Projet programmation 2
Introduction à Scala 3

Vincent Lafeychine
(Édition 2025 - 2026)

Ce document vous propose une introduction au langage de programmation Scala.

Certaines parties de ce document font référence à des liens de la documentation officielle de Scala :

- Les chapitres sont liés à des chapitres du Book, se présentant sous la forme de leçons.
- Les sous chapitres sont liés à des pages du Tour, se présentant sous la forme de fiches concises.
- Les fonctions présentées sont liées à des pages de l'API.

Ce document ne propose qu'un aperçu des éléments qui semble essentiel à découvrir. Ainsi, n'hésitez pas à explorer davantage la documentation officielle et à poser des questions si besoin.

Table des matières

1. Hello world ! *(Book: A Taste of Scala — A First Look at Types — Control structures)*

Scala est un langage de programmation objet construit par-dessus Java: tout ce qui est disponible en Java l'est aussi en Scala. Contrairement à Java, il est également possible d'écrire dans un style fonctionnel, ce qui permet d'améliorer grandement la lisibilité du code.

1.1. `sbt` (*Simple Build Tool*) *(Site internet de `sbt`)*

`sbt` est un outil pour compiler et lancer efficacement du code Scala (ou Java).

Pour l'utiliser, veuillez respecter la hiérarchie de dossiers suivante :

```
build.sbt
src/
    main/
        scala/
            <fichiers source>
        resources/
            <sprites>
```

Le fichier `build.sbt` contient les options de compilation, il contiendra la version du langage Scala :

```
scalaVersion := "x.y.z"
```

Écrivez un fichier `Main.scala` dans le dossier `src/main/scala/` contenant :

```
@main def main =
    println("Hello world!")
```

Vous lancerez la commande `sbt` à la racine de votre projet, ce qui lancera un interpréteur.

Cet interpréteur vous permet de :
- compiler votre projet avec `compile` ;
- lancer votre projet avec `run` ;
- ouvrir un *top-level* interactif (`REPL`) avec `console`.

En entrant `run` dans l'interpréteur, `sbt` va automatiquement :
- installer la bonne version du compilateur Scala et toutes les dépendances nécessaires ;
- vérifier si des changements ont eu lieu dans votre projet et compiler si besoin, puis ;
- lancer votre projet et afficher `Hello world!`.

Il est conseillé de laisser `sbt` ouvert lorsque vous développerez votre projet, `sbt` étant assez lent à s'initialiser. Il est préférable de tester rapidement avec le `REPL` et de construire les exemples plus avancés dans des fichiers.

1.2. Introduction et syntaxe *(Tour: Basics)*

Pour cette section, vous pouvez utiliser le `REPL` avec la commande `console`.

Recopiez ligne par ligne les commandes ci-dessous en observant leur résultat :

```
1 + 1
```

Pour déclarer une nouvelle variable mutable, on utilise le mot-clé `var` :

```
var x = 0
x = x + 1
x += 1
```

Pour déclarer une nouvelle variable immuable, on utilise le mot-clé `val` :

```
val y = 0
y = y + 1    // Erreur: 'y' est une variable immuable
```

On définit une fonction avec `def` :

```
def plusOne(n: Int): Int = n + 1

plusOne(1)
```

On peut également omettre le type de sortie, et Scala l'inférera :

```
def plusOne(n: Int) = n + 1
```

Cependant, nous vous encourageons à expliciter les types des fonctions pour plus de lisibilité.

Pour des fonctions de plusieurs lignes, on utilisera une indentation avec des espaces, similaire à Python :

```
var x = 0

def count() =
    x += 1
    println("Counter: " + x)          // Concaténation avec l'opérateur +
    println(s"Counter: ${x}")         // Interpolation de chaînes de caractères
```

La syntaxe des boucles `for` est la suivante :

```
for x <- 0 to 10 do
    println(x)
```

- *Écrivez une fonction* `def fibIter(n: Int): Int` *calculant la factorielle de n avec une boucle.*
- *Écrivez une fonction* `def fibRec(n: Int): Int` *calculant la factorielle de n avec des appels récursifs.*

1.3. Gestion des packages

Lors de la réalisation d'un projet, il est nécessaire de regrouper les différents composants entre eux. Scala, à travers l'utilisation de *packages*, permet de séparer ces composants dans différents fichiers.

Il est coutume de suivre une hiérarchie de dossiers pour les *packages* : les fichiers du package `x` se trouvent dans le dossier `src/main/scala/x/` et les fichiers du package `x.y.z` dans le dossier `src/main/scala/x/y/z/`.

