

# Formation Scala

Olivier GIRARDOT

Lateral-Thoughts

15 Janvier 2017

# Qui suis-je ?

## En quelques mots

- Développeur Java, Scala, Python
- Expert Big Data - Apache Spark, Elasticsearch, Kafka

## Mon expérience en Scala

- Code en scala depuis 2011
- Plusieurs projets en scala (Applications web, Datalake, Clients lourds...)
- Speaker - Formateur

# Qui êtes-vous ?

## En quelques mots

- Prénom
- Expérience en Java
- Utilisation de Java 8/Collection Guava

## Sur la formation

- Des attentes particulières pour la formation ?
- Des questions que vous vous posez sur Scala ?

# Jour 1 : Bases de Scala

## Ce que vous allez apprendre

- Qu'est-ce que Scala et pourquoi c'est utilisé
- Faire un Hello World en Scala
- Qu'est ce que le REPL
- Programmation impérative en Scala
- Modèle Objet de Scala
- Programmation fonctionnelle en Scala
- Le pattern matching

## Jour 2 : API en Scala (1)

### Ce que vous allez apprendre

- Les opérations sur les collections
- For Comprehension
- Les types algébriques
- Concurrency
- Comparaison Java 8/Scala

## Jour 3 : API en Scala (2)

### Ce que vous allez apprendre

- Les opérations sur les collections
- For Comprehension
- Les types algébriques
- Concurrency
- Comparaison Java 8/Scala

## Section 2

# Qu'est ce que Scala

# Qu'est ce que Scala ?

## En quelques mots

- Un langage de programmation multiparadigme (fonctionnel et objet)
- Un langage de programmation compilant en Bytecode JVM
- Un langage de programmation avec typage statique

## Status du langage

- Version actuelle : 2.12.4
- Site web : <http://www.scala-lang.org/>



# Scala

## Quelques faits

- Apparue en 2004
- Inventé par Martin Odersky
- Soutenu par Typesafe / Lightbend

## But de Scala

- Avoir un langage statiquement typé qui unifie et généralise l'orienté objet avec la programmation fonctionnelle
- Avoir un langage qui favorise les abstractions et la composition afin de pouvoir coder de la même manière une petite ou une grosse application

# Un langage fonctionnelle et objet

## Un langage fonctionnel

- Immutabilité
- Fonctions traitées comme des valeurs

## Un langage objet

- Système de classe avec méthode et données
- Polymorphisme ; héritage et génériques

# Un langage sur la JVM

## Compile en bytecode

- Permet d'utiliser Scala partout où il y a une JVM

## Intercompatibilité avec Java

- Utilisation de bibliothèque Java en Scala
- Mapping entre les types Java et les types Scala

# Un exemple de programme Scala

## Listing 1: Exemple de Programme en Scala

```
1 object ExampleProgramInScala extends App {  
2  
3   val computator = new Computator(1,2)  
4   println("sum of 1 + 2 :" + computator.compute())  
5  
6   val aList = 0::List(1,2,3)  
7   println("A sum of list : " + aList.map(_ + 1).foldLeft(0)((x, y) => x + y))  
8  
9 }  
10  
11 class Computator(x:Int, y:Int) {  
12  
13   def compute() = x + y  
14  
15 }
```

# Utilisateurs

## Ils utilisent Scala

- Twitter : recherche d'utilisateurs, API de streaming, interaction avec le Social Graph
- Netflix : API en scala
- La Poste : Process industriels et Datalake
- AXA : Applications Web (Lift), Datalake
- MFGLabs (Havas) : Data-Mining, Applications web
- ...

# L'écosystème Scala

## Les bibliothèques Scala

- Le système de build : SBT
- Pour les applications web : Play! Framework 2
- Pour les API REST : Play! Framework 2, Akka HTTP
- Pour la gestion de la concurrence : Akka
- Pour le calcul distribué : Spark

# Pourquoi utiliser Scala ?

- Une syntaxe plus légère que Java
- Toute la puissance de la programmation fonctionnelle
- Un langage adapté à la gestion de la concurrence
- Tout Java accessible

# Ressources

## Quelques liens

- Le site officiel : <http://www.scala-lang.org/>
- Le site de Lightbend : <https://www.lightbend.com/>
- Javadoc : <http://www.scala-lang.org/api/current/#package>

## Quelques livres

- *Programming in Scala* par Martin Odersky, Lex Spoon, Bill Verner
- *Scala in Depth* par Joshua Suereth
- *Functionnal Programming in Scala* par Paul Chiusano et Runar Bjarnason



# Bilan

## Ce que vous avez vu

- Qu'est-ce que Scala
- Avantages de Scala
- Qui utilise Scala
- L'écosystème Scala
- Des ressources pour Scala

## Section 3

# Mon premier programme Scala

# Le REPL Scala

## Qu'est ce que le REPL ?

- REPL veut dire Read-Eval-Print Loop
- C'est une console permettant de lancer des instructions Scala

## Comment lancer le REPL

- Lancer la VM
- Ouvrir une console
- Taper **scala** ou **sbt console**

```
Welcome to Scala version 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

# Exemples dans le REPL scala

```
scala> 1+2
res0: Int = 3

scala> res0 + 4
res1: Int = 7

scala> def function(x:Int) = x + 1
function: (x: Int)Int

scala> function(2)
res2: Int = 3

scala>
```

# Astuces dans le REPL

## Pour sauter des lignes

SHIFT + ENTER

## Quelques commandes utiles

- **:help** L'aide avec la liste des commandes
- **:quit** Quitter
- **:history** L'historique des commandes tapées

# Ammonite : Un REPL plus complet

## Limitations du REPL standard

- Pas de coloration syntaxique
- On ne peut pas éditer les lignes déjà tapées lorsqu'on fait du multiligne
- On ne peut pas importer des bibliothèques externes directement

## Solution : Ammonite REPL

<http://lihaoyi.github.io/Ammonite/#Ammonite-REPL>

