Functional Data Programming

Via SCALA

COURS 1

MOHAMED HOUSSAINE Rahamane

eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

Plan

- Programmation fonctionnelle
  - Introduction et contexte
  - Concepts fondamentaux (Immuabilité, Récursion, Fonctions d'ordres supérieurs…)

- SCALA
  - Concepts propres au Scala
  - In depth (Fonctions, Singleton, Pattern Matching…)

Objectifs

1. Définir et expliquer les concepts de base de la programmation fonctionnelle
2. Rédiger des programmes purement fonctionnels en utilisant la récursivité, la correspondance de motifs (pattern matching) et les fonctions d'ordre supérieur
3. Utiliser les fonctionnalités de Scala pour la manipulation des données
4. Interpréter et appliquer les concepts d'immutabilité
5. Concevoir des structures de données immuables
6. Combiner la programmation fonctionnelle avec les objets et les classes

Programmation Fonctionnelle : Introduction

Comment?

Paradigme

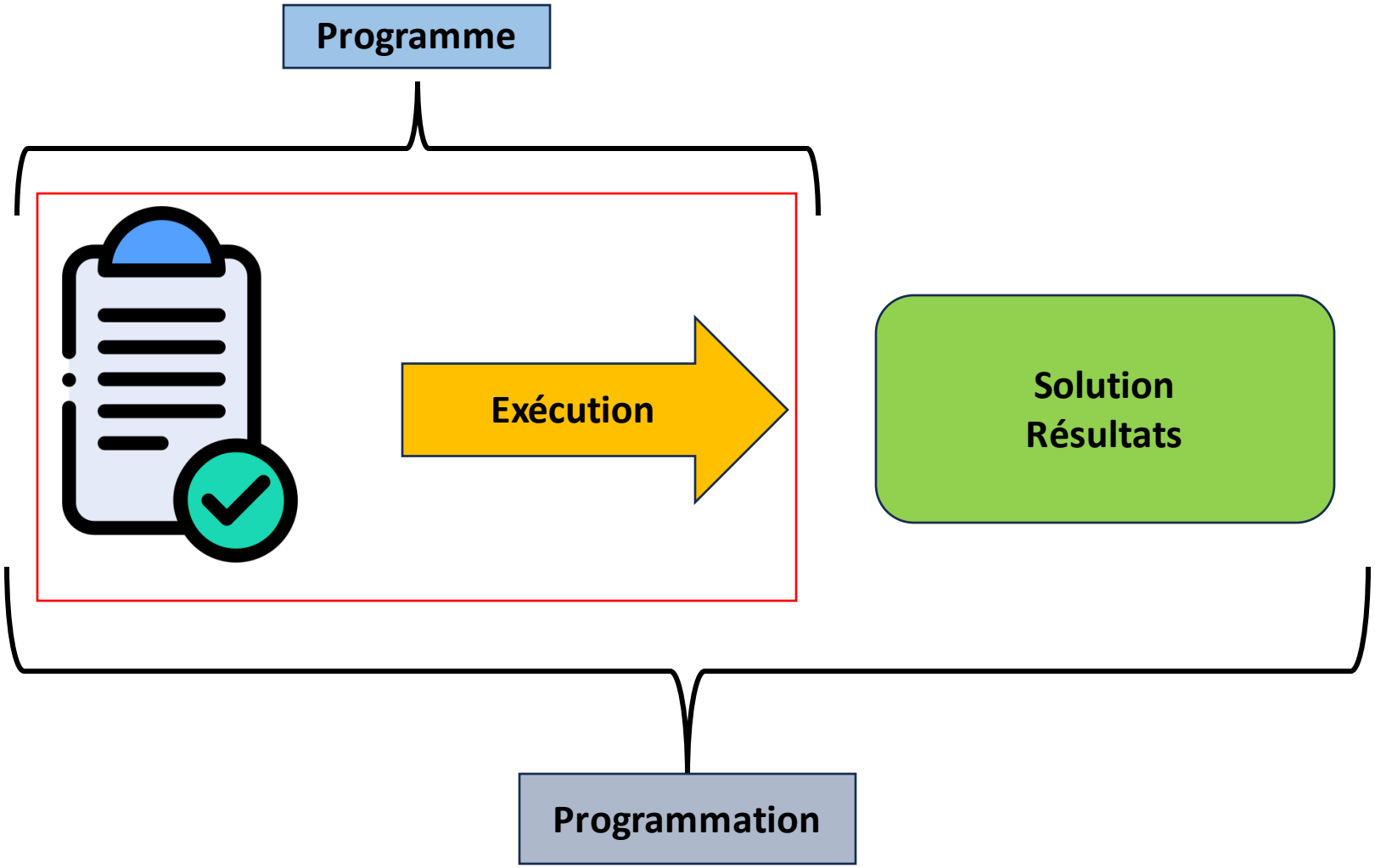
Comment faire?

Impératif

Quoi faire?

Déclaratif

Fonctionnelle



Programmation Fonctionnelle : Introduction

PROGRAMMATION **FONCTIONNELLE**

Instructions **Exécution** **Résultats** **Méthode Approche** **Fonction(s)** **F(x)**

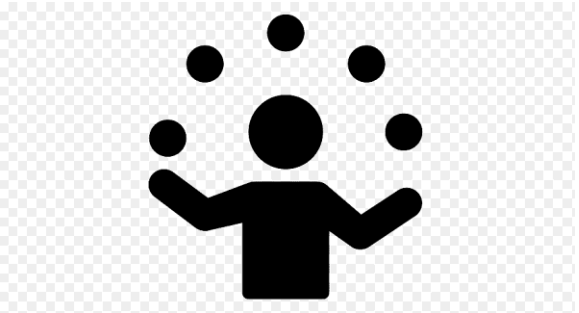
Programme **"Quoi faire?"**

Approche de la programmation de style déclaratif qui se concentre sur l'utilisation de fonctions mathématiques pour résoudre des problèmes.

