Introduction à Scala
Projet programmation 2

Emile Contal Nathan Grosshans Mathieu Hilaire Juraj Kolčák
Vincent Lafeychine Amélie Ledein Louis Lemonnier Stefan Schwoon

20 janvier 2023

Table des matières

1 Le langage Scala et `sbt` (Book)

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 web)

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

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

```
src/
    main/
        scala/
            yourfiles.scala
        resources/
            (yourimages.png)
```

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

```
scalaVersion := "3.2.1"
```

`sbt` ne recompile que les sources qui ont été modifiées, ce qui fait gagner beaucoup de temps.

Vous mettrez éventuellement vos fichiers auxiliaires (par exemple `images/`) dans le dossier `resources/`.

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

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

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

Lorsque vous taperez `compile` ou `run`, `sbt` compilera vos fichiers dans un dossier `target/`.

Nous allons ignorer le dossier `target/` dans `git`. Copiez le fichier `.gitignore` à la racine du projet :

```
# bloop and metals
.bloop
.bsp

# metals
project/metals.sbt
.metals

# scala 3
.tasty
.scala_dependencies
*.class

# sbt
project/project/
project/target/
target/
```

1.2 Introduction et syntaxe (Tour)

Pour commencer, lancez l'interpréteur Scala en tapant `sbt`, puis `console`, et 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
```

On peut déclarer des valeurs immuables avec `val` :

```
val y = 0
y = y + 1
```

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

plusOne(1)
```

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

Une exception est faite lorsque votre fonction ne renvoie rien (`Unit`).

Pour des fonctions de plusieurs lignes, on utilisera la syntaxe suivante (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
```

On remarque que lorsque la fonction ne prend pas d'argument, on peut omettre les parenthèses lors de son appel et de sa définition. Cela indique que la fonction n'effectue pas d'effets de bord, comme l'accès à des champs dans un objet (voir section 2 sur la programmation objet).

Une syntaxe des boucles `for` est la suivante :

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

▶ *Écrivez une fonction calculant la factorielle d'un entier de deux manières différentes : avec une boucle et récursive.*

2 Programmation orientée objet (Book)

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 ;
- dans une interface graphique, les fenêtres, les boutons, etc ;
- dans un jeu graphique, les différents acteurs.

Un objet dispose des données (des `val` et `var`) qui définissent l'état interne de l'objet et des *méthodes* (des `def`). Les méthodes permettent d'interagir avec l'objet, elles peuvent modifier les données internes, renvoyer de l'information et interagir avec d'autres objets. L'exécution d'un programme démarre avec un certain nombre d'objets, dont un qui contient la méthode `main`. Pendant l'exécution, d'autres objets peuvent être créés.

2.1 Classes (Tour)

On regroupe des objets similaires dans une *classe*. Par exemple, dans une interface graphique, les fenêtres formeraient une classe. On écrit donc du code en décrivant le comportement des classes.

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

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

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

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

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

```
val b = Bicycle()

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

b.brake()
```

La première ligne crée une instance de `Bicycle` en effectuant un appel avec le nom de la classe.

Dans le paradigme objet, chaque instance possède ses propres données.

Du coup, on peut créer deux vélos en même temps, chacun ayant son propre compteur :

```
val b = Bicycle()
val c = Bicycle()

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

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

Les objets peuvent prendre des paramètres lors de leur création :

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

val b = Bicycle("Moulinette")
```

Il est à noter que les paramètres sont précédés de `var` (mutable) ou de `val` (immutable).

Si un telle annotation n'est pas mise, le paramètre ne sera accessible que dans le corps de l'objet.

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

2.2 Héritage

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

Dans une sous-classe, on ne décrit que les différences avec la super-classe. Par exemple, considérons un vélo de route (qui est plus rapide que les autres) ou un vélo rouillé (qui fait du bruit lors du freinage) :

```
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")
```

Le mot-clé `override` spécifie que la méthode remplace celle de la super-classe. On peut toutefois réutiliser une fonction remplacée, par exemple `move` dans `RoadBicycle` est équivalent à :

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

Une méthode qui accepte comme argument un objet d'une classe `C` peut travailler sur n'importe quelle sous-classe de `C`, tout en utilisant les méthodes remplacées de la sous-classe.

▶ *Regroupez les lignes concernant* `move` *et* `brake` *dans* `main` *dans une méthode* `travel` *qui prend comme paramètre un* `Bicycle` *et la distance à parcourir. Utilisez-la avec deux vélos différents.*

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

