Scala

m2iformation.fr

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction Scala

Introduction à Scala : Un Langage Polyvalent pour la Programmation Moderne

Scala (Scalable Language) est un langage de programmation polyvalent, créé par **Martin Odersky** en 2004. Il combine les paradigmes de **programmation orientée objet (POO)** et de **programmation fonctionnelle (FP)**, ce qui le rend à la fois expressif, flexible et puissant. Conçu pour fonctionner de manière transparente avec la **JVM (Java Virtual Machine)**, Scala peut utiliser et étendre l'écosystème Java tout en offrant des fonctionnalités plus modernes et concises.

🔥 Pourquoi utiliser Scala ?

Scala est utilisé par de nombreuses entreprises prestigieuses, comme **Twitter, LinkedIn, Airbnb et Spotify**, principalement pour des applications à haute performance et pour le traitement de gros volumes de données (Big Data) avec des frameworks comme **Apache Spark**.
Les avantages de Scala incluent :

- **Syntaxe concise et expressive** : Moins de code, mais plus de clarté.
- **Interopérabilité avec Java** : On peut utiliser les bibliothèques Java directement.
- **Support de la programmation fonctionnelle** : Fonctions d'ordre supérieur, immuabilité, expressions lambda, etc.
- **Parallélisme et Concurrence** : Grâce à des outils comme **Akka** (pour les systèmes d'acteurs), Scala permet de créer des applications parallèles et distribuées.
- **Utilisation dans le Big Data** : Scala est le langage principal d'**Apache Spark**, ce qui le rend crucial pour la manipulation et le traitement de grandes quantités de données.

🚀 Caractéristiques principales de Scala

1. **Double Paradigme (Objet + Fonctionnel)**
   - **POO** : Classes, objets, héritage, polymorphisme, etc.
   - **FP** : Fonctions pures, immuabilité, récursion, monades, etc.
2. **Syntaxe concise et lisible**
   - Les accolades `{}` et les points-virgules `;` ne sont pas obligatoires.
   - Utilisation d'expressions au lieu d'instructions (`val result = if (x > 0) x else -x`).
3. **Immutabilité par défaut**
   - Les variables sont définies par `val` (constantes immuables) ou `var` (variables modifiables).

```
val x = 10 // immuable, on ne peut pas changer x
var y = 20 // mutable, on peut modifier y
```

4. **Fonctions d'ordre supérieur**
   - Les fonctions peuvent être transmises en tant qu'arguments et retournées par d'autres fonctions.

```
def applyFunction(f: Int => Int, x: Int): Int = f(x)
val result = applyFunction(x => x * 2, 10) // retourne 20
```

5. **Collections Riches et Expressives**
   - Collections immuables par défaut (List, Map, Set, etc.).
   - Opérations fonctionnelles sur les collections (map, flatMap, filter, reduce, etc.).

```
val numbers = List(1, 2, 3, 4, 5)
val doubled = numbers.map(_ * 2) // List(2, 4, 6, 8, 10)
```

6. **Interopérabilité avec Java**
   - Les bibliothèques Java peuvent être appelées et utilisées dans le code Scala.

```
val now = java.time.LocalDate.now()
println(s"Today's date is: $now")
```

7. **Gestion des Concurrences**
   - La bibliothèque **Akka** permet la programmation asynchrone et distribuée avec un modèle d'acteurs.
   - Les **futures et promises** permettent la programmation asynchrone de manière plus simple.

📚 Exemple de base en Scala

Voici un programme "Hello, World!" classique en Scala.

```scala
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, World!")
  }
}
```

Explications :

- **object** : Déclare un objet singleton (équivalent d'une classe statique).
- **def main** : Fonction d'entrée du programme.
- **println** : Affiche du texte à la console.

💡 Les Cas d'Utilisation de Scala

- **Big Data** : Avec **Apache Spark**, Scala est le langage par excellence pour l'analyse des données massives.
- **Systèmes distribués** : Grâce à **Akka** et au modèle d'acteurs, on peut créer des systèmes distribués et résilients.
- **Développement d'API web** : Avec des frameworks comme **Play Framework** (similaire à Spring Boot).
- **Machine Learning** : En utilisant des bibliothèques comme **Breeze** et **Spark MLlib**.
- **Applications de trading financier** : La capacité de Scala à gérer la concurrence en fait un bon candidat pour les systèmes de trading haute fréquence.

🛠️ Outils de Développement

- **IntelliJ IDEA** (avec le plugin Scala)
- **Scala REPL** (l'interpréteur interactif Scala)
- **sbt** (Scala Build Tool) pour la compilation et la gestion des dépendances.

Différents modes d'utilisation de Scala Compilation, Script et Interpréteur

Scala est un langage flexible qui peut être utilisé de trois manières principales selon le besoin du développeur. Ces modes d'utilisation permettent de tirer parti de sa polyvalence et de sa compatibilité avec la JVM. Voici une **explication détaillée** de ces trois modes d'exécution de Scala.

🔥 1️⃣ Mode Compilé

Dans ce mode, le code Scala est **compilé** en **bytecode** (le même format que Java) pour être exécuté par la **JVM (Java Virtual Machine)**. Ce mode de fonctionnement est **similaire à Java**.

🟦 Comment ça fonctionne ?

1. **Écriture du code** : On écrit un fichier source en Scala avec l'extension **.scala**.
2. **Compilation** : On utilise l'outil `scalac` pour compiler le fichier Scala en un fichier **.class**.
3. **Exécution** : La **JVM** exécute le bytecode contenu dans le fichier `.class`.

🟦 Exemple

Fichier source : `HelloWorld.scala`

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, World!")
  }
}
```

Étapes de compilation et d'exécution

1. **Compiler le fichier**

```
scalac HelloWorld.scala
```

Cette commande génère un fichier **HelloWorld.class** qui contient le bytecode (compréhensible par la JVM).

2. **Exécuter le fichier compilé**

```
scala HelloWorld
```

Cela exécute le bytecode en utilisant la **JVM** et affiche :

```
Hello, World!
```

Avantages du mode compilé

- **Performance optimisée** : Le code compilé est plus rapide à exécuter que le script ou l'interpréteur.
- **Réutilisation** : On peut exécuter plusieurs fois le fichier compilé sans avoir besoin de le recompiler.
- **Interopérabilité avec Java** : Le fichier **.class** généré peut être utilisé par d'autres applications Java.

🔥 2️⃣ Mode Script

En mode **script**, Scala se comporte un peu comme les langages **Python, Bash ou Perl**. Ici, on écrit un fichier **.scala**, mais **sans méthode main** ni structure formelle d'objet ou de classe. Le script est exécuté directement à la volée, sans qu'il soit nécessaire de le compiler.

📘 Comment ça fonctionne ?

1. On écrit un fichier **script.scala**.
2. On exécute directement le fichier avec la commande `scala`.
3. Le fichier **n'est pas compilé** en `.class`, mais est exécuté à la volée.

📘 Exemple

Fichier script : `script.scala`

```
println("Hello, World!")
val x = 42
val y = x * 2
println(s"Le double de $x est $y")
```

Exécution du script