```
10:38:01 vincent: ~ % amm
Loading...
Welcome to the Ammonite Repl 0.5.0
(Scala 2.11.7 Java 1.8.0_66)
@ val x = 1
x: Int = 1
@ if (x > 0) {
  true
} else {
  false
}
res1: Boolean = true
@
```

# Quelques caractéristiques de Ammonite REPL

## Edition sur plusieurs lignes

```
@ if (1 > 0) {  
  true  
} else {  
  false  
}
```

## Import de bibliothèques

```
load.ivy("org.scalaz" %% "scalaz-core" % "7.1.1")
```

## Quelques commandes utiles

- **help** L'aide avec la liste des commandes
- **quit** Quitter
- **history** L'historique des commandes tapées

# Installation

## Installation

```
curl -L -o amm http://git.io/vBTzM  
chmod +x amm
```

## Lancement

```
./amm
```



# Hello World

## Hello World en Java

### Listing 2: Hello World en Java

```
1 public class HelloWorld {  
2  
3     public static void main(String[] args) {  
4         System.out.println("Hello World");  
5     }  
6 }
```

## Hello World en Scala

### Listing 3: Hello World en Scala

```
1 object HelloWorld extends App {  
2     println("Hello World")  
3 }
```

# Bilan

## Ce que vous avez vu

- Le REPL (Read-Evaluate-Print Loop)
- Lancer le REPL Scala
- Jouer avec le REPL
- Ammonite REPL, un REPL plus complet en Scala
- Hello World en Scala

## Section 4

# Les Bases de la programmation en Scala

# Le Main

## La base du programme

### Listing 4: Un Main Vide

```
1 object FizzBuzz extends App {  
2  
3 }
```

---

## Commentaires

- Un object est un singleton, il permet notamment de stocker des méthodes statiques
- Il étend le trait (l'équivalent d'une Interface Java) App, qui permet d'éviter d'avoir à déclarer une fonction main, tout sera tapé dans le corps de l'object

# Les déclarations

```
1 val x = 3
2 val text:String = "hello"
3 var i = 2
4 i = i + 1
5 def addOne(x:Int) = x + 1
```

## Commentaires

- var et val déclarent des variables, def déclare des fonctions
- var déclare des variables mutables, on peut les modifier une fois déclarées (comme en Java), val déclare des variables immutables, on ne peut pas les modifier une fois déclarées
- le type est déclaré après le nom de la variable, et non avant comme en Java
- le type peut être inféré, c'est à dire qu'on n'a pas besoin de le déclarer explicitement comme en Java

# Les types de données

```
1 val aBoolean:Boolean = true
2 val anInt:Int = 3
3 val aDouble:Double = 3.0
4 val aText:String = "coucou"
5 val aList:List[Int] = List(1,2,3)
6 val anArray:Array[Int] = Array(1,2,3)
7 val aSet:Set[Int] = Set(1,2,3)
8 val aMap:Map[Int,String] = Map(1 -> "a", 2 -> "b", 3 -> "c")
```

## Commentaires

- Les types sont les mêmes que les types Java
- Les types génériques sont déclarés avec des crochets
- On n'a pas besoin de new pour les types comme List, Array, Set ou Map

# Output

```
1 println("coucou")
2 println(3)
3 print("coucou"); print("beuh")
4 println("coucou"); println("beuh")
```

---

## Commentaires

- `println` permet d'afficher une valeur et de retourner à la ligne, c'est l'équivalent de `System.out.println()`
- `print` affiche la valeur mais ne retourne pas à la ligne
- on peut passer n'importe quel objet en argument à `print` et à `println`
- pour séparer plusieurs instructions sur une même ligne, on utilise le point-virgule

# Les fonctions

```
1 def sum(x:Int, y:Int) = x + y
2 def display(text:String) { println(text) }
3 def addOne(x:Int) = {
4     x + 2
5     x + 1
6 }
7 def addTwo(x:Int):Int = x + 2
```

## Commentaires

- les arguments sont séparés par une virgule, avec le type déclaré après le nom de l'argument
- il n'y a pas besoin de return, c'est la dernière instruction exécutée qui détermine le type
- les fonctions qui ne retournent rien sont de type de retour `Unit` (équivalent à `void` en Java)



# Conditionnel if/then/else

```
1 val absOfX = if (x > 0) x else -x
2 if (y > 0) {
3   println(y)
4   -y
5 } else {
6   y
7 }
```

---

## Commentaires

- on n'a pas besoin d'accolades pour faire un if/then/else, et le mot clé then n'existe pas
- on peut attribuer le résultat d'un if/then/else à une variable
- lorsqu'il y a plusieurs lignes, on doit utiliser les accolades

# Les opérateurs sur les Booléens et les Entiers

```
1 true && false
2 true || false
3 true == false
4 !true
5
6 2 + 2
7 2 * 2
8 2 / 2
9 2 - 2
10 2 % 2
```

---

## Commentaires

- Ce sont les mêmes opérateurs qu'en Java !

# Boucle for