```
val b = new Bicycle("Moulinette")
println(b)
```

Par défaut, les objets affichent 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 Trait (Tour)

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:
    var counter: Double = 0

    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`*.*

▶ *Modifiez* `travel` *pour accepter tout véhicule.*

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 `C`, `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: ...
```

3 Collections et programmation fonctionnelle (Book — Book)

Scala offre de nombreuses classes pour stocker des collections de données. On discutera de ces classes et de leurs opérations.

3.1 Introduction : tableaux (API)

Commençons avec un type usuel, les `Array` :

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

println("First element: ${a(1)}")
```

Il existe plusieurs possibilités pour déclarer un tableau :

```
val a = Array(1, 3, 5)
val b = Array[String]("good", "bad", "ugly")
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 le spécifie explicitement (`String`). Dans le troisième cas, on utilise explicitement le *companion object* afin de créer un tableau de 3 entiers, initialisé avec la valeur 0.

Deux choses à remarquer :

- `Array` (et les autres collections) sont un exemple d'une classe *polymorphe*, 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[_]`.
  Le fait de déclarer `a` comme `val` veut dire que `a` designera toujours le même objet pendant sa vie, même si le contenu de cet objet peut muter. Ainsi, une déclaration telle que `var f = 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` de `Int` de taille 3 :

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

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

3.2 Divers types de listes (Book)

Scala propose deux types de structures de données. Le contenu des classes peut être mutable (`scala.collection.mutable._`) ou immuable (`scala.collection.immutable._`).

Par exemple, `List` donne une liste `immutable`, c'est-à-dire que suite à la déclaration, toute tentative de modification échoue :

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

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

// Erreur:
a(2) = 5
```

`List` est réalisée par des listes chaînées. Du coup, `a(i)` n'est pas une opération en temps constant. Pour des accès aléatoires, `Vector` (parmi d'autres) est plus efficace.

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 Tableaux associatifs

Scala supporte aussi des tableaux associatifs qui stockent des paires (*clé*, *valeur*) ; on peut alors retrouver la valeur à partir de la clé. La classe `Map` est alors paramétrée par les types des clés et des valeurs. Il en existe des versions mutables et immuables.

Exemple : créer un `Map` (par défaut non-mutable) des `String` aux `Int` avec deux paires :

```
val m = Map("a" -> 3, "c" -> 5)
```

Par la suite `m("a")` donne 3.

Créer un tableau vide initialement mais mutable, en spécifiant les types paramétrés explicitement :

```
val n = scala.collection.mutable.Map[String, Int]()
```

Rajouter (ou mettre à jour) des valeurs, par exemple `n("a") = 5`.

3.4 Opérations (Book)

On est souvent amené à traverser les collections de données.

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

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

for (i <- a):
    println(i)
```

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

```
a.foreach(x => println(x))
a.foreach(println(_))
```

Il est à noter que la seconde forme n'est possible que pour les fonctions anonymes ayant un paramètre qui n'apparaît qu'une seule fois.

Des opérations permettent de générer de nouvelles listes :

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

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

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

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

Dans la deuxième ligne, le type de retour de `find` est un type `Option[T]`, que nous allons détailler dans la section 4.3, qui peut soit être `None`, soit `Some(x: T)` où `T` est le type des éléments de `a`.

Dans la dernière ligne, 0 est une valeur initiale ; la fonction est d'abord appliquée sur la valeur initiale et le premier élement, puis sur le résultat et le deuxième élément et ainsi de suite.

Avec `reduceLeft`, c'est le même principe, mais en omettant la valeur initiale. Du coup, `a.reduceLeft((a, b) => (a + b))` donne le même résultat dans ce cas. On remarque que `a.reduceLeft(_+_)` fonctionne aussi en plus court, et qu'il existe simplement `a.sum` (ainsi que `a.min`, `a.max`).

▶ *Faites la concaténation des éléments d'un `Array[String]`. Trouvez aussi la longueur de la chaîne la plus courte.*

Dernière remarque, un tableau multidimensionnel peut être ramené à un tableau simple par `flatten`.

3.5 *For comprehensions* (Tour)

Scala offre du sucre syntaxique pour une notation compacte des boucles imbriqués, les *for comprehensions*. Considérons la classe suivante qui représente une cellule avec coordonnées x, y qui contient une valeur v.

```
class Cell(val x: Int, val y: Int):
    // La valeur de la cellule n'est pas accessible depuis l'extérieur
    private var v: Int = 0

    // On autorise la lecture de v: val x = cell.v
    def v: Int = v

    // On autorise l'écriture de v: cell.v = x
    def v_=(n: Int) =
        v = n
```

La boucle suivante initialise une grille 8×8 de cellules :

```
var g = Array.ofDim[Cell](8, 8)

for (x <- 0 to 7; y <- 0 to 7) do
    g(x)(y) = Cell(x, y)
```

Une *for comprehension* peut aussi filtrer des valeurs :

```
for (x <- 0 to 7; y <- 0 to 7; if x - y == 3) do
    g(x)(y).v = 1
```

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

```
val cells = for (x <- 0 to 7; y <- 0 to 7) yield g(x)(y)
```

L'opération précédente peut maintenant être exprimée par l'une des lignes suivantes :

```
for (c <- cells) do
    if (c.x - c.y == 3) then
        c.v = 1

for (c <- cells; if c.x - c.y == 3) do
    c.v = 1

cells.filter(c => c.x - c.y == 3).foreach(_.v = 1)
```

▶ *Mettez* `v` *à* `x + y` *dans toutes les cellules de* `g`*. Calculez la somme de toutes les valeurs* `v` *dans* `g`*.*

4 *Objects*, *case class* et *pattern matching*

4.1 *Companion objects* (Book)

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(val radius: Double):
  def area: Double = Circle.calculateArea(radius)

object Circle:
  private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)

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

Il est à noter 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)

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(re + c.re, im + c.im)

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

println(d)
println(c == e)
```

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 leur identité référentielle ;
- Une méthode `toString` définie automatiquement (voir le résultat de `println`).

Du coup, 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 `==`).

▶ *Comparer avec le comportement de* `==` *et* `println` *quand* `Complex` *est une classe ordinaire.*

▶ *Programmer une classe abstraite* `Tree` *avec une méthode* `sum(): Int`*. Déclarer des sous-classes* `Node` *et* `Leaf` *de façon que par exemple* `Node(Node(Leaf(2), Leaf(3)), Leaf(5)).sum()` *donne la valeur* 10.

4.3 *Pattern matching* (Tour)

Tout comme en OCaml, il est possible d'écrire des *pattern matching*. Le *matching* est particulièrement aisé pour les *case class*. Avec la classe `Tree` de la section 4.2, il est possible d'écrire :

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

def sum(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, on peut écrire :

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

Dans chaque `case` d'un *pattern matching*, on peut ajouter des tests sur les valeurs avec la syntaxe :

```
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.*

▶ *Ajoutez le cas* `X` *parmi les expressions arithmétiques puis définissez une fonction qui évalue la dérivée d'une expression par rapport à* `X`*.*

5 Programmation polymorphe

Dans la section 3 on a vu des *classes polymorphes* telles que `Array[String]` ou `List[Int]`. Dans ces cas, `Array` et `List` sont des classes qui fournissent une fonctionnalité commune pour des objets de différents types. Dans cette section, on étudie quelques exemples où ces classes paramétrées pourront être utiles, et leur fonctionnement de base.

5.1 Polymorphisme (Book)

Disons qu'on a envie de déclarer une classe de graphes. Dans nos graphes chaque sommet possède un nom, et il y a un sommet appelé racine :

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

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

Jusqu'ici, tout marche bien. Maintenant, supposons qu'on a envie d'utiliser cette classe dans de multiples contextes : par exemple dans un cas, les sommets ne portent que des noms, dans d'autres ils sont équipés d'un poids, dans encore un autre cas il s'agit d'un arbre généalogique où les sommets portent des informations sur des personnes (nom, prénom, date de naissance, ...).

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

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

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

`Graph` accepte bien `n` car `ExtNode` est une sous-classe de `Node`. Mais ne serait-il pas intéressant d'afficher le poids de la racine ? Ajoutons la ligne suivante :

```
println(g.root.weight)
```

Or, cela ne passe pas. En fait, le type inféré pour l'attribut `root` est logiquement `Node` car `root` possède ce type. Or, pour obtenir le poids de `root` il faut que Scala sache qu'il s'agit d'un `ExtNode`.

