
Scala

Scala

Scala ?

- Hybride Objet / fonctionnel
- Compilé en **bytecode**
  - code intermédiaire entre les *instructions machines* et le *code source*, qui n'est pas directement exécutable
- S'exécute dans la machine virtuelle Java
- Un langage **Scalable**
  - Adapté pour des petits scripts
  - Mais aussi pour des gros traitements distribués
- **Concis** (Programmation fonctionnelle et typage)

Avec Scala …

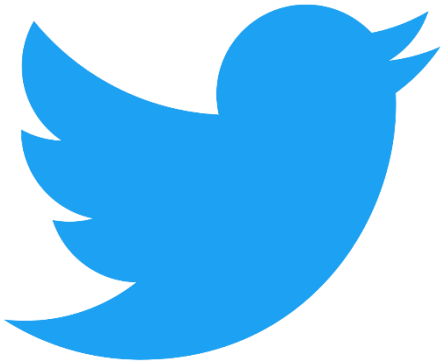
Réutiliser l'existant Java

- Librairies
- App servers

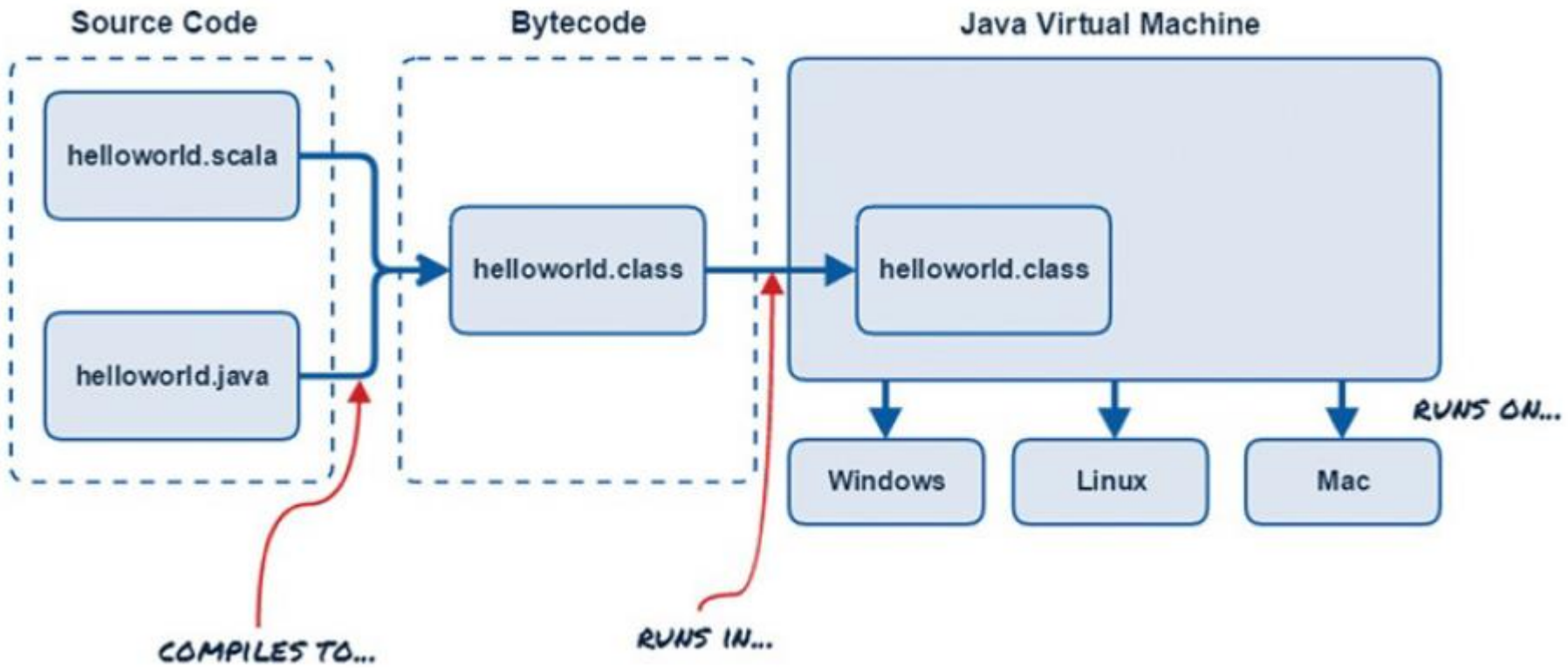
Scala

Utilisé par :

