



Devoxx France 2023

+

De Java à Scala 3,
des lambdas, à la
programmation
fonctionnelle

Code
Language

+

Devoxx France 2023

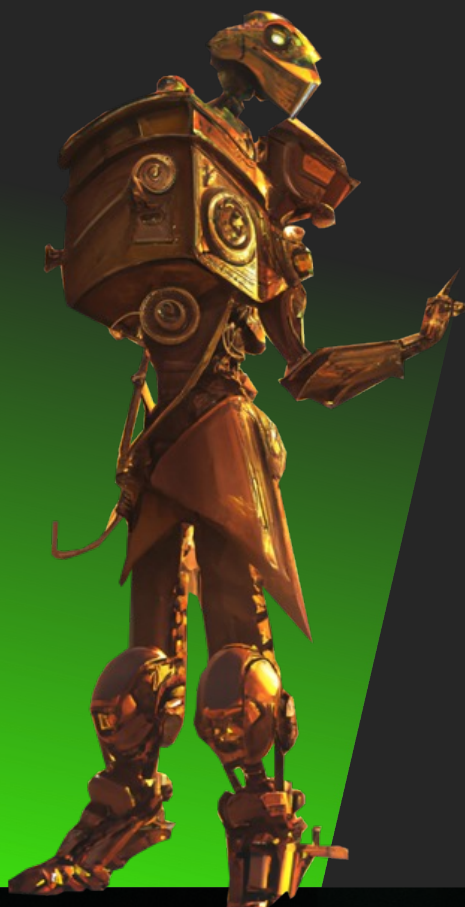
De Java à Scala 3, des lambdas, à la programmation fonctionnelle

Ugo BOURDON &
Jonathan WINANDY

Scalaistes

Devoxx France 2023

Sommaire



Introduction

D'où l'on parle, on vend le sujet, Java

Définitions : Règles et loi sur les fonctions

Principes de la programmation fonctionnelle

Définitions : Immutabilité et transparence référentielle

Concepts et avantages de l'immutabilité

Types algébriques de données (ADT)

Définitions, utilisations

Comparaison avec les approches en OOP

Plus de types

Types riches et polymorphisme ad-hoc

Systèmes d'effet (IO), programmation concurrente, ...

Structured concurrency

Conclusion

Ugo BOURDON

- Code en Scala à plein temps depuis 2011
- Travailleur indépendant
- CTO Performance-Immo (Éditeur logiciel dans différent domaine)
- Praticient eXtrême programming, DDD, Kanban
- *@ugobourdon*

Jonathan WINANDY

- Passionné de Scala
- Consultant en Data Eng (Kafka / Spark / Scala)
 - chez UNIVALENCE (depuis 2015)
 - audit de plateforme data
 - architecture (sous forme de code)
 - ...
 - enseignements en écoles d'ingénieur / universités
- Scala.IO
- *@ahoy_jon*

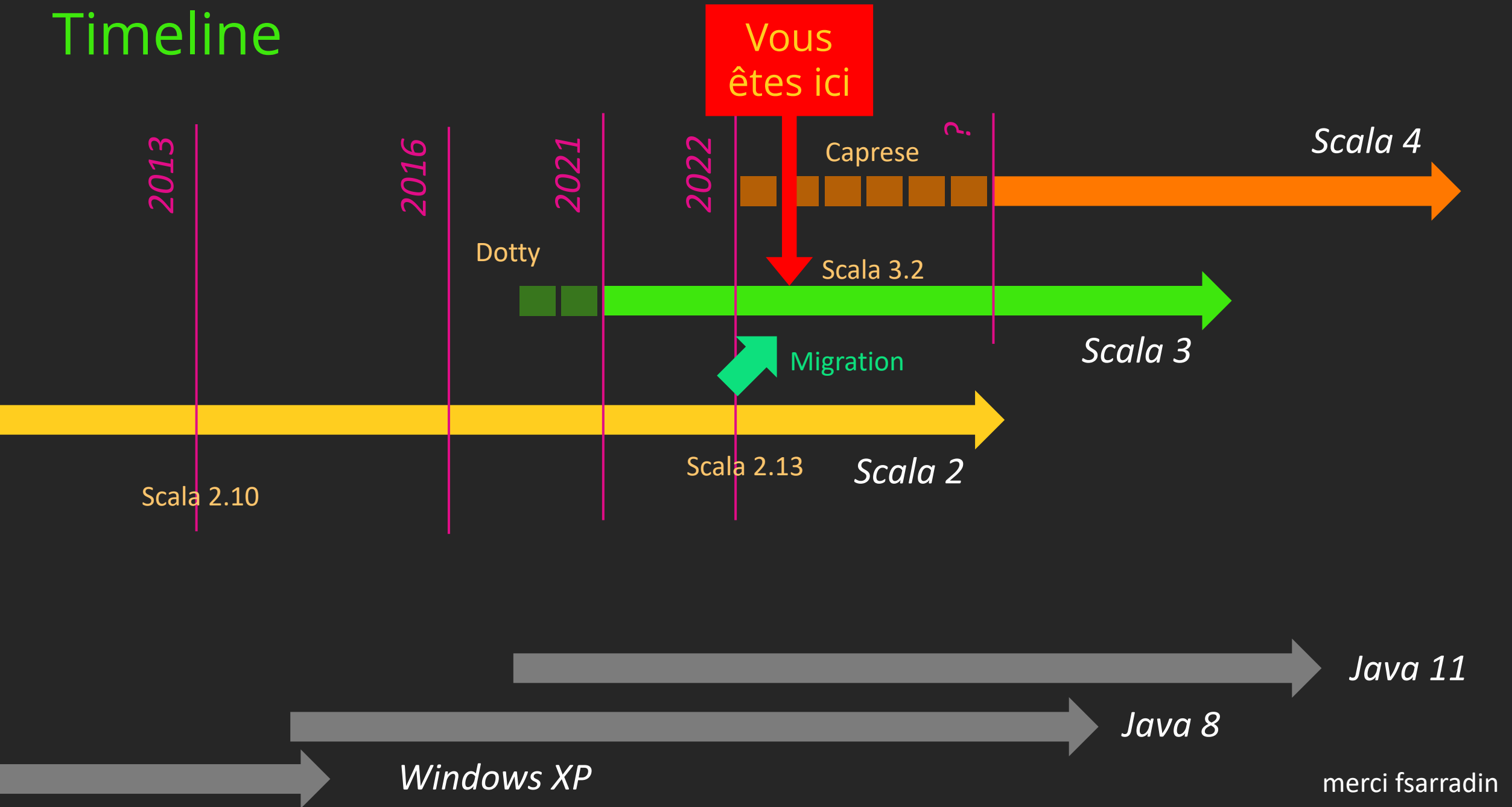
Organisation de la présentation

- Cela va durer 3h :
 - Si vous avez des questions, n'hésitez pas
 - Il y a une pause de 15 minutes au milieu
- Format
 - Présentation de quelques concepts clés de la programmation fonctionnelle
 - Démo ! On a une application test
 - https://github.com/SimplyScala/fpscala3_uno
 - On va pouvoir regarder ça ensemble
- On va vous donner notre point de vue
 - et si vous n'êtes pas d'accord, n'hésitez à nous le dire !