- *Placez la méthode* `plusOne` *dans le package* `counter` *(dans* `src/main/scala/counter/plusOne.scala`*) :*

```
package counter

def plusOne(n: Int): Int = n + 1
```

- *Utilisez la méthode* `plusOne` *en incluant le* package `counter` *:*

```
import counter.plusOne

@main def main = println(plusOne(0))
```

- *Lancez le projet en utilisant la commande* `run` *dans l'interpréteur.*

2. Programmation orientée objet (POO)

(Book: Domain Modeling → OOP Modeling — Methods)

L'idée de base est d'identifier les *objets* manipulés par un programme et de structurer la programmation autour de ceux-ci. Un objet peut représenter un objet naturel qui interagit avec d'autres objets ou bien une structure de données avec ses opérations.

Quelques exemples pour des objets :

- dans une base de données : les tableaux, les requêtes, etc ;
- dans une interface graphique : les fenêtres, les boutons, etc ;
- dans un jeu vidéo : les différents acteurs.

Un objet possède des données mutables (`var`) et immuable (`val`) qui définissent l'état interne de l'objet et des *méthodes* (`def`). Les méthodes permettent d'interagir avec l'objet, elles peuvent modifier les données mutables, renvoyer de l'information et interagir avec d'autres objets.

L'exécution d'un programme consiste en l'interaction de ces objets, créés lors de cette exécution.

2.1. Classes

(Tour: Classes)

On regroupe des éléments similaires dans une *classe*. Dans le cas d'une interface graphique, les composants d'une fenêtre, tel que des boutons cliquables, formeraient une classe. On écrit donc du code en décrivant le comportement des classes.

Comme exemple, nous allons prendre un vélo : on considère qu'il dispose d'un compteur kilométrique et qu'il permet de bouger et freiner.

```
// Même si votre projet ne sera développé que par des francophones,
//   écrivez votre projet (noms de variables, documentation...) uniquement en anglais.
class Bicycle:
    var counter: Double = 0

    def move() =
        counter += 1
        println(s"Counter: ${counter}")

    def brake() =
        println("I stop here!")
```

Dans la fonction `main`, on place le code suivant qui sert à pédaler 10 km.

```
@main def main =
    val b = Bicycle()

    while (b.counter < 10) do
        b.move()

    b.brake()
```

La première ligne de notre fonction `main` crée une instance de `Bicycle` en effectuant un appel avec le nom de la classe. La fonction qui porte le même nom que la classe et qui permet de créer une nouvelle instance de cette classe s'appelle un constructeur de la classe.

▸ *Lancez le projet en utilisant la commande `run` dans l'interpréteur.*

Dans le paradigme objet, chaque instance possède ses propres données. Ainsi, nous pouvons créer deux vélos en même temps, chacun ayant son propre compteur :

```
@main def main =
    val b1 = Bicycle()
    val b2 = Bicycle()

    while (b1.counter < 10) do
        b1.move()

    while (b2.counter < 5) do
        b2.move()
```

▸ *Définissez une méthode* `def travel(distance: Int)` *dans* `Bicycle` *qui n'utilise que la fonction* `move`*. L'objectif est d'utiliser* `b1.travel(10)` *et* `b2.travel(5)` *plutôt que de dupliquer les boucles* `while`*.*

Les objets peuvent prendre des paramètres lors de leur création :

```
class Bicycle(val name: String):
    ...

@main def main =
    val b = Bicycle("Moulinette")
```

Sur l'exemple ci-dessus, le constructeur de la classe `Bicycle` prend un paramètre `name`, qui sera nécessaire de fournir lors de la création de la classe.

Il est à noter que ces paramètres peuvent être précédés de `var` (mutable) ou de `val` (immuable). Si aucune annotation n'est mise, le paramètre sera immutable et uniquement accessible dans le corps de l'objet.

▸ *Faites afficher le nom du vélo dans* `move`*.*

2.2. Héritage

(Tour: Classes)

Un aspect intéressant de la programmation orientée objet est le partage de code. Si certains objets d'une classe ont des comportements différents ou supplémentaires par rapport aux autres membres, il convient de les regrouper dans une sous-classe en formant donc une hiérarchie.

Dans une sous-classe, on ne décrit que les différences avec la super-classe. La terminologie « classe enfante » (sous-classe) et « classe parente » (super-classe) existe également.