```
1 var j = 0;
2 while (j < 10) {println(j);j = j+1}
3 for (j <- 0 to 10) {println(j)}
4 for (j <- List("a","b","c")) println(j)
```

---

## Commentaires

- L'opérateur ++ n'existe pas (en fait il fait autre chose, la concaténation de listes)
- Pour itérer sur un set d'éléments, on utilise l'opérateur <-
- Encore une fois, les accolades ne sont pas obligatoires

# FizzBuzz

## Itération 1

- Afficher les nombres de 1 à 100

## Itération 2

- Dans la boucle qui affiche les nombres, remplacer le nombre affiché par "Fizz" lorsque celui-ci est divisible par 3

## Itération 3

- Dans la boucle qui affiche les nombres, remplacer le nombre affiché par "Buzz" lorsque celui-ci est divisible par 5

## Itération 4

- Dans la boucle qui affiche les nombres, remplacer le nombre affiché par "FizzBuzz" lorsque celui-ci est divisible par 3 et 5

## Autres exercices disponibles

### Exercices simples

- **Savings** : Calculer combien d'année il faut pour atteindre une somme donnée en partant d'une somme de départ et d'un taux d'intérêt
- **Nombre d'entiers impairs** : Compter le nombre d'entiers impairs dans une liste d'entiers donnée.
- **Trim de String** : Supprimer tous les espaces en tête et en queue d'une chaîne de caractères

### Exercices plus avancés

- **Syracuse** : Trouver la plus longue séquence nécessaire pour revenir à 1 en appliquant la suite de Syracuse pour un nombre compris entre 1 et 100
- **Eratosthène** : Afficher tous les nombres premiers entre 1 et 100
- **Fibonacci** : Afficher le centième élément de la suite de Fibonacci

# Imports

## En Java

```
1 import com.package.MyClass
2 import com.package.*
```

## En Scala

```
1 import com.package.MyClass
2 import com.package._
```

## Commentaires

- Les imports fonctionnent de la même manière en Scala et en Java
- La seule différence, c'est que pour importer toutes les classes d'un paquet, on utilise `._` en Scala alors qu'on utilise `.*` en Java

# Input de l'utilisateur

```
1 import scala.io.StdIn.readLine
2 val input:String = readLine("taper une chaine de caracteres > ")
3 import scala.io.StdIn.readInt
4 val inputInt:Int = readInt()
```

---

## Commentaires

- Pour l'input on a besoin d'importer des fonctions
- On peut lire et retourner différents types
- Seule la fonction qui lit et retourne une chaîne de caractères permet de taper un message de prompt

# C'est plus/C'est moins

## Les règles du jeu

- L'ordinateur génère un nombre aléatoire entre 1 et 100,
- L'utilisateur doit ensuite le deviner
- À chaque fois que l'utilisateur tape un mauvais nombre, l'ordinateur lui répond si le nombre à deviner est supérieur ou inférieur au nombre que l'utilisateur a tapé
- Une fois que l'utilisateur a trouvé le nombre, l'ordinateur lui affiche un message disant combien de fois l'utilisateur a essayé de deviner le chiffre avant de trouver le bon



# Bilan

## Ce que vous avez vu

- Syntaxe de base de Scala
- Déclaration de fonctions en Scala
- La Programmation Impérative en Scala
  - If/Then/Else
  - Boucles

## Section 5

# Le Modèle Objet en Scala

# Déclaration d'une classe en Java

```
1 public class MyClass {
2     private final String name;
3     private final int number;
4
5     public MyClass(String name, int number) {
6         this.name = name;
7         this.number = number;
8     }
9
10    public String getName() {
11        return name;
12    }
13
14    public int getNumber() {
15        return number;
16    }
17 }
```

---

# Déclaration d'une classe en Scala

```
1 class MyClass(val name:String, val number:Int)
2
3 val myClassInstance = new MyClass("coucou",3)
4 myClassInstance.name
```

---

## Commentaires

- 17 lignes pour Java, une ligne pour Scala
- On déclare directement les champs dans la déclaration de la classe
- Le constructeur avec tous les champs est offert

# Déclaration de méthodes en Scala

```
1 class MyClass(val aField:String) {  
2   def myMethod:String = aField  
3   def myOtherMethod1:String = this.myMethod  
4   def myOtherMethod2 = myMethod  
5 }
```

## Commentaires

- Une méthode est une fonction dans une classe
- Si on veut faire référence à l'instance de l'objet dans une méthode, on utilise le mot-clé `this`
- Lorsqu'il n'y a pas d'ambiguïté, on peut omettre ce mot-clé

# Scope des méthodes en Scala

```
1 package myOuterPackage.myInnerPackage
2
3 class MyClass(val aField:String) {
4   private[this] def objectPrivateMethod = aField
5   private def classPrivateMethod = aField
6   private[myInnerPackage] def innerPackagePrivateMethod = aField
7   private[myOuterPackage] def outerPackagePrivateMethod = aField
8   protected def protectedMethod = aField
9   def publicMethod = aField
10 }
```

## Commentaires

- Nous avons les mêmes niveaux d'étendus qu'en Java : private, package, protected, public
- Il n'y a pas de mot-clé public, une méthode déclaré sans mot-clé est publique.
- Il y a deux niveaux en plus d'étendus en Scala : object-private et package-specific
  - object-private, le champ n'est accessible qu'à partir de l'instance d'où la méthode est appelée, et pas des autres instances de la classe
  - package-specific, c'est comme le niveau package, mais sur un package qui est au-dessus du package de la classe

# Scope des champs en Scala

