Codez plus facilement, sans bugs avec la programmation fonctionnelle

C’est quoi la programmation fonctionnelle ?

Et pourquoi l’apprendre ?

C’est un paradigme de programmation

Au même titre que la Programmation Impérative, la POO… (Puristes s’abstenir) 👀

On y manipule deux choses :

- Des fonctions
- Des données

Mise à l’écart pendant 20 ans mais redevient à la mode depuis une dizaine d’années

Répond particulièrement bien aux défis modernes de l’informatique

- Scalabilité
- Programmation concurrente et parallèle

Utile et utilisée dans de nombreux domaines

IA

- Apache Spark
- Databricks
- Microsoft SynapseML

Cloud, Réseaux sociaux, Streaming...

- Microsoft Azure, Clever Cloud
- Twitter, Linkedin, Facebook, WhatsApp...
- Netflix, Disney+, Spotify...

Jeu

- Ankama
- DevSisters

Pourquoi ces entreprises utilisent-elles la FP ?

Découvrons comment grâce à la programmation fonctionelle :

👩‍💻 Avoir un code clair

🚧 Esquiver de nombreux bugs

🚀 Coder plus vite

Langage utilisé pour ce Talk

Le problème

Objectif

Créer un jeu de blackjack en ligne.

Règles (en gros) :

- La mise est décidée par le croupier
- Chaque joueur commence avec deux cartes
  - Cartes de 2-9
  - Roi (10 points)
  - As (1 ou 11 points au choix)
- Chaque joueur peut piocher une troisième carte
- Pour gagner, la somme des cartes doit être inférieure à 21 et supérieure à celle du croupier
- Chacun est contre le croupier et pas contre les autres joueurs

Les étapes d’un round

Happy path + potentielles erreurs (du point de vue du serveur) + effets (Packets TCP, requêtes en BDD…)

- Le serveur indique aux joueurs la mise et les cartes de départ
  - Internet (e.g TCP)
- Le joueur envoie son choix au serveur : piocher une carte, doubler la mise ou ne rien faire
  - Message (e.g JSON) malformé
  - Choix invalide
  - Internet
- Si le joueur pioche, le serveur renvoie la carte piochée
  - Internet
- Le serveur envoie le résultat de la partie
  - Internet

Le code n’est finalement pas si simple

```
val player = ???

val userMessage = getPlayerMessage(player)
val choice = decodePlayerChoice(userMessage)

if choice == DrawCard then
  val card = drawCard()
  player.addCardToHand(card)
  sendPlayerMessage(player, encodeCard(card))
else if choice == DoubleDown then
  player.bet *= 2

sendPlayerMessage(player, encodeResult(getRoundResult()))
```

Charge mentale :

- getPlayerMessage: Blocage + Erreurs possibles (e.g réseau)
- decodePlayerChoice: Erreurs possibles (choix/message invalide)
- drawCard: Erreurs possibles (plus de carte dans la pile)
- sendPlayerMessage: Blocage + Erreurs possibles (e.g réseau)

Vraiment pas simple

```
val player = ???

val userMessage = getPlayerMessage(player)
val choice = decodePlayerChoice(userMessage)

if choice == DrawACard then
  val card = drawCard()
  player.addCardToHand(card)
  sendPlayerMessage(player, encodeCard(card))
else if choice == DoubleDown then
  player.bet *= 2

sendPlayerMessage(player, encodeResult(getRoundResult()))
```


```
val player = ???

val userMessage = getPlayerMessage(player)
val choice = decodePlayerChoice(userMessage)

if choice == DrawACard then
  val card = drawCard()
  player.addCardToHand(card)
  sendPlayerMessage(player, encodeCard(card))
else if choice == DoubleDown then
  player.bet *= 2

sendPlayerMessage(player, encodeResult(getRoundResult()))
```


En même temps


```
val player = ???

val userMessage = getPlayerMessage(player)
val choice = decodePlayerChoice(userMessage)

if choice == DrawACard then
  val card = drawCard()
  player.addCardToHand(card)
  sendPlayerMessage(player, encodeCard(card))
```

```
val player = ???

val userMessage = getPlayerMessage(player)
val choice = decodePlayerChoice(userMessage)

if choice == DrawACard then
  val card = drawCard()
  player.addCardToHand(card)
  sendPlayerMessage(player, encodeCard(card))
```

Plusieurs questions

- Peut-on s’assurer de bien avoir géré toutes les erreurs ?
- Comment gérer le cas où plusieurs jouent sur une même partie ?
- Peut-on gérer plusieurs utilisateurs en même temps

Le tout, en ayant un code plus clair ?