Motivation de la présentation

- 10 ans de Scala 2.10
 - Sortie de Scala 3 en 2021
- bientôt 10 ans de Java 8
 - Sortie de Java 17 en 2021
- Petits étonnements sur la trajectoire de Java !
 - L'idée c'est de vous montrer que cela vaut le coup de passer de l'autre côté.
 - Et surtout vous fournir les clés pour vous y mettre bientôt maintenant que Scala est beaucoup plus facile à prendre en main !

Timeline



Scala 3

- **Syntaxe simplifiée** (python)
- Nouveau système de types (types union, intersection, opaques, etc.)
- Implicits redessinés (syntaxe "given/using" et extensions)
- *Nouveau système de macros et métaprogrammation*
- Support natif des **énumérations**
- Améliorations du **pattern matching**
- *Performances et compilation optimisées*
- Compatibilité ascendante avec Scala 2
 - la blague c'est que Scala 3 tourne sur Scala 2.13 et inversement

Syntaxe à la Python ?

`Significant Indentation Syntax` :

- moins de `{}`, plus de `:`
- les deux syntaxes fonctionnent

```
case class State(n: Int,  
                 minValue: Int,  
                 maxValue: Int):  
  def inc: State =  
    if n == maxValue then this  
    else this.copy(n = n + 1)
```

Java 8, 11, 17

- Java 8 (sorti en mars 2014) :
 - Expressions lambda
 - Interfaces fonctionnelles
 - Références de méthode
 - Stream API
- Java 11 (sorti en septembre 2018) :
 - Syntaxe "var" pour les variables locales (introduite en Java 10)
- Java 17 (sorti en septembre 2021) :
 - Pattern Matching pour l'opérateur instanceof (JEP 394)
 - Sealed classes (JEP 397)
 - Records (JEP 395) - structures de données immuables et concises

Evolutions à venir en Java

- Amber
 - Simplification des syntaxes et amélioration de la qualité de code
- Valhalla
 - amélioration des représentations mémoires
- Loom
 - Threads virtuels

Scala et sa complexité polymorphique

- Scala est un langage plus complexe que Java en raison de sa nature hautement polymorphe.
- Néanmoins :
 - Tous les concepts sont disponibles depuis plus de 10 ans.
 - L'interopérabilité est complète avec Java
 - Et cela fonctionne sur 1.8
- Et la programmation concurrente est radicalement meilleure en Scala

Pourquoi il ne faut pas utiliser Scala en 2023 ?

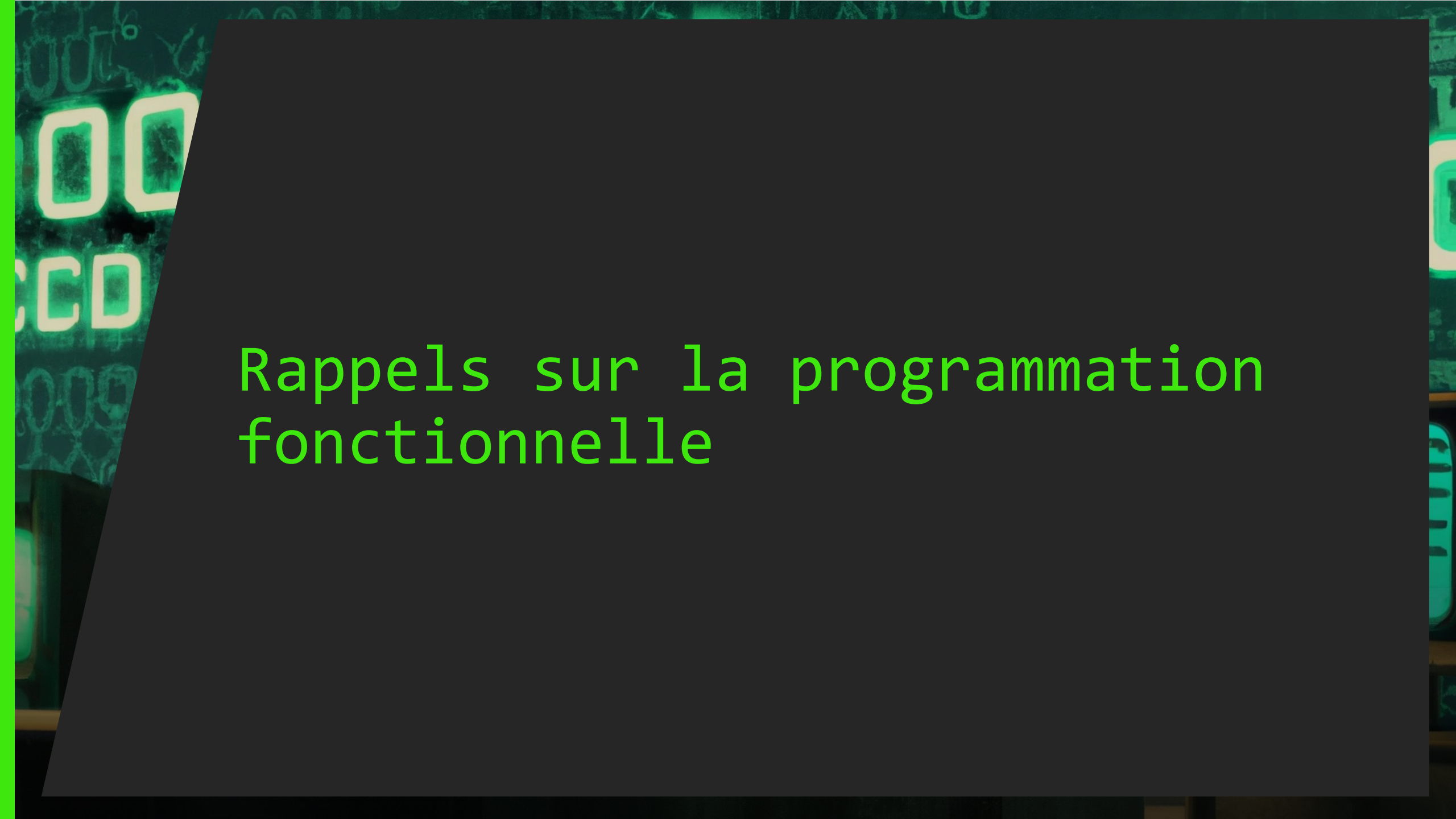
- On ne connaît pas Scala
 - Et c'est bien dommage
- La stabilité des outils sur Scala 3
 - mais Scala 2.13 est génial
- Twitter !
- Spring Boot
 - Même si on fait du "Spring Boot" en mieux
 - Mais que cela nécessite un Scalaiste avec 5ans d'XP
- On ne connaît pas Scala
 - Et mon architecte n'aime pas Scala

Les trucs sympas !