Par exemple, considérons un vélo de route (qui est plus rapide) et un vélo rouillé (qui fait du bruit lors du freinage) en remplaçant les méthodes de la super-classe en utilisant le mot-clé `override` :

```
class RoadBicycle(name: String) extends Bicycle(name):
    override def move() =
        counter += 1.5
        println(s"${name}: *moving*")

class RustyBicycle(name: String) extends Bicycle(name):
    override def brake() =
        println("eek")
```

On peut toutefois réutiliser une fonction remplacée en utilisant le mot-clé `super`. Par exemple, il est possible d'écrire la méthode `move` dans `RoadBicycle` de la manière suivante :

```
override def move() =
    counter += 0.5
    super.move()
```

Toute classe possède une méthode `toString`, la représentation canonique textuelle de chaque objet:

```
@main def main =
    val b = Bicycle("Moulinette")
    println(b)
```

Par défaut, les objets affichent le nom de la classe dont ils sont issues, ainsi que leur adresse dans la mémoire.

▸ *Redéfinissez la méthode `toString` pour afficher le nom et le compteur d'une bicyclette.*

2.3. Modificateurs de visibilité

(Tour: Classes)

Chaque classe doit être responsable des éléments et méthodes qui lui appartiennent.

Par exemple, le compteur `counter` de notre vélo ne devrait pas être accessible à la modification :

```
@main def main =
    val b = Bicycle("Moulinette")
    b.counter = 42       // Il faut interdire cette modification.
    println(b.counter)   // Il faut cependant autoriser cette lecture.
```

Il est possible d'écrire `counter` de façon immuable, mais cela nous empêcherait d'écrire `move` :

```
class Bicycle:
    val counter: Double = 0

    def move() =
        counter += 1      // Erreur: 'counter' est une variable immuable
```

Le paradigme objet offre la possibilité de restreindre la visibilité des données en utilisant les mots-clés :

- `private` : Empêchant à tout autre objet d'accéder à l'élément ;
- `protected` : Empêchant à tout autre objet d'accéder à l'élément, sauf aux sous-classes.

Il est à noter que ces mots-clés fonctionne sur les variables d'une classe, mais également ses fonctions.

Une première étape est donc d'utiliser `protected` sur la variable `counter`, laissant la possibilité aux classes `RoadBicycle` et `RustyBicycle` de modifier la variable :

```
class Bicycle:
    protected val counter: Double = 0
```

Une fois fait, il n'est plus possible d'accéder depuis l'extérieur de la classe :

```
@main def main =
    val b = Bicycle("Moulinette")
    b.counter = 42       // Ne fonctionne pas : Parfait.
    println(b.counter)   // Ne fonctionne pas : Il faut autoriser cette lecture.
```

Pour remédier à cela, nous allons ajouter une fonction `counter`, appelée *getter*, à notre classe `Bicycle` qui permettra d'accéder à l'élément `counter` :

```
class Bicycle:
    private val counter: Double = 0

    def counter: Double = counter

    // Erreur: Double définition de 'counter' ('var counter' et 'def counter').
```

Le code ci-dessus ne va pas compiler, `counter` faisant référence à la fois à la variable et à notre nouvelle fonction *getter*. Il est commun de lever l'ambiguïté en préfixant la variable par un *underscore* :

```
class Bicycle:
    // Tout accès de '_counter' passera maintenant par le nouveau getter `counter`.
    private val _counter: Double = 0

    def counter: Double = _counter

    ...
```

Il est important de remarquer que la méthode `counter` ne possède pas de parenthèses. En effet, lorsqu'une fonction ne prend pas d'argument et n'effectue aucun effet de bord (ce qui est le cas ici), il est commun d'omettre les parenthèses dans sa définition ainsi que lors de son appel pour expliciter cette caractéristique.

Il est également possible de modifier la façon dont une variable est modifiée en définissant un *setter* :

```
class Bicycle:
    private var _counter: Double = 0

    // Notez la syntaxe utilisée ici pour définir un setter (b.counter = newValue)
    def counter_=(newValue: Double): Unit =
        if newValue > 0 then
            _counter += newValue

    ...
```

Définir un *getter* et un *setter* est très courant en programmation objet afin de conserver les invariants que chaque classe souhaite préserver (comme l'exemple du *setter* de `counter` qui empêche `counter` d'être négatif). Faites donc bien attention aux attributs de visibilités de vos variables et méthodes !

2.4. Traits

(Tour: Traits)

Un trait définit un ensemble de méthodes que des objets vont devoir implémenter, cela permet de regrouper des comportements communs.