FOURSQUARE



Compiler et exécuter un programme Scala ou java



Variables and Primitives in Scala

Scala Value Classes

Scala Value	Description	Java Equivalent
`Byte`	8-bit signed integer	`byte`
`Short`	16-bit signed integer	`short`
`Int`	32-bit signed integer	`int`
`Long`	64-bit signed integer	`long`
`Char`	16-bit single Unicode character	`char`
`String`	Array of characters	`String`
`Float`	32-bit single-precision float	`float`
`Double`	64-bit double-precision float	`double`
`Boolean`	True or false	`boolean`
`Unit`	Disregarded return value	`void`

Variables and Primitives in Scala

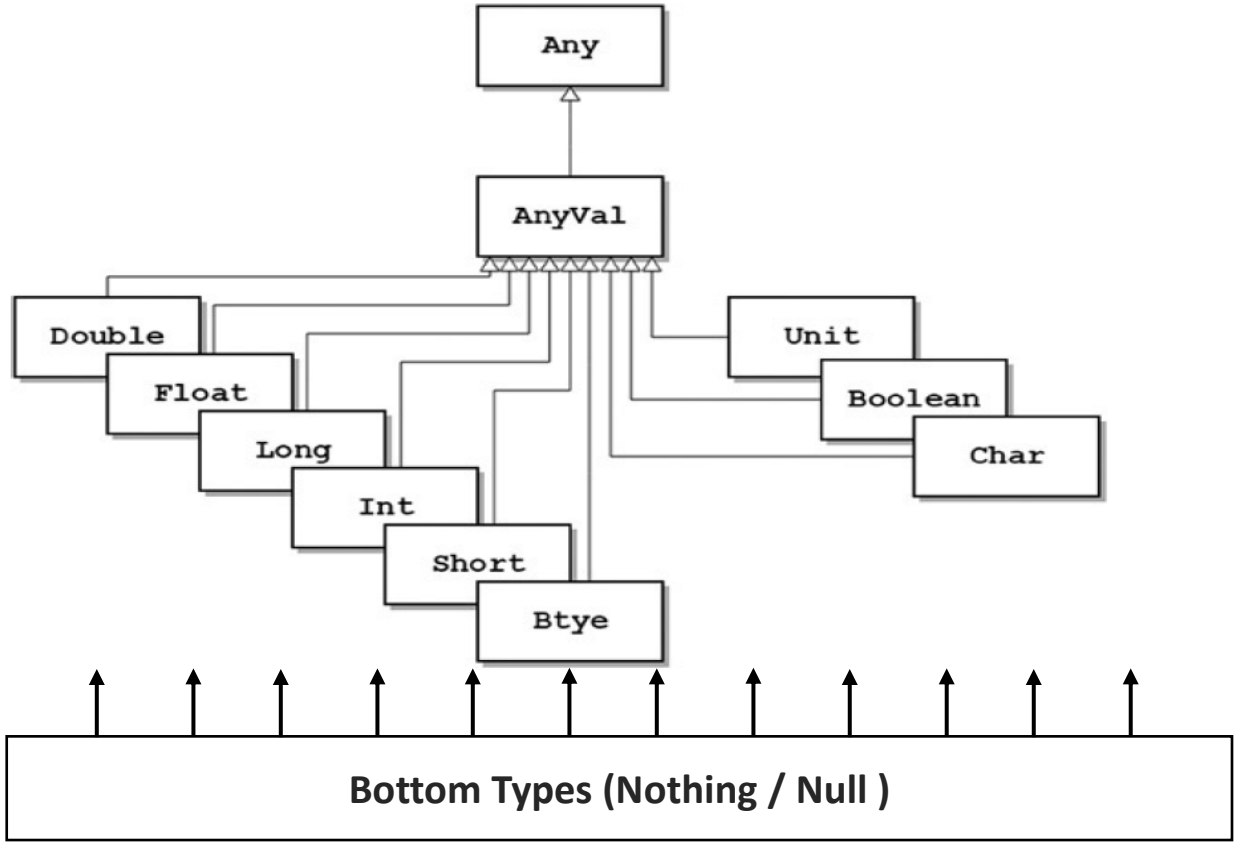
Les classes de valeur dans **Scala** sont des sous-classes de la classe **AnyVal**