```
scala script.scala
```

Explications

- **Pas besoin de "main"** : Contrairement au mode compilé, ici on ne crée pas de méthode `main`, Scala exécute les instructions directement.
- **Scripts rapides** : Utile pour des **tests rapides** ou des **petits scripts utilitaires**.
- **Pas de fichiers .class** : Scala **ne produit pas de bytecode** ici.
- **Pas de classes nécessaires** : On écrit du code "au vol" comme on le ferait dans un script shell ou un script Python.

Avantages du mode script

- **Rapide à tester** : Pas besoin de compilation.
- **Parfait pour le prototypage** : Si vous souhaitez tester une idée de code ou de logique, c'est rapide.
- **Simplicité** : Pas besoin de `object` ni de `main`, juste des instructions.

🔥 3️⃣ Mode Interpréteur (REPL)

Le mode **interpréteur** (ou **REPL** - Read, Eval, Print, Loop) est un environnement interactif de développement pour **tester rapidement des expressions Scala**. Ce mode est utilisé par les développeurs pour **tester des concepts ou des fonctionnalités** avant de les écrire dans un fichier.

“ 🟦 **Signification de REPL** :

- **R**ead : Lis une ligne de code que l'utilisateur entre.
- **E**val : Évalue la ligne de code.
- **P**rint : Affiche le résultat de l'expression.
- **L**oop : Attend la saisie d'une nouvelle ligne de code. ”

🟦 Comment ça fonctionne ?

1. Tapez la commande `scala` pour ouvrir l'interpréteur.
2. Tapez directement des instructions dans la console.
3. Le résultat de chaque expression est immédiatement affiché.

Exemple d'utilisation

```
scala
```

Vous entrez dans le **REPL**. Vous verrez quelque chose comme :

```
Welcome to Scala 3.2.0 (Java HotSpot(TM) 64-Bit Server VM, Java 17.0.1).
Type in expressions for evaluation. Or try :help.
```

Entrez des commandes :

```
scala> println("Hello, World!")
Hello, World!

scala> val x = 10
val x: Int = 10

scala> x * 3
res0: Int = 30

scala> List(1, 2, 3).map(_ * 2)
res1: List[Int] = List(2, 4, 6)
```

📊 Comparaison des 3 modes

Critère	Mode Compilé	Mode Script	Mode Interpréteur (REPL)
Structure	Nécessite `object` et `main`	Aucune structure requise	Aucune structure requise
Compilation	Oui, produit un fichier `.class`	Non	Non, tout est en mémoire
Exécution	Via `scala NomClasse`	`scala script.scala`	REPL interactif (ligne par ligne)
Vitesse	Rapide (pré-compilé)	Un peu plus lent	Instantané mais pour petites expressions
Cas d'usage	Applications complètes	Scripts simples	Prototypage rapide, tests interactifs
Interopérabilité	Compatible Java	Compatible Java	Compatible Java
Utilisation	Production	Utilitaire	Prototypage

💡 Quelle approche choisir ?

- **Mode compilé** : Si vous développez une application professionnelle ou un projet complexe.
- **Mode script** : Si vous voulez automatiser des tâches ou exécuter du code sans le structurer.
- **Mode REPL** : Si vous souhaitez tester des idées, apprendre Scala ou expérimenter de nouvelles fonctionnalités.

Qu'est-ce que sbt ? 🤔

sbt (pour **"Scala Build Tool"**) est un **outil de construction** (build tool) pour les projets Scala et Java. Il est similaire à **Maven** ou **Gradle** dans le monde Java, mais il est **conçu spécialement pour Scala**. sbt est l'outil principal utilisé par les développeurs Scala pour **compiler, exécuter, tester et gérer les dépendances** d'un projet.

1 Compilation du code

- Compile automatiquement le code source Scala et Java de votre projet.
- Les fichiers `.scala` sont compilés en bytecode `.class`, utilisable par la JVM.

Commande :

```
sbt compile
```

2 Exécution des programmes

- Exécute votre application Scala.
- Vous pouvez lancer directement votre programme en utilisant la commande `run`.

Commande :

```
sbt run
```

3 Gestion des dépendances

- sbt permet d'**ajouter des bibliothèques externes** (comme des bibliothèques de calcul, de base de données, etc.).
- Ces dépendances sont définies dans un fichier nommé `build.sbt`.

Exemple de dépendance dans build.sbt :

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.16" % Test
```

Avec ce fichier, sbt télécharge automatiquement la bibliothèque **Scalatest** et l'ajoute au projet.

4 Lancer les tests

- sbt peut exécuter vos **tests unitaires** ou **tests d'intégration**.
- Les frameworks de tests comme **ScalaTest** ou **JUnit** sont compatibles avec sbt.

Commande :

```
sbt test
```

5 Gestion des packages (jar)

- sbt vous permet de créer un **fichier .jar** qui peut être exécuté avec la JVM.
- Ce fichier jar contient toutes les classes nécessaires à l'exécution de votre programme.

Commande :

```
sbt package
```

Cela produit un fichier `mon-projet.jar` que vous pouvez exécuter avec :

```
java -jar mon-projet.jar
```

6 Mode interactif

- sbt offre un **mode interactif** qui vous permet de taper des commandes ligne par ligne, un peu comme une "console" interactive.

Commande :

```
sbt
```

Dans ce mode, vous pouvez taper des commandes comme `compile`, `run`, `test`, etc.

Structure d'un projet sbt

Lorsqu'on crée un projet sbt, voici la structure typique du répertoire :

```
mon-projet/
├── build.sbt           (fichier de configuration principale)
├── project/            (répertoire des fichiers de configuration de sbt)
├── src/
│   ├── main/
│   │   └── scala/      (répertoire pour le code source Scala)
│   └── test/           (répertoire pour les tests unitaires)
└── target/             (répertoire où sbt met les fichiers compilés)
```

📜 Fichier build.sbt (le fichier le plus important)

Le fichier `build.sbt` est le fichier de configuration principal de votre projet. C'est ici que vous définissez :

- **Le nom du projet**
- **La version de Scala**
- **Les dépendances** (comme les bibliothèques externes)
- **Les options de compilation** (par exemple, activer des options de débogage)

Exemple de fichier `build.sbt`

```
name := "MonProjetScala"

version := "0.1.0"

scalaVersion := "2.13.12"

libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "3.2.16" % Test,
  "com.typesafe.akka" %% "akka-actor" % "2.6.20"
)
```

Explications :

- **name := "MonProjetScala"** : nom du projet
- **version := "0.1.0"** : version du projet
- **scalaVersion := "2.13.12"** : version de Scala utilisée
- **libraryDependencies** : liste des bibliothèques à importer (ici, on ajoute **Scalatest** et **Akka**)

⚙ Installation de sbt

Pour utiliser sbt, vous devez l'installer sur votre système.

1️⃣ **Sous Linux** (Ubuntu, Debian) :

```
sudo apt update
sudo apt install sbt
```

2️⃣ **Sous macOS** :

```
brew install sbt
```

3️⃣ **Sous Windows** :

- Installez **sbt** à partir du site officiel de sbt.