Par exemple, un trait `Vehicle` générique regroupe des moyens de déplacements spécifiques comme les vélos ou les trains. Il ne sera pas possible d'instancier un véhicule, mais il sera possible d'instancier un vélo ou un train.

Un voyage se fait en bougeant jusqu'à ce qu'on ait parcouru une certaine distance. Considérons donc la déclaration suivante :

```
trait Vehicle:
    private val _counter: Double = 0

    def counter: Double = _counter

    def counter_=(newValue: Double): Unit =
        if newValue > 0 then
            _counter += newValue

    def move(): Unit

    def brake() =
        println("I stop here!")
```

La méthode `move` n'est pas définie dans le trait `Vehicle`. Toute sous-classe de `Vehicle` devra spécifier le comportement concret de `move` dont on ne connaît que le type.

▸ *Faites de `Bicycle` une sous-classe de `Vehicle` et ajoutez une autre sous-classe `Scooter`.*

Une même classe peut hériter d'une seule super-classe, mais de multiples traits.

Par exemple, les déclarations suivantes sont possibles si `B` est une classe et si `C` et `D` sont des traits :

```
class A extends B: ...
class A extends C: ...
class A extends B with C: ...
class A extends B with C with D: ...
```

En Scala, il est coutume d'écrire un fichier pour chaque classe/trait, nommé avec le nom de votre classe/trait :

▸ *Écrivez les différents classes/traits de cette section dans les fichiers correspondants.*

3. Collections et programmation fonctionnelle

(Book: Domain Modeling → FP Modeling — Collections — Functional Programming)

Scala propose deux types de structures de données. Le contenu des classes peut être :

- mutable, voir le *package* `scala.collection.mutable`, ou ;
- immuable, voir le *package* `scala.collection.immutable`.

Nous allons explorer les collections suivantes :

- les tableaux, du *package* `scala.Array` (équivalent à `scala.collection.mutable.ArraySeq`) ;
- les listes, du *package* `scala.collection.immutable.List` et ;
- les tableaux associatifs, du *package* `scala.collection.immutable.HashMap`.

▸ *Avant de lire la suite, explorez les graphes du sous-chapitre « Three main categories of collections » du chapitre « Collections types » du* Book.

▸ *Identifiez dans les graphes où se situent les collections énumérées (en fond gris), ainsi qu'à quel trait elles appartiennent (en fond bleu).*

3.1. Les tableaux

(API: `scala.Array`)

Commençons avec un type usuel, les `Array` :

```
val arr = Array(1, 3, 5)

println(s"First element: ${arr(0)}")    // L'accès s'effectue comme un appel de fonction
```

Il existe plusieurs possibilités pour déclarer un tableau :

```
val a = Array(1, 3, 5)
val b = Array[String]("Monday", "Tuesday", "Wednesday")
val c = Array.ofDim[Int](3)
```

Les deux premières lignes créent un `Array` avec du contenu. Dans le premier cas, le type est déterminé implicitement par Scala (`Int`). Dans le deuxième cas, on spécifie le type explicitement (`String`). Dans le troisième cas, on utilise une méthode, dite statique (voir Section 4.1), de la classe `Array` afin de créer un tableau de 3 entiers, initialisé avec la valeur `0`.

Il y a deux choses importantes à remarquer :

- `Array` (et les autres collections que nous allons voir) sont des exemples de classe *polymorphe* (voir Section 5), paramétrée par un type (tel que `Int`, `String`, ou une classe quelconque car aucune contrainte n'est imposée).
- Dans les exemples ci-dessus, `a`, `b` et `c` sont des *références* à des objets du type `Array[_]`. Définir ces variables avec `val` indique que les variables garderont le même objet (ici, une instance de la classe `Array`), mais le contenu que possède l'objet peut muter. Ainsi, une déclaration telle que `val x = a` crée simplement une deuxième référence vers l'objet pointé par `a`.

Les tableaux multidimensionnels sont supportés par Scala grâce à la méthode `ofDim`.

Pour créer un `Array` de taille 5 dont les éléments sont des `Array` d'`Int` de taille 3 :

```
val g = Array.ofDim[Int](5, 3)

g(4)(2) = 84
```

3.2. Les listes

(API: `scala.collection.immutable.List`)

`List` est un conteneur immuable. Après son instantiation, toute tentative de modification échouera :

```
val a = List(1, 2, 3, 4)
val b = List(5, 6, 7, 8)

println(3 :: a)
println(a ++ b)

a(2) = 5     // Erreur: `update` n'est pas un membre de `List[Int]`
```

`List` est réalisée par des listes chaînées. Du coup, `a(i)` n'est pas une opération en temps constant. De plus, certaines opérations sur les `List` créent de nouveaux objets de type `List`, qui auront un coût de $O(n)$ pour n éléments.