- La communauté est hyper-active
 - les chans discords
 - Scala : <https://discord.com/invite/scala>
 - ZIO : <https://discord.com/invite/2ccFBr4>
 - Typelevel : <https://discord.com/invite/XF3CXcMzqD>
 - la communauté sur twitter qui répond
 - et qui explique en détail les concepts
 - événements en France
 - PSUG : <https://www.meetup.com/paris-scala-user-group-psug/>
 - Scala.IO : <https://scala.io/>
 - Les groupes de programmation fonctionnelle (Nantes, Lille, Lyon, Montpellier, ...)

Les trucs sympas !

- Les copains :
 - François Sarradin (Univalence / Scala.IO)
et Stéphane Tankoua (Fabernovel / PSUG)
 - Hands-On Labs de 3h demain de 10h45 à 14h15, salle Paris 201
 - Fabrice Sznajderman (MNT / Scala.IO / DevovxxFr)
 - François Armand (Rudder) / Stéphane Landelle (Gatling)
 - @NicolasRinaudo / @JulienTruffaut
 - Jules Ivanic / Matthieu Baechler / Stéphane Derosiaux
 - ...
 - Une autre présentation sur Scala vendredi 13h30 à 14h45
Salle Neuilly 253 : **Écoutez l'histoire de Sonos Voice et de ZIO...** par Pierre Baillet



Rappels sur la programmation fonctionnelle

La programmation fonctionnelle

- Un paradigme de programmation :
 - déclaratif
 - dérivé des mathématiques
- Transposition en informatique du concept de fonction pour faciliter la fiabilité des programmes.

Paradigme déclaratif

- Description du fonctionnement au lieu des détails de l'exécution.
- Des langages déclaratifs :
 - SQL
 - CSS
 - XSLT
 - R
 - ...

Dérivé des mathématiques

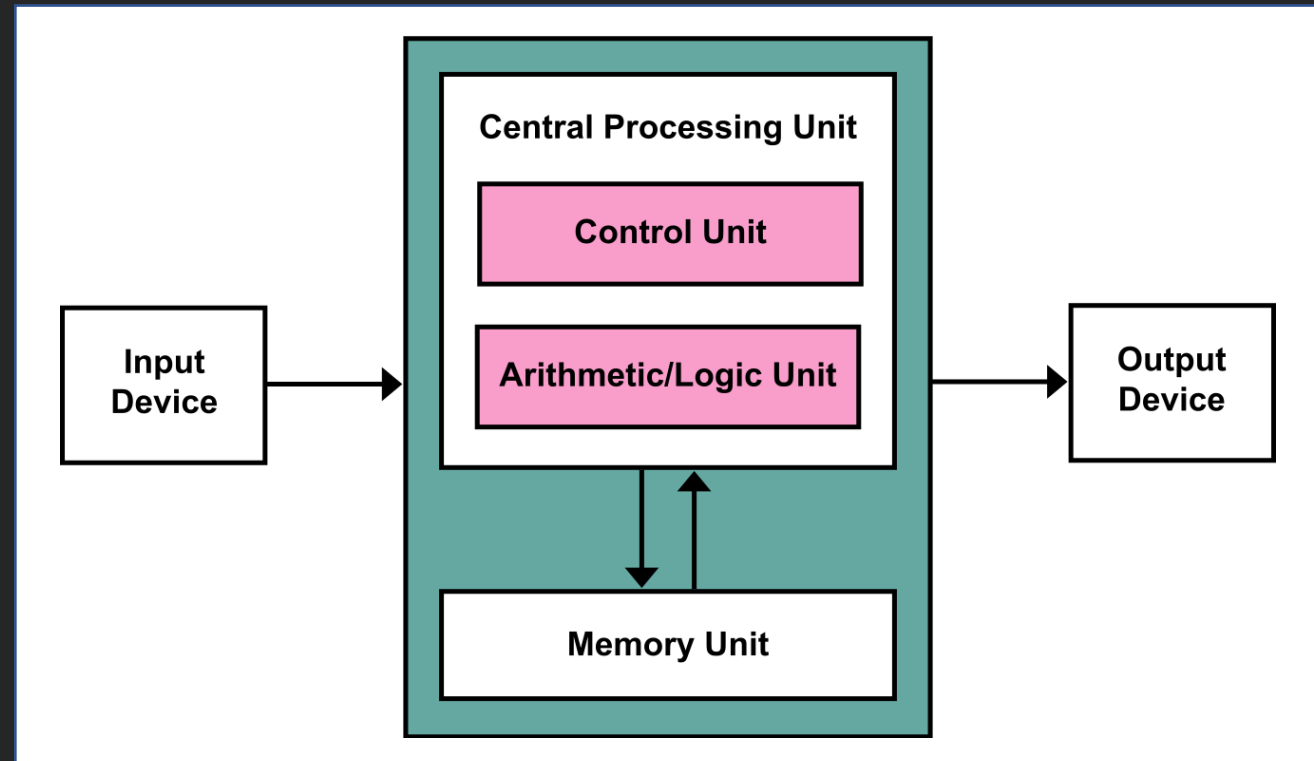
- Abstraction/Généralisation
 - Composition/Décomposition
 - Logique
 - Algèbre
-
- On va pouvoir utiliser les outils des mathématiques pour raisonner sur les programmes.

Autres paradigmes

- **La programmation impérative** : basée sur des séries d'instructions successives, dont certaines sont parfois étranges :
 - $x := x + 1$
- **La programmation logique** : basée sur des faits, des règles et des requêtes. eg :
 - `homme(socrates) ;`
`X => humain(X) si homme(X) ;`
`humain(socrates) ?`

Pourquoi la programmation impérative est-elle si populaire ?

- Familiarité
 - Architecture de von Neumann
 - "Can Programming Be Liberated from the von Neumann Style?" – John Backus
- Facilité
- Performance



Raisonnement approximatif

- Les limites physiques imposées limitent les capacités de raisonnement.



- Même si l'on opère dans des limites physiques, on gagne à rester le plus abstrait le plus longtemps possible !

Haskell vs Ada vs C++ vs ...

La programmation fonctionnelle est très souvent plus performante sur les évaluations inter-langages.

Mais, il y a des exemples de retour comme :
"Les fonctions de plus haut niveau (HOF) ne sont qu'une illusion, pas très utiles dans la vie réelle."

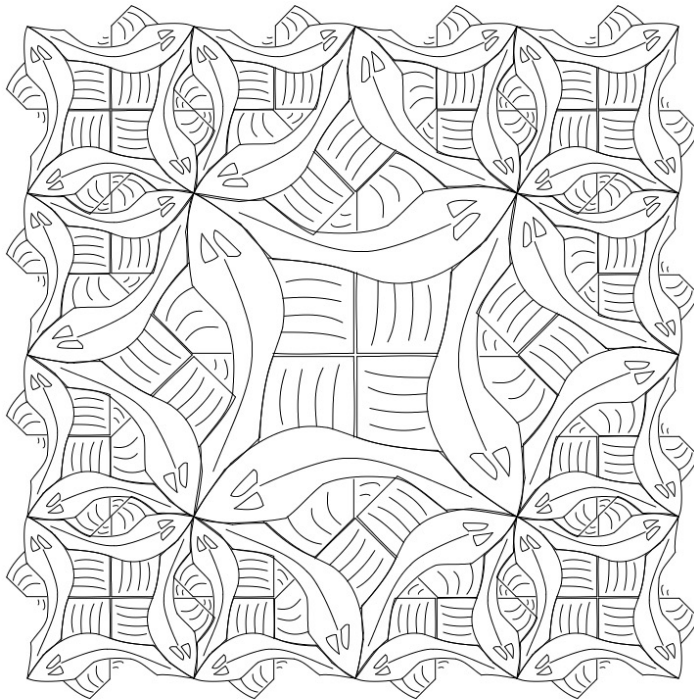
Il y a une résistance à s'imaginer que c'est un outil applicable et performant dans toutes les situations où l'on fait de la programmation.