AnyVal est une sous-classes de la classe **Any**

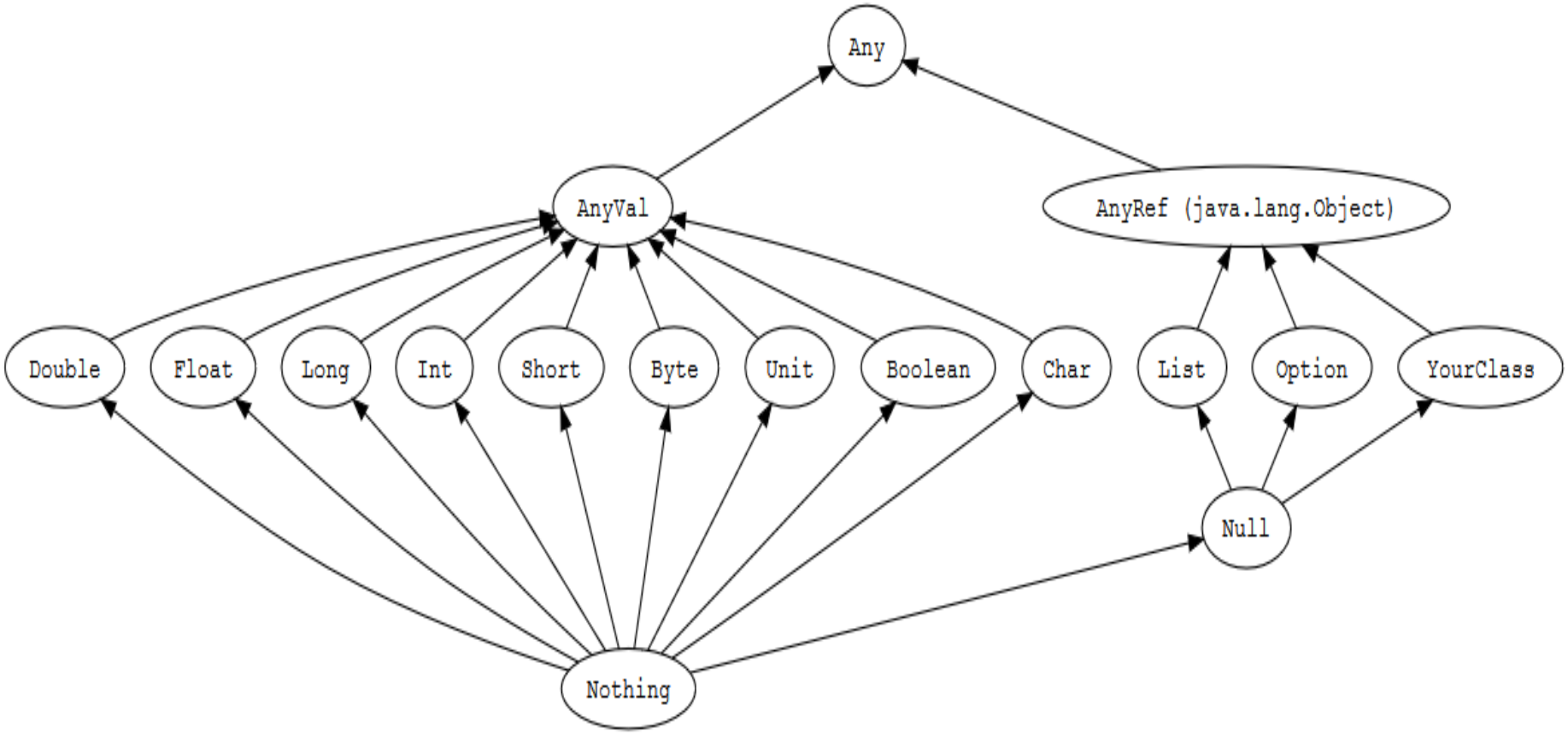
Les sous-classes héritent des membres (méthodes ou propriétés) de leurs classes parentes

Variables and Primitives in Scala

Hiérarchie des classes Scala pour les classes de valeur ou value classes.



Variables and Primitives in Scala



Ref: https://docs.scala-lang.org/resources/images/tour/unified-types-diagram.svg

Int Methods

Exemple :

```
scala> 46
res0: Int = 46

scala> 46.toString()
res1: String = 46

scala> 46 == 47
res2: Boolean = false

scala> 46 != 47
res3: Boolean = true
```

Int Methods

Exemple :

```
scala> 46.abs
res4: Int = 46

scala> 46.toString
res5: String = 46

scala> 46.toHexString
res6: String = 2e

scala> 46.isValidInt
res7: Boolean = true
```

Les opérations basiques sur Scala

Toute opération correspond à une méthode

```
> 0 to 2 // Ce qui est écrit
```

- `0.to(2) // Ce qui est exécuté (il existe une méthode 'to' dans la class Int)`

```
> x-1 // Ce qui est écrit
```

- `x.-(1) // Ce qui est exécuté (il existe une méthode '-' dans la class Int)`

```
0::liste // Ce qui est écrit

> liste.::(1) // Ce qui est exécuté (il existe une méthode '::' dans la class Int)
```