FUNCTIONAL
PROGRAMMING
A SIMPLE STEP-BY-STEP

En programmation fonctionnelle, ces types de fonctions n’existent pas.

Ou du moins, sont évités au maximum.

Nous utilisons à la place des fonctions ✨ pures ✨

Qu’est-ce qu’une fonction pure ?

Une fonction pure est une fonction :

- Totale
- Déterministe
- Sans effet de bord

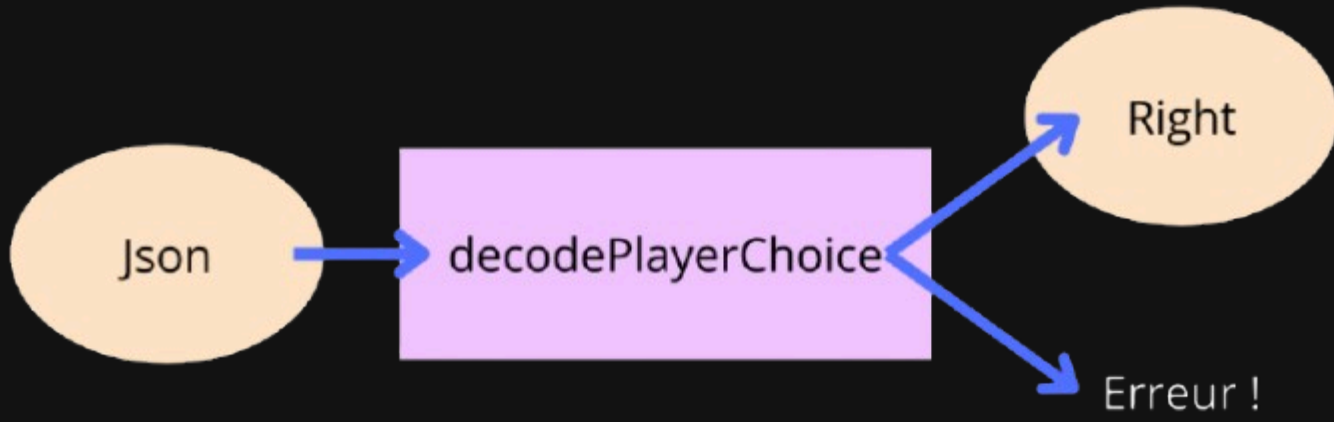
Comme toutes les fonctions mathématiques

```
def plus(x: Int, y: Int): Int = x + y
```

Fonction totale

Fonction non totale

```
def decodePlayerChoice(message: Json): PlayerChoice = ???
```



Au fond, une erreur n’est qu’une valeur parmi d’autres

```
def decodePlayerChoice(message: Json): Either[InvalidChoiceError, PlayerChoice] = ???
```

`Either[L, R]` veut dire "une valeur de type `L` (Left) ou de type `R` (Right)"

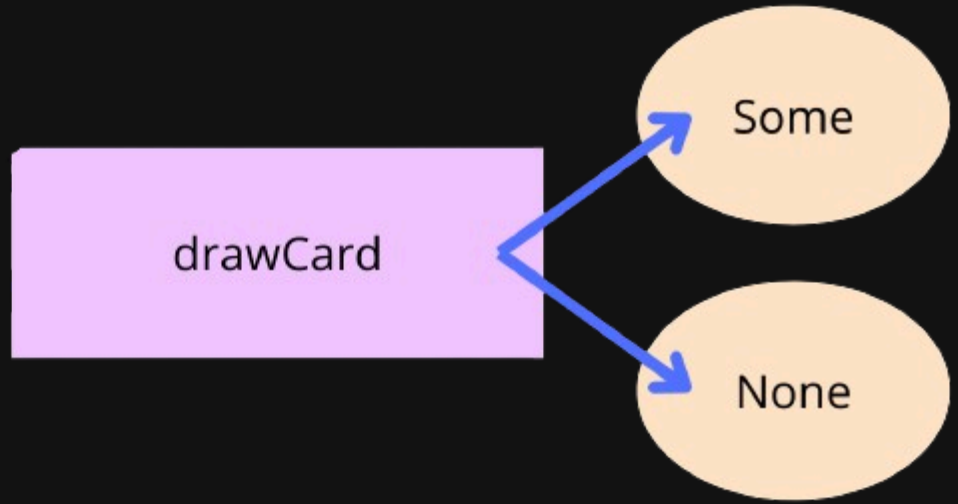
Plus possible d’oublier de gérer le cas d’erreur

```
val choice: PlayerChoice = decodePlayerChoice(json)
```

```
-- [E007] Type Mismatch Error: ----------------------------------------------------
1 |val choice: PlayerChoice = decodePlayerChoice(json)
  |                           ^^^^^^^^^^^^^^^^^^^^^^^^
  |                       Found:    Either[InvalidChoiceError, PlayerChoice]
  |                       Required: PlayerChoice
  |
  | longer explanation available when compiling with `-explain`
1 error found
```