Les différentes collections se distinguent donc par les opérations possibles sur les objets et leur efficacité. Un comparatif des collections permet de sélectionner la collection adéquate.

3.3. Les tableaux associatifs

(API: `scala.collection.immutable.HashMap`)

La classe `HashMap` implémente les ensembles associatifs qui stockent des paires (clé, valeur); on peut alors retrouver la valeur à partir de la clé. La classe `HashMap` est alors paramétrée par les types des clés et des valeurs.

La classe `HashMap` n'est pas disponible dans le *prelude* (l'ensemble des classes, fonctions ou variables définies par défaut dans Scala), il va être nécessaire d'importer la classe `HashMap` (voir Section 1.3):

```
import scala.collection.immutable
```

Créons un tableau associatif (par défaut non mutable) des `String` aux `Int` avec deux paires :

```
val days = immutable.HashMap("Monday" -> 1, "Tuesday" -> 2)

println(days("Monday"))
```

Créons un tableau associatif initialement vide, mais mutable, en explicitant obligatoirement les types :

```
import scala.collection.mutable

val attempt = mutable.Map()              // Map[Nothing, Nothing] -> Tableau inutile

val days = mutable.Map[String, Int]()    // Map[String, Int]
days("Wednesday") = 3
```

3.4. Opérations sur les conteneurs

On est souvent amené à traverser des collections de données. Le paradigme fonctionnel permet d'enchaîner les traversements de collections tout en y appliquant des modifications.

Il est possible de récupérer chaque élément d'un tableau grâce à un *for comprehension* :

```
val arr = Array(5, 2, 8, 1)

for (i <- arr) do
    println(i)
```

Une méthode similaire est d'utiliser la méthode `foreach` :

```
def printElem(x: Int): Unit = println(x)

arr.foreach(printElem)          // Appel avec une fonction explicitement définie
arr.foreach(x => println(x))    // Appel avec une fonction anonyme
arr.foreach(println(_))         // Appel avec une fonction anonyme
```

Il est à noter que la dernière syntaxe n'est possible que pour les fonctions anonymes ayant un seul paramètre qui n'apparaît qu'une seule fois dans le corps de la fonction.

Des opérations permettent de générer de nouvelles listes :

```
// Transformation de la liste
arr.map(2 * _)

// Filtrer la liste en fonction du prédicat
arr.filter(_ >= 5)
```

Des opérations permettent d'effectuer des calculs sur les éléments :

```
arr.exists(_ >= 7)
arr.find(_ >= 7)
arr.count(_ <= 7)
arr.foldLeft(0)((a, b) => a + b)
```

- *Retrouvez la page de l'API contenant l'ensemble de ces fonctions (regardez l'indice d'abord, puis la solution).*
- *Indice : Rechercher la page du Trait le plus générique concernant les collections (solution de l'indice).*

Quelques remarques :

- Le type de retour de `find` est un type `Option[T]`, qui peut soit être `None`, soit `Some(x: T)`, où `T` est le type des éléments de `arr`. Nous allons détailler cela dans la section sur le *pattern matching* (Section 4.3).
- La fonction `foldLeft` réduit successivement vers une seule valeur, avec `0` comme valeur initiale. La fonction `foldLeft` applique ensuite la fonction `f` sur la valeur initiale et le premier élément, puis sur le résultat et le deuxième élément, et ainsi de suite : `f(0, f(arr(0), f(arr(1), ...)))`.

La fonction `reduceLeft` est similaire à `foldLeft`, mais omet la valeur initiale :

```
arr.reduceLeft((a, b) => (a + b))
```

La syntaxe `arr.reduceLeft(_ + _)` est également possible. De plus, il existe la méthode `sum` (`arr.sum`). Notez l'omission des parenthèses sur la méthode `sum`, indiquant qu'aucun effet de bord n'a lieu sur l'objet.

En utilisant au mieux les méthodes de l'objet `Array[String]` :

- *Faites la concaténation des éléments en un seul `String`, en séparant les éléments par un espace.*
- *Trouvez la longueur de la chaîne la plus courte.*

3.5. *For comprehensions*

(Tour: For comprehensions)

Scala offre une notation compacte des boucles imbriquées avec les *for comprehensions*.

