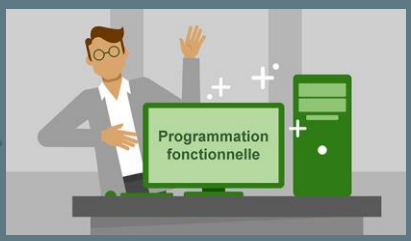
EN DIRECT AVEC LEMARSON

Les bonnes pratiques de la programmation fonctionnelle

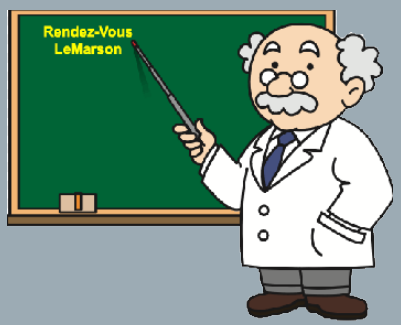


Émission animée
par Claude Marson

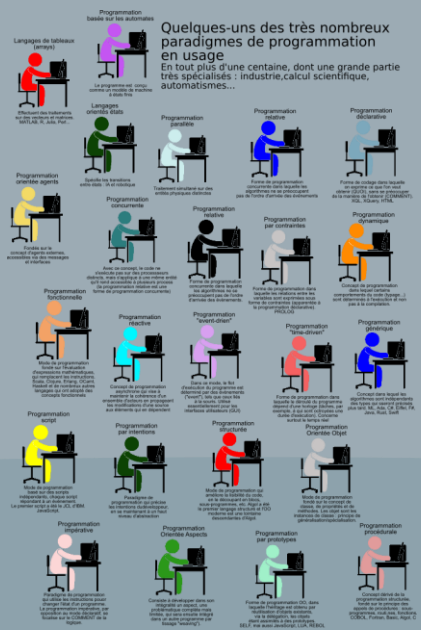
Le sommaire aujourd'hui

- Ce n'est pas un cours de programmation, nous tentons seulement de dégager les principaux concepts de programmation fonctionnelle et de les expliquer
- Nous n'entrerons pas dans les polémiques qui consiste à savoir si tel langage est fonctionnel ou non
- Les langages n'implémentent pas nécessairement les concepts fonctionnels de la même manière

- La mouvance déclarative
- Les grands principes
- La programmation fonctionnelle, c'est quoi ?
- A la base de tout, les fonctions
- La composabilité
- Fonction d'ordre supérieur : expressivité
- Fonctions lambdas et anonymes
- Fonction pure : principe clé de la programmation fonctionnelle
- La "currification" et les applications partielles
- La transparence référentielle
- L'immutabilité
- Les fonctions de première classe ("citoyenneté")
- La récursivité
- Les "closures"
- Les "pour" et les "contre"
- Le déploiement des concepts



La programmation fonctionnelle



La programmation fonctionnelle

La mouvance déclarative

- Principe de la programmation déclarative : création d'applications à l'aide de composants logiciels indépendants du contexte et ne comportant aucun état interne (Wikipedia). Donc l'appel à ces composants avec les mêmes arguments produisent les mêmes résultats, quel que soit le contexte d'appel
- On décrit le quoi et pas le comment
- Plusieurs formes :
  - Programmation descriptive, à expressivité réduite, qui décrit des structures de données : HTML, SQL
  - Programmation fonctionnelle : une application est un montage de fonctions mathématiques : LISP, Scala, Haskell, CAML
  - Programmation logique, pour laquelle les composants de l'application sont des relations logiques : Prolog
  - Programmation par contraintes

La programmation fonctionnelle

Les grands principes

Fonctions pures
Des fonctions qui ne font pas intervenir des variables qui se situent en dehors de son scope.

Curryfication
Remplacement d'une fonction à N arguments par N focntions à un seul argument.

Transparence référentielle
Caractéristique d'une fonction qui fournit le même résultat quel que soit le contexte, qui doit donc être pure et déterministe.