Commandes principales de sbt

Commande	Description
`sbt compile`	Compile le code source Scala
`sbt run`	Exécute le programme principal
`sbt test`	Exécute les tests unitaires
`sbt clean`	Nettoie les fichiers compilés
`sbt package`	Crée un fichier JAR du projet
`sbt console`	Ouvre une console interactive de Scala
`sbt update`	Met à jour les dépendances

🔥 Exemple de projet complet avec sbt

1️⃣ Créer un dossier de projet :

```
mkdir MonProjet
cd MonProjet
```

2️⃣ Créer le fichier `build.sbt` :

```
name := "MonProjet"

version := "0.1.0"

scalaVersion := "2.13.12"

libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.16" % Test
```

3 **Créer les répertoires du projet** :

```
mkdir -p src/main/scala
mkdir -p src/test/scala
```

4 **Créer un fichier Scala dans** `src/main/scala/HelloWorld.scala` :

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Bonjour tout le monde !")
  }
}
```

5 **Compiler et exécuter le projet** :

```
sbt compile
sbt run
```

Sortie attendue :

```
[info] running HelloWorld
Bonjour tout le monde !
```

🧐 Pourquoi utiliser sbt ?

- **Automatisation** : Automatise les tâches de compilation, de test et de gestion des dépendances.
- **Gagne du temps** : Plus besoin de télécharger les dépendances manuellement.
- **Gestion des dépendances** : Télécharge automatiquement les bibliothèques.
- **Facilité d'intégration** : Fonctionne avec **IntelliJ IDEA** et d'autres IDE.
- **Utilisation avancée** : Permet d'exécuter des commandes interactives et des scripts.

⚔️ Différences entre sbt et Maven

Caractéristique	sbt	Maven
Syntaxe	Simple et concise (fichier .sbt)	XML (pom.xml)
Vitesse	Plus rapide (compilation incrémentale)	Plus lent (recompile tout)
Langue cible	Scala (mais compatible Java)	Java principalement
Approche	Interactif (console)	Exécution par commandes

⚙ Problèmes courants avec sbt

1️⃣ Problème : sbt est lent

Solution : Activez la compilation incrémentale (déjà activée par défaut) et utilisez un "daemon" pour garder sbt actif.

Commande :

```
sbt --client
```

2️⃣ Problème : sbt ne trouve pas la bonne version de Scala

Solution : Assurez-vous que la bonne version est spécifiée dans `build.sbt` :

```
scalaVersion := "2.13.12"
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Syntaxe de base

1. Les commentaires

Les commentaires permettent d'ajouter des notes dans le code.

- **Commentaire sur une seule ligne** :

```
// Ceci est un commentaire sur une seule ligne
```

- **Commentaire multi-lignes** :

```
/*
  Ceci est un commentaire
  sur plusieurs lignes
*/
```

2. Variables et constantes

On peut déclarer des variables ou des constantes à l'aide de `val` et `var`.

- `val` (constante) : ne peut pas être modifiée après sa déclaration.

```
val pi: Double = 3.14 // constante immuable
```

- `var` (variable) : sa valeur peut changer.

```
var age: Int = 25
age = 26 // OK, on peut changer la valeur
```

- **Inférence de type** : il est possible de ne pas spécifier le type, Scala le déduit automatiquement.

```
val name = "John" // Scala comprend que le type est String
```

3. Types de données de base

Voici les principaux types de base en Scala :

Type	Description	Exemple
`Int`	Nombre entier	`val x: Int = 42`
`Double`	Nombre à virgule flottante	`val x: Double = 3.14`
`Float`	Nombre flottant (32 bits)	`val x: Float = 2.5f`
`Boolean`	Vrai ou faux	`val flag = true`
`Char`	Caractère	`val letter: Char = 'A'`
`String`	Chaîne de caractères	`val greeting = "Hello"`

4. Opérateurs

Les opérateurs permettent de manipuler des valeurs.

- **Opérateurs arithmétiques** : `+`, `-`, `*`, `/`, `%`

```
val a = 10 + 5   // 15
val b = 10 * 2   // 20
val c = 10 / 3   // 3 (division entière)
val d = 10 % 3   // 1 (reste de la division)
```

- **Opérateurs de comparaison** : `>`, `<`, `>=`, `<=`, `==`, `!=`

```
val isGreater = 10 > 5  // true
val isEqual = 10 == 10  // true
```

- **Opérateurs logiques** : `&&` (ET), `||` (OU), `!` (NON)

```
val result = (10 > 5) && (5 < 10) // true
```

5. Structures conditionnelles

Les structures conditionnelles permettent d'exécuter du code en fonction de certaines **conditions**.

5.1. If / Else

```
val age = 18
if (age >= 18) {
  println("Adulte")
} else {
  println("Mineur")
}
```

5.2. If / Else If / Else

```
val note = 85
if (note >= 90) {
  println("Excellent")
} else if (note >= 70) {
  println("Bien")
} else {
  println("Insuffisant")
}
```

5.3. If ternaire (expression)

En Scala, **if/else** est une **expression** (comme en Python) et peut renvoyer une valeur.

```
val age = 20
val status = if (age >= 18) "Adulte" else "Mineur"
println(status) // Adulte
```

6. Boucles

6.1. Boucle For (standard)

La **boucle** `for` permet d'itérer sur une plage de valeurs.

Exemple 1 : Itération simple

```
for (i <- 1 to 5) {
  println(i) // Affiche 1, 2, 3, 4, 5
}
```

- `1 to 5` : génère une plage de 1 à 5 (inclus).

Exemple 2 : Plage exclusive (to vs until)

```
for (i <- 1 until 5) {
  println(i) // Affiche 1, 2, 3, 4 (sans 5)
}
```

Exemple 3 : Ajout d'une condition (garde)

```
for (i <- 1 to 10 if i % 2 == 0) {
  println(i) // Affiche les nombres pairs : 2, 4, 6, 8, 10
}
```

Exemple 4 : For imbriqué

```
for (i <- 1 to 3; j <- 1 to 2) {
  println(s"i=$i, j=$j")
}
```

6.2. Boucle While

La boucle **while** continue tant qu'une condition est **vraie**.

Exemple :

```
var i = 1
while (i <= 5) {
  println(i) // Affiche 1, 2, 3, 4, 5
  i += 1
}
```

6.3. Boucle Do While

La boucle **do-while** exécute le code **au moins une fois**, même si la condition est fausse.

Exemple :

```
var i = 1
do {
  println(i) // Affiche 1
  i += 1
} while (i <= 5)
```

6.4. Break et continue (solution alternative)

Contrairement à d'autres langages, Scala n'a pas de `break` ou `continue` explicites. On utilise la méthode `return` dans une boucle **for** ou on utilise des structures de contrôle avec des **exceptions**.

Simuler un break :

```
import scala.util.control.Breaks._

breakable {
  for (i <- 1 to 10) {
    if (i == 5) break // Arrête la boucle ici
    println(i) // Affiche 1, 2, 3, 4
  }
}
```

6.5. Boucle avec Yield (retourne une collection)

La boucle `yield` permet de **retourner une collection**.