Autre exemple : gestion de l’absence de valeur

```
def drawCard(): Option[Card] = ???
```



Fonction déterministe

Fonction non déterministe

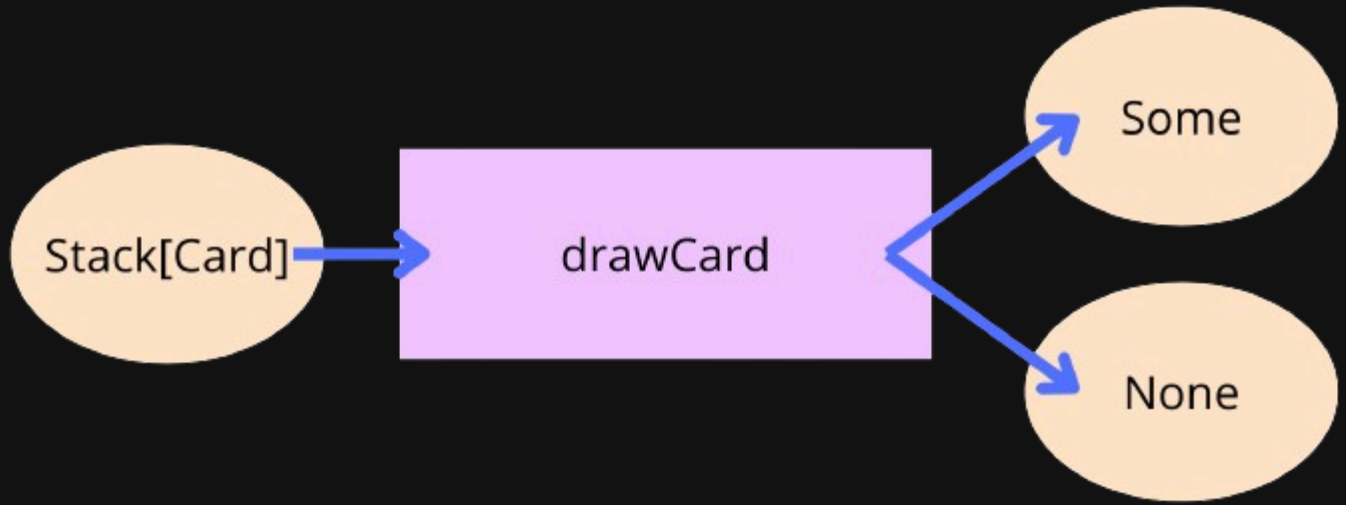
```
val cardsPile: mutable.Stack[Card] = mutable.Stack(cardA)

def drawCard(): Option[Card] = stack.pop()
```

```
drawCard() //Some(cardA)
drawCard() //None
```

Il faut rendre les dépendances explicites

```
def drawCard(cardsPile: mutable.Stack[Card]): Option[Card] = cardsPile.pop()
```