Programmation Fonctionnelle : Contexte

Complexité croissante des logiciels

Défis dans la mise en place des architectures



Difficultés dans la maintenance des architectures

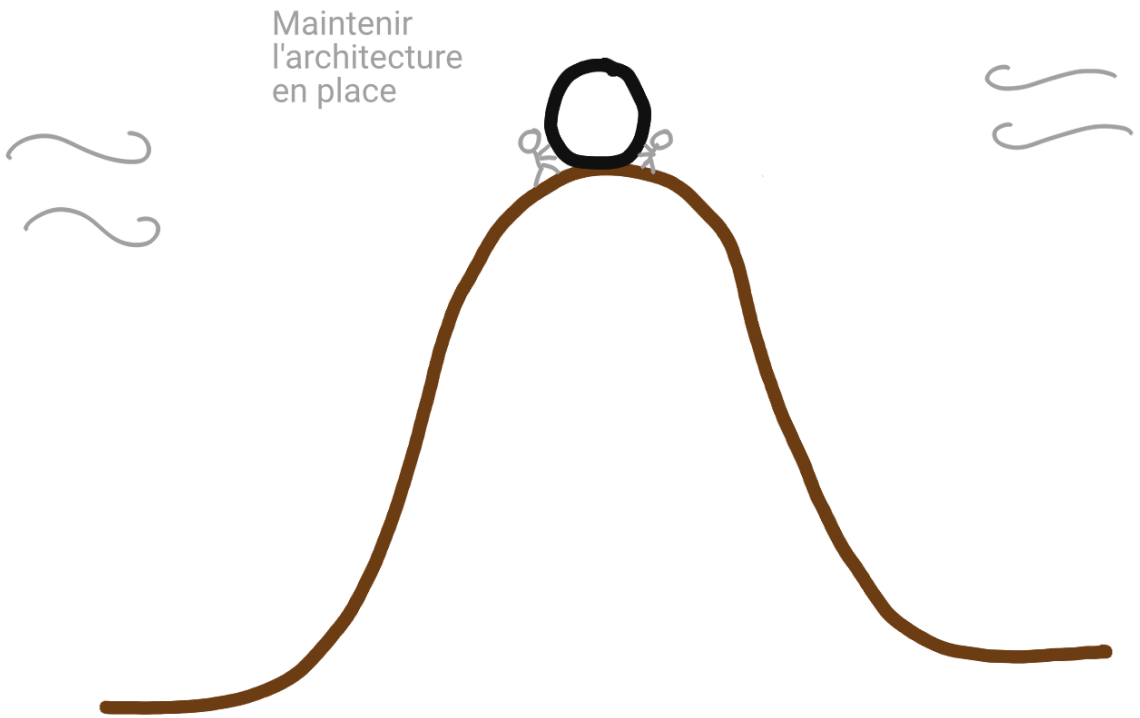
Stratégies pour maintenir l'architecture