Fonctions d'ordre supérieur
Des fonctions qui s'emboîtent les unes dans les autres, comme des poupées russes.

Immutabilité
Se dit des objets, variables, classes, dont les paramètres ne changent pas.
Principe essentiel en PF, appliqué dans tous les langages qui s'y réfèrent.

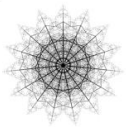
Lambdas et fonctions anonymes
Des fonctions qui n'ont pas de nom et que l'on intègre dans le code de différentes manières, telle qu'une variable en JavaScript.

Composabilité
Recouvre les combinaisons des fonctions, la construction en "lego" d'une nouvelle fonction. En relation directe avec les fonctions mathématiques.

Fermetures ("closures")
Fonctions qui accèdent aux variables non locales, qui constituent son environnement lexical.

Applications partielles
ou fonctions partielles,
sont des fonctions qui agissent sur un nombre réduit d'arguments, les autres étant valorisés explicitement.

Fonctions de première classe
Fonctions qui ont le même statut qu'une variable. Elle peut être typée, nommée, affectée...

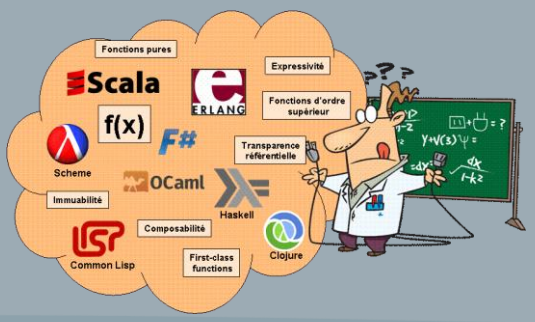


Récursivité
Le fait qu'une fonction peut s'appeler elle-même, dans un algorithme qui peut être récurrent.

La programmation fonctionnelle

La programmation fonctionnelle, c'est quoi ?

- Fondée sur les fonctions, appartient à la mouvance la la programmation déclarative
- Ensemble de bonnes pratiques, plus qu'un véritable langage, bien que Scala, Haskell… inspirées du formalisme mathématique
- Volonté d'améliorer la lisibilité et la robustesse du code, plus que les performances
- La plupart des langages modernes comportent des artefacts fonctionnels
- Origine en 1930 : travaux d'Alonzo Church, qui avait imagine un système formel (λ-calcul, lambda-calcul), fondé sur des fonctions et applications, avec les premières références à la récursivité
- Les langages fonctionnels utilisent des types et des structures de données de haut niveau, tels que des listes extensibles, sur lesquels on peut appliquer des traitements globaux, la concaténation de listes par exemple, le tout en une seule ligne de code.



La programmation fonctionnelle

A la base de tout, les fonctions

- La programmation classique est fondée sur le principe de la machine à états. Le langage ayant pour finalité de faire évoluer les variables de cette machine, dans une série d'itérations.
- La programmation fonctionnelle est fondée sur la notion de fonction mathématique, $y = f(x_1, x_2 \ldots x_n)$, qui a un ensemble de valeurs x_1, $x_2 \ldots x_n$, fait correspondre une seule valeur y. Les mathématiciens disent qu'une fonction mathématique fait correspondre à un n-uple, une seule valeur en résultat.
- En programmation fonctionnelle, le programme principal est considéré comme une fonction, qui fait appel à d'autres fonctions, qui elles-mêmes…etc.
- Le principe de la PF est donc de définir des fonctions que l'on organise les unes par rapport aux autres : une fonction pouvant en contenir une autre, voire en produire une comme résultat