** HOF : Fonction qui prend d'autres fonctions en paramètre*

Note : Les designs patterns en OOP

Design Pattern POO	Équivalent fonctionnel PF
Singleton	Fonction pure / Module
Strategy	Fonction d'ordre supérieur (Higher-order function)
Factory Method / Abstract Factory	Fonction de construction / avec paramètres
Template Method	Fonction d'ordre supérieur (Higher-order function)
Builder	Fonction de construction partielle
Adapter	Fonction partielle / Composition de fonctions
Decorator	Composition de fonctions
Command	Fonction (closure)
Chain of Responsibility	Composition de fonctions / Monades
Observer	Fonctions de rappel (callback) / Stream
Memento	Immutabilité / Copie fonctionnelle
Visitor	Pattern matching / Fonction d'ordre supérieur

Un exemple : Square Limit - M.C. Escher



120 lignes de code :

- 45 pour les rotations, et fusions d'images
- 50 pour définir le poisson
- 20 pour définir le problème
- 5 pour initialiser le fichier

P00 et/ou PF

- Encapsulation autour d'un état interne
- Le couplage se fait avec des `signaux`
- Relations précises d'entrées vers une sortie
- Les données sont immuables

***Modélisations
et Interactions***

***Transformations
et Règles***

Note : les bonnes pratiques en POO consistent souvent à tendre vers le FP

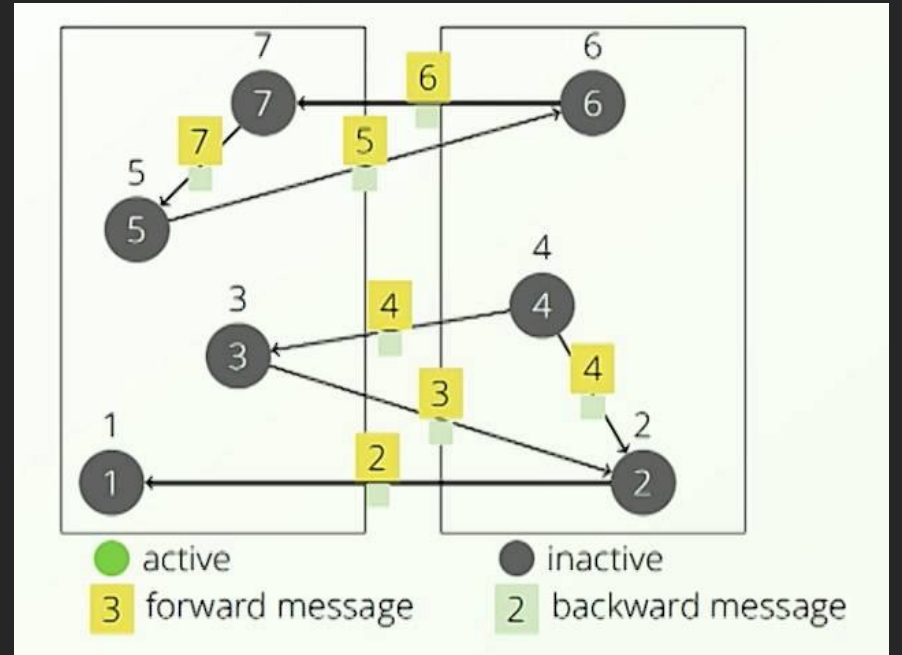
à quoi ressemble la P00 en PF ?

```
obj      : Self
          ET
{
  getAtr  : Self          =>      Atr
  setAtr  : (Self, Atr)    => Self
  cmd     : Self          => (Self, Return1)
  method  : (Self, Arg1, Arg2) => (Self, Return2)
}
```

```
abstract class MyClass(data: Data) {
  def getAtr: Atr
  def setAtr(atr: Atr): MyClass
  def cmd: (MyClass, Return1)
  def method(arg1: Arg1, arg2: Arg2): (MyClass, Return2)
}
```


Note : "Pregel"

Pregel est un modèle de calcul pour du graph distribué, par exemple, PageRank.



```
def pregel[A](  
    initialMsg: A,  
    maxIterations: Int,  
    activeDirection: EdgeDirection  
)(  
    vprog: (VertexId, VD, A) => VD,  
    sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],  
    mergeMsg: (A, A) => A  
): Graph[VD, ED]
```


Note : les "lambdas"

- Les lambdas ne sont pas suffisantes pour faire de la programmation fonctionnelle.
- Il y a des problèmes de composition avec la **mutation**.

```
let result = 0;  
const add = x => {  
    result += x; return result;  
};  
console.log(add(2)); // 2  
console.log(add(3)); // 5
```



Règles et lois sur les fonctions

Le terme "fonction"

En programmation, en général, le terme "fonction" est souvent utilisé de manière interchangeable avec "procédure", qui désigne une suite d'instructions.