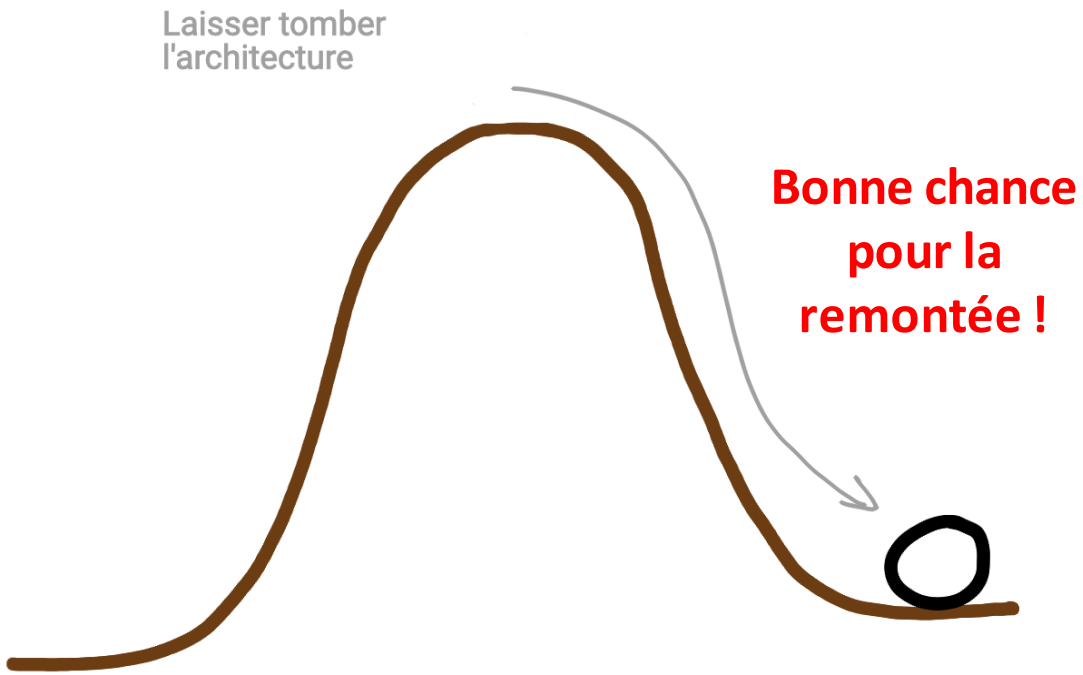
Alternatives proposées

Programmation Fonctionnelle : Contexte

Alternatives proposées

Plus robuste sur le long terme



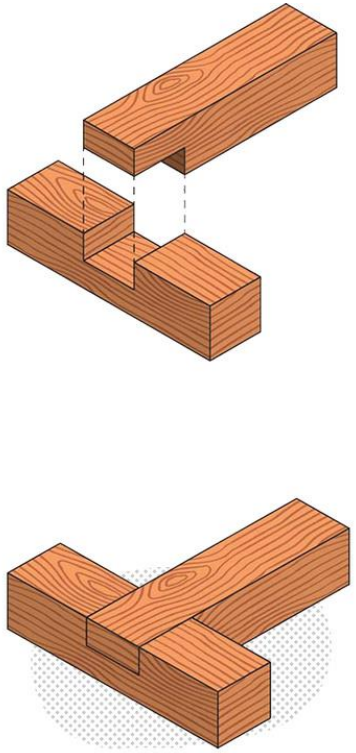


Bonne chance pour la remontée !

Programmation Fonctionnelle : Contexte

Solution à la complexité croissante : La Programmation Fonctionnelle (PF)

Programmation Fonctionnelle



- Modulaire
- Prédictible
- Robuste

Programmation Objet Orienté

Programmation Fonctionnelle : Concepts fondamentaux

Concept n°1

Immutabilité

```
x = 5
x = x + 1
y = x + 1
```

A = Constante 1
B = Constante 2

Programmation Fonctionnelle : Concepts fondamentaux

Concept n°2

Récursivité

```
def fib(n: Int): Int = {
    if (n <= 1) 1
    else fib(n - 1) + fib(n - 2)
}
```

~~for / while~~

Suite de Fibonacci

Programmation Fonctionnelle : Concepts fondamentaux

Concept n°3

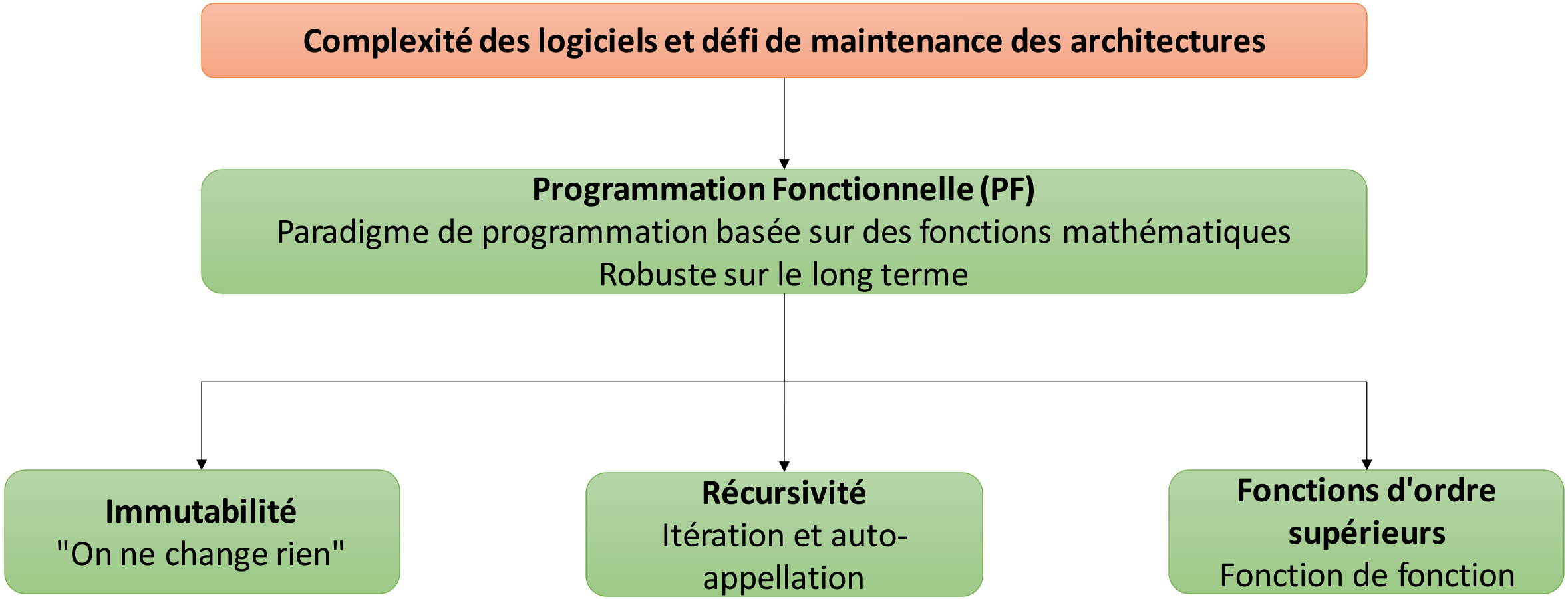
Fonctions d'ordres supérieurs

```
def showOutput(f: () => Unit): Unit = {
        f()
}

def printX(): Unit = {
        println("hello X")
}

showOutput(printX)
```