C'est à ce moment où l'utilité des classes polymorphes s'avère. On change la déclaration ainsi :

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

Ici, `N` est un type qui paramètre la classe `Graph` qui sera spécifié lorsqu'on instancie un graphe. D'ailleurs, `g.root` est de type `N`. Le code suivant marche alors sans problème car le type de `g` devient `Graph[ExtNode]` et `g.root` devient `ExtNode`.

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

En effet, dans cet exemple, Scala devine que `N` vaut `ExtNode` dans `Graph(n)` car `n` possède ce type.

5.2 Types bornés

Supposons que la classe `Graph` devrait fournir des fonctionnalités plus intéressantes, par exemple afficher le nom d'un sommet.

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

Or, ceci ne fonctionne plus car `root` est désormais du type `N`, et Scala ne connaît rien sur ce type ; en particulier Scala est incapable d'inférer que ce type possède un `name`.

La solution consiste à spécifier que `N` doit être une sous-classe de `Node`.

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

Inversement, l'opérateur `>:` spécifie une contrainte de super-classe. Prenons le code suivant :

```
class A

class B extends A:
    def mergeWith[T](x: T) =
        List(x, this)

val a = A()
val b = B()
List(a, b)
b.mergeWith(a)
```

On constate que le type inféré de `List(a, b)` est `List[A]` mais que `b.mergeWith(a)` donne `List[Any]`. Bien que ces deux listes sont composées des mêmes éléments : lorsque Scala compile la méthode `mergeWith`, l'ancêtre commun plus proche de `T` et `B` n'est que `Any`.

La modification suivante permet de résoudre le problème :

```
class B extends A:
    def mergeWith[T >: B](x: T) =
        List(x, this)
```

On exige alors que `T` soit une super-classe de `B` ce qui permet à Scala d'inférer le type `List[T]` pour `mergeWith`. D'ailleurs, cette contrainte n'empêche pas de passer des sous-classes de `B`, le résultat de `mergeWith` sera toutefois du type `List[B]`.

```
class C extends B

val c = C()
b.mergeWith(c)
c.mergeWith(c)
```

5.3 Bornes multiples

Revenons sur l'exemple d'un graphe. Désormais on le spécifie avec une liste de sommets et arêtes :

```
class Node(val name: String)
class Edge(val from: Node, val to: Node)
class ExtNode(val name: String, val weight: Int) extends Node(name)

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

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

val n1 = new ExtNode("n1", 5)
val n2 = new ExtNode("n2", 3)
val edge = new Edge(n1, n2)
val g = new 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 dernière ligne ; logiquement car `successors` travaille avec une liste de `Edge` qui ne relie que des `Node`.

Il convient alors de paramétrer la classe `Edge` elle aussi :

```
class Edge[N](val from: N, val to: N)

class Graph[N <: Node](nodes: List[N], edges: List[Edge[N]]):
    ...

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

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 de `n1`.

On définit alors une sous-classe d'arêtes :

```
class ExtEdge[N](from: N, val label: String, to: N) extends Edge(from, to)

val n1 = ExtNode("n1", 5)
val n2 = ExtNode("n2", 3)
val edge = ExtEdge(n1, "a", n2)
val g = Graph(List(n1, n2), List(edge))

g.outgoing(n1).foreach(e => println(e.label))
```

Mais bien sûr cela échoue car `outgoing` renvoie toujours `List[Edge]`.

La solution consiste alors à donner deux paramètres interdépendants à la classe `Graph` :

```
class Graph[N <: Node, E <: Edge[N]](nodes: List[N], edges: List[E])
```