```
1 class MyClass(  
2   privateImmutable1:String,  
3   private val privateImmutable2:String,  
4   protected val protectedImmutable:String,  
5   val publicImmutable:String,  
6   private var privateMutable:String,  
7   protected var protectedMutable:String,  
8   var publicMutable:String  
9 )
```

## Commentaires

- On peut ajouter des scopes sur les champs, comme sur les méthodes
- Un champ déclaré comme `val` sera immutable
- Un champ déclaré comme `var` sera mutable

# À vous de jouer

## Création de la classe Person

- La classe Person aura 3 champs
  - `name:String`, qui sera immutable et public
  - `gender:String`, qui sera mutable et object-private
  - `birthYear:Int`, qui sera immutable et private
- Elle aura aussi les méthodes suivantes
  - `isOlder(anotherPerson:Person):Boolean` : détermine si la personne est plus vieille qu'une autre personne, public
  - `getGender:String` : retourne le sexe de la personne, package private



# Héritage en Scala

```
1 class MyClass
2 class MyOtherClass extends MyClass
3
4 class MyClassWithField(aField:String)
5 class MyOtherClassWithFields1 extends MyClassWithField("coucou")
6 class MyOtherClassWithFields2(anotherField:String) extends MyClassWithField(
    anotherField)
```

## Commentaires

- Pour étendre une classe, on utilise le mot-clé `extends`
- Comme en Java, on ne peut hériter que d'une seule classe à la fois
- Si la classe parente a des champs, il faut aussi les initialiser lors de la déclaration de la classe enfant
- On peut soit les initialiser avec une constante, ou avec un champ de la classe enfant

# Relation Parent-Enfant

```
1 class MyClass {
2   def sayHello = "Hello"
3   def sayHelloToClass = sayHello + " MyClass"
4 }
5
6 class MyOtherClass extends MyClass {
7   override def sayHello = "coucou"
8   override def sayHelloToClass = super.sayHello + " MyOtherClass"
9 }
```

## Commentaires

- Pour surcharger une méthode, on est obligé d'utiliser le mot-clé `override`
- Contrairement à Java, où l'annotation `override` est optionnelle
- Si on veut appeler une méthode de la classe parente, on utilise le mot-clé `super`

# Classe Abstraite

```
1 abstract class MyClass {  
2   def sayHello:String  
3 }  
4  
5 class MyOtherClass extends MyClass {  
6   override def sayHello = "coucou"  
7 }
```

---

## Commentaires

- Pour déclarer une classe abstraite, il faut ajouter le mot-clé `abstract`
- Une classe abstraite peut avoir des méthodes non implémentées
- Dans ce cas, il faut que la méthode soit typée explicitement

# À vous de jouer

## Création de la classe VeryImportantPerson

- La classe VeryImportantPerson hérite de Person
- Les trois champs ne changent pas et sont les mêmes que pour Person
- La méthode isOlder sera surchargée et renverra toujours false
- La méthode getGender devient public et retourne la même chose que sa méthode parente

# Les traits

```
1 trait MyTrait
2 trait MyOtherTrait {
3   def display() {println("Hello")}
4 }
5 class MyClassWithTrait() extends MyTrait with MyOtherTrait
```

---

## Commentaires

- Les traits sont l'équivalent des interfaces en Java, en plus puissant
- On peut définir des fonctions dans les traits (un peu comme les implémentations defaults en Java 8)
- On implémente une classe avec plusieurs traits avec `extends` pour le premier trait puis `with` pour les traits suivants

# Héritage d'un trait et d'une classe

```
1 class MyClass
2 trait MyTrait
3 trait MyOtherTrait
4
5 class MyClassWithTrait extends MyTrait with MyOtherTrait
6 class MyInheritedClassWithTrait extends MyClass with MyTrait
```

## Commentaires

- On utilise le mot-clé `extends` pour hériter d'un trait
- Si l'on veut hériter en plus d'autres traits, on utilise le mot-clé `with`
- Il n'y a pas de distinctions dans la déclaration d'héritage entre les trait et les classes héritées
- Contrairement à Java où on a `extends` pour les classes et `implements` pour les interfaces

# Les objets

```
1 object MyObject {  
2   def display() {println("Hello")}  
3 }
```

---

## Commentaires

- Une classe objet est l'équivalent d'un singleton, on ne peut pas instancier un object
- C'est ici qu'on peut stocker les méthodes statiques
- Un objet peut étendre une classe et/ou un/des traits

# Companion Object