Considérons la classe `Cell` qui représente une cellule avec coordonnées (x, y) contenant une valeur v :

```
class Cell(val x: Int, val y: Int):
    // La valeur '_active' de la cellule n'est pas accessible depuis l'extérieur
    private var _active: Boolean = false

    def isActive: Boolean = _active

    def activate() =
        _active = true

    // On réécrit la méthode `toString`
    override def toString(): String =
        s"($x, $y) = ${if _active then "T" else "F"}"
```

Initialisons une grille 6×6 de cellules, et définissons une fonction pour visualiser la grille :

```
val grid = Array.ofDim[Cell](6, 6)

def printGrid(grid: Array[Array[Cell]]) =
    grid.map(_.mkString(", ")).foreach(println)
```

Une *for comprehension* permet d'itérer sur chaque *générateur*, tout en filtrant des valeurs si besoin :

```
for (x <- 0 to 5; y <- 0 to 5) do
    grid(x)(y) = Cell(x, y)

for (x <- 0 to 5; y <- 0 to 5; if Math.abs(x - y) == 2) do
    grid(x)(y).activate()
```

Avec le mot-clé `yield`, une *for comprehension* peut construire une liste :

```
val cells = for (x <- 0 to 5; y <- 0 to 5; if grid(x)(y).isActive) yield grid(x)(y)
```

Cependant, itérer avec les positions d'une grille n'est pas conseillé, il est préférable d'avoir un moyen sûr d'itérer sur l'ensemble des cellules sans en oublier (ou pire, itérer en dehors des bornes).

En utilisant `flatten`, la grille est « aplatie » en une liste sur lequel il est maintenant possible d'itérer :

```
val cells = for (cell <- grid.flatten; if cell.isActive) yield cell
```

▸ *Essayez d'itérer en dehors des bornes : Que se passe-t'il ?*

4. Propriétés sur les objets en Scala

(Book: Domain Modeling → Tools — Control structures)

4.1. *Companion objects*

(Tour: Singleton objects)

Une méthode n'a pas tout le temps besoin d'une instance d'un objet pour être invoquée. Par exemple, c'est le cas de la méthode `ofDim` de l'objet `Array` dans la Section 3 sur les collections.

Ces méthodes sont dites *statiques*, et sont représentées en Scala à travers les *companion objects* :

```
class Circle(radius: Double):
    def area: Double = Circle.areaFormula(radius)

object Circle:
    def areaFormula: (Double => Double) = radius => Math.PI * Math.pow(radius, 2.0)

println(Circle(5.0).area)
```

Notons une nouvelle fois que les parenthèses de la fonction `area` ont été omises. En effet, cette fonction ne prend aucun paramètre et n'effectue aucun effet de bord.

4.2. *Case class*

(Tour: Case classes)

Les *case class* sont des classes avec des propriétés particulières. Il convient de penser leurs instances comme des n-uplets immuables.

Regardons l'exemple suivant :

```
case class Complex(re: Double, im: Double):
    def size: Double =
        scala.math.sqrt(re * re + im * im)

    def +(c: Complex): Complex =
        Complex(re + c.re, im + c.im)

val c = Complex(1, 2)
val d = c + Complex(-2, 3)
val e = Complex(1, 1)

println(d)
println(c == e.copy(im = 2))
```

Un nombre complexe est défini entièrement par ses parties réelle et imaginaire. Ces données sont connues lors de la création d'une instance de `Complex`. Les opérations sur les nombres complexes ne changent pas leur état interne (contrairement à l'exemple de la bicyclette).

Les *case class* sont équipées des opérations suivantes :

- L'opérateur == comparant les paramètres de deux instances et non leurs adresses en mémoire ;
- Une méthode `toString` définie automatiquement (voir le résultat de `println`) ;
- Une méthode `copy`, qui prend comme arguments optionnels l'ensemble des paramètres de la *case class*.

Ainsi, les paramètres d'une *case class* sont des `val` immuables, et la pratique de déclarer des `var` à l'intérieur est découragée (ceux-ci ne seraient pas pris en compte par ==).

- *Comparez avec le comportement de == et `toString` lorsque `Complex` est une classe ordinaire.*
- *Définissez une classe abstraite `Tree` avec une méthode `sum: Int`, puis définissez les sous-classes `Node` et `Leaf` de manière que le programme `Node(Node(Leaf(2), Leaf(3)), Leaf(5)).sum()` renvoie* 10.

4.3. *Pattern matching*

(Tour: Pattern matching)

Tout comme en OCaml, il est possible d'écrire du *pattern matching* sur des *case class*.

Avec la *case class* `Tree` demandée en exercice de la Section 4.2, il est possible d'écrire :

```
val t = Node(Node(Leaf(1), Leaf(2)), Leaf(3))

def sumTree(t: Tree): Int =
    t match
        case Leaf(v) => v
        case Node(l, r) => sum(l) + sum(r)
```

Lorsque le `match` est le seul élément de la fonction, il est possible de réduire la syntaxe comme suit :

```
def sumTree: Tree => Int =
    case Leaf(v) => v
    case Node(l, r) => sum(l) + sum(r)
```

Dans chaque `case` d'un *pattern matching*, il est possible d'ajouter des tests sur les valeurs comme suit :

```
def f: Tree => Int =
    case Leaf(0) => 0
    case Leaf(v) if v < 0 => -1
    case Leaf(v) if v > 0 => 1
    case Node(l, r) => f(l) + f(r)
```

- *Définissez des classes représentant une expression arithmétique sur des entiers relatifs avec les opérateurs* `Add`*,* `Mul` *et* `Abs` *(valeur absolue).*
- *Définissez une fonction qui évalue l'expression.*

Le *pattern matching* est simple dans le cas des *case class*, il est également de faire du *pattern matching* avec d'autres types d'objets : N'hésitez pas à regarder le *Tour*.

5. Programmation polymorphe

(Book: Types and the type system — Contextual abstractions)

Dans la Section 3, on a vu des classes polymorphes telles que `Array[String]` ou `List[Int]`. Dans ces exemples, `Array` et `List` sont des classes qui fournissent une fonctionnalité commune pour des objets de différents types.

Dans cette section, nous étudierons quelques exemples dans lesquels ces classes paramétrées pourront être utiles, ainsi que leur fonctionnement de base.

5.1. Polymorphisme

(Tour: Generic classes — Polymorphic methods)

Définissions des graphes. Pour l'instant, un graphe est composé d'un seul nœud qui porte un nom :

```
class Node(val name: String)
class Graph(val root: Node)

val g = Graph(Node("racine"))
println(g.root.name)
```

Maintenant, supposons que l'on souhaite utiliser cette classe dans de multiples contextes, comme :

- dans notre exemple, où les sommets ne portent que des noms
- dans un calcul de distance, où les sommets sont équipés d'un poids
- dans un arbre généalogique, où les sommets portent des informations sur des personnes...

À défaut d'ajouter toutes les informations requises dans tous ces contextes à la seule classe `Node` (ce qui requiert une exhaustivité et rendrait cette classe peu lisible), on préfère la création des sous-classes.

À priori, aucun souci :

```
class WeightNode(name: String, val weight: Int) extends Node(name)

val g = Graph(WeightNode("racine", 5))
println(g.root.name)
```

À partir d'un `Graph`, il est possible de récupérer le champ `name`, car `WeightNode` est une sous-classe de `Node`. Cependant, essayons d'afficher le poids de la racine avec la ligne suivante :

```
println(g.root.weight)   // Erreur: Le membre 'weight' ne fait pas parti de 'Node'
```

Cela ne fonctionne pas, car le type de l'attribut `root` est `Node`, tel que définit par la classe `Graph`. Or, pour obtenir le poids de `root`, il faut que Scala comprenne qu'il s'agit d'un `WeightNode`.

C'est à cela que servent les classes polymorphes, changeons la déclaration de `Graph` :

```
class Graph[N](val root: N)
```

Ici, `N` est un type générique qui paramétrise la classe `Graph` qui sera spécifié lorsqu'on instancie un graphe.

Lorsque `N` correspond à `WeightNode`, `g` est de type `Graph[ExtNode]` et `g.root` devient `WeightNode` :

```
val g = Graph(WeightNode("racine", 5))
println(g.root.weight)
```

5.2. Types bornés et bornes multiples

(Tour: Upper Type Bounds)

Ajoutons une méthode à la classe `Graph` pour afficher le nom d'un sommet :

```
class Graph[N](val root: N):
    def printRootName() =
        println(root.name)   // Erreur: Le membre 'name' ne fait pas parti de 'N'
```

Or, ceci ne fonctionne plus car `root` est désormais du type `N`, qui ne possède pas de tel méthode `name`.