5.4 Covariance et contravariance (Book — Tour)

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(","))
```

Or, cela échoue car `m2` est du type `Matrix[B]` considéré incompatible avec le `Matrix[A]` attendu par `rowsums`. Si on modifie la définition de la matrice ainsi :

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

alors `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`.

6 Outils d'aide au développement

6.1 Documentation : Scaladoc (Référence)

Lorsque l'on effectue un projet, il est essentiel que chaque participant *documente* les divers éléments de code qu'il produit (classes, méthodes, etc.) afin que les autres (ou lui-même) puissent les utiliser comme des « boîtes noires », sans devoir directement lire le code pour savoir comment il fonctionne.

Scaladoc (fourni avec Scala) est un système générant de la documentation (sous forme de pages HTML) à partir de commentaires spécifiques que le programmeur ajoute aux fichiers sources.

Devant chaque élément (classe, méthode, attribut, etc.) que l'on souhaite documenter, on place un bloc de commentaire débutant par `/**`, terminant par `*/` et contenant généralement un texte descriptif court, un texte descriptif plus long (qui peut néanmoins être omis) et d'autres informations introduites par des étiquettes, comme par exemple `@param` pour ajouter un descriptif lié à un paramètre d'un constructeur ou d'une méthode :

```
/**
 * Descriptif court de la classe ‘C‘.
 *
 * Descriptif long de la classe ‘C‘.
 * Celui-ci peut s'étaler sur plusieurs lignes et paragraphes.
 *
 * @author Informations concernant l'auteur.
 *
 * @constructor
 * Descriptif court du constructeur primaire de la classe ‘C‘.
 *
 * @param p Descriptif du paramètre de ce constructeur.
 */
class C(p: Int)
{
  /**
   * Descriptif court de l'attribut ‘a1‘.
   *
   * Descriptif long de l'attribut ‘a1‘.
   */
  var a1 = 0

  /** Descriptif court de l'attribut ‘a2‘. */
  val a2 = Array(293762, 89072398, 9872723, 2389723)

  /**
   * Descriptif court de la méthode ‘m‘.
   *
   * @param p Descriptif du paramètre ‘p‘ de la méthode ‘m‘.
   * @return Descriptif de ce qui est retourné par la méthode ‘m‘.
   */
  def m(p: Int): Int = p
}
```

Pour générer la documentation pour un ensemble de fichiers sources, il suffit alors de lancer la commande `scaladoc` sur ces fichiers, ce qui aura pour effet de produire une arborescence de pages de documentation en HTML affichables à l'aide d'un simple navigateur web (la racine de l'arborescence correspondant au fichier `index.html`).

Scaladoc propose de nombreuses autres étiquettes pour ajouter une multitude d'informations de types divers et supporte également le « *markup* » pour formatter le texte.

6.2 Formatage : Scalafmt (Site web)

Afin d'éviter des incohérence dans le style appliqué au code, *scalafmt* permet d'appliquer un style automatiquement et uniformément à l'ensemble d'un projet. `scalafmt` s'intègre à `sbt` ainsi qu'à plein d'IDE et permet également d'être utilisé en ligne de commande.

Pour utiliser la commande `scalafmt` depuis `sbt`, il suffit d'ajouter la ligne suivante dans le fichier `project/plugins.sbt` :

```
addSbtPlugin("org.scalameta" % "sbt-scalafmt" % "2.5.0")
```

6.3 Utilisation de pipeline

Afin de s'assurer que les modifications apportés soit formatté avec `scalafmt` et compile correctement, il peut être intéressant d'utiliser une *pipeline*.

Une *pipeline* est une suite d'opérations qui s'exécute automatiquement à chaque fois qu'un *commit* est envoyé.

Voici un exemple de configuration de *pipeline* pour le service `GitLab` :

```
image: "vlafeychine/scala-native-sbt"

variables:
  SBT_OPTS: "-Dsbt.global.base=sbt-cache/sbtboot
             -Dsbt.boot.directory=sbt-cache/boot
             -Dsbt.ivy.home=sbt-cache/ivy"

cache:
  untracked: true
  paths:
    - "sbt-cache/ivy/cache"
    - "sbt-cache/boot"
    - "sbt-cache/sbtboot"
    - "sbt-cache/target"

build:
  script:
    - sbt "scalafmtCheck; compile"
```

L'image utilisée est `vlafeychine/scala-native-sbt` qui contient les instructions suivantes :

```
FROM  sbtscala/scala-sbt:eclipse-temurin-17.0.5_8_1.8.2_3.2.1

RUN   apt-get -y update && \
      apt-get -y install clang libcsfml-dev openssh-client
```

7 Utilisation de la bibliothèque SFML pour Scala

SFML est une bibliothèque graphique écrite en C++. Afin de pouvoir l'utiliser en Scala, il va falloir utiliser un *binding* qui permet de faire le lien entre les deux langages.

De plus, Scala Native sera utilisé pour compiler le code Scala en un exécutable natif.

7.1 Installation de Scala Native (Site web)

Pour installer Scala Native, il suffit d'ajouter la ligne suivante dans le fichier `project/plugins.sbt` :