Exemple 1 : Carrés des nombres

```
val squares = for (i <- 1 to 5) yield i * i
println(squares) // Vector(1, 4, 9, 16, 25)
```

Exemple 2 : Filtrer et transformer

```
val evenSquares = for (i <- 1 to 10 if i % 2 == 0) yield i * i
println(evenSquares) // Vector(4, 16, 36, 64, 100)
```

6.6. Plage de valeurs

Vous pouvez créer des plages de valeurs **(Range)** avec des intervalles.

Exemple :

```
val range = 1 to 10 // Plage de 1 à 10
val range2 = 1 until 10 // Plage de 1 à 9
```

Continuons avec la **syntaxe de base de Scala** sans aborder l'**orientation objet (OOP)**. Nous allons explorer les notions essentielles qui vous permettront de **maîtriser les bases de Scala**.

7. Les Fonctions

Les fonctions sont des **citoyens de première classe** en Scala, ce qui signifie qu'on peut :

- Les déclarer
- Les affecter à des variables
- Les passer comme arguments
- Les retourner comme résultats

7.1. Déclaration de fonction simple

La syntaxe de base pour une fonction est la suivante :

```
def nomDeLaFonction(parametre1: Type1, parametre2: Type2): TypeDeRetour = {
  // Corps de la fonction
  expression
}
```

Exemple 1 : Fonction qui additionne deux nombres

```
def addition(a: Int, b: Int): Int = {
  a + b
}
println(addition(2, 3)) // Affiche 5
```

7.2. Fonction sans paramètre

Si la fonction n'a pas de paramètres, on peut se passer des parenthèses.

```
def salutation(): String = "Bonjour, Scala !"
println(salutation())
```

7.3. Fonction avec paramètre par défaut

On peut définir des **valeurs par défaut** pour les paramètres.

```
def bonjour(nom: String = "Utilisateur"): String = {
  s"Bonjour, $nom !"
}
println(bonjour())           // Affiche "Bonjour, Utilisateur !"
println(bonjour("Alice"))    // Affiche "Bonjour, Alice !"
```

7.4. Fonction anonyme (lambda)

Les fonctions anonymes sont souvent utilisées avec des **fonctions d'ordre supérieur** comme `map`, `filter`, etc.

```
val carre = (x: Int) => x * x
println(carre(4)) // Affiche 16
```

On peut utiliser `_` pour simplifier les fonctions anonymes :

```
val doubler = (_: Int) * 2
println(doubler(5)) // Affiche 10
```

7.5. Fonction sans le mot-clé `return`

En Scala, la dernière expression d'une fonction est automatiquement retournée (comme en Python).

```
def multiplication(a: Int, b: Int): Int = a * b
```

8. Expressions (tout est expression)

En Scala, **tout est une expression**, ce qui signifie que même les **if**, **boucles** et **match** retournent des valeurs.

Exemple 1 : If comme expression

```
val age = 20
val categorie = if (age >= 18) "Adulte" else "Mineur"
println(categorie) // Adulte
```

Exemple 2 : For comme expression (avec yield)

```
val nombres = for (i <- 1 to 5) yield i * i
println(nombres) // Vector(1, 4, 9, 16, 25)
```

9. Pattern Matching (remplace le switch-case)

Le **pattern matching** permet de **matcher des cas** comme un **switch-case** mais avec des **capacités avancées**.

Exemple 1 : Simple pattern matching

```
val jour = "Lundi"
val typeDeJour = jour match {
  case "Samedi" | "Dimanche" => "Week-end"
  case "Lundi" => "Début de la semaine"
  case _ => "Jour de semaine"
}
println(typeDeJour) // Affiche "Début de la semaine"
```

Exemple 2 : Pattern matching avec types

```
def analyse(x: Any): String = x match {
  case i: Int => s"Un nombre entier : $i"
  case s: String => s"Une chaîne de caractères : $s"
  case _ => "Type inconnu"
}
```

10. Collections

Scala propose de nombreuses **collections immuables** (par défaut) comme **List**, **Set**, **Map**, **Vector**, etc.

10.1. List (liste immuable)

Les listes en Scala sont **immuables**, ce qui signifie qu'on ne peut pas les modifier directement.

Création de liste

```
val maListe = List(1, 2, 3, 4, 5)
```

Opérations sur les listes

```
val doublees = maListe.map(x => x * 2)
println(doublees) // List(2, 4, 6, 8, 10)
```

10.2. Set (ensemble de valeurs uniques)

Les **Sets** contiennent des valeurs **uniques**.

Création de Set

```
val monSet = Set(1, 2, 2, 3)
println(monSet) // Set(1, 2, 3)
```

10.3. Map (associations clé-valeur)

Les **Map** permettent de stocker des paires **clé-valeur**.

Création de Map

```
val map = Map("nom" -> "Alice", "age" -> 25)
println(map("nom")) // Alice
```

11. Opérations sur les collections

Les **collections immuables** peuvent être transformées sans modifier la collection d'origine.

11.1. `map`

Applique une fonction à chaque élément d'une collection.

```
val maListe = List(1, 2, 3)
val doublees = maListe.map(_ * 2)
println(doublees) // List(2, 4, 6)
```

11.2. `filter`

Filtre les éléments d'une collection selon une condition.

```
val nombres = List(1, 2, 3, 4, 5)
val pairs = nombres.filter(_ % 2 == 0)
println(pairs) // List(2, 4)
```

11.3. `reduce`

Combine les éléments de la collection pour obtenir un résultat unique.

```
val total = List(1, 2, 3, 4).reduce(_ + _)
println(total) // 10
```

11.4. `foreach`

Parcourt chaque élément d'une collection.

```
val maListe = List(1, 2, 3)
maListe.foreach(println)
```

12. Opérations de contrôle avancées

Les **break** et **continue** ne sont pas présents en Scala. On utilise des structures spéciales.

12.1. Break

Pour quitter une boucle, on utilise la fonction `break` de **scala.util.control.Breaks**.

```
import scala.util.control.Breaks._

breakable {
  for (i <- 1 to 10) {
    if (i == 5) break
    println(i) // Affiche 1, 2, 3, 4
  }
}
```

12.2. Continue

On peut imiter **continue** en utilisant un **if**.

```
for (i <- 1 to 5) {
  if (i == 3) {
    // On saute cette itération
  } else {
    println(i)
  }
}
```

13. Import et packages

Pour utiliser des fonctionnalités externes (comme `break`), on peut importer des packages.

```
import scala.util.control.Breaks._
```

14. Interpolation de chaînes

On peut **insérer des variables** dans les chaînes de caractères avec **s"..."**.

```
val nom = "Alice"
val message = s"Bonjour, $nom !"
println(message) // Bonjour, Alice !
```

15. Conversion de types

Scala permet de convertir les types explicitement.

```
val x = 42
val y = x.toString // Convertit Int en String
```

16. Entrée utilisateur

Pour lire l'entrée de l'utilisateur, on utilise `scala.io.StdIn.readLine()`.

```
import scala.io.StdIn
val nom = StdIn.readLine("Entrez votre nom : ")
println(s"Bonjour, $nom !")
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

POO

1 Classe et Attributs

La **classe** est la **base** de la POO. Elle représente un **modèle** à partir duquel on peut créer des **objets**.

Définition d'une classe

