p.x_=(p.y()) en java
```

Les méthodes d'instances

Définition

En Scala, une méthode d'instance est une fonction déclarée dans une classe. Le mot clé `this` désigne la référence sur l'objet appelant.

Déclaration

```
class Point(var x:Int, var y:Int) {
  def move(dx: Int, dy: Int) {
     x = x + dx //ou this.x=this.x + dx
     y = y + dy //ou this.y=this.y + dy
  }
 }
```

Appel de méthode sur un objet

```
val p = new Point(0,0)

p.move(1,1) // appel possible

p move (1,1)// autre appel possible
```

Héritage

```
class Point3D(x: Int, y: Int, var z :Int) extends Point(x, y){

 def move(dx: Int, dy: Int, dz: Int) {
  x = x + dx
  y = y + dy
  z = z + dz
 }
 override  def toString: String = "(" + x + ", " + y + ", " + z +")"
}
```

Remarques

- `x` et `y` sont considérés ici comme des paramètres du constructeur (pas de `val`/`var`)
- Toute redéfinition de méthode doit être précédée du mot clé `override`
- Une classe hérite par défaut de `AnyRef`
- Impossible d'hériter de plusieurs classes

Restreindre l'héritage

Deux types de restrictions d'héritage d'une classe `A`

- `final class A` : définitions de sous-classes interdites
- `sealed class A` : possible de définir des sous-classes directes de `A` seulement dans le fichier source où `A` a été définie.
  $\Rightarrow$ utile si on veut définir exhaustivement l'ensemble des sous-types

Dans fichier op.scala

```
sealed abstract class Operator {...}

class Plus extends Operator {...}
class Fois extends Operator {...}
```

Dans fichier mod.scala

```
class Mod extends Operator {...} // erreur de compilation
```

Les singletons

Définition

Un singleton est une classe qui n'a qu'une seule instance.

Caractéristiques

- Remplace les champs statiques de Java
- Ne peut pas être instancié mais peut étendre une classe
- Se déclare avec le mot clé `object`

```
object Demo {
   def main(args: Array[String]) {
      val point = new Point(10, 20)
      printPoint
   }

   def printPoint{//S'appelle avec  Demo.printPoint
         println ("Point x location : " + point.getX);
         println ("Point y location : " + point.getY);
   }
}
```

Les packages

Définition

Un package est un objet qui défini un ensemble de classe, object, et d'autres packages.

Exemple

Fichier Essai.scala

```
package A

package B{
  class E
  package C{
    class F
    object O
  }
}
```

Particularité Scala

- Il n'y a aucune contrainte sur l'arborescence des fichiers sources sur le système de fichiers
- L'arborescence sur le système de fichiers sera créée à la compilation pour les binaires

Import des packages

Syntaxe

`import p.I` où `I` détermine l'ensemble des noms des membres de `p` à importer

Exemples d'import

- tous les membres de p : `import p._`
- le membre x de p : `import p.x`
- le membre x de p renommé a : `import p.{x =>a}`
- les membres x et y de p : `import p.{x, y}`
- un membre interne z d'un de membre p2 de p : `import p.p2.z`

Objets implicitement importés

les packages `java.lang` et `scala`, l'objet `scala.Predef`

Visibilité des membres

Problème

Les champs sont par défaut publics

$\Rightarrow$ visibles depuis toute portion de code (perte d'encapsulation)

Modificateur de visibilité

Dans une classe A, un champ :

- `private[X]` est visible dans X et à tous ses sous-membres
- `protected[X]` est visible dans X, à tous ses sous-membres et aux instances des sous classes de A

X peut être :

- A lui même (valeur par défaut)
- un nom de `package`, de `class`, d'`object` qui contient A
- `this` : la visibilité se réduit à l'instance courante

Visibilité des membres : exemple

```
package P{
 class A ( <VISI>  val x:Int){
   def a(aa:A)=...
 }
 class B extends A{
   def b(ba:A, bb:B)=...
 }
}
```