La programmation fonctionnelle

La composabilité

- Même principe qu'en mathématiques :
- y = f(x) et z=g(x)
- Si x s'y prête, z = g(f(x)), le résultat de la fonction f est l'argument de la fonction z, on dit que les fonctions sont composées
  - 1ère fonction : `val f = (x : Int) => x + 2` // consiste à incrémenter la variable x de la valeur 2
  - 2ème fonction : `val g = (x : Int) => x * 5` // multiplie par 5 la valeur de la variable (les deux fonctions ont le même type).
- La composition des deux fonctions revient à les "emboîter" :
  - `val h = (f :Int => Int, g: Int) => g(f(x)` // la première fonction f fournit un entier à la deuxième fonction g
- En Scala : `def compose[A, B, C](f: A => B, g: B => C): (A => C) = x =>g(f(x))`

La programmation fonctionnelle

Fonction d'ordre supérieur : expressivité

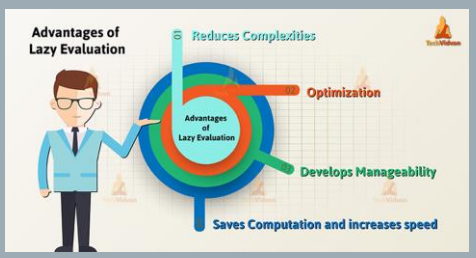
- Les fonctions dites d'ordre supérieures, HoF (Higher Order Functions prennent en entrée une ou plusieurs autres fonctions et retournent en sortie une fonction. C'est l'emboîtement des poupées russes.
- Les fonctions dites d'ordre supérieur prennent en entrée une ou plusieurs fonctions et peuvent retourner des fonctions en sortie : c'est l'expressivité ou l'effet gigogne des poupées russes. L'un des aspects les plus pertinents de la programmation fonctionnelle.

EN DIRECT AVEC LEMARSON

La programmation fonctionnelle

Fonctions lambdas et anonymes

- Fonctions anonymes : fonctions qui n'ont pas de noms, à la place de leur nom, on trouve les instructions, introduites avec une syntaxe particulière.
- Fonctions lambdas : une fonction anonyme, sa définition se fait sans déclaration explicite du type de retour, ni de modificateurs d'accès ni de nom. C'est un raccourci syntaxique qui permet de définir une méthode directement à l'endroit où elle est utilisée.
- La manière d'évaluer une expression permet aussi de distinguer les langages fonctionnels où qui se réfèrent comme tels



La programmation fonctionnelle

Fonction pure : principe clé de la programmation fonctionnelle

- Un programme peut comporter de nombreuses méthodes ou fonctions
  - Une variable globale est définie de façon générique et peut être accédée par n'importe quelle méthode ou fonction
  - Une variable est locale, lorsqu'elle n'existe que dans le corps de la méthode ou fonction
- **Une fonction pure est une fonction qui n'entraîne pas d'effets de bords**, c'est-à-dire qu'elle ne récupère ni ne modifie des variables, qui ne sont pas à sa portée.
- Pour être pure, une fonction doit être prévisible : on doit connaître sa valeur de retour avant qu'elle soit exécutée
- Une fonction pure doit prendre au moins un argument et retourner quelque chose
- La pureté est absolue et il n'est pas possible de ne coder qu'avec des fonctions pures, mais l'objectif de la programmation fonctionnelle est de minimiser les effets indésirables des fonctions impures
- Les bons programmeurs ne dépassent pas 20 % de code impur et atteignent parfois 10 %
- L'usage des fonctions pures rend le code plus propre, plus facile à mettre au point

En JavaScript :

```
let x = 10;
function ajouteDeX(monNombre) {
     return monNombre + x;
}

function modifieX(nouveauX){
     x = nouveauX;
}
```

A l'exécution :

```
ajouteDeX( monNombre: 5); //15
modifieX( nouveauX: 8);
ajouteDeX( monNombre: 4); //12
```

Les deux fonctions sont impures, car x est défini à l'extérieur de leur scope

La programmation fonctionnelle

Fonction pure : principe clé de la programmation fonctionnelle

```
f(i) {
        return i + 2 ;
}
f(1) -> 3
f(1) -> 3
```