Les fonctions dans la programmation fonctionnelle

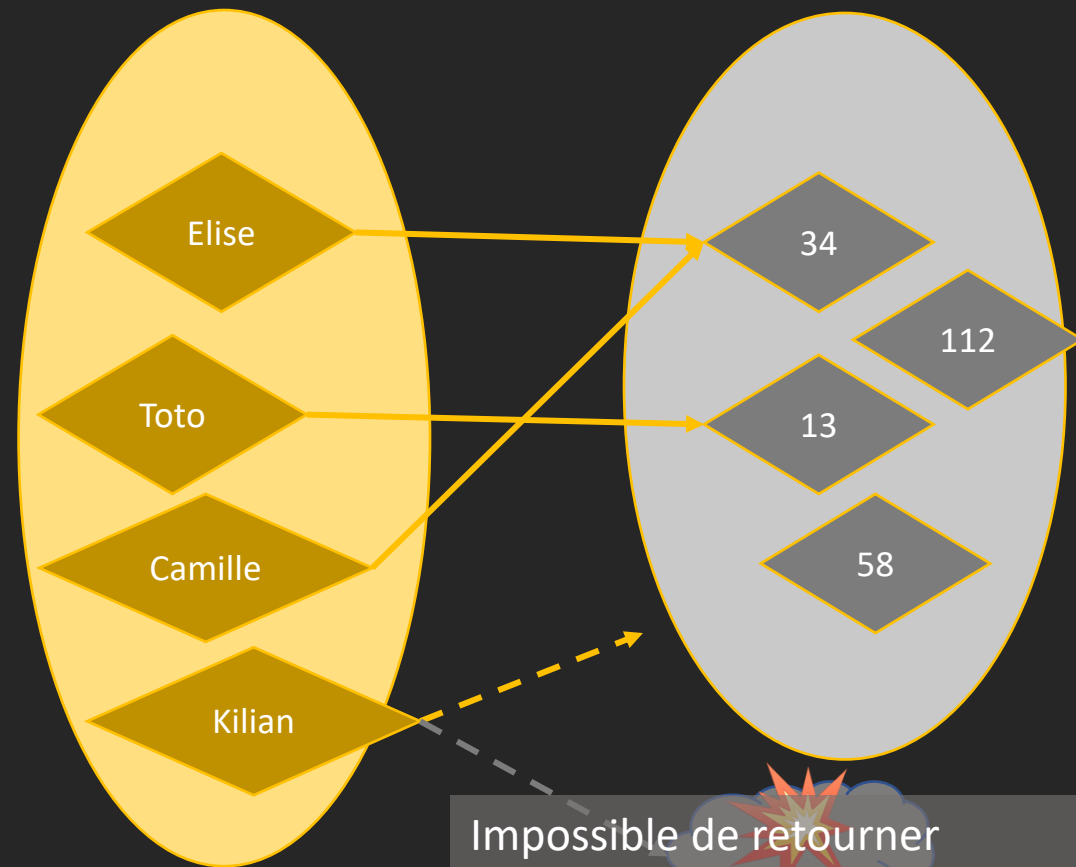
Les règles pour la définition de fonction :

1. Les entrées produisent toujours une sortie.
2. Une entrée identique produit la même sortie.
3. Les fonctions sont pures et n'ont pas d'effets de bord.

Règle 1 : Les entrées produisent toujours une sortie

Entrée (domaine)
(ensemble de définition)

Sortie (co-domaine)
(ensemble des valeurs prises)

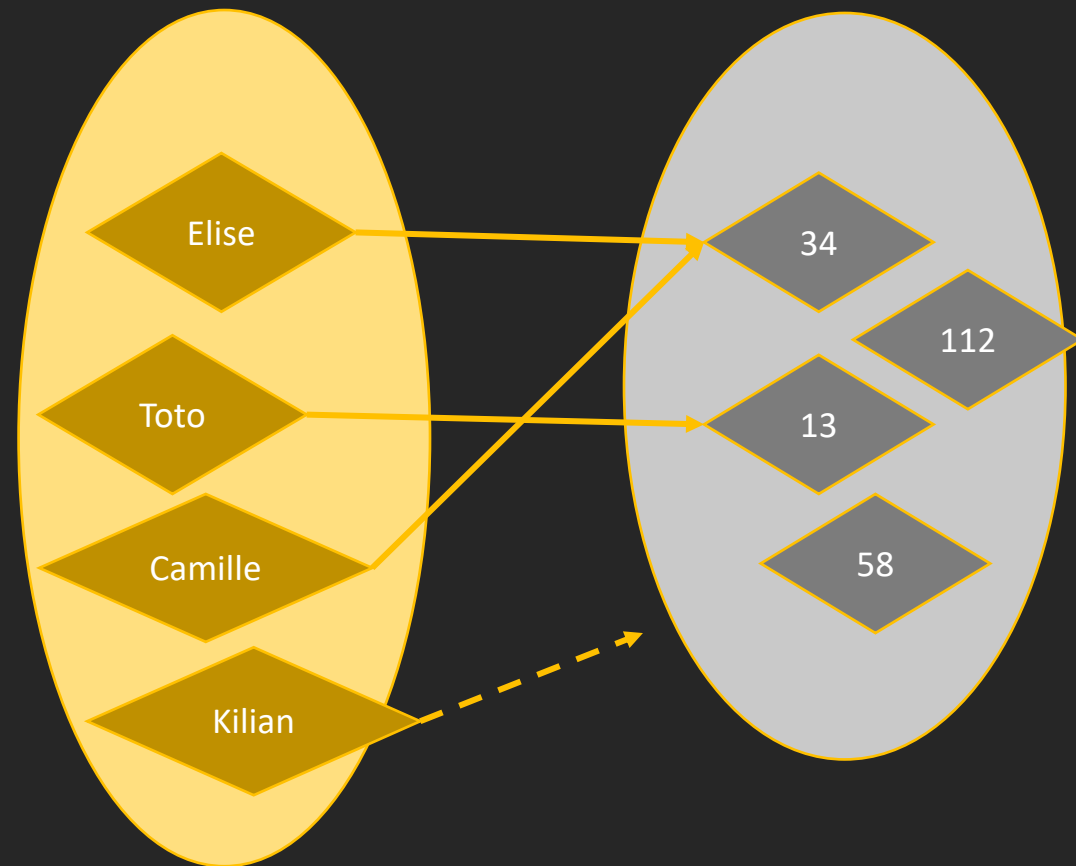


Impossible de retourner
une sortie invalide

Règle 2 : Une entrée identique produit la même sortie (pour la même fonction)

Entrée (domaine)
(ensemble de définition)

Sortie (co-domaine)
(ensemble des valeurs prises)



Règle 3 : Les fonctions sont pures et n'ont pas d'effets de bord.

Exemple d'effets de bord :

- Modification d'une variable
- Modification d'une structure de données en place
- Affectation d'un champ dans un objet
- Lancement d'une exception ou arrêt avec une erreur
- Affichage dans la console ou lecture d'une entrée utilisateur
- Lecture ou écriture dans un fichier
- Dessin sur l'écran

“Malheureusement que des trucs utiles”

Les fonctions dans la programmation fonctionnelle

Les règles pour la définition de fonction :

1. Les entrées produisent toujours une sortie.
(une fonction est complète)
2. Une entrée identique produit la même sortie.
(une fonction est déterministe)
3. Les fonctions sont pures et n'ont pas d'effets de bord.
(une fonction est "pure")

Comment écrire un programme sans effet ?

un contrainte sur “**la manière d'écrire** les programmes”

!=

n'est pas une “restrictions sur les formes de programmes pouvant être écrites”

Et de nos jours, on a des ‘god-type’, comme

```
ZIO[Deps, Error, Return]
```

qui permettent de représenter les programmes et de manipuler la definition et l'exécution avec des fonctions pures.

Comment écrire un programme sans effet ?

Donc, on utilise des expressions pour représenter les effets. Exemple :

```
def runNTimesPrint[Return](n: Int,  
                           prg: Task[Return],  
                           beforeMsg : Int => String,  
                           afterMsg  : (Int, Return) => String): Task[Seq[Return]] = {  
  ZIO.foreach(1 to n)(i => printLine(beforeMsg(i)) *> prg.tap(res =>  
    printLine(afterMsg(i, res))))  
}
```

et une fonction pure de transformation de programme
... (qui s'écrit en une ligne de code inférieure à 120 chars)

The background features a dark blue gradient with a large, lighter blue triangle on the left side. The triangle contains a faint, glowing binary code pattern. The text is centered within the triangle.