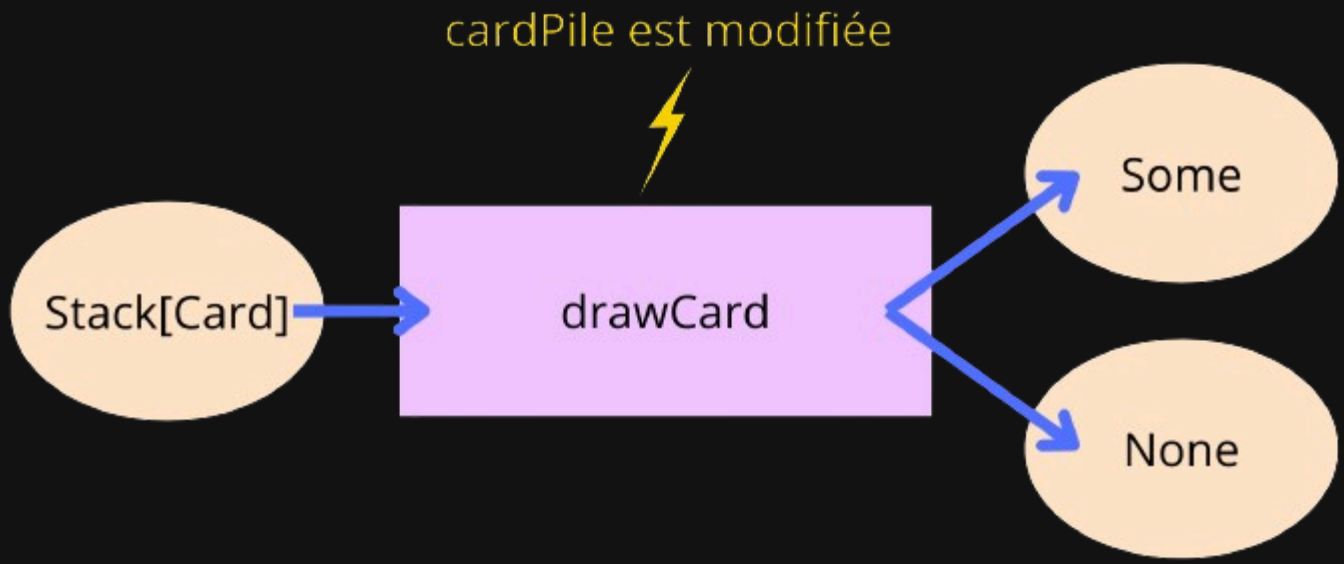


Fonction sans effet de bord

drawCard n’est pas encore pure

La pile est potentiellement modifiée à chaque appel !

```
def drawCard(cardsPile: mutable.Stack[Card]): Option[Card] = cardsPile.pop()
```



Il faut renvoyer le nouvel état de la liste/pile

Cette fois, `drawCard` est réellement pure.

```
def drawCard(cardsPile: List[Card]): Option[(Card, List[Card])] =
  if cardsPile.size > 0 then Some((cardsPile.head, cardsPile.tail))
  else None
```

Qu’est-ce qu’on y gagne ?

Est-ce qu’on ne se serait pas compliqué la vie pour rien ?

Transparence référentielle

Tout appel d'une fonction pure peut-être remplacé par son résultat

```
if decodePlayerChoice(json) == DrawCard then ???
else if decodePlayerChoice(json) == DoubleDown then ???
else ???
```

Il est donc très simple de factoriser plusieurs appels de fonctions en un seul. Aussi pratique pour la mise en cache de données.

```
val choice = decodePlayerChoice(json)

if choice == DrawCard then ???
else if choice == DoubleDown then ???
else ???
```

Avec cette seule propriété, tout devient plus simple

Nos fonctions des valeurs, des briques que nous pouvons composer comme bon nous semble

Exemple : calculer la valeur d'une main

Exemple impératif tel qu'on le ferait en Java ou Python

```
enum Card:
  case Number(value: Int)
  case Jack
  case Queen
  case King
  case Ace

def cardValue(card: Card): Int = card match
  case Number(value)       ⇒ value
  case Jack | Queen | King ⇒ 10
  case Ace                 ⇒ 11
```

```
def handValue(hand: List[Card]): Int =
  var result = 0

  for card ← hand do
    result += cardValue(card)

  return result
```

Mais nous pouvons maintenant faire mieux

```
def handValue(hand: List[Card]): Int = hand
  .map(cardValue)
  .reduce((a, b) ⇒ a + b) //ou plus simple: reduce(_ + _)
```

- `map` est une méthode de `List` qui prend une autre fonction `A ⇒ B` en paramètre et l'applique à chaque élément. Ici :
  - `A` est `Card`
  - `B` est `Int`
- `reduce` est une méthode de `List` qui prend une autre fonction `(A, A) ⇒ A` en paramètre et l'applique entre les éléments. Ici :
  - `A` est `Int`

Exemple plus complexe : Pseudos du top 10 féminin

```scala
enum Gender:
  case Male, Female, Other

case class Player(name: String, gender: Gender)

//true = gagné, false = perdu
case class Game(id: Int, winners: Map[Player, Boolean])

def top10female(players: List[Player], games: List[Game]): List[Player] =
  players
    .sortBy(player ⇒ games.count(_.winners.getOrElse(player, false)))
    .takeRight(10)
    .map(_.name)
```

C’est ce qu’on appelle des monades

Et ce pattern ne s’applique pas qu’aux listes !