Fonctions composées

$f \circ g$

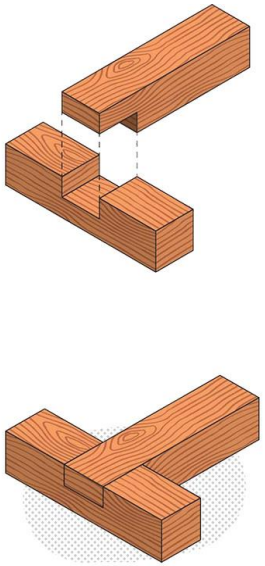
Programmation Fonctionnelle : RECAP



SCALA : Introduction

SCALA : Introduction

Programmation Fonctionnelle



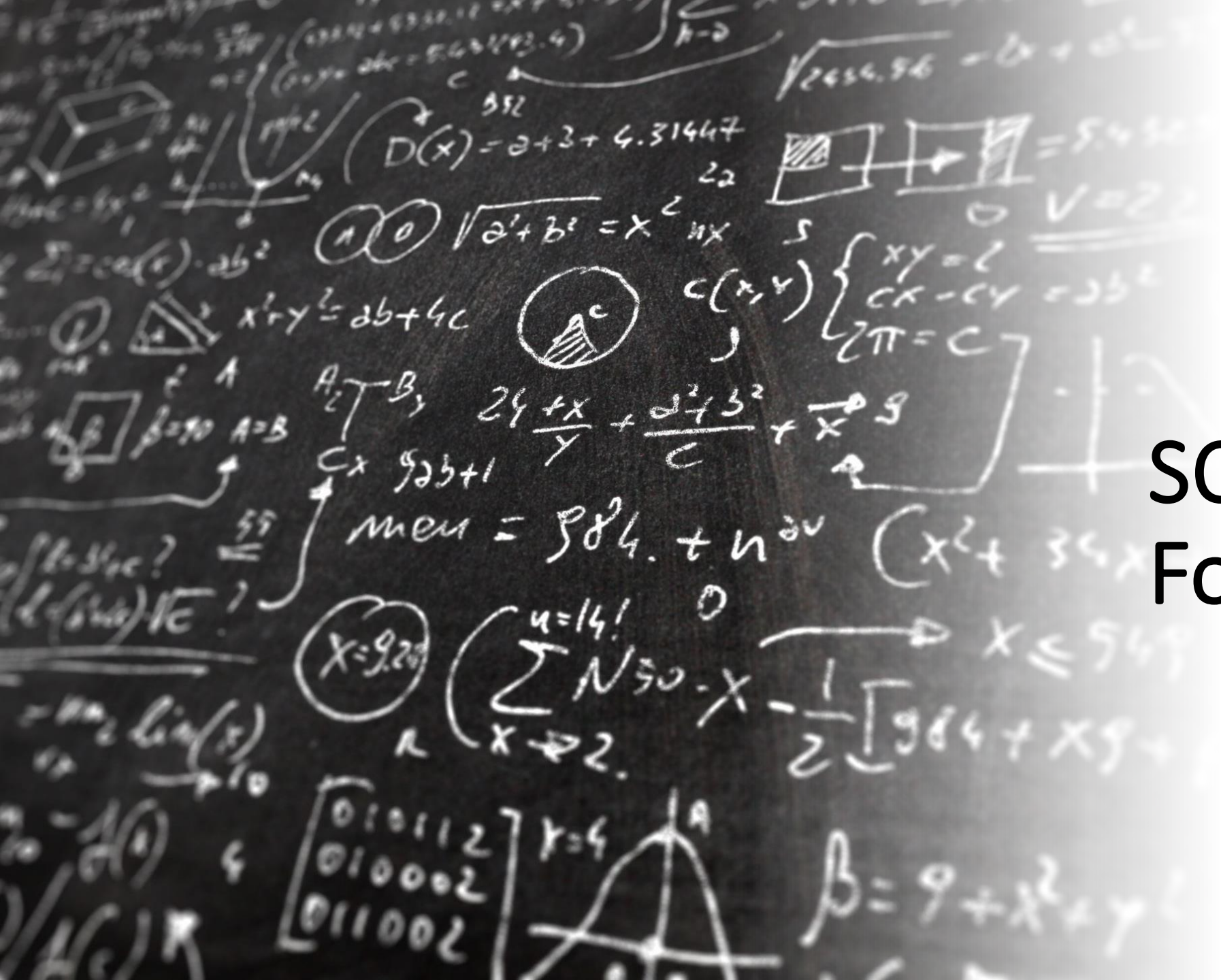
+

Programmation Objet Orienté

immutabilité, fonctions pures…

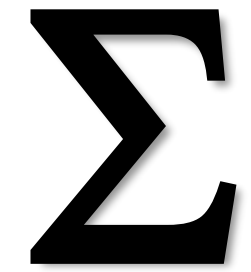
encapsulation, abstraction…

Très peu de lignes
Syntaxe concise



SCALA : Fonctions

SCALA : Fonctions



SCALA : Fonctions

Utilisation de la keyword "def" pour déclarer une fonction	`def greet(): Unit = {`
	`  println("Hello!")`
	`}`

Nom de la fonction en camel case et peut inclure divers caractères	`def calculateDiscountPrice(): Double = {`
	`  // Code pour calculer le prix avec réduction`
	`}`

Liste des paramètres avec leurs types de données	`def calculateArea(width: Double, height: Double): Double =`
	`{`
	`  // Code pour calculer l'aire d'un rectangle`
	`}`

Définition du type de retour de la fonction :	`def isEven(number: Int): Boolean = {`
	`  // Code pour vérifier si le nombre est pair`
	`}`

Possibilité de déclarer une fonction avec ou sans opérateur d'égalité "="	`def greet(): Unit = {`
	`  println("Hello!")`
	`}`
	
	`def greet() {`
	`  println("Hello!")`
	`}`

Le corps de la fonction est inclus entre des accolades "{}"	`def calculateArea(width: Double, height: Double): Double =`
	`{`
	`  val area = width * height`
	`  area`
	`}`

SCALA : Fonctions

Exercice 1
Définissez une fonction nommée **square** qui prend un entier en entrée et renvoie le carré de cet entier.