- Cette fonction est pure, car elle effectue un calcul uniquement sur des données internes et si on la rappelle plusieurs fois, elle donnera toujours le même résultat.
- Par contre,

```
j =3 ;
f(i) {
        j = i + j ;
        return j ;
}
f(1) -> 4
f(1) -> 5
```

- est une fonction impure, car elle fait intervenir la variable j, définie en dehors de son périmètre. Et si on l'appelle deux fois, elle donnera des résultats différents.
- Une fonction pure doit donc respecter trois principes :
  - ne pas entraîner d'effet de bord et donc modifier des variables définies en dehors de son scope
  - être prévisible, ce qui exclut « de facto » les appels asynchrones comme Ajax ou les « promesses » chères à JavaScript
  - toujours retourner quelque chose, prendre un ou plusieurs arguments en entrée et retourner un résultat.

La programmation fonctionnelle

La "curryfication" et les applications partielles

- « curryfication », du nom d'Haskell Curry, mathématicien américain, connu pour ses travaux sur la logique mathématique et créateur du langage éponyme.
- L'idée de Curry est simple : plutôt que de concevoir des fonctions portant sur N arguments et pouvant ne pas être pures, il vaut mieux les remplacer par N fonctions qui n'agissent que sur un seul paramètre.
- Ne pas confondre avec le concept d'application partielle, qui consiste seulement à **réduire** le nombre des arguments.
- En Python, les `functools.partial` : on décide d'utiliser une fonction f(v1, v2, v3, v4) avec moins d'arguments, telle que f(v1, v2), les autres variables étant fixées.
- Python comporte en natif plusieurs fonctions qui sont prêtes à être « partiellisées ».
- Ex d'une fonction à cinq arguments :

```
def maFonction(arg1, arg2, arg3, arg4, arg5) :
print(arg1, arg2, arg3, arg4, arg5)
```

- on pourra fixer certains arguments :

```
from functools import partial
liste1 = partial(maFonction, 10) # ce qui revient à fixer arg1 à la valeur 10
maFonction(2,3,4,5)
```

retournera la suite 10, 2, 3, 4, 5

EN DIRECT AVEC LEMARSON

La programmation fonctionnelle

La transparence référentielle

- L'exécution d'une fonction ne dépend pas de l'environnement dans laquelle elle se trouve et donnera toujours les mêmes résultats.
- Elle doit être pure et déterministe. Le déterminisme voulant dire que si on fournit les mêmes arguments à une fonction, elle fournira toujours le même résultat.
- La fonction addition en Scala :

```
def addition(x : Int) = {
return x +1
}
```

est déterministe. Si on saisit une succession de **print(addition(10)**, on obtiendra toujours 11.

- La fonction :

```
x = 12
def addition() = {
x = x + 1
return x
}
```

- n'est pas déterministe. Si on saisit **print(addition()** deux fois de suite, on obtiendra successivement 13 et 14.
- On voit le lien avec les fonctions pures, car le non déterminisme de la fonction est lié au fait que x est défini en dehors de la fonction.

La programmation fonctionnelle

L'immutabilité

- Une variable (latin "variabilis", "changeant") dite immutable, n'est jamais modifiée, dès lors qu'elle a été déclarée et initialisée.
- La grande différence avec un langage classique, est que si on doit modifier une variable, on va créer une autre version, sans toucher à la première.
- Avantage : ça limite les effets de bords.
- On parle aussi de classe et d'objet immutable, qui étend le concept à toutes les variables qui peuvent être exportées ou visibles de l'extérieur, les autres devant être privées.
- Avantages :
  - ils ne peuvent pas être perturbés dans un fonctionnement parallèle par des threads concurrents
  - on peut les mettre en cache
  - ils constituent d'excellentes clés pour les Map et Set
  - il n'est pas nécessaire d'en faire une copie « défensive » pour parer à toute éventualité, etc.