```
1 class MyClass(aField: String)
2
3 object MyClass {
4   def apply(aField: String) = new MyClass(aField)
5 }
```

---

## Commentaires

- Si un objet a le même nom qu'une classe, il est appelé companion object de cette classe
- Dans ce cas, il doit être déclaré dans le même fichier que la classe
- Il est surtout utilisé pour stocker des méthodes permettant de construire des instance de la classe
- Ici par exemple, il permet de créer des instances de MyClass sans utiliser le mot-clé `new`



# À vous de jouer

## Création du companion object de la classe Person

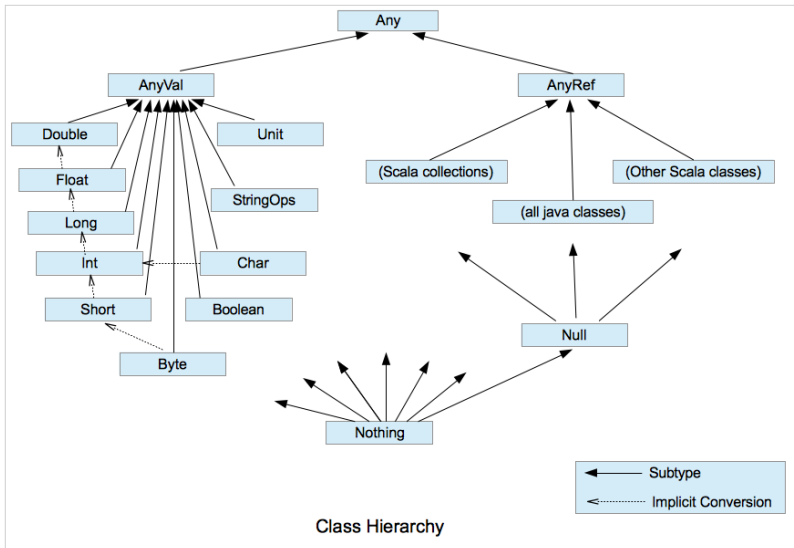
- Créer le companion object de la classe Person
- Ajouter une méthode `apply` permettant de créer une instance de Person sans le mot-clé `new`

# Quelques petites choses sur les classes scala

## Remarques

- Comme toute classe Java étend la classe `Object`, toute classe Scala étend la classe `Any`
- Il n'y a pas de types primitifs en Scala, tout est vraiment objet
- Il existe une classe `Nothing` qui est fille de toutes les autres classes, c'est l'équivalent de `null` en Java
- On peut mettre plusieurs classes/traits/objects dans le même fichier

# La hiérarchie de type



## Remarques sur la hiérarchie

- Tous les types Scala en rapport avec les types primitifs Java étendent `AnyVal`
- Tous les autres types Scala étendent la classe `AnyRef`, qui est la classe `Object` en Java
- Comme en Java, il y a des conversions implicites entre les différents types représentant des valeurs numériques
- La classe `StringOps` étend `AnyVal`, cette classe permet de faire des opérations que l'on peut trouver sur les séquences sur une chaîne de caractère en convertissant ladite chaîne en tableau de `Char`

# Les Véhicules

## Instructions

- Nous avons 3 véhicules : Moto, Bicycle, Car
- Chaque véhicule implémente la méthode `makeNoise`, qui affiche le bruit du véhicule : la voiture fait `VroumVroum`, la moto fait `BroumBroum` et le vélo fait `DringDring`
- Nous avons un seul conducteur, qui a une méthode `drive` qui prend en argument un véhicule et qui retourne le bruit que fait ce véhicule

# Implémentation Java

## Listing 5: Vehicle.java

```
1 public interface Vehicle {  
2     void makeNoise();  
3 }
```

---

## Listing 6: Car.java

```
1 public class Car implements Vehicle {  
2     @Override  
3     public void makeNoise() {  
4         System.out.println("VroumVroum");  
5     }  
6 }
```

---

## Listing 7: Driver.java

```
1 public class Driver {  
2     public static void drive(Vehicle vehicle) {  
3         vehicle.makeNoise();  
4     }  
5 }
```

---

# Pour aller plus loin

## Instructions

- Ajouter une méthode `fix` qui affiche "C'est réparé" pour tous les véhicules
- Ajouter une classe `Helicopter` qui implémente `Vehicle` et qui affiche "FlapFlap" lorsqu'on appelle la méthode `makeNoise`
- Ajouter une classe `Plane` qui implémente `Vehicle` et qui affiche "Vraoum" lorsqu'on appelle la méthode `makeNoise`
- Ajouter un trait `CanFly`, qui contient une méthode `takeOff` qui affiche "ça décolle". `Plane` et `Helicopter` doivent implémenter ce trait.

# Les génériques en Scala

## Génériques en Java

```
1 public class Generic<A, B> {  
2  
3     private A field;  
4  
5     public A execute(B argument) {  
6         return field;  
7     }  
8  
9     public <C> A anotherFunction(C argument) {  
10        return field;  
11    }  
12 }
```

---

## Génériques en Scala

```
1 class GenericScala[A,B](field:A) {  
2  
3     def execute(argument:B):A = field  
4  
5     def anotherFunction[C](argument:C):A = field  
6  
7 }
```

---



# Bilan

## Ce que vous avez vu

- La Programmation Objet en Scala
  - Déclaration d'une classe
  - Faire hériter une classe d'une autre
  - Les Traits, les interfaces de Scala
  - Les Objects
- Les Generics

## Section 6

## La Programmation Fonctionnelle

# La Programmation Fonctionnelle

## Définition

- Paradigme de programmation dans lequel, au lieu d'avoir un programme séquentiel modifiant des états comme en programmation impérative, le programme est vu comme une suite de fonctions imbriquées qui ne modifient jamais un état et ne possèdent qu'une seule sortie.

## Grands Principes

- Immutabilité : une fois une variable définie, elle ne change plus de valeur. C'est comme si on avait des final sur les champs partout en Java
- Fonctions d'ordre supérieur : une fonction est une valeur qui peut être passée en paramètre d'autres fonctions

# Programmation fonctionnelle contre Programmation impérative

## programmation impérative

```
1 def length(aList:List[String]) = {  
2   var size = 0  
3   for (i <- aList) {  
4     size = size+1  
5   }  
6   size  
7 }
```

## programmation fonctionnelle

```
1 def functionalLength(aList:List[String]):Int = {  
2   if (aList.isEmpty) {  
3     0  
4   } else {  
5     1 + functionalLength(aList.tail)  
6   }  
7 }
```

# Les effets de bord

## Exemple

```
1 def functionWithSideEffect() {println("Je suis un effet de bord")}
```

## Qu'est-ce qu'un effet de bord ?

- En programmation fonctionnelle, on fait en sorte qu'aucune fonction n'ait d'effet de bord
- Un effet de bord, c'est lorsqu'une fonction modifie un état en plus de retourner quelque chose
- Dans l'exemple ci-dessus, on modifie l'état de l'écran en plus de ne rien retourner
- Une fonction n'ayant pas d'effets de bord est dite pure
- Une fonction qui ne retourne rien (soit de type `Unit`) n'est pas pure, exception faite de la fonction qui ne fait rien.

# La transparence référentielle

## Exemple