```
addSbtPlugin("org.scala-native" % "sbt-scala-native" % "0.4.9")
```

Ainsi que d'ajouter la ligne suivante dans le fichier `build.sbt` :

```
enablePlugins(ScalaNativePlugin)
```

Lors du développement, il est conseillé d'activer les paramètres suivants dans le fichier `build.sbt` :

```
import scala.scalanative.build.*

nativeConfig ~= {
    _.withIncrementalCompilation(true)
        .withLTO(LTO.none)
        .withMode(Mode.debug)
}
```

7.2 Installation du *binding* (GitHub)

Comme le *binding* n'est pas encore mature, il n'est pas disponible sur le dépôt `Maven Central`. Cependant, le projet est disponible sur le dépôt `GitHub Package`.

Pour pouvoir accéder au dépôt, il faut un token d'accès à mettre dans le fichier `~/.gitconfig` ou `~/.config/git/config` :

```
[github]
  token = <token>
```

Vous pouvez utiliser le token suivant `token = "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"`, ou en générer un en suivant les explications sur le projet.

Afin de récupérer le projet avec `sbt`, il faut utiliser le plugin `sbt-github-packages` en ajoutant la ligne suivante dans le fichier `project/plugins.sbt` :

```
addSbtPlugin("com.codecommit" % "sbt-github-packages" % "0.5.3")
```

Pour terminer, il suffit de configurer le dépôt dans le fichier `build.sbt` :

```
githubSuppressPublicationWarning := true
githubTokenSource := TokenSource.GitConfig("github.token")

resolvers += Resolver.githubPackages("lafeychine")
libraryDependencies += "io.github.lafeychine" %%% "scala-native-sfml" % "0.2.3"
```

Vous pouvez maintenant utiliser le *binding* dans votre projet :

```
import sfml.graphics.*
import sfml.window.*

@main def main =
    scala.util.Using.Manager { use =>
        val window = use(RenderWindow(VideoMode(1024, 768), "Hello world"))

        val texture = use(Texture())
        texture.loadFromFile("cat.png")

        val sprite = use(Sprite(texture))

        while window.isOpen() do
            for event <- window.pollEvent() do
                event match {
                    case _: Event.Closed => window.closeWindow()
                    case _               => ()
                }

            window.clear(Color.Black())

            window.draw(sprite)

            window.display()
    }
```

Pour plus d'informations et de tutoriels, les liens suivants peuvent être utile :

- Les tutoriels : https://lafeychine.github.io/scala-native-sfml/docs/index.html
- L'API : https://lafeychine.github.io/scala-native-sfml/index.html

À noter que le projet est encore en développement et que les fonctionnalités ne sont pas encore portées. Si vous rencontrez des problèmes, vous pouvez ouvrir une issue sur le dépôt ou me contacter directement sur Discord (v-lafeychine#9801).

7.3 Notes sur les ressources

La bibliothèque SFML manipule des ressources qui doivent être libérées au bon moment. Scala Native dispose d'un mécanisme de *garbage collector*, ainsi, les ressources ne sont pas libérées immédiatement.

Le *binding* hérite chaque ressource du trait `Resource` qui indique au développeur que la ressource doit être libérée, ce qui permet d'utiliser l'un des paradigmes suivants :

- Manipuler les ressources à travers le bloc `Using` qui libère les ressources englobés par la fonction `use` à la fin du bloc, comme montré dans l'exemple.
- Appeler explicitement la méthode `close` lorsque la ressource n'est plus utilisée.

Afin de détecter les oublis de libérations mémoires, une solution est d'utiliser le projet `LeakSanitizer` (requiert une version 12 ou plus du compilateur `clang`).

Il suffit d'ajouter les paramètres suivants dans le fichier `build.sbt` :

```
nativeConfig ~= {
    ...
        .withCompileOptions(CompileOption("-fsanitize=leak"))
        .withLinkingOptions(LinkingOption("-fsanitize=leak"))
}
```