```
class Person(nom: String, age: Int) {
  def afficherInfo(): Unit = {
    println(s"Nom: $nom, Âge: $age")
  }
}
```

Explications

- **`class Person(nom: String, age: Int)`** : C'est la **classe** `Person` avec deux **attributs** (`nom` et `age`).
- **`afficherInfo`** : Méthode qui affiche les informations de la personne.

Utilisation

```
val personne = new Person("Alice", 25)
personne.afficherInfo()
```

Sortie

```
Nom: Alice, Âge: 25
```

2 Accès (Public, Private, Protected)

Les **attributs** et **méthodes** d'une classe peuvent être :

- **public** (par défaut) : accessible de partout.
- **private** : accessible uniquement **à l'intérieur de la classe**.
- **protected** : accessible dans la **classe et les classes dérivées**.

Exemple

```
class BankAccount(private var balance: Double) {
  // Méthode publique
  def deposit(amount: Double): Unit = {
    balance += amount
  }

  // Méthode protégée
  protected def getBalance(): Double = balance

  // Méthode privée
  private def secretOperation(): Unit = {
    println("Opération secrète !")
  }
}
```

3 Constructeurs (Principal et Secondaire)

Le **constructeur principal** est défini dans la déclaration de la classe.
Les **constructeurs secondaires** se définissent avec `def this`.

Constructeur principal

```
class Person(nom: String, age: Int) {
  def afficherInfo(): Unit = {
    println(s"Nom: $nom, Âge: $age")
  }
}
```

Constructeur secondaire

```
class Person(nom: String, age: Int) {
  def this(nom: String) = {
    this(nom, 18) // Appel au constructeur principal
  }
}
```

Utilisation

```
val personne1 = new Person("Alice", 25)
val personne2 = new Person("Bob") // Utilise le constructeur secondaire
```

4 Objets (Singletons)

Un **objet** Scala est un **singleton** (une seule instance) et est défini par le mot-clé `object`.

```
object Utility {
  def printHello(): Unit = {
    println("Bonjour !")
  }
}
```

Utilisation

```
Utility.printHello()
```

5 Méthodes et Comportements

Les **méthodes** définissent le **comportement** des objets.
Une méthode se définit à l'intérieur de la classe avec le mot-clé `def`.

Exemple

```
class Person(nom: String, age: Int) {
  def sePresenter(): Unit = {
    println(s"Bonjour, je m'appelle $nom et j'ai $age ans.")
  }
}
```

6 Héritage

Héritage = réutiliser les méthodes et attributs d'une **classe parent** dans une **classe enfant**.
L'héritage est défini avec le mot-clé `extends`.

```
class Person(nom: String, age: Int) {
  def sePresenter(): Unit = {
    println(s"Bonjour, je m'appelle $nom et j'ai $age ans.")
  }
}

class Student(nom: String, age: Int, classe: String) extends Person(nom, age) {
  def afficherClasse(): Unit = {
    println(s"Je suis en classe : $classe")
  }
}
```

7 Polymorphisme

Le **polymorphisme** permet de **redéfinir** le comportement d'une méthode héritée.

```
class Person(nom: String) {
  def sePresenter(): Unit = println(s"Je suis $nom")
}

class Student(nom: String) extends Person(nom) {
  override def sePresenter(): Unit = println(s"Je suis l'étudiant $nom")
}
```

8 Classes Abstraites

Une **classe abstraite** ne peut pas être instanciée.
On utilise le mot-clé `abstract`.

```
abstract class Animal {
  def crier(): Unit
}

class Chien extends Animal {
  def crier(): Unit = println("Wouf Wouf !")
}
```

9 Interfaces et Traits

Un **trait** est similaire à une **interface** en Java, mais il peut contenir des méthodes **implémentées**.

```
trait Volant {
  def voler(): Unit
}

trait Nageur {
  def nager(): Unit = println("Je nage !")
}

class Canard extends Volant with Nageur {
  def voler(): Unit = println("Je vole !")
}
```

Utilisation

```
val canard = new Canard()
canard.voler()
canard.nager()
```

10 Encapsulation

L'**encapsulation** signifie que les données sont **cachées** et **protégées**.
On utilise `private` et des **getters/setters**.

```
class BankAccount(private var balance: Double) {
  def deposit(amount: Double): Unit = balance += amount
  def getBalance: Double = balance
}
```

Qu'est-ce qu'une exception ?

Une **exception** est une erreur qui interrompt l'exécution normale d'un programme. Par exemple :

- Diviser un nombre par zéro.
- Accéder à un objet qui n'existe pas (null pointer).
- Lire un fichier qui n'est pas présent sur le disque.

Ces erreurs doivent être "capturées" pour éviter que le programme ne plante.

👉 **Exemple simple :**

```
val resultat = 10 / 0 // Problème ! Division par zéro.
```

Ce code va **crasher** avec une erreur de type **ArithmeticException**. Ce genre d'erreur peut être capturée et gérée.

Attraper et traiter les exceptions (try, catch, finally)

En Scala, on utilise trois mots-clés :

- **try** : On met le code risqué à l'intérieur.
- **catch** : On capture les erreurs et on dit quoi faire.
- **finally** : Ce code est toujours exécuté (pour nettoyer des ressources).

🛠️ Exemple simple

```
try {
  val resultat = 10 / 0 // Division par zéro
} catch {
  case e: ArithmeticException => println("Attention ! Division par zéro.")
  case e: NullPointerException => println("Attention ! Référence null.")
  case e: Exception => println(s"Erreur inattendue : ${e.getMessage}")
} finally {
  println("Fin du programme. On nettoie les ressources.")
}
```

3. Les outils modernes de Scala

En plus de `try`, `catch`, `finally`, Scala propose des outils plus propres et plus sûrs :

- **Option** : Pour dire qu'une valeur peut exister ou non.
- **Either** : Pour renvoyer soit un résultat, soit une erreur.
- **Try** : Pour encapsuler les erreurs sans planter le programme.

Option (pour éviter les NullPointerException)

Un **Option** représente soit une valeur, soit "rien" (**None**). Plutôt que d'avoir un **null**, on a une valeur qui "peut exister".