```
1 def functionWithoutReferentialTransparency():Int = scala.util.Random.nextInt(10)
```

## Qu'est-ce que la transparence référentielle ?

- En programmation fonctionnelle, on fait en sorte que les fonctions soient référentiellement transparentes
- La transparence référentielle, c'est quand on peut remplacer une fonction par son résultat sans modifier le comportement du programme global
- Dans l'exemple, ce n'est pas le cas, car la fonction renvoie toujours quelque chose de différent à chaque fois, on ne peut donc pas changer la fonction par son résultat
- La transparence référentielle est intéressante parce que cela permet de cacher les résultats d'une fonction (memoïsation)
- Une fonction sans argument n'est pas référentiellement transparente, à part une fonction qui renvoie une constante.

# Les fonctions comme valeurs

## Exemple

```
1 def functionTakingAnotherFunctionAsArgument(f: Int => Int):Int = {  
2   return f(0) + f(1)  
3 }  
4  
5 def addOne(x:Int) = x + 1  
6  
7 functionTakingAnotherFunctionAsArgument(addOne)
```

## Commentaires

- En programmation fonctionnelle, les fonctions sont des valeurs comme les autres
- On peut donc, comme dans l'exemple présenté, les passer en argument d'autres fonctions
- Lorsqu'on passe une fonction en argument, on doit exprimer son type de la manière suivante :  $(\text{TypeArg1}, \text{TypeArg2}) \Rightarrow \text{TypeRetourné}$

# Les Lambdas

## Exemple

```
1 def functionTakingAnotherFunctionAsArgument(f: Int => Int):Int = {  
2   return f(0) + f(1)  
3 }  
4  
5 functionTakingAnotherFunctionAsArgument(x => x + 1)  
6 functionTakingAnotherFunctionAsArgument(_ + 1)
```

## Commentaires

- On n'est pas obligé de définir une fonction avant de la passer en argument d'une autre fonction
- On peut directement passer la fonction, on appelle cela une fonction anonyme.
- Pour cela il y a deux méthodes, soit avec les arguments explicites :  
    `x => x + 1`
- Soit avec `_`, qui peut être utilisé lorsque on n'a pas besoin d'explicitement l'argument : `_ + 1`



# Déclaration de fonctions internes

## Exemple

```
1 def outerFunction(x:Int) = {  
2   def innerFunction(a:Int, b:Int) = {  
3     a + b  
4   }  
5  
6   innerFunction(x, 1)  
7 }
```

## Commentaires

- On peut définir des fonctions à l'intérieur d'autres fonctions

# La récursivité

## Le problème

- Pour faire une boucle en programmation impérative, soit on incrémente un entier, soit on fait appel à un itérateur
- Mais ces deux méthodes ne respectent pas l'immuabilité (l'entier change) ou la transparence référentielle (la méthode next de l'itérateur n'est pas transparente référentiellement)
- Comment fait-on pour faire une boucle en programmation fonctionnelle ?

## La Solution : la récursivité

- On appelle la même fonction plusieurs fois sur des espaces de plus en plus petits

# Exemple de récursivité

## Sans récursivité

```
1 def nonRecursiveFunction(n:Int) = {  
2   var i = 0  
3   while (i < n) {  
4     println("je fais quelque chose")  
5     i = i+1  
6   }  
7 }  
8 nonRecursiveFunction(10)
```

---

## Avec récursivité

```
1 def recursiveFunction(n:Int):Unit = {  
2   if (n <= 0) {  
3     return  
4   } else {  
5     println("je fais quelque chose")  
6     recursiveFunction(n-1)  
7   }  
8 }  
9 recursiveFunction(10)
```

---

# Retour sur l'exemple de récursivité

## L'exemple

```
1 def recursiveFunction(n:Int):Unit = {  
2   if (n <= 0) {  
3     return  
4   } else {  
5     println("je fais quelque chose")  
6     recursiveFunction(n-1)  
7   }  
8 }  
9 recursiveFunction(10)
```

## Commentaires

- C'est comme le raisonnement par récurrence vu en cours de mathématiques en terminal : initialisation, hérédité.
- Il faut faire bien attention à la condition d'arrêt (ici  $n \leq 0$ )
- Le type de retour de la fonction doit être explicite (ici `Unit`)

# Les Appels de récursivité

## La fonction length

```
1 def length[A](aList:List[A]):Int = {  
2   if (aList.isEmpty) 0 else 1 + length(aList.tail)  
3 }  
4  
5 length(List(0,2,1))
```

---

## Appels à la fonction length

- Premier appel : `length(List(0,2,1))`
- Deuxième appel : `1 + length(List(2,1))`
- Troisième appel : `1 + 1 + length(List(1))`
- Quatrième appel : `1 + 1 + 1 + length(List())`
- Cinquième appel : `1 + 1 + 1 + 0`

# À vous de jouer

## La factorielle

- La factorielle d'un entier naturel  $n$  est le produit des nombres entiers strictement positifs inférieurs ou égaux à  $n$
- $factorielle(0) = 1$
- $factorielle(n) = n \cdot (n - 1) \cdot (n - 2) \dots \cdot 2 \cdot 1$

# Le problème de la récursivité

## Essayez le code suivant dans le REPL

```
1 def length(range:Range):Int = {  
2   if (range.isEmpty) {  
3     0  
4   } else {  
5     1 + length(range.tail)  
6   }  
7 }  
8 length(1 to 40000)
```

## Le résultat