Les Variables Mutable & Immutable

Variable Mutable : Modifiable

Variable Immutable : Non Modifiable

Les Variables Mutable & Immutable

Exemple :

```
scala> var mutablevar = 21
mutablevar: Int = 21

scala> mutablevar = mutablevar + 1
mutablevar: Int = 22

scala> val immutablevar = 21
immutablevar: Int = 21

scala> immutablevar = immutablevar + 1
:21: error: reassignment to val

scala> val immutablevar2 = immutablevar + 1
immutablevar2: Int = 22
```

Scala

Les boucles

La boucle while:

```
while (CONDITION) EXPRESSION
```

Exemple :

```
var i : Int = 0
while(i < 10) {
    println(i)
    i = i+1
}
```

Les boucles

La boucle for :

```
for (i <- MIN to MAX) EXPRESSION
```

Exemple :

```
for(i <- 0 to 9)
    println(i)
```

Les fonctions

Exemple :

```
def max(x : Int, y : Int) = if (x > y) x else y
```

// Equivalents :

```
def opp(x : Int) : Int = -x
def opp(x : Int) : Int = { return -x; }
```

Fonctions anonymes

```
// On crée une liste
val liste = List( "bleu" , "rouge" , "vert" , "blanc")
liste: List[String] = List(bleu, rouge, vert)

// On filtre la liste à partir d'un prédicat « commence par un 'b' »
liste.filter( s => s.startsWith("b"))
res42: List[String] = List(bleu,blanc)

// On compte combien de mots correspondent au prédicat « commence par un 'b' »
liste.count( s => s.startsWith("b"))
res44: Int = 2
```

Les procedures

Évitez la syntaxe de la procédure, car elle a tendance à être source de confusion.

```
// n'écrivez pas de cette façon
def sayHello(aQui: String) {
        println(s"hello $aQui")
}
// Utilisez Unit
def sayHello(aQui: String): Unit = {
        println(s"hello $aQui")
}
```

Currying

Fonction où seuls quelques arguments sont spécifiés

- Il faut fournir les autres au programme

Concepts courants dans les langage fonctionnels

Currying

Exemple :

```
// La fonction prend deux paramètre et retourne 'vrai' si le reste de la division est null
> def nDividesM(m : Int)(n : Int) = (n % m == 0)
nDividesM: (m: Int)(n: Int)Boolean

//La fonction suivante fixe un paramètre à 2 et attends le deuxième
> val isEven = nDividesM(2)_
isEven: Int => Boolean = <function1>

// revient à appeler nDividesM(2,4)
> println(isEven(4))
True

// revient à appeler nDividesM(2,5)
> println(isEven(5))
false
```

Data Structures

Scala prend en charge plusieurs structures de données communes appelées collections.

Les collections peuvent être considérées comme des conteneurs d'objets, contenant généralement les données

Certains types de collection peuvent être modifiables ou immuables.

Les collections prises en charge dans Scala incluent des listes, sets, tuples et maps.

Les tableaux

Les éléments d'un tableaux sont **mutables**

```
val a = new Array[String](2) // On initialise un tableau
a: Array[String] = Array(null, null)

a(0) = "Java"
a(1) = "rocks"

a(0) = "Scala" // On modifie le premier élément d'un tableau

a // On affiche le tableau
res22: Array[String] = Array(Scala, rocks)
```

Lists

Les listes dans Scala sont des ensembles d’éléments linéaires.

Les listes contiennent un nombre arbitraire d'éléments (zéro ou plus)

Lists

Exemple :

Listes avec un seul type

```
val listofints = List(1, 2, 3)
listofints: List[Int] = List(1, 2, 3)
```

Listes avec types mixtes

```
val listofanys = List("Jeff", "Aven", 46)
listofanys: List[Any] = List(Jeff, Aven, 46)
```

Lists

Exemple :

Déclarer des listes et utiliser des fonctions

```
val listofints = 1 :: 2 :: 3 :: Nil
listofints: List[Int] = List(1, 2, 3)

listofints.filter(_ > 1)
res0: List[Int] = List(2, 3)
```

Accéder aux éléments d'une liste

```
scala> listofints(0)
res2: Int = 1
```

Scala

Lists

Modifier une liste

Les **listes** dans scala sont **immuables**, ce qui signifie que vous ne pourrez pas les modifier.

Vous pouvez cependant créer de **nouvelles listes** en **ajoutant** les objets existants à votre **liste** à l'aide des opérateurs
: : et : : :

Lists

Exemple :

```
var listofints = 1 :: 2 :: 3 :: Nil
listofints: List[Int] = List(1, 2, 3)

val listofints2 = listofints ::: 4 :: Nil
listofints2: List[Int] = List(1, 2, 3, 4)

val listofints3 = 0 :: listofints2
listofints3: List[Int] = List(0, 1, 2, 3, 4)

val listofints4 = listofints3 ::: 5 :: Nil
listofints4: List[Int] = List(0, 1, 2, 3, 4, 5)
```

Sets

Les Sets en Scala contiennent un ensemble d'éléments uniques.

Contrairement aux listes, les ensembles ne contiennent pas de doublons.

Les Sets ne sont pas ordonnés, vous ne pouvez pas accéder à leurs éléments par position.

Sets

Exemple :

```
val setofints1 = Set(1,2,3,4)
setofints: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

val setofints2 = Set(4,5,6,7)
setofints: scala.collection.immutable.Set[Int] = Set(4, 5, 6, 7)

val setofints3 = Set(1,2,3,4,4)
setofints: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
```

Sets

Un Set est une collection bien définie d'objets distincts.

Les Sets fournissent des méthodes pour le min, max, count, find et filter, ainsi que les opérations telles que diff, intersect et union

Sets

Exemple :

```
setofints1.min
res0: Int = 1

setofints1.max
res1: Int = 4

setofints1.intersect(setofints2)
res2: scala.collection.immutable.Set[Int] = Set(4)

setofints1.union(setofints2)
res3: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 7, 3,
4)

setofints1.diff(setofints2)
res4: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

Sets

Les Sets sont **immuables** mais peuvent être **mutables** en les spécifiant avec **scala.collection.mutable.Set** explicitement.

Sets

Exemple :

```
var mutableset = scala.collection.mutable.Set(1,2,3,4)
mutableset: scala.collection.mutable.Set[Int] = Set(2, 1, 4, 3)

mutableset += 5
res0: scala.collection.mutable.Set[Int] = Set(2, 1, 4, 3, 5)
```

Scala

Tuples

Les tuples sont des ensembles ordonnés de valeurs;

vous pouvez penser aux tuples comme des enregistrements d'une table d'une base de données relationnelle, où chaque élément ou valeur peut être référencé par position.

La position de la valeur dans le tuples a une certaine pertinence, contrairement à la position d'un élément dans une liste.

Les tuples peuvent contenir des objets .

De même que les listes, les tuples sont des structures de données immuables.

Tuples

Exemple

```
val a_tuple = ("Jeff", "Aven", 46)
a_tuple: (java.lang.String, java.lang.String, Int) =
(Jeff,Aven,46)

val another_tuple = new Tuple3("Jeff", "Aven", 46)
another_tuple: (java.lang.String, java.lang.String, Int) =
(Jeff,Aven,46)
```

Notez dans le dernier exemple que j'ai utilisé le nom de classe Tuple3;

si le tuple contenait quatre éléments, j'aurai utiliser le nom de classe Tuple4, et ainsi de suite.

Scala

Tuples

Après avoir créé un tuple, vous pouvez accéder à n'importe quel champ en position.

Contrairement aux listes ou aux tableaux qui se basent sur zéro pour le premier élément

- La position de l'élément dans un tuple est à base unique (ce qui signifie qu'on commence avec 1).
- Vous pouvez également déclarer les noms de champs et accéder aux éléments par leur nom

Tuples

Exemple :

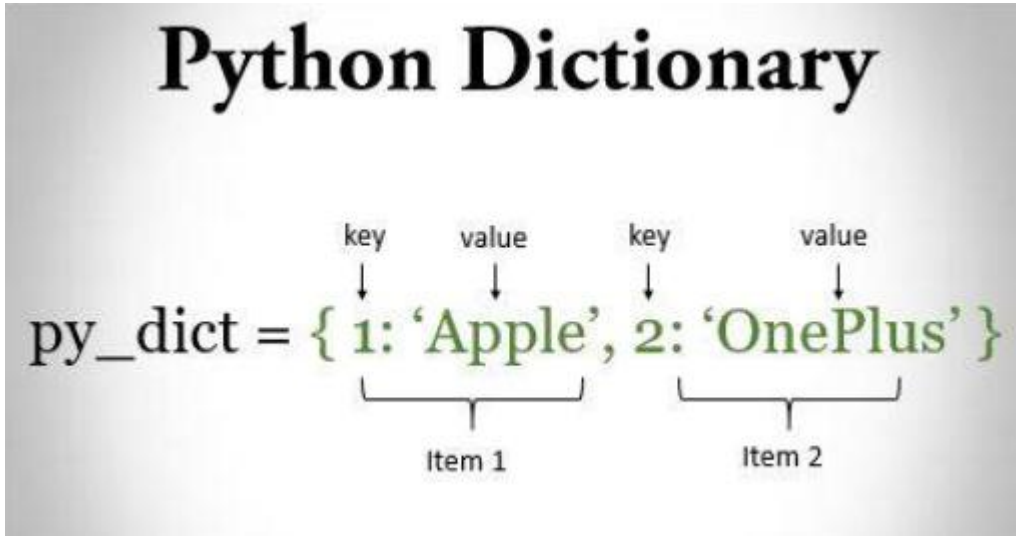
```
val a_tuple = ("Jeff", "Aven", 46)
a_tuple: (java.lang.String, java.lang.String, Int) = (Jeff,Aven,46)