Exercice 2
Modifiez la fonction précédente pour qu'elle accepte maintenant une liste d'entiers et renvoie une liste contenant les carrés de chaque entier.

Hint : liste.map(element => Opération sur l'élément)

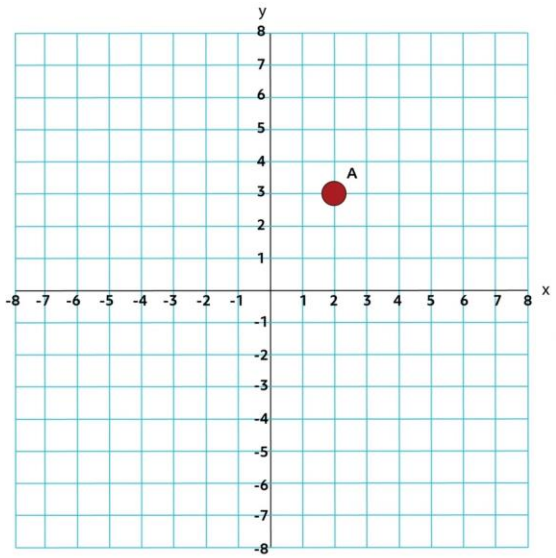
SCALA : Classes

SCALA : classes

Définition

```scala
class Student // définition = class + identifiant
val student1 = new Student // Création d'un  nouvel étudiant (méthode 1)
val student2 = Student() // Création d'un  nouvel étudiant (méthode 2)
```

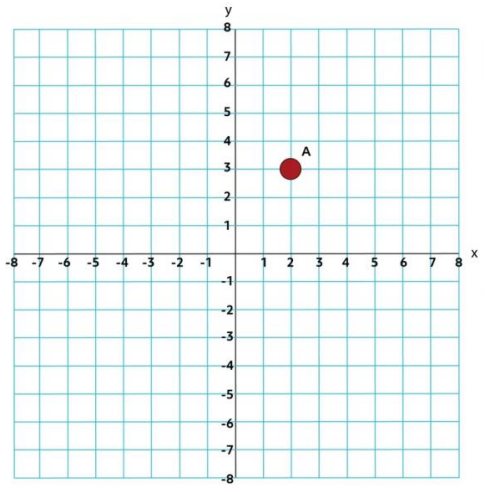
Classe = variables + méthodes




```scala
// default constructor
class Point(var x: Int, var y: Int) {

// class body
  // Fonction (méthode) pour déplacer le point
  def move(dx: Int, dy: Int): Unit = {
    x = x + dx
    y = y + dy
  }

  // Fonction (méthode) pour afficher le point
  override def toString: String =
    s"($x, $y)"
}

val point1 = new Point(2, 3) //val point1 = Point(2, 3)
println(point1.x) (): scala.Unit  // affiche 2
println(point1) (): scala.Unit    // affiche (2, 3)
```

SCALA : Classes

Paramètres facultatifs
Affichage dans l'ordre




```scala
class Point(var x: Int = 0, var y: Int = 0){
  def move(dx: Int, dy: Int): Unit = {
    x = x + dx
    y = y + dy
}
  override def toString: String =
    s"($x, $y)"
}

val origin = new Point // x et y sont tous deux définis sur 0
val point1 = new Point(1) // x est défini sur 1 et y est défini sur 0
val point2 = new Point(y=2) // x est défini sur 0 et y est défini sur 2
println(point1) (): scala.Unit // prints (1, 0)
println(point2) (): scala.Unit // prints (0, 2)
```

SCALA : Classes

Membres privés et la syntaxe getter/setter
i.e. accès data / modifier data


```scala
class Balle:
  private var _x = 0
  private var _y = 0
  private val limite = 100

  def x: Int = _x
  def x_=(nouvelleX: Int): Unit =
    if nouvelleX < limite then
      _x = nouvelleX
    else
      afficherAvertissement()

  def y: Int = _y
  def y_=(nouvelleY: Int): Unit =
    if nouvelleY < limite then
      _y = nouvelleY
    else
      afficherAvertissement()

  private def afficherAvertissement(): Unit =
    println("ATTENTION : En dehors des limites de l'écran")
end Balle

val balle1 = Balle()
balle1.x = 99
balle1.y = 101
```

// Variables privées pour stocker les coordonnées de la balle

// Taille maximale de l'écran

// Getter pour obtenir la coordonnée x de la balle

//Setter pour modifier la coordonnée x de la balle

// Vérifie si la nouvelle coordonnée est dans les limites de l'écran

// Getter pour obtenir la coordonnée y de la balle

//Setter pour modifier la coordonnée y de la balle

// Vérifie si la nouvelle coordonnée est dans les limites de l'écran

// Crée une nouvelle balle

// Modifie les coordonnées de la balle (x=99, y=101)

SCALA : Fonctions

Exercice 1

Créez une classe **Personne** avec les propriétés suivantes :

nom (de type String)
age (de type Int)

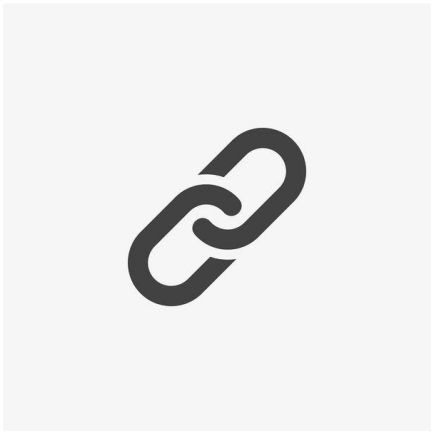
Définissez un constructeur qui initialise ces propriétés. Ajoutez une méthode **afficherInfos()** qui imprime le nom et l'âge de la personne.

Exercice 2

Créez deux instances de la classe **Personne** et initialisez-les avec des valeurs différentes (âge). Appelez la méthode **afficherInfos()** pour chaque instance pour afficher les informations de chaque personne.

SCALA : String

SCALA : String



```
var str = "Hello World!" // (str : String =)
Or
val str = "Hello World!" // (str : String =)
```

Afficher la chaîne

```
println(str) // afficher la chaîne de caractères
```

Longueur de la chaîne

```
var len = str.length()
```

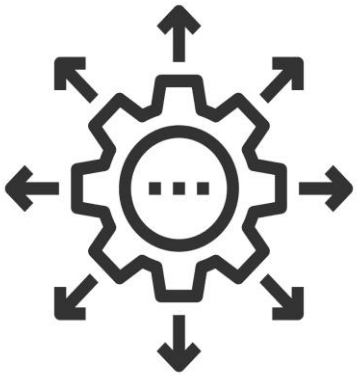
Concaténation

```
str1.concat(str2)

println(str1 + str2)
```

SCALA : Singleton Object

SCALA : Singleton Object

Appeler une méthode in class : différent



```scala
// Un programme Scala simple pour illustrer
// le concept d'objet singleton

class CoffeeMachine:

  // Variables
  var waterLevel = 1000 // Niveau d'eau en millilitres
  var coffeeBeans = 500 // Quantité de grains de café en grammes

  // Méthode qui prépare une tasse de café
  def makeCoffee(): Unit =
    if waterLevel >= 200 && coffeeBeans >= 10 then
      waterLevel -= 200 // Consomme 200 ml d'eau
      coffeeBeans -= 10 // Consomme 10 grammes de grains de café
      println("Une tasse de café a été préparée avec succès!")
    else
      println("Impossible de préparer une tasse de café. Vérifiez les
          niveaux d'eau et de café.")

// Objet singleton
object Main:
  def main(args: Array[String]): Unit =
    // Création d'un objet de la classe CoffeeMachine
    val coffeeMachine = new CoffeeMachine()
    coffeeMachine.makeCoffee()
```

SCALA : Singleton Object

Appeler une méthode in objet singleton : toujours le même

```scala
// Un programme Scala pour illustrer
// comment appeler une méthode à l'intérieur d'un objet singleton

// Objet singleton nommé CoffeeMachine
object CoffeeMachine {

    // Variables de l'objet singleton
    var waterLevel = 1000 // Niveau d'eau en millilitres
    var coffeeBeans = 500 // Quantité de grains de café en grammes

    // Méthode de l'objet singleton
    def makeCoffee(): Unit = {
        println("Préparation d'une tasse de café...")
        // Code de préparation de café
        println("Une tasse de café a été préparée avec succès!")
    }
}

// Objet singleton nommé Main
object Main {
    def main(args: Array[String]): Unit = {

        // Appel de la méthode de l'objet singleton CoffeeMachine
        CoffeeMachine.makeCoffee()
    }
}
```

SCALA : Companion Object

Objet compagnon:

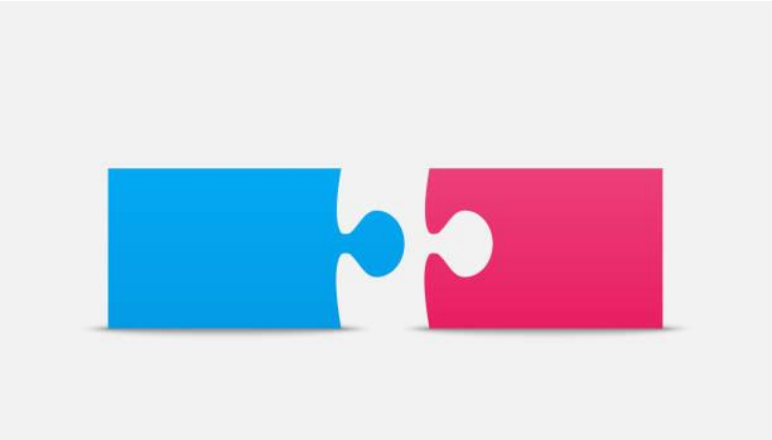
- **= class**
- **Accès au variables et méthodes privés**


```scala
// Classe représentant la machine à café
class CoffeeMachine:

  // Attributs de la machine à café
  var coffeeLevel = 100 // Niveau de café initial

  // Méthode pour servir du café
  def serveCoffee(): Unit =
    println("Votre café est servi !")
    coffeeLevel -= 10 // Diminution du niveau de café

// Objet compagnon de la classe CoffeeMachine
object CoffeeMachine:

  // Méthode principale pour tester la machine à café
  def main(args: Array[String]): Unit =
    val machine = CoffeeMachine() // Création d'une instance de
        CoffeeMachine
    machine.serveCoffee() // Appel de la méthode serveCoffee depuis le
        compagnon
```

SCALA : Pattern Matching

SCALA : Pattern Matching

~~if/else~~



Syntaxe

```
import scala.util.Random

val x: Int = Random.nextInt(10)

x match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
```