```
java.lang.StackOverflowError  
at scala.collection.immutable.Range.hasStub(Range.scala:71)  
at scala.collection.immutable.Range.longLength(Range.scala:72)  
at scala.collection.immutable.Range.<init>(Range.scala:89)  
at scala.collection.immutable.Range$Inclusive.<init>(Range.scala:433)  
at scala.collection.immutable.Range$Inclusive.copy(Range.scala:436)  
at scala.collection.immutable.Range.drop(Range.scala:202)  
at scala.collection.immutable.Range.tail(Range.scala:229)  
at .length(<console>:11)  
at .length(<console>:11)  
...
```

# La récursivité terminale

- On a ce problème parce que la récursivité n'est pas terminale
- Du coup, l'état à l'instant où on appelle la fonction récursive va être stocké à chaque itération
- La solution, c'est de transformer cette récursivité en récursivité terminale, c'est à dire qu'il faut que l'appel à la fonction soit la dernière instruction du scope



# Length en récursivité terminale

```
1 def length(range:Range):Int = {  
2   def innerLength(innerRange:Range, acc:Int):Int = {  
3     if (innerRange.isEmpty) {  
4       acc  
5     } else {  
6       innerLength(innerRange.tail, acc+1)  
7     }  
8   }  
9   innerLength(range, 0)  
10 }
```

## Commentaires

- Nous voyons ici un pattern très important pour transformer une récursivité en récursivité terminale
- On crée une fonction interne qui a pour argument des arguments ayant le même type que les arguments de la fonction externe et un accumulateur qui représente le résultat
- Cet accumulateur va être incrémenté à chaque appel de la fonction interne et retourné lorsque l'on rencontre la condition d'arrêt
- Après il suffit d'appeler la fonction interne avec les arguments de la fonction externe et l'accumulateur initialisé à la valeur d'arrêt

# Annotation tailrec

```
1 import scala.annotation.tailrec
2
3 @tailrec
4 def length(range:Range):Int = {
5   if (range.isEmpty) 0 else 1 + length(range.tail)
6 }
```

```
<console>:12: error: could not optimize @tailrec annotated method length: it
contains a recursive call not in tail position
    def length(range:Range):Int = if (range.isEmpty) 0 else 1 + length(range.
      tail)
```

## Commentaires

- Il existe une annotation `@tailrec` demandant au compilateur de vérifier si une récursivité est terminale
- Si la récursivité n'est pas terminale sur une fonction annotée, le compilateur stoppe la compilation sur une erreur
- Pour utiliser cette annotation, il faut importer `scala.annotation.tailrec` et placer un `@tailrec` juste avant la fonction dont on veut vérifier la récursivité terminale

# À vous de jouer

## Exercices

- Réécrivez le FizzBuzz en fonctionnel
- Réécrivez le C'est plus C'est moins en fonctionnel
- Écrivez une fonction `reverse` qui renverse une liste, par exemple, `reverse(List(1,2,3))` doit renvoyer `List(3,2,1)`
- Écrivez une fonction `map` qui applique une fonction sur une liste, par exemple `map(List(1,2,3), x => x+1)` doit renvoyer `List(2,3,4)`
- Écrivez une fonction `filter` qui filtre une liste en fonction d'un prédicat, par exemple `filter(List(4,2,3), x => x > 2)` doit renvoyer `List(4,3)`

## Quelques éléments supplémentaires

### Quelques fonctions utiles sur les listes Scala

- `isEmpty` : retourne vrai si la liste est vide, faux sinon
- `head` : retourne le premier élément d'une liste, lève une exception si la liste est vide
- `tail` : retourne la liste privée de son premier élément
- `::` : ajoute un élément en tête de la liste, par exemple `1::List(2,3)` renverra la liste `List(1,2,3)`

## Bonus : Currification et fonctions partielles

### Qu'est ce que la curryfication ?

- La curryfication consiste à décomposer une fonction ayant  $n$  arguments en  $n$  fonctions ayant un seul argument
- Chacune de ces fonctions intermédiaires renvoient une fonction partielle
- Une fonction partielle est une fonction dont une partie des arguments a été fixée, mais pas tous

### Exemple

```
1 def sum(x:Int, y:Int, z:Int) = x + y + z
2 def curryfiedSum(x: Int)(y: Int)(z:Int) = x + y + z
3
4 // definit une fonction (Int, Int) => Int qui calcule 10 + y + z
5 def partialFunction1 = curryfiedSum(10)_
6
7 // definit une fonction Int => Int qui calcule 10 + 10 + z
8 def partialFunction2 = curryfiedSum(10)(10)_
```

# Bonus : les Fermetures

## Exemple