Substitution et transparence référentielle

Substitution et transparence référentielle

La substitution est le mécanisme d'évaluation des expressions.

eg :

```
def inc(x: Int): Int = x + 1  
  
val b1 = inc(1) + inc(1) + inc(2)  
  
val i1 = inc(1)  
val b2 = i1 + i1 + inc(2)
```

Expression	Value
inc(1)	2
inc(1)	2
2 + 2	4
inc(2)	3
4 + 3	7

Substitution et transparence référentielle

Une fonction transparente référentiellement peut être remplacée par sa valeur de retour sans changer le comportement du programme.

Pour tout x , $f(_)$, et $y := f(x)$,
on peut remplacer $f(x)$ dans tout le programme par y .

```
trait WhatIsMyJob {  
  def decompose(pb: ComplexProblem): List[LessComplexProblem]  
  
  def findSolution(pb: LessComplexProblem): Solution  
  def recompose(actions: List[Solution]): Application  
  
  def main(complexProblem: ComplexProblem): Application =  
    decompose(complexProblem) map findSolution andThen recompose  
}
```

Notre boulot de programmeur
fonctionnel

Pourquoi la transparence référentielle est-elle importante ?

Dans un paradigme déclaratif, on peut réécrire les expressions pour obtenir un résultat.

Ce processus de réécriture est possible grâce au modèle de substitution. Cela permet de manière transparente (équivalente) :

- de refactoriser le code
- de faire des optimisations
- ...

De manière générale, cela permet de manipuler des expressions au lieu des instructions.

Contre exemple

```
import scala.util.Random
```

```
List(Random.nextInt(100),  
Random.nextInt(100), Random.nextInt(100))  
// returns List(30, 42, 90)
```

```
val value = Random.nextInt(100)  
List(value, value, value)  
// returns List(30, 30, 30)
```

Un exemple

Un calculette avec des variables.

```
val _1: Expr = 1.e // Nombre(1)
val x: Expr = "x".e // Variable("x")
val b: Expr = "b".e // Variable("b")
val expr2: Expr = (x * _1) + ("x".e * b) // x *
1 + x * b
```

```
val r1 = optimFull(expr2)
assert(x == "x".e)
assert(r1 == x * (_1 + "b".e)) // x * (1 + b)
```

The background of the slide features a dark, textured surface with glowing green binary code (0s and 1s) and the letters 'CCD' in a bold, sans-serif font. A large, dark gray rectangular area covers the majority of the slide, with the text 'Types ?' centered within it.

Types ?

Types ?

- En programmation fonctionnelle, les types servent à décrire les valeurs possibles
 - Les types peuvent être vus comme des énumérations de valeurs
 - Chaque type définit un ensemble fini ou infini de valeurs possible
- L'idée c'est de trouver le type le moins compliqué (cardinalité faible) qui représente correctement nos expressions

Types ?

Type

Nothing

Unit

Boolean

Option[Boolean]

Either[Unit, T]

Unit | T

enum Couleur:
case Rouge, Vert, Bleu

(Boolean, Couleur)

Boolean | Couleur

Char

String

List[Boolean]

Valeurs

Aucune valeur (sous-type de tous les autres types)

()

true, false

None, Some(true), Some(false)

Left(()), Right(T) - où T est une valeur quelconque de type T

() ou T - où T est une valeur quelconque de type T

Couleur.Rouge, Couleur.Vert, Couleur.Bleu

(true, Rouge), (true, Vert), (true, Bleu), (false, Rouge), (false, Vert), (false, Bleu)

true, false, Rouge, Vert, Bleu

'a', 'b', 'c', ..., 'A', 'B', 'C', ..., '0', '1', '2', ..., '\n', '\t', ... (tous les caractères Unicode)

"", "a", "ab", "abc", ..., (toutes les séquences possibles de caractères)

[], [true], [false], [true, true], [true, false], [false, true], [false, false], ...

Enum (Scala 3)

Avec Scala, vous pouvez déclarer des types associés à une succession de valeurs.

```
enum Couleur:  
  case ROUGE, VERT, BLEU
```

```
enum Couleur(valeur: Int):  
  case ROUGE extends Couleur(1)  
  case VERT   extends Couleur(2)  
  case BLEU   extends Couleur(3)
```

```
// Utilisation de l'énumération  
Couleur.ROUGE // ROUGE  
Couleur.ROUGE.valeur // 1
```

Somme des types

- Couleur = Rouge + Vert + Bleu
- Animal = Dog(name) + Cat(name)

```
enum Animal:  
    case Dog(name: String)  
    case Cat(name: String)  
  
val pet: Animal = Animal.Dog("Felix")
```

Somme des types

- Il existe plusieurs façons de définir une somme de type :
- Sous-typage : $A, B <: A, C <: A ; A = B + C$
 - Enumerations : $E = E1 + E2 + E3 + \dots$
 - sealed
- Either : $\text{Either}[B, C] = \text{Left}[B] + \text{Right}[C]$
- Union type : $B \mid C = B + C$

Somme des types : Option

- Un des usages fréquents de la somme de types, c'est la gestion des valeurs éventuellement définies (null)

```
val name: String | Null // (scala 3)
val name: Option[String]
```

```
def toOpt(str: String | Null): Option[String] =
  str match
    case null => None
    case x:String => Some(x)
```

Produits (ou records/tuples)

- Le produit des types permet de mettre plusieurs valeurs de types différents ensemble :

```
case class Product(name: String,  
                   price: Double,  
                   quantity: Int)
```

```
//usage  
val product = Product("test", 10, 1)  
product.name // "test"  
product.price // 10.0d  
product.quantity // 1
```

style nommé

style positionnel

```
val product = (1, "abc", true)  
product._1 // 1  
product._2 // "abc"  
product._3 // true
```


ADT : Type de donnée algébrique

l'ADT est une construction qui n'est pas spécifique à la programmation fonctionnelle

- L'ADT est une composition de :
 - Somme de type
 - Records / Produit avec :
 - Repetitions (List / Option)
 - types primitifs et autres types

```
enum Personnage:  
  case SuperHero(nom: String, pouvoir: String, amis: List[Personnage])  
  case Humain(nom: String, age: Int, amiProche: Option[Personnage])  
  
val superman = SuperHero("Superman", "Force surhumaine", Nil)  
val batman = SuperHero("Batman", "Intelligence et richesse", List(superman))  
val lois = Humain("Lois Lane", 35, Option(superman))  
val clark = Humain("Clark Kent", 35, Option(lois))
```

Pattern matching avec extraction

- On va se servir de la structure des ADT pour faire des matching imbriqués, tout en destructurant

```
def optim(expr: Expr): Expr = expr match
  case Mult(Nombre(0), _) => Nombre(0) // 0 * x == 0
  case Mult(_, Nombre(0)) => Nombre(0) // x * 0 == 0
  // (a * b) + (a * d) == a * (b + d)
  case Plus(Mult(a, b), Mult(c, d)) if a == c => a * (b + d)
  // ...
  case Plus(left, right) => Plus(optim(left), optim(right)) // x + 0 == x
  case Mult(left, right) => Mult(optim(left), optim(right)) // x * 1 == x
  case _ => expr // x == x
```

Extension methods

- Permet des Nouvelles méthodes à des **types existants** sans modifier leur code source.

```
case class Circle(x: Double,  
                  y: Double,  
                  radius: Double)
```

```
extension (c: Circle)  
  def circumference: Double = c.radius * math.Pi * 2
```



Note : IA ?