Option

Et si on pouvait facilement traiter les valeurs absentes ?

```
def getDBPassword(
  programArguments: Map[String, String],
  configFile: Map[String, String],
  environmentVariables: Map[String, String]
): Option[String] =
  programArguments
    .get("--db-password")
    .orElse(configFile.get("db.password"))
    .orElse(environmentVariables.get("DB_PASSWORD"))

val passwordOrDefault = getDBPassword(programArguments, configFile, environmentVariable).getOrElse("default_password")
```

Configuration

Et si on pouvait facilement déclarer la configuration de notre programme ?

```
case class BlackjackConfig(maxPlayerByRound: Option[Int], maxConcurrentRounds: Int)

val config = (
  prop("config.maxPlayerByRound").or(env("MAX_PLAYER_BY_ROUND")).option,
  prop("config.maxConcurrentRounds").or(env("MAX_CONCURRENT_ROUNDS")).default(1)
).mapN(BlackjackConfig.apply)
```

Tests

Et si on pouvait automatiquement générer nos données de test ?

```
val genderGenerator: Generator[Gender] = oneOf(Gender.values)

val playerGenerator: Generator[User] = (
  stringGenerator,
  genderGenerator
).mapN(User.apply)

val gameGenerator: Generator[Game] = (
  idGenerator,
  playerGenerator
    .zip(booleanGenerator)
    .repeat
    .map(_.toMap)
).mapN(Game.apply)
```

Jusqu’à même des programmes

Et si on pouvait lancer en parallèle des programmes, les relancer s’ils échouent, etc. ?

```
def pingServer(url: String): ZIO[HttpClient, ConnectionError, Int] = ???

val lowestPing: ZIO[HttpClient, ConnectionError, Int] =
  pingServer(euServer)
    .race(usServer)     //Le plus rapide à répondre entre EU et US est gardé
    .timeout(5.seconds) //Si les deux serveurs prennent plus de 5s, on échoue
    .retryN(3)          //On réessaie 3 fois si aucun des deux serveurs ne réussit
```

Live coding

Encodage JSON

Composition automatique des briques

Quand la programmation fonctionnelle rencontre la programmation logique

Pour certaines briques, la composition est "évidente"

Il n'y a pas pas 12 façons différentes de générer des `User` pour nos tests

```
val genderGenerator: Generator[Gender] = oneOf(Gender.values)

val playerGenerator: Generator[User] = (
  stringGenerator,
  genderGenerator
).mapN(User.apply)

val gameGenerator: Generator[Game] = (
  idGenerator,
  playerGenerator
    .zip(booleanGenerator)
    .repeat
    .map(_.toMap)
).mapN(Game.apply)
```

Une machine pourrait le faire à notre place !

Au fait, pourquoi notre langage ne pourrait pas le faire tout seul ?

Certains langages fonctionnels ont cette capacité

Nous pouvons déclarer des preuves que notre langage va automatiquement assembler pour produire ce que nous voulons.

Axiomes

```
given stringGenerator: Generator[String] = oneOf('a' to 'z' ++ 'A' to 'Z')
  .repeat(7)       //List('R', 'a', 'p', 'h', 'a', 'e', 'l')
  .map(_.mkString) //"Raphael"

given intGenerator: Generator[Int] = ???
```

Théorèmes

```
given listGenerator[A](using aGenerator: Generator[A]): Generator[List[A]] =
  aGenerator.repeat(n)
```

Pour tout `A`, si `A` peut être généré, alors une `List[A]` peut être générée.

```
given Generator[EmptyTuple] = _ ⇒ EmptyTuple

given [H, T <: Tuple](using headGenerator: Generator[H], tailGenerator: Generator[T]): Generator[H *: T] = seed ⇒
  headGenerator.generate(seed) *: tailGenerator.generate(seed)
```

Pour tout `A1`, ..., `AN`, si ces types sont générables alors le tuple `(A1, ..., AN)` aussi. Cet exemple est récursif.

```
def derived[A](using m: Mirror.ProductOf[A], fieldsGenerator: Generator[m.MirroredElemTypes]): Generator[A] =
  fieldsGenerator.map(m.fromTuple)
```

Pour tout `F1`, ..., `FN` les types des champs d'une `case class A`, si ces types sont générables alors `A` aussi.

Application à notre jeu de Blackjack

```
val genderGenerator: Generator[Gender] = derived[Gender]

val playerGenerator: Generator[Player] = derived[Player]

val gameGenerator: Generator[Game] = derived[Game]
```

Ou encore plus simple :

```
enum Gender derives Generator:
  case Male, Female, Other

case class Player(name: String, gender: Gender) derives Generator

case class Game(id: Int, winners: Map[Player, Boolean]) derives Generator
```

Résultat:

- Génération de la logique pour nous
- Aucun impact sur les performances
- Si ce n'est pas possible, nous sommes prévenus avant même de lancer le code !

Merci de m’avoir écouté

Des questions ?