```
1 var x = 3
2 def addX(y: Int) = x + y
3
4 addX(3) // retourne 6
5 x = 4
6 addX(3) // retourne 7
```

---

## Qu'est ce qu'une fermeture ?

- C'est lorsqu'une fonction fait appel à des variables définies à l'extérieur d'elle même
- Dans l'exemple, la fonction addX fait appel à une variable x définie à l'extérieur de son scope
- Si la variable x change, le résultat de la fonction change

## Bonus : Curry-Howard

### La correspondance de Curry-Howard

- Correspondance entre la logique mathématique et les types des programmes dans un cadre fonctionnelle.
- Les types des fonctions sont des théorèmes
- Les implémentations des fonctions sont les preuves : si une fonction peut être implémentée, c'est un théorème valide

### Exemple

- $f: A \Rightarrow A$  : c'est vrai, et il n'y a qu'une fonction qui puisse faire cela, c'est l'identité
- $g: \text{List}[A] \Rightarrow A$  : c'est faux, car aucune fonction pure et référentiellement transparente ne peut retourner cela (pensez à la liste vide)

# Bilan

## Ce que vous avez vu

- Les Principes de la Programmation Fonctionnelle
  - L'immuabilité
  - Les fonctions d'ordre supérieur
  - La programmation sans effet de bord
  - La transparence référentielle
- La récursivité et la récursivité terminale
- Curryfication et fermeture
- Le théorème de Curry-Howard



## Section 7

# Le Pattern Matching

# Le Pattern Matching

## Qu'est ce que c'est ?

- C'est l'équivalent d'un `switch` en java, mais en beaucoup plus puissant
- Par exemple, c'est utilisable sur énormément de types, alors qu'en Java le `switch` est limité aux Strings, aux entiers et aux enums
- On peut même définir ses propres classes sur lesquelles faire du pattern matching
- On peut faire des conditions dans le pattern matching
- C'est très utilisé en programmation fonctionnelle.

# Exemple switch Java contre Pattern Matching Scala

## Switch en Java

```
1 int i = 1;
2 switch (i) {
3     case 0:
4         System.out.println("c'est un zero");
5         break;
6     case 1:
7         System.out.println("c'est un un");
8         break;
9     default:
10        System.out.println("c'est autre chose");
11 }
```

## Pattern Matching en scala

```
1 val i = 1
2 i match {
3     case 0 => println("c'est un zero")
4     case 1 => println("c'est un un")
5     case _ => println("c'est autre chose")
6 }
```

# Détails sur le pattern matching

## Exemple

```
1 val i = 1
2 i match {
3   case 0 => println("c'est un zero")
4   case 1 => println("c'est un un")
5   case _ => println("c'est autre chose")
6 }
```

## Commentaires

- Il n'y a pas de break
- le cas default est défini par le caractère `_`, ce caractère désigne quelque chose qu'on ne veut pas nommer
- on aurait pu le remplacer par un nom quelconque

# Pattern Matching conditionnel

## Exemple

```
1 x match {  
2   case a if a > 0 => println("c'est superieur a zero")  
3   case _ => println("c'est inferieur a zero")  
4 }
```

## Commentaires

- Je peux donner un nom à ma variable histoire de la réutiliser derrière
- Je peux mettre une condition sur un cas de pattern matching

# Le Pattern Matching est assignable

## Exemple

```
1 val absX = x match {  
2   case a if a > 0 => x  
3   case _ => -x  
4 }  
5  
6 def abs(x:Int) = x match {  
7   case a if a > 0 => x  
8   case _ => -x  
9 }
```

## Commentaires

- Le pattern matching, comme le conditionnel `if`, peut retourner une valeur
- Je peux donc définir une fonction qui ne contient que du pattern matching

# Le Pattern Matching s'applique à plein d'objets

## Exemple

```
1 def length[A](aList:List[A]):Int = aList match {  
2   case Nil => 0  
3   case x::xs => 1 + length(xs)  
4 }
```

## Commentaires

- Le pattern matching s'applique à pleins d'objets, dont liste
- Nil veut dire liste vide, cela évite d'avoir à taper `xs if xs.isEmpty`
- `x::xs` correspond à la tête (x) et la queue (xs) de la liste

# À vous de jouer

## Exercices

- Réécrivez le FizzBuzz avec du pattern matching
- Réécrivez le C'est plus C'est moins avec du pattern matching
- Réécrivez la fonction `reverse` qui renverse une liste, avec du pattern matching
- Réécrivez la fonction `map` qui applique une fonction sur une liste, avec du pattern matching
- Réécrivez la fonction `filter` qui filtre une liste en fonction d'un prédicat, avec du pattern matching



# Utiliser ses propre objets

## Sealed Trait

- C'est un trait déclaré avec le mot clé `sealed`
- Lorsqu'un trait est déclaré `sealed`, ce trait ne peut être étendu que par les classes déclarées dans le même fichier
- Cela permet au pattern matching de connaître toutes les implémentations de ce trait, et donc de pouvoir couvrir ce trait

## Case class

- C'est une classe déclarée avec le mot clé `case`
- Lorsque l'on crée ce type de classe, on a gratuitement une méthode permettant de créer une instance sans le mot clé `new`
- On a aussi une méthode `unapply` qui va permettre de pattern matcher sur les champs de l'objet de manière transparente

# Exemple d'utilisation de ses propres objets

```
1 sealed trait Vehicle
2 case class Car(seat:Int) extends Vehicle
3 case class Bicycle() extends Vehicle
4
5 def seatSpeaker(vehicle:Vehicle) = vehicle match {
6   case Bicycle() => println("un velo n'a pas de sieges !!!")
7   case Car(1) => println("un voiture avec une place, c'est petit")
8   case Car(x) if x > 1 => println("La on commence a parler")
9   case Car(_) => println("une voiture avec un nombre de places negatif ou nul ???")
10 }
```

---

## Quelques remarques finales sur les case class

- En Scala on utilise beaucoup les case class, car cela enrichit les classes sans effort
- On peut créer des case class sans les faire étendre des sealed trait
- Attention, dans les case class on peut accéder aux champs. Ensuite cela est mitigé par le fait que normalement, tout est immutable
- En plus du constructeur sans new et du pattern matching, on a aussi l'opérateur d'égalité qui compare champ par champ

# Bilan

## Ce que vous avez vu

- Le Pattern Matching
  - Conditionnel dans le Pattern Matching
- Les sealed trait
- Les case class