La solution consiste à spécifier que le type générique `N` doit être une sous-classe de `Node` :

```
class Graph[N <: Node](val root: N):
    def printRootName() =
        println(root.name)
```

Ajoutons des classes spécifiques concernant les graphes, spécifions une liste d'arêtes et de sommets :

```
class Node(val name: String)
class WeightNode(name: String, val weight: Int) extends Node(name)

class Edge(val from: Node, val to: Node)

class Graph[N <: Node](val nodes: List[N], val edges: List[Edge]):
    def outgoing(n: N) =
        edges.filter(_.from == n)

    def successors(n: N) =
        outgoing(n).map(_.to)

val n1 = WeightNode("n1", 5)
val n2 = WeightNode("n2", 3)
val edge = Edge(n1, n2)
val g = Graph(List(n1, n2), List(edge))

g.successors(n1).foreach(n => println(n.name))
```

Cet exemple fonctionne, mais échoue si on remplace `name` par `weight` dans la fonction `println` de la dernière ligne. En effet, la méthode `successors` travaille avec une liste de `Edge` qui ne relie que des `Node`.

▸ *Réparez cet exemple, en ajoutant uniquement des bornes supplémentaires aux types.*

Supposons maintenant qu'on veut utiliser la classe `Graph` dans un contexte où les arêtes portent des étiquettes, et qu'on a envie de récupérer les étiquettes sur les arêtes sortantes d'un sommet.

On définit alors une sous-classe d'arêtes :

```
class LabelEdge(from: Node, val label: String, to: Node) extends Edge(from, to)

val n1 = WeightNode("n1", 5)
val n2 = WeightNode("n2", 3)
val edge = LabelEdge(n1, "a", n2)
val g = Graph(List(n1, n2), List(edge))

g.outgoing(n1).foreach(e => println(e.label))  // Erreur: Le membre 'label' ne fait
                                               //   pas parti de 'Edge'
```

▸ *Réparez cet exemple, en ajoutant uniquement des bornes supplémentaires aux types.*

5.3. Covariance et contravariance

(Tour: Variances)

Dans l'exemple précédent, `Edge[A]` et `Edge[B]` sont deux classes distinctes et incomparables. Même si `B` était une sous-classe de `A`, cela ne fait pas de `Edge[B]` une sous-classe de `Edge[A]`, ou inversement.

Parfois, un tel comportement est néanmoins désirable; regardons l'exemple suivant qui réalise une matrice d'un type `T` quelconque (`f` étant une fonction qui fournit le contenu de la matrice) :

```
class Matrix[T](height: Int, width: Int, f: (Int, Int) => T):
    val data = IndexedSeq.tabulate[T](width,height) { (i, j) => f(i,j) }
```

Disons qu'on utilise une matrice de `A`, où `A` possède une donnée `n` entier, et qu'on souhaite calculer les sommes des `n` dans toutes les lignes avec `rowsums` :

```
class A(val n: Int)

def rowsums(m: Matrix[A]) =
    m.data.map(_.map(_.n).sum)

val m1 = Matrix(3, 3, (i, j) => A(i * 3 + j))
println(rowsums(m1).mkString(", "))
```

Supposons maintenant qu'on possède une sous-classe `B` de `A`. Puisque toute instance de `B` possède, elle aussi, un `n`, on veut naturellement utiliser `rowsums` avec une matrice de `B` :

```
class B(n: Int) extends A(n)

val m2 = Matrix(3, 3, (i, j) => B(i * 3 + j))
println(rowsums(m2).mkString(", "))  // Erreur: 'm2' est du type 'Matrix[B]',
                                     //   le type attendu étant 'Matrix[A]'
```

Or, cela échoue car `m2` est du type `Matrix[B]` considéré incompatible avec `Matrix[A]`, attendu par `rowsums`. Modifions, comme suit, la définition de la matrice :

```
class Matrix[+T](...):
    ...
```

Dans ce cas, `Matrix[B]` sera considéré comme une sous-classe de `Matrix[A]` (car `B` est sous-classe de `A`), et `rowsums` accepte bien `m2`. Dans ce cas, on dit que `Matrix` est *covariant*.

Dans certains cas rares, on a besoin de l'effet inverse (*contravariance*): une déclaration de la forme `class Matrix[-T](...)` ferait de `Matrix[A]` une sous-classe de `Matrix[B]` si `B` est sous-classe de `A`.

Vous voilà fin prêt pour développer votre propre projet en Scala !

Rendez-vous la semaine prochaine pour découvrir la librairie graphique.

En attendant, si vous le souhaitez, continuez de parcourir les différents liens donnés dans ce document.