In Function

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
matchTest(3)  // returns other
matchTest(1)  // returns one
```

SCALA : Pattern Matching

Exercice :
Écrivez un programme en Scala qui simule le lancer d'un dé à six faces. Utilisez la fonction Random.nextInt() pour générer un nombre aléatoire entre 0 et 5, représentant le résultat du lancer de dé.

Ensuite, utilisez une expression de correspondance (match) pour interpréter le résultat du lancer de dé et afficher un message approprié en fonction du nombre obtenu.
Si le nombre obtenu est inférieur à 1 ou supérieur à 5, affichez un message indiquant que le résultat est invalide. Enfin, affichez le message résultant.

SCALA : Pattern Matching

Utilité dans les Case Class

Case Class : classe spéciale pour modélisation des données immuables et automatisme

```
sealed trait Forme // déclaration pour définir une hiérarchie de type sellé

// classe carré avec un côté spécifié
case class Carre(cote: Int) extends Forme

// classe cercle avec un rayon spécifié
case class Cercle(rayon: Double) extends Forme

// classe triangle avec un périmètre spécifié
case class Triangle(périmètre: Int) extends Forme

…. Afficher les valeurs de chaque forme
```

SCALA : Pattern Matching

Pattern Guards : expressions booléennes après Matching

Syntaxe

```
variable match {
  case Pattern1 if condition1 =>
    // traitement si Pattern1 correspond et condition1 est vraie

  case Pattern2 if condition2 =>
    // traitement si Pattern2 correspond et condition2 est vraie
  // Autres cas et traitements
}
```

SCALA : Pattern Matching

Pattern Guards : expressions booléennes après Matching

Écrivez un bloc **match** qui intègre une variable **animal** (n'oubliez pas '**sealed trait**') et affiche un message spécifique si **animal** est un **chien** ou un **chat**.

Veillez à utiliser des cas comportant des motifs afin de déterminer si **animal** est un **chien** ou un **chat**, et à utiliser des **conditions** (ex: nom de l'animal) pour afficher un message spécifique (*'C'est X , le chien (ou chat)!'*).

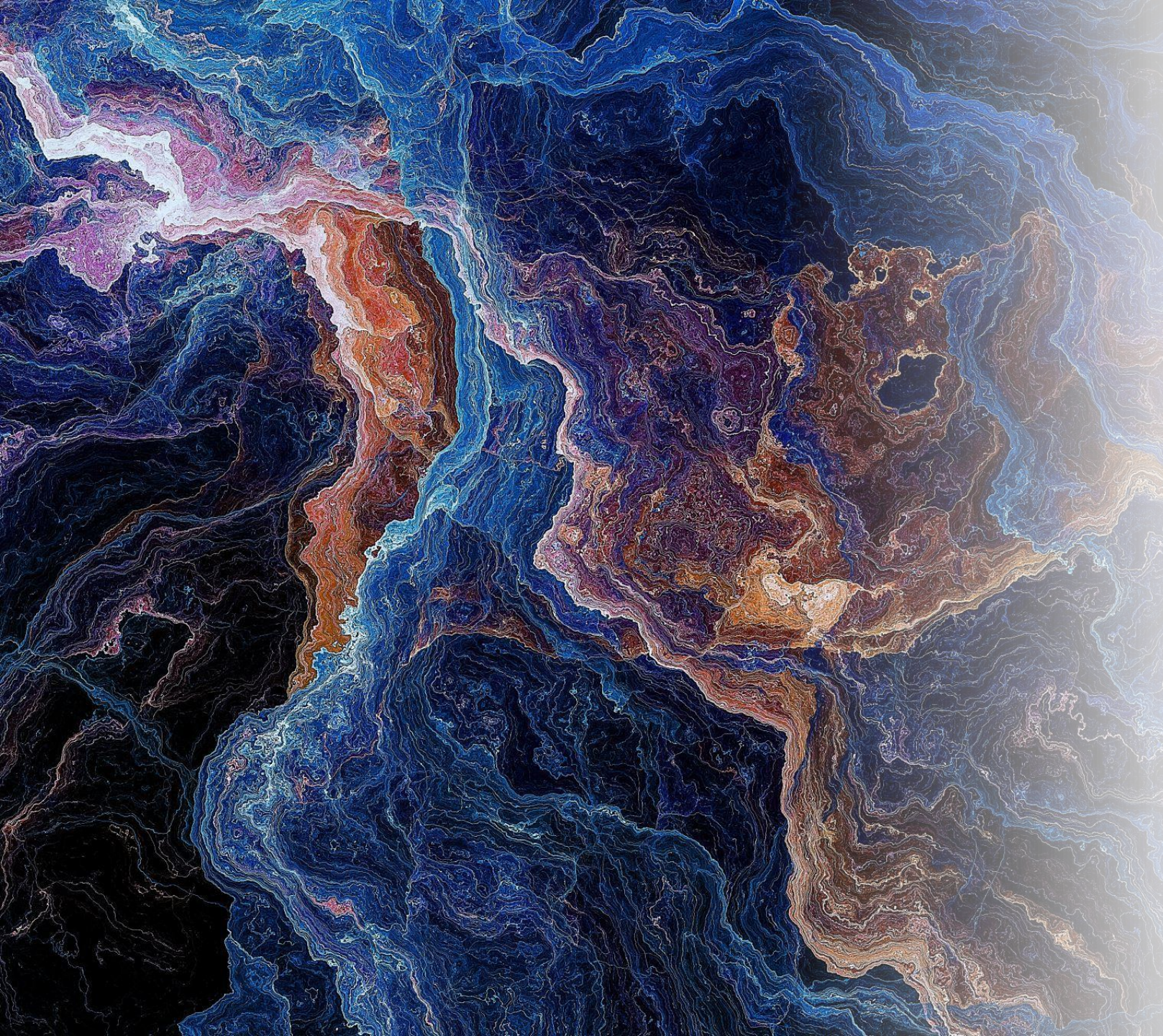
SCALA : Pattern Matching

Matching sur le type: correspondre son objet à son type

Écrivez un code illustrant deux variétés de **fruits** : les **pommes** et les **bananes**. Utilisez des **classes** afin de représenter ces **fruits**. Veillez à ce que chaque fruit ait sa méthode qui lui est **spécifique** : "croquer" ou "laver" pour une **pomme**, "peler" ou "éplucher" pour une **banane**.

Par la suite, créez une **fonction** appelée **actionFruit** qui sélectionne un fruit en paramètre et renvoie un message décrivant l'action spécifique à ce fruit.

Finalement, créez deux **instances** de fruits (une pomme et une banane) et utilisez la fonction "actionFruit" pour afficher les **résultats** de l'interaction avec chaque fruit.



SCALA : Regular expression (Regex)

SCALA : Regex

Regular expression ou expression régulière : chaîne de caractères utlisées pour chercher un pattern
String --> Regex : '.r'

Syntaxe

```
import scala.util.matching.Regex

val sunPattern: Regex = "soleil".r

sunPattern.findFirstMatchIn("Le soleil brille aujourd'hui")
match {
  case Some(_) => println("La phrase contient le mot
'soleil'")
  case None    => println("La phrase ne contient pas le mot
'soleil'")
}
```

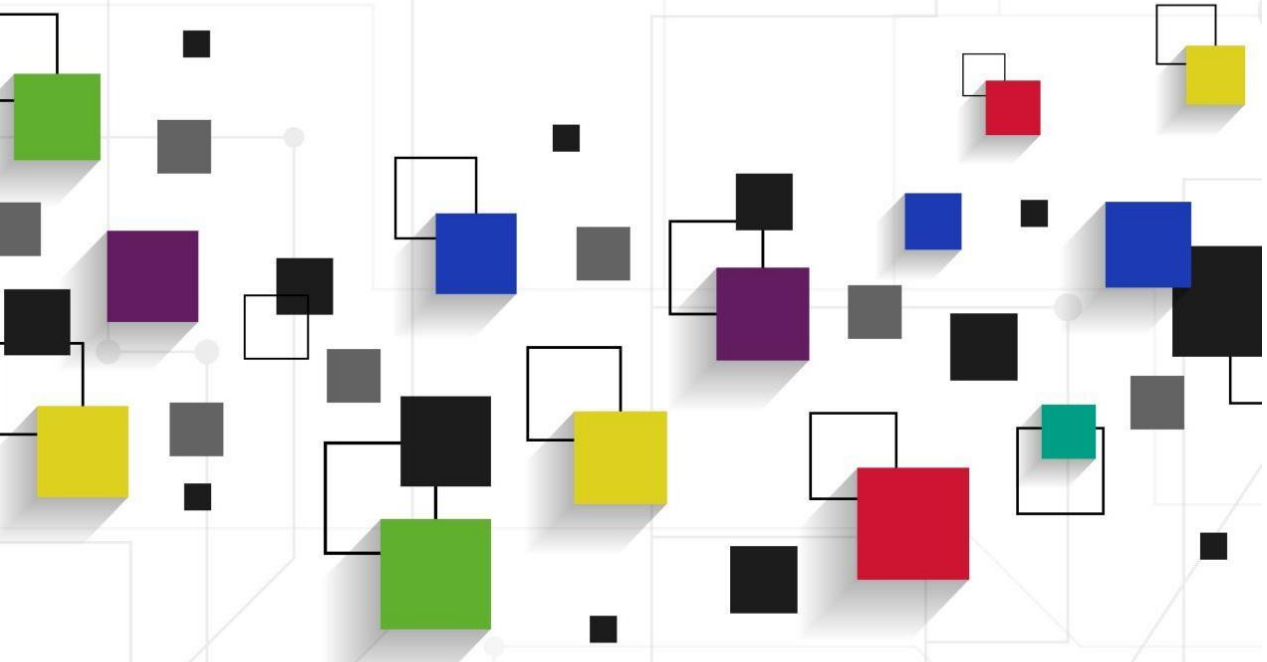
SCALA : Regex

Regular expression ou expression régulière : chaîne de caractères utlisées pour chercher un pattern

String --> Regex : '.r'

Syntaxe

```
def saveContactInformation(contact: String): Unit = {
  val emailPattern = """^(\w+)@(\w+(.\w+)+)$""".r
  val phonePattern = """^(\d{3}-\d{3}-\d{4})$""".r

….
"écrivez le code manquant en utilisant les expressions régulières emailPattern
et phonePattern pour effectuer le matching sur la variable contact et imprimer
le message approprié en fonction du type de contact."
….

}

saveContactInformation("123-456-7890")
saveContactInformation("JeanDupont@exemple.domaine.com")
saveContactInformation("2 Rue de la Lune, Mars, Voie Lactée")
```



SCALA
: Statements, Loop, Exceptions

SCALA : Statements, Loop, Exceptions

Statements (Instructions)

```
val x = 5  // Déclaration et affectation d'une variable
println("Hello, world!")  // Appel de fonction pour afficher du texte
```

Loop (boucle)

```
for (i <- 0 until 5) {
    println(i)  // Affiche les nombres de 0 à 4
}
```

Exceptions

```
try {
    val result = 1 / 0  // Division par zéro
} catch {
    case e: ArithmeticException => println("Impossible de diviser par zéro !")
}
```