ADT pour représenter le jeu de UNO

En Scala 3, avec les enums

tu peux séparer Rank dans deux énums pour avoir les ranks avec et sans couleurs ?

Action doit être une énum

Avec un sealed trait en commun pour les Ranks

Avec ça

enum Card:

case ColoredCard(color: Color, rank: RankWithColor)

case SpecialCard(rank: SpecialRank)

enum RankWithColor extends Rank

Tout le code

Voici une représentation possible du jeu de UNO en Scala 3 avec des enums et un sealed trait pour les Ranks :

scala

Copy code

enum Color:

case Red, Green, Blue, Yellow

enum RankWithColor:

case Number(value: Int)

require(value >= 0 && value <= 9, "Number must be between 0 and 9")

enum SpecialRank:

case Skip, Reverse, DrawTwo, Wild, WildDrawFour

sealed trait Rank

object Rank:

given Conversion[RankWithColor, Rank] = identity

given Conversion[SpecialRank, Rank] = identity

enum Card:

case ColoredCard(color: Color, rank: RankWithColor)

case SpecialCard(rank: SpecialRank)

Note : IA ?

Le calcul des règles métier se passe beaucoup mieux.


tu peux déterminer qui est entrain de gagner ? None si la partie n'est pas en cours, Option[Cons[Joueur]] avec plusieurs joueurs si au même nombre de carte.

Juste le code, pas d'exemple, la sortie est une liste non vide

En fait pour le calcul, tu peux juste trouver le nombre de carte minimum et faire un filtre

Voici une fonction pour déterminer qui est en train de gagner dans la partie de Uno :

scala

 Copy code

```
import fp.scala.uno.domain.models._
import fp.scala.uno.domain.models.joueurs._
import fp.scala.utils.models.nel._

def joueurEnTete(partie: PartieDeUno): Option[NEL[Joueur]] = partie match {
  case PartieEnCours(joueurs, _, _, _) =>
    val minCartes = joueurs.joueurs.map(_.main.length).min
    val joueursEnTete = joueurs.joueurs.filter(_.main.length == minCartes)
    Some(NEL.fromListUnsafe(joueursEnTete))
  case _ => None
}
```

The background of the slide features a dark, textured surface with glowing green binary code (0s and 1s) and the letters 'CCD' in a bold, sans-serif font. A large, dark gray triangular shape is positioned on the left side of the slide, pointing towards the center.

Systeme d'effet

La frontière

La gestion des entrées/sorties est un défi important dans la programmation fonctionnelle.

Pour être utilisable, un programme doit gérer les entrées/sorties, mais les mécanismes impliqués sont *“impurs”*.

*La bibliothèque standard de Scala ou le langage ne possède pas, **aujourd’hui**, de type spécifique pour gérer les entrées/sorties.*

Les solutions en Scala

- **Ne pas gérer les effets !**
- C'est le future[A]
 - Scala 2's Future & Try
 - Akka + Akka Streams + Akka ecosystem
- Les effets comme valeur
 - Cats Effect + FS2 (basé sur les typeclasses / 2017)
 - ZIO + ZIO Stream + ZIO ecosystem (plus direct / 2018)
- Les travaux en cours pour pousser plus loin
 - PRE-SIP: Suspended functions and continuations
 - Caprese (Scala 4)'s capabilities

Qu'est-ce qu'un Task[A]/Effect[A] ?

- Un effet est une représentation fonctionnelle, d'un fragment de programme
 - qui si l'on exécute au bout moment
 - jusqu'au bout
 - va produire une valeur de type A
- *De la même manière que la List[A] est une représentation fonctionnelle persistante de 0 ou n valeurs de type A dans un ordre donné.*

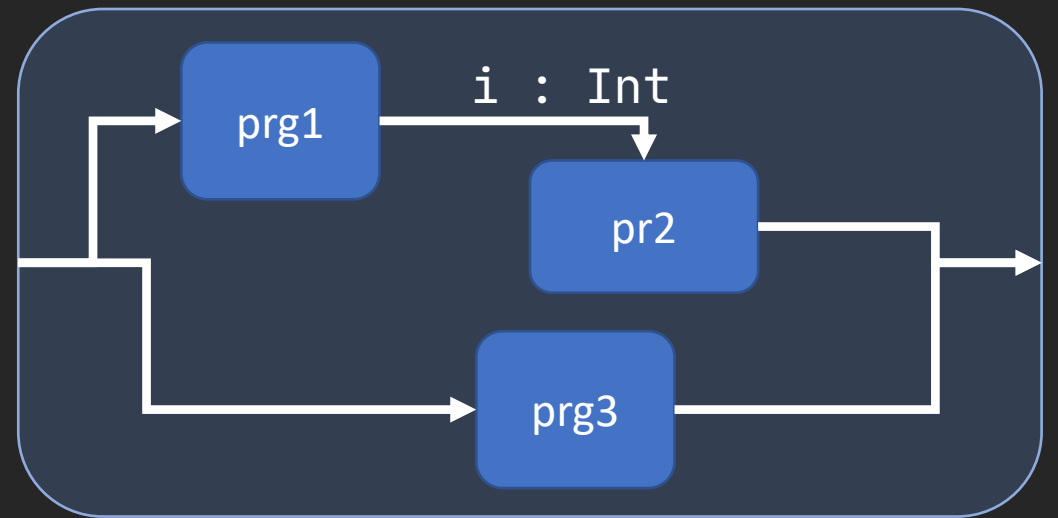
Qu'est-ce qu'un Task[A]/Effect[A] ?

- Modèle de contrôle clair
- Suppression des erreurs de synchronisation
- Parallélisation facile

```
val prg1: Task[Int]
def prg2(i: Int): Task[String]

val prg3: Task[String]

val prg4: Task[String] =
  (prg1 flatMap prg2) race prg3
```



Pourquoi les Future[A] cela ne marche pas ?

- Evaluation immédiate :
 - On voit que l'on a défini le Future, c'est déjà lancé, on ne peut pas déclarer que l'on va faire quelque chose avant ou après.
- La gestion d'erreur

Et Loom ?

- Ben c'est des green-threads, mangez-en !
- Pour information, Loom vient de sortir et devrait passer en LTS en septembre 2023.
 - cela veut dire que l'on va récupérer un nouveau runtime pour nos effets Scala !
- Mais ... ça change rien :
 - On utilise déjà des greenthreads/fibre en Scala
 - C'est comme les lambdas ...