La programmation fonctionnelle

La transformation d'une classe "normale" en immutable

- La plupart des langages prévoient des dispositions pour transformer une classe en classe immutable.
- Revient à prévoir des mécanismes pour créer une nouvelle instance, dès lors qu'un paramètre de la classe change. Toute modification de la classe entraînera la création d'une nouvelle instance, tout comme un langage procédural créera une nouvelle version de chacune des variables modifiées.
- L'exemple en Scala illustre cette organisation :

```
case class Point(x: Int, y: Int) {
def deplacementAbscisse(v : Int): Point = copy(x = x + v)
def deplacementOrdonneee(v : Int) : Point = copy(y = y + v)
}
```

- Les fonctions **deplacementAbscisse** et **deplacementOrdonnee**, créent chacune une nouvelle instance de la classe Point grâce à un « **copy** ».
- De cette manière, les données x et y, que l'on décrirait en Java avec un final, ne sont pas modifiées et le principe d'immutabilité est respecté.

La programmation fonctionnelle

L'immutabilité en Java

- Un élément, une variable, est immutable si sa valeur ne change jamais
- Ex de Java
  - Un élément immutable est doté du qualificatif ***final,*** il ne peut pas être modifié dans la suite du programme (pas de ***setter*** ailleurs).
  - Il peut s'appliquer aux attributs ou aux méthodes d'une classe
  - Les éléments mutables : variables de classe non finales, les objets distants, les service stateful, les bases de données, créent un couplage fort , ce que l'on veut éviter.
  - La référence à this ne doit jamais être exportée.
  - Tous les champs qui se réfèrent à un objet mutable doivent être privés et ne seront pas exportés
- Ex de deux classes dont le but est de filtrer une liste d'entiers par rapport à un seuil (threshold)

```
class MutableThreshold {
    private int threshold;
    public void setThreshold(int t) {
        threshold = t;
    }
    public List filter(List ints) { /* loop over ints to filter according to threshold */
}
class ImmutableThreshold {
    private final int threshold;
    public ImmutableThreshold(int t) {
        threshold = t;
    }
    public List filter(List ints) { /* loop over ints to filter according to threshold */
}
```

Exemples d'immutabilité en Java, avec ***final***

```
public class MaClasse {
     public final void uneMethode() {
     …
     }
} // cette méthode ne peut être redéfinie dans une classe dérivée
```

L'immutabilité peut être appliquée sur des variables locales, dont on est sûr qu'elles ne changeront pas

```
public static int ajouterCinq(final int a) {
     final int b = 5;
     return a + b;
}
```

Une classe finale ne peut être dérivée :

```
public final class MaClasse {
…
}
```


La programmation fonctionnelle

Les fonctions de première classe ("citoyenneté")

- Une fonction est de première classe, quand on peut la traiter comme n'importe quelle variable : un entier, une chaîne de caractères… On dit qu'elle a le même statut.
- Elle peut être transmise en tant qu'argument à d'autres fonctions, peut être retournée par une autre fonction et peut être affectée en tant que valeur à une variable.
- On peut donc agir dessus de la même manière que sur une variable classique :
- On peut la nommer, la typer et l'affecter :
  `x := sin;`
- On peut la définir et la créer à la demande :
  `x := (function x-> x+1);`
- On peut la passer en argument à une autre fonction :
  `f(sin);`
- Elle peut être le résultat d'une fonction :
  `(f(5))(7);`
- Elle peut être stockée dans une structure de données quelconque :
  `array := {log, sin, cos, tan};`

La programmation fonctionnelle

La récursivité

- Une fonction est dite récursive, quand elle peut s'appeler elle-même, dans une sorte de boucle dans laquelle la fonction « travaille » sur des arguments différents.
- En mathématiques, c'est approximativement ce que désigne le raisonnement par récurrence, bien qu'il y ait quelques différences sémantiques.
- Le meilleur exemple pour comprendre ce qu'est le calcul par récursivité est celui d'une factorielle !n, qui comme chacun sait est le produit des n premiers nombres, si n est un nombre entier :
- **!n = n(n-1)(n-2)…3.2.1**