```
val nom: Option[String] = Some("Alice") // Il y a une valeur
val age: Option[Int] = None // Il n'y a pas de valeur

// On vérifie si on a une valeur
nom match {
  case Some(valeur) => println(s"Le nom est $valeur")
  case None => println("Pas de nom")
}
```

Try (pour attraper les erreurs sans try-catch)

Le **Try** est un outil moderne. Il encapsule un résultat "réussi" (**Success**) ou "en erreur" (**Failure**). Plus besoin d'utiliser `try-catch` !

Exemple classique :

```scala
import scala.util.{Try, Success, Failure}

val resultat = Try(10 / 0) // Provoque une erreur

resultat match {
  case Success(value) => println(s"Succès, le résultat est $value")
  case Failure(exception) => println(s"Échec ! L'erreur est : ${exception.getMessage}")
}
```

Either (pour renvoyer des erreurs et des succès)

Either peut contenir deux types de valeurs :

- **Left** (erreur)
- **Right** (succès)

C'est très utile quand une méthode peut renvoyer une erreur au lieu d'un simple résultat.

Exemple pratique :

```
def division(a: Int, b: Int): Either[String, Int] = {
  if (b == 0) Left("Impossible de diviser par zéro")
  else Right(a / b)
}

val resultat = division(10, 0)
resultat match {
  case Left(erreur) => println(s"Erreur : $erreur")
  case Right(valeur) => println(s"Succès, le résultat est $valeur")
}
```

4. Comparaison des outils de gestion d'erreurs

Méthode	Type	Description
`try-catch-finally`	**Impératif**	Capturer les erreurs comme en Java
Option	**Fonctionnel**	Évite `null` et gère l'absence de valeurs
Try	**Fonctionnel**	Évite `try-catch`, encapsule succès/erreur
Either	**Fonctionnel**	Renvoyer succès ou erreur (Right/Left)

1. Qu'est-ce qu'un générique ?

Un **générique** est une **variable de type**. Au lieu de définir une classe ou une méthode qui ne fonctionne qu'avec un type précis (**Int**, **String**, etc.), on la rend **générique** pour qu'elle puisse fonctionner avec **n'importe quel type**.

📘 Exemple simple

Imaginons qu'on veuille créer une "boîte" pour stocker **n'importe quel type de valeur**.
Si on la crée **sans génériques**, on est limité à un seul type, comme **Int** ou **String**.
Mais si on la rend **générique**, elle peut stocker **tout type** : **Int**, **String**, **Double**, etc.

🛠️ Exemple (Boîte de type fixe)

```
class BoiteInt(val contenu: Int) {
  def afficher(): Unit = println(s"Le contenu est $contenu")
}

val b = new BoiteInt(42)
b.afficher() // Affiche : Le contenu est 42
```

👉 **Problème** : On ne peut pas utiliser la boîte avec un **String** !

2. Comment rendre une classe générique ?

On peut rendre la **classe générique** en utilisant **[T]**.
T est un **type paramètre** (comme une variable, mais pour les types).
Il remplace le type **Int**, **String**, etc.
🛠️ **Exemple (Boîte générique)**

```
class Boite[T](val contenu: T) {
  def afficher(): Unit = println(s"Le contenu est $contenu")
}

val boiteInt = new BoiteteString = new Boite[String]("Bonjour")

boiteInt.afficher()    // Affiche : Le contenu est 42
boiteString.afficher() // Affiche : Le contenu est Bonjour
```

Explications :

1. `class Boite[T]` : `T` représente **n'importe quel type**.
2. `val contenu: T` : Le contenu de la boîte peut être un **Int**, **String**, etc.
3. On crée une `Boite[Int]` (pour les entiers) et une `Boite[String]` (pour les chaînes de caractères).

3. Génériques avec les méthodes

Les méthodes aussi peuvent être **génériques** ! On déclare le type `[T]` **après le nom de la méthode**.

🛠️ Exemple (Méthode générique)

```scala
def afficherDouble[T](valeur: T): Unit =
  println(s"Double affichage : $valeur $valeur")

afficherDouble(42)       // Double affichage : 42 42
afficherDouble("Scala")  // Double affichage : Scala Scala
```

Explications :

- `def afficherDouble[T](valeur: T)` : Cette méthode accepte un **T** (n'importe quel type).
- On peut utiliser `T` pour afficher des **Int**, **String**, **Double**, etc.

4. Génériques avec les fonctions anonymes

Les **fonctions anonymes** (les lambdas) peuvent aussi utiliser des génériques.

🛠️ Exemple (Fonction lambda générique)

```
val afficherDeuxFois: [T] => T => Unit =
  [T] => (valeur: T) => println(s"$valeur $valeur")

afficherDeuxFois(10)      // 10 10
afficherDeuxFois("Hello") // Hello Hello
```

Explications :

1. La syntaxe **[T] => T => Unit** est une manière de dire "On accepte **n'importe quel type** `T`".
2. **(valeur: T) =>** est une fonction lambda qui prend un **T**.
3. Ça fonctionne avec **Int**, **String**, et **Double**, etc.

5. Les bornes de type (upper bound, lower bound)

Tu peux contrôler les **types autorisés** pour **T**.
On utilise les **bornes supérieures (`<:`)** et **bornes inférieures (`>:`)**.

🔹 Borne supérieure (`<:`)

On impose que **T doit être un sous-type** d'un type donné.

```
class Animal
class Chien extends Animal
class Chat extends Animal

class BoiteAnimaux[T <: Animal](val contenu: T) {
  def afficher(): Unit = println(s"L'animal est un $contenu")
}

val chien = new BoiteAnimaux[Chien](new Chien) // OK
// val string = new BoiteAnimaux[String]("Bonjour") // Erreur !
```

Explications :

- `T <: Animal` signifie que **T doit être un Animal ou un sous-type d'Animal**.
- On peut mettre un **Chien** ou un **Chat** dans la boîte, mais pas un **String**.

Borne inférieure (`>:`)

On impose que **T doit être un super-type** d'un type donné.

```scala
def afficherMessage[T >: Int](message: T): Unit =
  println(s"Le message est $message")

afficherMessage(42)      // OK, car Int >= Int
afficherMessage(3.14)    // OK, car Double est au-dessus de Int
afficherMessage("Scala") // OK, car String est au-dessus de Int
```

Explications :

- `T >: Int` signifie que **T doit être un super-type de Int**.
- On peut utiliser **Int**, **Double**, et **String** car ils sont au-dessus de **Int**.

6. Cas concrets d'utilisation

1. **Les Collections (List, Option, etc.)**
   Les listes en Scala utilisent des génériques.

```scala
val listeEntiers: List[Int] = List(1, 2, 3)
val listeStrings: List[String] = List("a", "b", "c")
```

2. **Les méthodes utilitaires**

```scala
def maximum[T <: Ordered[T]](a: T, b: T): T =
  if (a > b) a else b

val max = maximum(10, 20) // 20
```

3. **Les classes immuables**

```scala
case class Boite[T](contenu: T)
val boite = Boite("Bonjour")
```

7. Exercice

💪 **Objectif** : Écris une classe générique **Pile[T]** (comme une pile d'assiettes) avec les méthodes suivantes :

1. **push**(élément: T) : ajoute un élément en haut de la pile.
2. **pop**() : retire le dernier élément de la pile.
3. **top**() : retourne l'élément en haut de la pile.

1. Qu'est-ce qu'une extension en Scala ?

En quelques mots, une **extension** permet d'ajouter de **nouvelles méthodes** ou **de nouvelles fonctionnalités** à une classe **déjà existante sans la modifier**. Cela s'appelle aussi **extension methods** en anglais.

👉 **Pourquoi est-ce utile ?**

- Tu n'as pas besoin de changer le code de la classe d'origine.
- Tu peux personnaliser des classes sans en créer de nouvelles.
- Cela rend le code plus propre, plus lisible et plus modulaire.

2. Comment créer une extension en Scala ?

En Scala 3, les extensions sont plus simples à écrire qu'en Scala 2. Tu as deux manières de faire :

Méthode 1 : Utiliser `extension`

C'est la manière la plus propre et la plus simple d'écrire une extension.

🛠️ Exemple simple : Ajouter une méthode `double` à Int