```
class C extends P.B{
   def c(ca:P.A, cb:P.B, cc:C)=...
 }
```

Valeur de VISI	dans a()		dans b()			dans c()			
	this.x	aa.x	this.x	ba.x	bb.x	this.x	ca.x	cb.x	cc.x
private[this]	✓	✗	✗	✗	✗	✗	✗	✗	✗
private[A]	✓	✓	✗	✗	✗	✗	✗	✗	✗
protected[this]	✓	✗	✓	✗	✗	✓	✗	✗	✗
private[P]	✓	✓	✓	✓	✓	✗	✗	✗	✗
protected[A]	✓	✓	✓	✗	✓	✓	✗	✗	✓
protected[P]	✓	✓	✓	✓	✓	✓	✗	✗	✓
∅	✓	✓	✓	✓	✓	✓	✓	✓	✓

Les classes abstraites

Définition

- La classe abstraite Scala est équivalente à la classe abstraite Java
  $\Rightarrow$ Pas d'instanciation possible
- Possibilité de définir des méthodes sans implémentation.

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

Les traits

Caractéristiques

- Type sans attribut et sans constructeur
- Les méthodes peuvent être abstraite ou avoir une implémentation par défaut
- Un trait ou une classe peut implanter plusieurs traits (mot clé `with`)
- Un trait peut hériter d'une classe (par défaut `AnyRef`)
  Trait `T` héritant de classe `C` ⇒ Toute classe `X` implantant `T` doit être un sous-type de `C` par le lien d'héritage

```
trait Movable {
   def move(dx: Int, dy:Int): Unit
}
trait Drawable {
        def draw(): Unit
}
```

Implantation de trait(s) :

```
class Point(var x: Int, var y: Int) extends Movable with Drawable
```

Les case classes

Déclaration

```
case class Personne(nom:String, prenom: String){ ... }
```

Caractéristiques

- Instanciation simplifiée : pas besoin de `new`

```
scala> val p = Personne("Bon","Jean")
p: Personne = Personne(Bon,Jean)
```

- Les instances sont comparées par valeur et non par référence

```
scala> val p2 = Personne("Bon","Jean")
p2: Personne = Personne(Bon,Jean)

scala> p2==p
res0: Boolean = true
```

- Objet immutables par convention (les attributs `var` sont dépréciés)
  $\Rightarrow$ une modification implique une nouvelle instance

Type générique

Déclaration

```
class Panier[T]{ ... }//T est globale à la classe

def f[U](i:Int) : U = {...}//U est local à une fonction/méthode
```

Gestion de la covariance de sous type

Si `Pomme` est un sous type de `Fruit`, est ce que `Panier[Pomme]` est un sous type de `Panier[Fruit]` ?

oui si on déclare `class Panier[+T]{...}`

Contra-variance de sous-type

Si `A` est un sous type de `B`, alors un `Array` de `B` est un sous type d' `Array` de `A` si on déclare `Array[-A]`

exemple : `trait Function1[-T1, +R] extends AnyRef`

Borner les types génériques

Borne supérieure (*Upper bound*)

```
class Panier[T <: Fruit]{ ... }
```

⇒ `T` doit être égal à `Fruit` ou un sous-type de `Fruit`

Borne inférieure (*Lower bound*)

```
class MaClasse[T >: Machin]{ ... }
```

⇒ `T` doit être égal à `Machin` ou un type parent de `Machin`

Borne de vue (*View bound*)

```
class MonAutreClass[T <% Legume]{ ... }
```

⇒ `T` doit pouvoir être convertit implicitement vers un `Legume`

(cf. conversion implicites)

Pattern Matching

Définition

Structure de contrôle qui permet l'aiguillage du flux d'exécution à partir d'un filtrage de motif sur une expression donnée

$\Rightarrow$ généralisation du switch du Java

```
<exprX> match {
    case pattern1 [if <guard1>] => <expr1match>
    case pattern2 [if <guard2>] => <expr2match>
    ...
    case patternN [if <guardN>] => <exprNmatch>
}
```

Attention

- Évaluation dans l'ordre de déclaration : "le 1er qui matche gagne"
- Le type des patterns doit être celui de expX ou bien un sous-type
- Si aucun pattern ne correspond $\Rightarrow$ scala.MatchError

Pattern Matching

Pour une expression X, un pattern peut être

- Un littéral : `2`, `true`, `"abc"`, `(3,4)`
- Un identifiant de constante : `MAXINT`, `EmptySet`
- Un pattern de variable : `n`, `(x,y)`, `(x,y,z)`
  le nom de la variable est liées à la valeur de X
- Le Joker `_` : correspond à n'importe quel motif de X
  aucun lien entre un nom de variable et la valeur
- Un constructeur $C(arg_1, arg_2, ..., arg_n)$ d'une case classe
  les arguments du contructeur sont aussi des patterns

Pattern Matching : exemples

Exemple 1

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case a if a >= 3 && a<=10 => "between␣3␣and␣10"
  case _ => "many"
}
```

Exemple 2

```
def matchTest(x: Any): Any = x match {
  case 1 => "one"
  case "two" => 2
  case y: Int => "scala.Int␣of␣value␣"+y
  case Personne("Jean",nom) => "ce␣Jean␣s'appelle␣"+nom
  case (a:Int,b:Int)  if a > b => "tuple2␣et␣a␣>␣b"
  case (_,_) => "c'est␣un␣tuple2"
  case (3,b:Int,Personne(_,_)) =>  42+b
}
```

Les conversions implicites

Définition

Conversion implicite d'un objet de type `S` vers un objet de type `T`.

```
implicit def nomPasImportant(a:S):T = ...
```

La conversion de `S` vers `T` se fait implicitement ssi $\exists$ **exactement une** méthode implicite `S=>T` importée dans le code client :

$\Rightarrow$ les méthodes de T paraissent accessibles sur un objet S

$\Rightarrow$ "simule" l'extension de n'importe quelle classe (même final)

```
class IntWrapper(val original: Int) {
  def isPair = original % 2 == 0
}
implicit def IntToIntWrapper(value: Int) = new IntWrapper(value)
implicit def IntToString(value: Int):String = value.toString

500.length //ceci compile, appel à length de String, renvoie 3
3.isPair //ceci compile, appel à isPair de IntWrapper, renvoie false

def f(s:String) =  s.charAt(0)
f(800) // fonctionne très bien également
```

Scala Standard Library

Les principaux packages

Nom du package	Caractéristiques	Exemple de types
`scala`	package racine implicitement importé	`Any, Int, Nothing, Unit ..` `Option, Tuple, Array, ...` `App, ..`
`scala.collection`	l'API des collections	`Vector, List, HashMap ..`
`scala.concurrent`	primitives de programmation concurrente	`Futures, Promises ..`
`scala.io`	opérations d'I/O	`Codec, Source ..`
`scala.math`	fonctions basiques et types numériques additionnels	`BigDecimal, BigInt ..`
`scala.sys`	Interaction avec la JVM	`SystemProperties ..`
`scala.util.matching`	Expressions régulières	`Regex ..`

Scala Standard Library : La classe Any

Définition

Classe racine de la hiérarchie des types

Méthodes offertes

- `def equals(arg0 :Any) :Boolean` ⇒ Comparer l'objet appelant à arg0
- `def hashCode :Int` ⇒ Retourne un hachage de l'objet appelant.
- `def toString :java.lang.String` ⇒ Retourne une représentation string de l'objet
- `final def isInstanceOf[T0]:Boolean` ⇒ Teste si l'objet est de type T0.
- `final def asInstanceOf[T0]:T0` ⇒ Caster l'objet appelant en T0
- `final def ==(arg0 :Any) :Boolean` ⇒ alias de equals
- `final def != (arg0 :Any) :Boolean` ⇒ renvoie `!(this == arg0)`

Scala Standard Library : les Tableaux

Spécification

Stockage **séquentiel** d'un **nombre fixe** respectant un même type

```
final class Array[T] extends java.io.Serializable
                             with java.lang.Cloneable
```

Méthodes

- Lecture d'une case `i` : `def apply(i:Int):T`
- Écriture dans une case `i` : `def update(i:Int, x:T):Unit`
- Taille du tableau : `def length:Int`
- + des fonctions `implicit` pour considérer les Array comme des Collections

Exemple

```
var z = new Array[String](3)
var y = Array("Jupiter","Saturne","Neptune")
z(0) = "Toto" //sucre syntaxique pour z.update(0,"Toto")
println( y(2)) // sucre syntaxique pour y.apply(2)
```

Scala Standard Library : les Tableaux à plusieurs dimensions

Particularité par rapport à Java

Pas de type spécifique Scala pour un tableau à N dimensions

⇒ Obligation de déclarer des objet de type `Array[Array[Array[ ... ]]]`

Déclaration avec la méthode `Array.ofDim`

```
scala> val matrix = Array.ofDim[Int](2,2)
matrix: Array[Array[Int]] = Array(Array(0, 0), Array(0, 0))

scala> val cube = Array.ofDim[Int](2,2,2)
cube: Array[Array[Array[Int]]] = .......


scala> val x =cube(0)(1)(1)
x: Int = 0
```

Scala Standard Library : Les Tuples

Spécifications

Objet **immutable** combinant un **nombre fixe d'objets** pouvant être de **types différents**.

Déclaration

```
val a = new Tuple3(1, "hi", A(2))//or val t = (1, "hi", A(2))
val b = new Tuple4(1, "hi", A(2),3.2)//or val t = (1, "hi", A(2),3.2)
//... until Tuple22
```

Accès

- Accès au ième élément ($i \geq 1$) d'un tuple `t` : `t._i`.
  Exemple :

  ```
  val sum = b._1 + b._2 + b._3 + b._4
  ```

- Swapper les éléments d'un tuple2 :

  ```
  val couple = ("Scala", 2.12)
  println("Swapped Tuple: " + couple.swap )
  ```

Scala Standard Library : les Options

Spécifications

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

- Type enveloppe contenant une valeur optionnelle.
- Sous-types directs :
  - `object None extends Option[Nothing]` : enveloppe vide
  - `final case class Some[+A](value:A) extends Option[A]` : enveloppe contenant un objet de type `A`
- conversion implicite vers l'API des collection

Cas d'utilisation

```
def racine(d: Double) : Option[Double] =
     if (d >= 0)  Some(math.sqrt(d))
     else None

var x = racine(5)
 x match {
     case Some(a) => println("sqrt(" + x + ")=" + a)
     case None => println("calcul␣impossible␣de␣"sqrt(" ␣+␣x␣+␣")")
}
```

Les Collections Scala : les packages

Le package `scala.collection`

Package racine :

- API générique offerte par l'ensemble des Collections
- Uniquement composé de `trait` ou de `abstract class`

Le package `scala.collection.immutable`

Implémentations garantissant un état interne constant d'une Collection après son instanciation

Le package `scala.collection.mutable`

Implémentations proposant une API enrichie de méthodes qui permettent la modification interne des collections (ajout, suppression, ...).

Attention : les implémentations par défaut sont les immutables

Les Collections Scala : les 3 familles de collections

Les Set

Collections sans doublons.

Les Seq

Collections associant à chaque élément un indice de position de 0 à $size - 1$

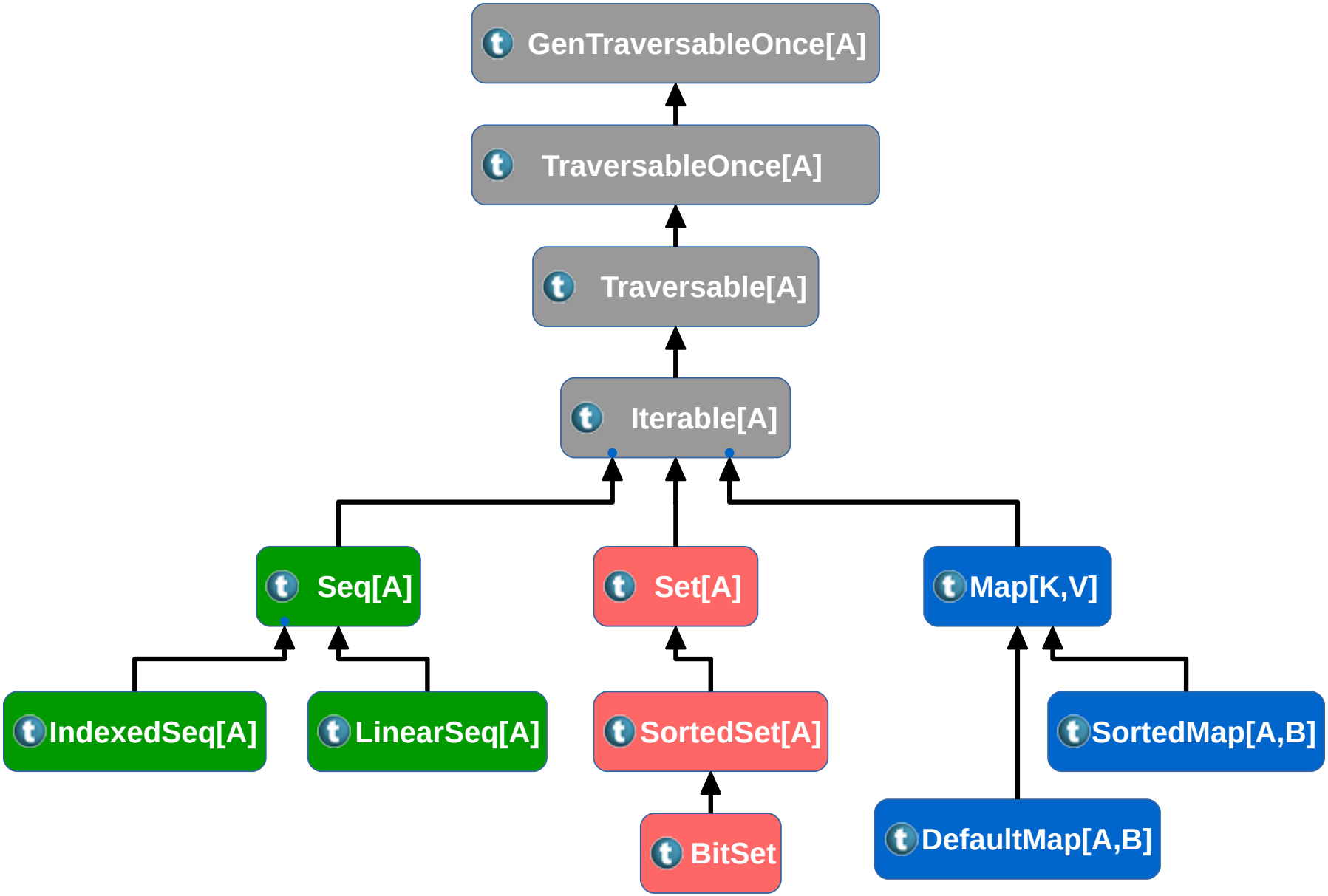
$\Rightarrow$ Doublons possibles

Les Map

Collections de couples (clé, valeur) assurant l'unicité des clés. Un élément peut s'exprimer avec la syntaxe :

- `(key,value)`
- ou bien `key -> value`

Les Collections Scala : le package `scala.collection`



Les Collections Scala : Instanciation

Principe

Chaque type de collection peut être créé en instanciant l'implémentation par défaut de ce type en écrivant : `TypeVoulu(elem1,elem2,elem3, ....)`

Exemples

```
scala> val c1 = Traversable(1, 2, 3)
c1: Traversable[Int] = List(1, 2, 3)

scala> val c2 = Iterable("x", "y", "z")
c2: Iterable[String] = List(x, y, z)

scala> val c3 = Map("x" -> 24, "y" -> 25) //ou Map(("x",24),("y",25))
c3: scala.collection.immutable.Map[String,Int] = Map(x -> 24, y -> 25)

scala> val c4 = Set(0, 1, 2)
c4: scala.collection.immutable.Set[Int] = Set(0, 1, 2)

scala> val c5= SortedSet("hello", "world")
c5: scala.collection.SortedSet[String] = TreeSet(hello, world)

scala> val c6 = IndexedSeq(1.0, 2.0)
c6: IndexedSeq[Double] = Vector(1.0, 2.0)

scala> val c7 = LinearSeq('a', 'b', 'c')
c7: scala.collection.LinearSeq[Char] = List(a, b, c)
```

Les Collections Scala : le trait Traversable

Méthode abstraite offerte

Appliquer un traitement sur l'ensemble des éléments :

```
abstract def foreach(f:(A) =>Unit):Unit
```

Méthodes offertes (héritées des traits supérieurs)

Type d'opération	**Exemples de méthode**
Concaténation	`++`
Transformation	`map, flatMap, collect`
Conversion	`toArray, toList, toIterable toSeq, toSet, toMap`
Copie	`copyToBuffer, copyToArray`
Taille	`isEmpty, nonEmpty, size`
Récupération d'éléments	`head, last, find headOption, lastOption`
Sous-collection	`tail, init, slice, take, drop` `takeWhile, dropWhile, filter filterNot, withFilter`
Division	`splitAt, span, partition, groupBy`
Test sur les éléments	`exists, forall, count`
Réduction	`foldLeft, foldRight, /:, :\, reduceLeft, reduceRight`

Les Collections Scala : le trait Iterable

Méthode abstraite offerte

Obtenir un objet itérateur qui décrit comment parcourir la collection :

```
abstract def iterator:Iterator[A]
```

Implémentation de `foreach`

```
def foreach[U](f: Elem => U): Unit = {
  val it = iterator
  while (it.hasNext) f(it.next())
}
```

Les Collections Scala : Les Seq

Définition

Collections associant à chaque élément un indice de position de 0 à $size - 1$

Méthodes offertes

Type d'opération	**Exemples de méthode**
Indexation et longueur	`apply, isDefinedAt, length, indices, lengthCompare`
Recherche d'index	`indexOf, lastIndexOf, indexOfSlice, lastIndexOfSlice indexWhere, lastIndexWhere, segmentLength`
Ajout	`+:, :+, padTo`
Mis à jour	`updated, patch`
Tri	`sorted, sortWith, sortBy`
Inverse	`reverse, reverseIterator, reverseMap`
Comparaison	`startsWith, endsWith, contains corresponds`
Multi-ensemble	`intersect, diff, union, distinct`

Méthodes supplémentaires offertes par les mutables

Ajout	`+=, ++=, insert, +=:, ++=:, insertAll`
Suppression	`-=, trimStart, trimEnd, clear`

Les Collections Scala : Les Set

Définition

Collections sans doublons.

Méthodes offertes

Type d'opération	**Exemples de méthode**
Test	`contains, apply, subsetOf`
Ajout	`+, ++`
Suppression	`-, --`
Ensemble	`intersect, union, diff` ou bien respectivement `& | &~`

Méthodes supplémentaires offertes par les mutables

Ajout	`+=, ++=, add, update`
Suppression	`-=, --=, remove, retain, clear`

Les Collections Scala : Les Map

Définition

Collections de couples (clé, valeur) assurant l'unicité des clés.

Méthodes offertes

Type d'opération	**Exemples de méthode**
Recherche	`apply, get, getOrElse, contains, isDefinedAt`
Ajout	`+, ++, updated`
Suppression	`-, --`
Sous-collection	`keys, keySet, keysIterator, values, valuesIterator`
Transformation	`filterKeys, mapValues`

Méthodes supplémentaires offertes par les mutables

Ajout	`+=, ++=, put, getOrElseUpdate`
Suppression	`-=, --=, remove, retain, clear`
transformation	`transform`

Les Collections Scala : les implémentations Immutables

Les Sets

- `TreeSet` : ensemble utilisant un arbre rouge-noir
- `BitSet` : ensemble d'entiers représentant un vecteur de bits

Les Seqs

- `List` : accès tête de liste et ajout $\mathcal{O}(1)$, autre accès $\mathcal{O}(N)$
- `Stream` : accès paresseux, les éléments sont évalués si besoin
- `Vector` : accès à n'importe quel élément en $\mathcal{O}(1)$, ajout $\mathcal{O}(N)$
- `Queue` : accès aux extrémités $\mathcal{O}(1)$
- `Range` : séquence d'Int où les éléments sont espacés par un pas fixe

Les Maps

- `TreeMap` : l'ensemble des clés est un `TreeSet`
- `ListMap` : liste chaînée de couple clé/valeur

Les Collections Scala : les implémentations Mutables

Les Sets

- `HashSet` : ensemble construit à partir d'une table de hachage

Les Seqs

- `ArrayBuffer` : tableau de taille dynamique
- `ListBuffer` : liste chaînée
- `MutableList` : liste chaînée + pointeur du dernier nœud vide

Les Maps

- `HashMap` : l'ensemble des clés est un `HashSet`
- `WeakHashMap` : les entrées sont supprimés de la map quand leur clé n'est plus référencée