La programmation fonctionnelle

Les "closures"

- Une fermeture ou « closure » est une fonction qui accède à des variables qui se situent en dehors de son contexte local (« scope »). C'est le cas lorsque l'on intégre une fonction A dans une fonction B et que la fonction A accèdera aux variables de B.
- On dit que la fonction est "accompagnée" de son environnement lexical, constitué de l'ensemble des variables non locales, qu'elle capture, par valeur ou référence.
- Une « closure » peut être passée en argument d'une fonction, dans l'environnement où elle a été créée et on parlera de « passage vers le bas » ou renvoyée comme valeur de retour et on évoquera alors un « passage vers le haut ».

La programmation fonctionnelle

Le déploiement des concepts

- Scala (utilisé par LinkedIn et Twitter), Haskell (utilisé par Facebook et Google), Erlang (utilisé par Yahoo !, Facebook et GitHub), Common Lisp, Scheme, Ocaml, F# de Microsoft ou Clojure.
- D'autres langages ont été améliorés qui intègrent des paradigmes fonctionnels, tels que Python, PHP (fonctions « first-class », lambdas), Java (expressions Lambda depuis Java 8), C# (fonctions « first-class ») et C++ (depuis la version 11).
- Même JavaScript insère des concepts fonctionnels avec les « polyfill », de même que Swift, le nouveau langage pour iOS d'Apple ou encore Ruby.

La programmation fonctionnelle

Les "pour"

- Abstraction du concept. En utilisant des fonctions pures, on cache la complexité des problèmes et en cas de difficultés, on peut facilement cibler les fonctions « coupables », celles qu'il faut corriger.
- La PF est faite pour le traitement parallèle.
- Avec des développeurs habitués aux mécanismes fonctionnels, le temps de codage est réduit de manière significative.
- Elle incorpore des concepts d'évaluation paresseuse (« lazy evaluation »)
- La PF est particulièrement bien adaptée aux nouveaux domaines tels que l'IA
- Code plus court et plus simple

Les "contre"

- La PF n'est pas à l'aise avec les I/O, celles-ci entraînant fatalement des effets de bords.
- Inconvénient, présenté souvent comme un avantage, la récursivité. Qui peut aussi sembler trop gourmande en ressources, mémoire entre autre. Pour éviter cette dérive, il existe des solutions, mais qui induisent d'optimiser les compilateurs ou de réécrire la récursivité de manière spécifique.
- Les développeurs lui reprochent son formalisme mathématique. Qui parfois recouvre des pratiques moins complexes à appliquer qu'à expliquer.
- Reproche suprême, le fonctionnel n'est pas adapté aux traitements itératifs. D'abord parce que les machines ne se comportent pas en « fonctionnel » et qu'ensuite, nous travaillons sur des systèmes à états, pour lesquels la PF n'est pas exploitable.

La PF ne remplacera ni l'objet, ni l'impératif, ce sera seulement une question de circonstances

Nos prochains rendez-vous

Vendredi 14 février 2020	**La fin du scandale des certificats payants ?**
Vendredi 21 février 2020	**La justification financière des projets qualité des données**
Vendredi 28 février 2020	**ITIL v4, vous avez du temps à perdre ?**
Lundi 2 mars 2020	**Actualités du TI**
Lundi 9 mars 2020	**La fin des mots de passe**
Vendredi 13 mars 2020	**IBN, la programmation des réseaux**
Vendredi 20 mars 2020	**LLVM et les run time modernes**
Lundi 30 mars 2020	**Les réseaux LPWAN, le choix entre Lora et Sigfox**

Je vous remercie de votre attention et à bientôt