```
extension (x: Int)
  def double: Int = x * 2
```

Explications :

- `extension (x: Int)` signifie que l'on étend le type `Int` en lui ajoutant une méthode.
- La méthode `double` retourne la valeur de `x` multipliée par 2.

👉 **Utilisation :**

```
val nombre = 5
val resultat = nombre.double // 10
println(resultat) // Affiche : 10
```

Méthode 2 : Utiliser `given` et `implicit` (moins utilisé en Scala 3)

Dans cette méthode, tu utilises des `implicit classes` (surtout en Scala 2). Mais avec Scala 3, **on préfère la syntaxe "extension"**.

Exemple Scala 2 (ancienne méthode)

```
implicit class RichInt(x: Int) {
  def double: Int = x * 2
}
```

👉 **Utilisation :**

```
val resultat = 10.double // 20
println(resultat) // Affiche : 20
```

⚠️ **Attention** : Cette méthode est dépréciée en Scala 3. On préfère utiliser `extension`.

3. Plusieurs méthodes dans une seule extension

On peut définir **plusieurs méthodes** pour le même type.
On n'a pas besoin de faire un `extension` pour chaque méthode.

🛠️ Exemple : Plusieurs méthodes sur Int

```
extension (x: Int)
  def double: Int = x * 2
  def triple: Int = x * 3
  def isEven: Boolean = x % 2 == 0
```

👉 Utilisation :

```
val nombre = 6
println(nombre.double)  // 12
println(nombre.triple)  // 18
println(nombre.isEven)  // true
```

4. Extensions paramétrées (Génériques)**

🛠️ Exemple : Ajouter une méthode à toutes les listes

```
extension [T](liste: List[T])
  def headOption: Option[T] =
    if (liste.isEmpty) None
    else Some(liste.head)
```

👉 Utilisation :

```
val maListe = List(1, 2, 3)
val maListeVide = List.empty[Int]

println(maListe.headOption)      // Some(1)
println(maListeVide.headOption)  // None
```

Explications :

- `T` est un type générique (ça peut être Int, String, etc.).
- La méthode `headOption` renvoie la tête de la liste si elle existe, sinon `None`.
- On utilise `Option` au lieu de `null`, car c'est plus sûr.

5. Les extensions sur plusieurs types (multi-extensions)

Il est possible de créer des extensions **pour plusieurs types à la fois**.

🛠️ **Exemple : Ajouter une méthode commune à String et Int**

```
extension (x: Int | String)
  def repeat(times: Int): String =
    x match
      case i: Int => i.toString * times
      case s: String => s * times
```

👉 **Utilisation :**

```
val resultat1 = 3.repeat(2)  // "33"
val resultat2 = "Hi".repeat(3)  // "HiHiHi"

println(resultat1) // 33
println(resultat2) // HiHiHi
```

6. Pourquoi utiliser des extensions ?

👉 **Sans extension**

```
val nombre = 10
val double = nombre * 2
```

👉 **Avec extension**

```
val double = 10.double
```

Pourquoi c'est mieux ?

- **Code plus lisible** : `10.double` est plus expressif que `10 * 2`.
- **Réutilisable** : Tu peux ajouter tes méthodes à n'importe quel type (Int, String, List, etc.).
- **Propre et modulaire** : Au lieu de modifier la classe de base, tu ajoutes une extension **sans toucher au code d'origine**.

7. Cas d'utilisation concrets des extensions

1. **Formatage des chaînes de caractères**

```
extension (s: String)
  def toSnakeCase: String = s.replaceAll("([a-z])([A-Z])", "$1_$2").toLowerCase

println("HelloWorld".toSnakeCase) // "hello_world"
```

2. **Ajout de méthodes utilitaires à List[T]**

```
extension [T](liste: List[T])
  def containsAll(elements: List[T]): Boolean = elements.forall(liste.contains)

val liste = List(1, 2, 3, 4)
println(liste.containsAll(List(2, 3))) // true
println(liste.containsAll(List(2, 5))) // false
```

1. Qu'est-ce qu'un événement ?

Un **événement** est une **action ou un changement** qui se produit à un moment donné.
Exemples d'événements :

- **Un clic de souris** sur un bouton d'interface.
- **Une nouvelle connexion** à un serveur.
- **Un message envoyé** par un utilisateur sur un chat.

> **Programmation événementielle** signifie :
> ➡️ **Réagir automatiquement aux événements** au lieu de suivre une séquence d'actions prédéfinies.

2. Outils pour la programmation événementielle en Scala

En Scala, on utilise plusieurs outils pour capturer et réagir aux événements :

- **CallBacks** (fonctions de rappel)
- **Listeners** (écouteurs)
- **Observables / Observateurs** (RxScala)
- **Acteurs (Actors)** via **Akka**

Mais le plus **puissant et populaire** est **Akka**, car il permet de gérer des systèmes complexes et **concurrents**. On va le voir en détail.

3. Installer et configurer Akka en Scala

Pour utiliser **Akka**, il faut :

1. **Configurer le fichier build.sbt**.
2. **Importer les bibliothèques nécessaires**.

🟦 1. Ajouter Akka au fichier `build.sbt`

Voici le fichier `build.sbt` (fichier de configuration pour SBT, le gestionnaire de packages Scala).
Tu dois y ajouter les dépendances d'**Akka Actor** et **Akka Stream**.

```
ThisBuild / scalaVersion := "3.3.0" // Version de Scala

lazy val root = (project in file("."))
  .settings(
    name := "evenementiel-akka",
    version := "0.1",
    libraryDependencies ++= Seq(
      "com.typesafe.akka" %% "akka-actor-typed" % "2.7.0", // Acteurs typés
      "com.typesafe.akka" %% "akka-stream" % "2.7.0"         // Flux (streams) d'événements
    )
  )
```

🔹 2. Importer les bibliothèques

Pour **utiliser Akka dans le code**, tu dois importer les packages nécessaires.

```
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.ActorRef
```

4. Créer des acteurs (Actors) avec Akka

Dans **Akka**, un **acteur (Actor)** est un objet capable de **recevoir et réagir à des messages**.
Chaque acteur est indépendant et ne partage pas d'état avec les autres (c'est ce qui le rend "concurrent").

🛠️ Étape 1 : Créer un Acteur de type ChatBot

On va créer un acteur qui **réagit aux messages** comme un chatbot.
Il réagit à trois types de messages :

- **"Bonjour"** ➡️ Il répond "Salut à toi !"
- **"Comment ça va ?"** ➡️ Il répond "Je vais bien, et toi ?"
- **Tout autre message** ➡️ Il répond "Je ne comprends pas ce message."

🛠️ Code complet d'un acteur Akka (ChatBot)

```scala
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.Behaviors

object ChatBot {
  // 1. Comportement de l'acteur
  val comportement = Behaviors.receiveMessage[String] {
    case "Bonjour" =>
      println("Salut à toi !")
      Behaviors.same // L'acteur ne change pas son comportement
    case "Comment ça va ?" =>
      println("Je vais bien, et toi ?")
      Behaviors.same
    case message =>
      println(s"Je ne comprends pas le message : $message")
      Behaviors.same
  }

  def main(args: Array[String]): Unit = {
    // 2. Système d'acteurs
    val systeme = ActorSystem(comportement, "SystemeDeChat")