println(a_tuple._1)
Jeff

val(fname, lname, age) = a_tuple
fname: java.lang.String = Jeff
lname: java.lang.String = Aven
age: Int = 46

fname
res1: java.lang.String = Jeff
```

Scala

Maps

Les Maps sont des ensembles de paires de clés/valeurs.

Similaires aux dictionnaire en Python.



Maps

Les clés dans un Map sont uniques et toute valeur peut être récupérée en fonction de sa clé.

Les valeurs dans un Map peuvent ne pas être uniques.

Les Maps possèdent des **méthodes** pour retourner une collection contenant des clés ou des valeurs

Maps

Exemple :

```
val a_map = Map("fname" -> "Jeff","lname" -> "Aven","age" -> 46)
a_map: scala.collection.immutable.Map[java.lang.String,Any] =
Map(fname -> Jeff, lname -> Aven, age -> 46)

a_map.keys
res0: Iterable[java.lang.String] = Set(fname, lname, age)

a_map.values
res1: Iterable[Any] = MapLike(Jeff, Aven, 46)
```

Maps

Les Maps sont **immutables** par défaut mais peuvent être **mutables** avec la classe **scala.collection.mutable.Map**.

```
var mutablemap = scala.collection.mutable.Map("fname" ->
"Jeff","lname" -> "Aven","age" -> 46)
mutablemap: scala.collection.mutable.Map[java.lang.String,Any] =
Map(fname -> Jeff, age -> 46, lname -> Aven)

mutablemap.keys
res0: Iterable[java.lang.String] = Set(fname, age, lname)

mutablemap += ("city" -> "Melbourne")
res1: mutablemap.type = Map(city -> Melbourne, fname -> Jeff, age ->
46, lname -> Aven)

mutablemap.keys.foreach(x => println(x))
city
fname
age
lname
```

Scala

Maps

Attention :

- ✓ Map # map
- ✓ Map (Classe) # map(Méthode)

Scala

REPL

Read-Eval-Print loop

Ligne de commande permettant de faire du « pas à pas »

Parfait pour l'apprentissage et la mise au point

Pattern matching

Comme le switch

Puissant

Pattern matching

Scala `match` v Java `switch`

match

```
someFlag match {
   case 1 | 2 => doSomething()
   case 3 => doSomethingElse()
   case _ => doSomethingDefault()
}
```

switch

```
switch (someFlag) {
   case 1:
   case 2:
      doSomething();
      break;
   case 3:
      doSomethingElse();
      break;
   default:
      doSomethingDefault();
}
```

Pattern matching

Exemple :

```
import scala.util.Random

val x: Int = Random.nextInt(10)
x: Int = 6

def matchTest(x:Int): String = x match {
case 0 => "zero"
case 1 => "one"
case 2 => "two"
case _ => "many"
}

matchTest(3)
res55: String = many
```

Classes

Une **classe** est un type permettant de regrouper dans la même structure :

- ✓ les informations : **champs**, propriétés, attributs
- ✓ les procédures et fonctions : **méthodes**

Classes

La classe est un type structuré

Va plus loin que l'enregistrement

Les champs d'une classe peuvent être de type quelconque

Ils peuvent faire référence à d'instances d'autres classes

Classes

Termes techniques :

- ✓ **Classe** est la structure
- ✓ **Objet** est une instance de la classe (variable obtenue après **instanciation**)
- ✓ **Instanciation** correspond à la création d'un **Objet**

Classes

Les paramètres deviennent des membres publics à la classe

Exemple :

```
class Personne(val name: String, var age:Int) {} // Une classe simple
defined class Personne

var p1 = new Personne("Stephane",32) // Un objet
p1: Personne = Personne@62878d7c

var p2 = new Personne("Sophie",15) // Un autre objet
p2: Personne = Personne@1f7949dc

p2.age = 15 // Accès direct à un membre
```

Classes

Définir une classe

On peut définir une classe avec simplement le mot-clé **`class`** et un identifiant.

```
class Personne
```

```
val user1 = new Personne
```

`new` est utilisé pour créer une instance d'une Classe.

`Personne` a un constructeur par défaut qui ne prend aucun argument car aucun constructeur n'a été défini.

Classes

Pour ajouter un constructeur

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

  def move(dx: Int, dy: Int): Unit = {
    x = x + dx
    y = y + dy
  }
}

val point1 = new Point(2, 3)
point1.x  // 2
point1.y  // 3
```

Classes

Les constructeurs peuvent avoir des paramètres optionnels en fournissant une valeur par défaut.

```
class Point(var x: Int = 0, var y: Int = 0)

val origine = new Point  // x et y initialisés à 0
val point1 = new Point(5)
println(point1.x)  // prints 5

val point2 = new Point(y=2)
println(point2.y)  // prints 2
```

case class

Définir une **case classe**

On peut définir une case classe avec simplement le mot-clé **`case class`** et un identifiant plus un paramètre.

```
case class Book(isbn: String)

val frankenstein = Book("978-0486282114")
```

case class

Notez que le mot clé **new** n'a pas été utilisé pour instancier la case class Book.

Les **case class** ont une méthode **`apply`** par défaut qui prend en charge la construction de l'objet.

Lorsque vous créez une **case class** avec des paramètres, ceux-ci sont par défaut en **val** public.

case class

Exemple :

```
case class Message(sender: String, recipient: String, body: String)

val message1 = Message("guillaume@gmail.com", "pierre@gmail.com", "Ça va
?")

println(message1.sender)  // prints guillaume@gmail.com
message1.sender = "fred@gmail.com"  // this line does not compile
```

case class

Exemple :

```
case class Message(var sender: String, recipient: String, body: String)

val message1 = Message("guillaume@gmail.com", "pierre@gmail.com", "Ça va
?")

println(message1.sender)  // prints guillaume@gmail.com
message1.sender = "fred@gmail.com"  // ok
```

Programmation Fonctionnelle

La programmation fonctionnelle est de plus en plus répandue dans l'industrie.

Cette tendance est motivée par l'adoption de Scala comme principal langage de programmation pour de nombreuses applications.

Scala fusionne la **programmation fonctionnelle** et **orientée objet** dans un package pratique.

Il interagit de manière transparente avec Java et Javascript.

Scala est le langage d'implémentation de nombreux frameworks importants, notamment Apache Spark, Kafka .

Programmation Fonctionnelle

```java
public List<Product> getProductsByCategory(String category) {
    List<Product> products = new ArrayList<Product>();
    for (Order order : orders) {
        for (Product product : order.getProducts()) {
            if (category.equals(product.getCategory())) {
                products.add(product);
            }
        }
    }
    return products;
}
```

Java

```scala
def productsByCategory(category: String) = orders.flatMap(o => o.products).filter(p =>
p.category == category)
```

Scala

Programmation Fonctionnelle

Avantage de Scala → c'est qu'il combine POO et PF

Il cumule ainsi les avantages :

- De la POO : encapsulation, abstraction…
- De la PF : immutabilité, fonctions pures…

Programmation Fonctionnelle

Les **fonctions** : issue de la PF.

On parle ici de fonctions pures qui ne font que prendre des paramètres en entrée et sortir un résultat.

Il s'agit d'une valeur à laquelle on attache un comportement.

Programmation Fonctionnelle

```
val double = (x: Int) => x * 2

println(double(4)) // 8
```

Programmation Fonctionnelle

Les **méthodes** ou **fonctions** : issue de la POO.

Il s'agit de fonctions, généralement (mais pas toujours) plus complexes.

Programmation Fonctionnelle

```
def triple(x: Int) : Int = x * 3

println(triple(5)) // 15
```

Programmation Fonctionnelle

Il est facile de faire un amalgame entre les fonctions de la PF et POO tant elles sont similaires.

Quelles différences à prendre en compte ?

Programmation Fonctionnelle

Les fonctions de la PF ne sont évaluées qu'une seule fois, tandis que les méthodes/fonctions POO sont évaluées à chaque appel.

Ainsi, les fonctions PF sont en général plus performantes que les méthodes.

Programmation Fonctionnelle

Les fonctions PF ne sont que des valeurs : elles peuvent être composées et passées en paramètre d'autres fonctions. C'est un des principes fondamentaux de la programmation fonctionnelle.

À noter que les méthodes/fonctions POO ne sont pas exclusives à la POO et peuvent être utilisées dans un environnement fonctionnel.

Programmation Fonctionnelle

```
def sum(callback : (Int) => Int, range : Int) = {
    var i : Int = 0
    var res : Int = 0

    for(i <- 1 to range) {
        res += callback(i)
    }

    res
}

println(sum(double, 10)) // 110
```

Programmation Fonctionnelle

```
def sum(callback : (Int) => Int, range : Int) = {
    var i : Int = 0
    var res : Int = 0

    for(i <- 1 to range) {
      res += callback(i)
    }

    res
}

println(sum(double, 3)) // 12
```

Programmation Fonctionnelle

Les fonctions PF ne sont que des valeurs : elles peuvent être composées et passées en paramètre d'autres fonctions. C'est un des principes fondamentaux de la programmation fonctionnelle.

À noter que les méthodes/fonctions POO ne sont pas exclusives à la POO et peuvent être utilisées dans un environnement fonctionnel.

Les Structures

Les Structures

Scala propose 4 types de structures :

- **Les classes**
- **Les case classes**
- **Les traits**
- **Les objets**

Programmation Fonctionnelle

- **Les classes**

  Issue de la POO.

  Elles permettent d'encapsuler des données (valeurs et variables) ainsi que des comportements (fonctions et méthodes).

  Elles permettent de créer des instances mutables.

Programmation Fonctionnelle

- **Les case classes**

  Issue de la PF.

  Il s'agit de classes qui ne contiennent que des valeurs.

Programmation Fonctionnelle

- **Les traits**

  Il s'agit de structures pouvant contenir des données et des comportements, mais qui ne sont pas instanciables.

  Elles ont pour seul et unique but d'être **hérités** par des **classes**.

Scala

Programmation Fonctionnelle

- **Les traits**

Les données ou comportements peuvent avoir une implémentation, auquel cas le trait joue le rôle d'une classe abstraite ou/et interface.

https://www.w3schools.com/java/java_abstract.asp

https://www.w3schools.com/java/java_interface.asp

Programmation Fonctionnelle

- **Les traits**

```
trait Hamburger {
    // Deux variable définie
    val breadType : String = "Pain à burger"
    val steakType : String = "Boeuf"

    // Une valeur nécéssitant d'être définies
    val cheeseType : String

    // Une méthode définie
    def eat(): Unit = println("Eating...")

    // Une méthode nécéssitant d'être définie
    def waste(): Unit
}
```

Programmation Fonctionnelle

- **Les traits**

```
class BigGreen(pCheeseType : String, pSaladType :
String, pTomatoSlices : Int, pPicklesSlices : Int,
pOnionSlices : Int) extends Hamburger {
    override val steakType : String = "None"
    val saladType : String = pSaladType
    val tomatoSlices : Int = pTomatoSlices
    val picklesSlices : Int = pPicklesSlices
    val onionSlices : Int = pOnionSlices
    val cheeseType : String = pCheeseType

    def waste(): Unit = println("Wasted :(")
}
```

Programmation Fonctionnelle

- **Les traits**

```
val bigGreen = new BigGreen("Emmental", "Roquette", 2, 3, 4)
println(bigGreen.breadType) // "Pain à burger"
println(bigGreen.steakType) // "None"
println(bigGreen.saladType) // "Roquette"
println(bigGreen.cheeseType) // "Emmental"
bigGreen.eat() // "Eating..."
bigGreen.waste() // "Wasted :("
```

Programmation Fonctionnelle

- **Les objets**

  À ne pas confondre avec les instances de classes.

  Il s'agit de **singletons** principalement utilisés pour grouper les données et comportements statiques.

  Ils ne sont pas instanciables (pas de new) et s'appellent directement par leur nom « comme les classes statiques en Java ».

Programmation Fonctionnelle

- **Les objets**

  L'objet singleton est un objet qui est déclaré en utilisant le mot-clé objet à la place de la classe.

  Scala crée donc un objet singleton pour fournir un point d'entrée pour l'exécution de votre programme.

Programmation Fonctionnelle

- **Les objets**

```
object PersonneF{
        val sexe = "F"
        def getSexe = sexe
}


PersonneF.getSexe
```

Programmation Fonctionnelle

- **Les objets**

  Dans Scala, les objets singleton peuvent partager le nom d'une classe correspondante.

  Dans un tel scénario, l'objet singleton est appelé objet compagnon .

Programmation Fonctionnelle

- **Les objets**

  Par exemple, sous une classe **Factorial** on définit un objet compagnon (également nommé **Factorial**).

  Par convention, les objets compagnons sont définis dans le même fichier que leur classe compagnon.

Scala

Programmation Fonctionnelle

- **Les objets**

```
class HelloW{
      def sayHelloWorld(){
      println("Hello World");
}}


object HelloW{
     def sayHi(){
      println("Hi!");
      }
 }

HelloW.sayHi //Hi!

val maVar = new HelloW()
maVar.sayHelloWorld //Hello
World
```

SBT

Installation SBT