<https://www.youtube.com/watch?v=SJeAb-XEle8>

```
import java.util.concurrent.*;
```

```
public class LoomExample {  
    public static void main(String[] args) throws InterruptedException {  
        ExecutorService executor = Executors.newVirtualThreadExecutor();  
  
        Runnable task1 = () -> System.out.println("Hello from Task 1");  
        Runnable task2 = () -> System.out.println("Hello from Task 2");  
  
        executor.submit(task1);  
        executor.submit(task2);  
  
        executor.await...  
        executor.shutdown();  
    }  
}
```

```
import zio._  
import zio.Console.println
```

```
object ZIOMain extends ZIOAppDefault {  
    val task1 = println("Hello from Task 1")  
    val task2 = println("Hello from Task 2")  
  
    val concurrentTasks = task1.zip(task2)  
  
    override def run = concurrentTasks.exitCode  
}
```

Qu'est-ce qu'une "Fiber" ?

- Les fibers sont une primitive de concurrence beaucoup plus légère que les threads.

Tâches



Fiber



Exécuteur concurrent

Threads



Attention à l'attribution des exécutions dans la bonne "pool"

Il y a plusieurs types d'exécution :

- Limité par le CPU (CPU-bound)
- Entrée-sortie bloquante (Blocking IO)
- Sondage d'entrée-sortie non bloquante (Non-blocking IO polling)

Et chaque catégorie a une configuration et un usage optimal différents !

Opérateurs ou Fibers ?

```
val prg1: Task[Int]
def prg2(i: Int): Task[String]
val prg3: Task[String]
```

```
val prg4: Task[String] =
  (prg1 flatMap prg2) race prg3
```

```
for {
  f1 <- prg1.fork
  f3 <- blocking(prg3).fork
  v1 <- f1.join
  r   <- prg2(v1).race(f3.join)
} yield r
```

```
import zio.direct.*

defer {
  val get: IO[Nothing, Seq[Joueur]] = getJoueurs(req.joueurs)
  val joueurs: Seq[Joueur] = get.run
  //création de la commande de préparation de partie
  val unoCommand: UnoCommand.PreparerUnePartie =
    UnoCommand.PreparerUnePartie(joueurs, ListeDesCartes.pioche)

  //récupération du service
  val ch: UnoAPIDeps = ZIO.service[UnoCommandHandler].run

  //traitement de la commande
  val process: ZIO[Any, EndpointsError, Seq[RepositoryEvent[UnoEvent]]] =
    ch.processCommand(processUid, aggregateUid, unoCommand)
    .mapError(UnoAPIError.toEndpointsError)
  val events: Seq[RepositoryEvent[UnoEvent]] = process.run

  CRUDResult(aggregateUid.safeUUID)
}
```

mode direct



The background of the slide features a dark, textured surface with glowing green binary code (0s and 1s) and the letters 'CCD' in a large, bold, green font. A large, dark gray, semi-transparent rectangle is positioned over the right side of the image, serving as a backdrop for the title text.

Gestion des erreurs

On a un ADT pour ça aussi !

```
sealed trait Cause[+E]
```

```
object Cause {  
  case object Empty extends Cause[Nothing]  
  case class Fail[+E](value: E, trace: zio.StackTrace) extends Cause[E]  
  case class Die(value: Throwable, trace: zio.StackTrace) extends Cause[Nothing]  
  case class Interrupt(fiberId: zio.FiberId, trace: zio.StackTrace) extends Cause[E]  
  case class Then[+E](left: Cause[E], right: Cause[E]) extends Cause[E]  
  case class Both[+E](left: Cause[E], right: Cause[E]) extends Cause[E]  
}
```

<https://univalence.io/blog/drafts/zio-la-gestion-des-erreurs-avec-causee>



Gestion des ressources

Pourquoi on ne peut pas utiliser juste l'injection de dépendance ?

- Si on utilise la construction des dépendances sans modéliser le cycle de vie, on ne peut pas :
 - Gestion des erreurs et des interruptions
 - Portée des ressources
 - On veut utiliser que la ressource pendant un set d'appels concurrents, et ensuite la libérer
- Et en plus on a de la composition




John A De Goes  @jdegoes

Simpler laws for ZIO Environment:

1. For **global deps**, like **services that a class depends on**, use **constructor-based DI**, with **layers for creation & wireup**.
2. For **local deps**, which are **introduced & eliminated (resources, state, security, connections)**, use **ZIO Environment**.



John A De Goes  @jdegoes · Nov 23, 2021

The 3 Laws of ZIO Environment:

1. **Methods inside service definitions (traits) should NEVER use the environment**
2. **Service implementations (classes) should accept all dependencies in constructor**
3. **All other code ('business logic') should use the environment to consume services** [twitter.com/Krever01/statu...](https://twitter.com/Krever01/status/1498888888888888888)

6:32 PM · Dec 6, 2022

8 Retweets 1 Quote 55 Likes 12 Bookmarks



John A De Goes  @jdegoes · Dec 6, 2022

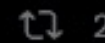
Replying to @jdegoes

Thus, when deciding whether to use ZIO Environment or something else, the fundamental question to ask is whether the Thing that You Need is created at application startup, or whether it is created locally.

If the former, use constructors (w/layers), otherwise, use Environment.



2



2



15

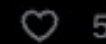


John A De Goes  @jdegoes · Dec 6, 2022

Advanced, optional point, to be ignored by most: use ZIO Environment for expressing business logic (including tests) or for eliminating threading-related boilerplate of dependencies (global or local) into third-party libraries.



2



5



Injection de dépendance, ainsi que le cycle de vie des ressources nécessaires au fonctionnement d'un programme reste "compliqué".

<https://degoes.net/articles/zio-environment>



CONCLUSION

Rappel des idées que l'on a abordées

- FP \neq Lambda
 - L'immutabilité compte beaucoup
- Depuis 5 ans, on a des modélisations hyper puissantes pour ce qui était pénible en fonctionnel
 - Et pas si compliqué / (peut-être pas facile au début)

Ressources

- <https://univalence.io/blog/>
- <https://www.fp-tower.com/>
 - <https://github.com/fp-tower/foundations>
 - <https://blog.fp-tower.com/foundations/generic-functions/generic-functions-part-1.html#1>
 - <https://blog.fp-tower.com/foundations/generic-functions/generic-functions-part-2.html#1>
- <https://nrinaudo.github.io/>
 - https://nrinaudo.github.io/talks/far_more_adt.html
- <https://zio.dev/>
- **Youtube :**
 - <https://www.youtube.com/@scalaio>
 - <https://www.youtube.com/@Ziverge>
 - https://www.youtube.com/watch?v=XUwynbWUlhg&list=PLvdARMfvom9C8ss18he1P5vOcocawm5uC&ab_channel=Ziverge
 - <https://www.youtube.com/@rockthejvm>
- Contenu spécifique :
 - <https://www.slideshare.net/deusaquilus/ziodirect-functional-scala-2022>
 - https://www.youtube.com/watch?v=g_jP47HFpWA&t=2245s&ab_channel=ScalaIOFR
 - <https://sderosiaux.medium.com/why-referential-transparency-matters-7c179424dab5>



MERCI
DE NOUS AVOIR
SUIVIS