    // 3. Envoi de messages
    systeme ! "Bonjour"
    systeme ! "Comment ça va ?"
    systeme ! "Au revoir"

    // Arrêter le système d'acteurs après 2 secondes
    Thread.sleep(2000)
    systeme.terminate()
  }
}
```

🔹 Explications du code

1. **Comportement de l'acteur** :
   - L'acteur **ChatBot** reçoit des **messages de type String**.
   - Pour chaque message reçu, il réagit selon son contenu.
2. **Système d'acteurs** :
   - On crée le système d'acteurs avec **ActorSystem**.
   - Le système d'acteurs gère la **concurrence** et l'**ordonnancement des messages**.
3. **Envoyer des messages à l'acteur** :
   - On utilise **systeme ! "Bonjour"** pour envoyer un message.
   - L'opérateur **!** envoie des messages **de manière asynchrone**.

> 🧐 **Question pour toi** :
> Que se passe-t-il si on ajoute ce message ?

```
systeme ! "Parle-moi de Scala"
```

5. Utiliser les Observables et Observateurs (RxScala)

RxScala permet de gérer des **flux de données**.
Un **observable** (comme un flux d'eau) peut être suivi par un **observateur** (comme une personne qui regarde le flux).

🛠️ **Étape 1 : Ajouter la dépendance dans build.sbt**

Ajoute cette ligne dans **build.sbt**.

```
libraryDependencies += "io.reactivex.rxjava3" %% "rxscala" % "0.28.0"
```

🛠️ **Étape 2 : Exemple simple avec un observable**

On crée un flux d'événements et on observe ce qui s'y passe.
On suit **5 nombres** et on les **double avant de les afficher**.

```
import rx.lang.scala._

val observable = Observable.from(1 to 5) // Flux des nombres 1 à 5

val subscription = observable
  .map(_ * 2) // Double chaque nombre
  .subscribe(
    x => println(s"Reçu : $x"),
    e => println(s"Erreur : ${e.getMessage}"),
    () => println("Flux terminé !")
```

📘 Explications du code

1. **Observable.from(1 to 5)** : On crée un flux contenant **1, 2, 3, 4, 5**.
2. **.map(_ * 2)** : Chaque nombre est multiplié par 2.
3. **subscribe** : On s'abonne au flux.
   - `x => println(s"Reçu : $x")` : Affiche chaque élément reçu.
   - `e => println(s"Erreur : ${e.getMessage}")` : Capture les erreurs (s'il y en a).
   - `() => println("Flux terminé !")` : S'exécute à la fin du flux.

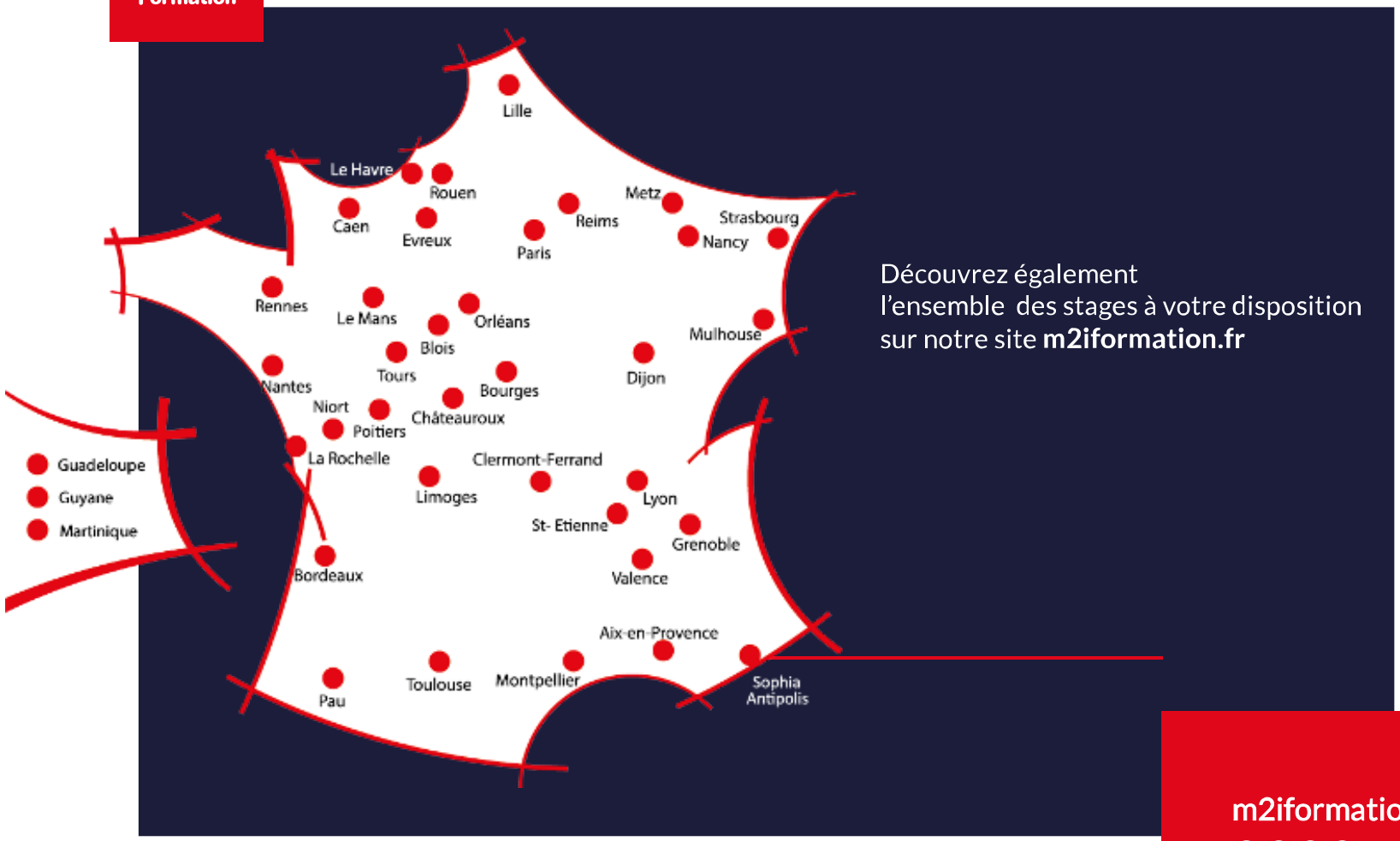
> 🧐 **Question pour toi** :
> Que se passe-t-il si on remplace **Observable.from(1 to 5)** par **Observable.from(1 to 100)** ?

6. Bonnes pratiques

1. **Évite les boucles bloquantes (while true).**
2. **Utilise les acteurs pour la concurrence.**
3. **Utilise les observables pour les flux de données.**

💪 Exercice

- Crée un acteur "Minuteur" qui affiche un compte à rebours de 10 secondes !



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr