Fil - Syntaxe

Kevin Traini

Table des matières

1 Introduction

Fil est un langage de programmation appartenant aux paradigmes fonctionnel, objet et événementiel. Dans la suite de ce document divers aspects syntaxique ainsi que des exemples sur ce langage seront donnés.

Ce langage est entièrement constitué d'expressions, c'est à dire que tout renvoie une valeur, les conditions, les boucles, les classes, ... Donc on peut mettre tout ce qu'on veut à droite d'une assignation.

2 Syntaxe du langage

2.1 Modules

Le langage est organisé en modules, chaque fichier est un module qui fait partie d'un package. Cette gestion permet d'éviter les collisions de noms entre plusieurs modules et package. De ce fait chaque fichier *.fil* commence toujours par la même ligne : `package <nom du package>`. Le nom du package représente le chemin d'accès au fichier. Par exemple si le fichier est dans le dossier *fr/fil/model*, alors le nom du package devra être : `fr.fil.model`.

Ce système de module permet donc de découper son code en plusieurs fichier et donc de mieux s'y retrouver. Il est alors possible d'importer un module dans un autre fichier via la ligne `import`. Il suffit alors de renseigner le nom du package du fichier ou directement le chemin vers le fichier. En reprenant l'exemple précédent avec le fichier *user.fil*, on a soit `import fr.fil.model.user`, soit `import "fr/fil/model/user.fil"`. Cet import va nous permettre donc d'accéder à ce qu'il y a dans le fichier *user.fil*.

A l'intérieur même de ce fichier, il est possible de choisir ce qui doit être visible à l'extérieur du module ou non. Par défaut rien n'est visible, pour rendre une fonction ou une classe visible, il faut rajouter le mot clé `export` devant la structure en question.

2.2 Types

Le langage possède 5 types primitif : *int*, *float*, *double*, *boolean* et *char*. Il est bien évidemment possible de rajouter des types via les classes. Par exemple, dans

la librairie standard il existe déjà une implémentation des chaînes de caractères qui se nomme *String*.

De plus, il est possible d'utiliser des pointeurs en rajoutant tout simplement une étoile * devant le nom du type.

2.3 Variables et constantes

Pour stocker en mémoire une valeur, un objet, un tableau, ... on peut déclarer des variables en utilisant la syntaxe : `var <nom de la variable> : <type>`. Pour initialiser cette variable, il suffit de rajouter `= <valeur>`. Il n'est pas obligatoire de renseigner le type de la variable, en effet le compilateur détectera tout seul le type de la valeur et saura que la variable est donc de ce type.

Pour déclarer une constante la syntaxe est quasiment la même, il faut juste remplacer `var` par `val`. Sauf que dans ce cas, il faut initialiser la constante en même temps que la déclaration.

2.4 Tableaux

Le type d'un tableau est tout simple : `<type>[]`. En soit un tableau est un pointeur, donc l'initialisation d'un tableau ressemble un peu à celle des objets. Par exemple pour un tableau d'entier on fait : `var tab = new int[20]`, et maintenant `tab` est un tableau de 20 entiers. Pour accéder à une case du tableau, il suffit ensuite de faire : `tab[<index>]`.

2.5 Fonctions

Une grande partie du langage est construite sur les fonctions, leurs déclarations sont assez simple : `fun <nom de la fonction>(<paramètres>): <type de retour>`. Comme pour les variables, le type de retour n'est pas obligatoire, il permet juste de lever les ambiguïtés. Les paramètres de la fonctions se déclarent avec leur nom et leur type de cette manière : `<nom du paramètre> : <type>`, il est également possible de rajouter une valeur par défaut au paramètre en ajoutant `= <valeur>`.

Pour le corps de la fonction il y a 3 possibilité : si la fonction ne fait que renvoyer directement une valeur (comme pour un getter de classe) on peut utiliser la syntaxe `= <valeur>`. Ou par exemple si la fonction ne fait qu'une seule

chose qui tient en une seule expression, alors on peut écrire `( <expression> )`. Enfin si on est dans aucun des cas précédents, il faut utiliser la syntaxe de base `{ <expression>... }`. Bien sûr on peut utiliser cette syntaxe dans tout les cas et ne jamais utiliser les 2 précédentes.

L'appel à une fonction se fait ensuite en renseignant son nom et ses paramètres dans l'ordre `<fonction>(<paramètres>)`. Les paramètres qui ont une valeur par défaut ne sont pas obligatoires, et il est également possible de renseigner les paramètres dans l'ordre que l'on veut à partir du moment où on les nomment : `<fonction>(<nom> = <valeur>, <nom> = <valeur>)`.

2.6 Lambda

Parfois il est lourd de créer une fonction qui ne vas servir qu'à un seul endroit, pour cela il est possible de créer un lambda (ou fonction fléchée). Le type d'un lambda s'écrit `(<type des paramètres>) -> <type de retour>`. Et le lambda en lui même s'écrit : `(<paramètres>) -> <corps>`. Le corps peut s'écrire exactement de la même manière que pour une fonction. Pour les paramètres il y a 2 cas : le type du lambda est connu et dans ce cas on peut mettre que le nom du paramètre, le type du lambda est inconnu et dans ce cas il faut renseigner le type de chaque paramètres de la même manière que pour une fonction.

L'appel à un lambda se fait ensuite exactement de la même manière que pour une fonction.

2.7 Classes

Le langage possède aussi une grande partie orientée objet. La création d'une classe peut se faire en une seule ligne : `class <nom de la classe>(<attributs>)`. Et voila, nous avons une classe, son constructeur, ses attributs et ses getters/setters. Par défaut une classe ne peut pas être héritée, pour modifier cela il faut rajouter le mot clé `open` devant la déclaration. Les attributs de la classe spécifié sur cette ligne, sont ceux pour le constructeur pas défaut. Mais il est tout à fait possible d'en rajouter dans le corps de la classe. Dans les 2 cas, la déclaration d'un attribut ce fait de cette manière : `<visibilité> var <nom> : <type>`. Comme pour les variables, il est possible de les déclarer en tant que constante avec `val`, et il est aussi possible d'initialiser l'attribut en rajoutant `= <valeur>`. Le champs *visibilité* au début va définir qui peut voir cet attribut :

— *public* → Tout le monde peut voir
— *internal* → Seul le module peut voir
— *protected* → Seul la classe et les classes filles peuvent voir
— *private* → Seule la classe peut voir

Pour chaque déclaration d'attribut, un getter et un setter sont automatiquement créés en fonction de comment l'attribut est déclaré : si c'est une constante il n'y aura pas de setter, et si l'attribut est *private* ou *protected* il n'y a pas de getter. Dans le cas où il est *internal* le getter (et setter) sont mis eux aussi en *internal*.

Une classe peut être déclaré abstraite (via le mot clé `abstract`, il n'y alors plus besoin de mettre le mot clé `open` pour l'ouvrir à l'héritage.

Une classe peut hériter d'une autre classe, cela se fait en rajoutant `: <nom de la classe mère>` juste avant l'accolade ouvrante. Si on veut réaliser des interfaces, il suffit alors de rajouter le nom de l'interface après la classe mère en séparant par une virgule. On peut en mettre autant qu'on veut.

Le corps de la classe se place entre accolades et contient toutes les déclarations d'attributs, de constructeurs, de méthodes.

Dans la première ligne de la classe, on a déjà défini le constructeur pas défaut, mais il ne fait qu'affecter une valeur à chaque attributs spécifié. Pour rajouter un comportement en plus à ce constructeur, il suffit de déclarer la méthode `constructor { }` et placer entre les accolades tout ce dont on a besoin. Il est également possible de rajouter d'autres constructeurs de cette manière `<nom de la classe>(<paramètres>) { }`. À l'intérieur d'un constructeur, il est tout à fait possible d'en appeler un autre en appelant `this(<paramètres>)`.

Vient ensuite la déclaration des méthodes : `<visibilité> fun <nom>(<paramètres>) : <type de retour>`. La méthode peut être définie abstraite (`abstract`) et ne pas avoir de corps, ou en avoir un et le reste fonctionne exactement de la même manière que pour les fonctions.

Pour créer un objet, il y a 2 cas, soit on créé un pointeur, soit on créé un simple objet. Pour l'objet simple il suffit de faire `<Nom de la classe>(<paramètres>)`, pour les pointeurs c'est pareil mais avec un `new` en plus.

Une fois l'objet créé, on peut accéder à ses attributs tout simplement en faisant `<objet>.<attribut>` pour le getter et `<objet>.<attribut> = <valeur>` pour le

setter. Et de même pour les méthodes : `<objet>.<fonction>(<paramètres>)`.

2.8 Interfaces

Tout langage objet qui se respecte se doit d'avoir la notion d'interface. En Fil, pour déclarer une interface on utilise `interface <nom>(<attributs>)`. Tout comme pour les classes, il est possible de lui rajouter des attributs et des méthodes (sans corps), mais pas des constructeurs. Il est également possible de lui faire réaliser d'autres interfaces.

2.9 Structures de contrôle

Il existe 2 structures de contrôles : `if` et `switch`. La première se construit très simplement : `if (<condition>)` suivi d'un corps entouré ou non d'accolades (s'il n'y a pas d'accolades on ne peut mettre qu'une seule expression) suivi par une répétition de `elif (<condition>)` suivi par un `else`.

La seconde s'écrit `switch (<condition>) { }`. Entre les accolades on peut écrire une succession de `case <valeur> -> { }` et un `default -> { }`.

2.10 Boucles

Il existe 2 types de boucle : `for` et `while`.

La boucle `while` s'écrit assez simplement `while (<condition>)`. Par contre il y a 2 manière d'écrire un `for` : via index ou via itérateur. La première est comme dans la plupart des langages : `for (var i = 0; i < N; i++)`, alors que la seconde ressemble plus au python : `for <elt> in <tableau>`.

2.11 Calcul

Pour les calculs, on a à disposition 7 types d'opérateur :

— *Les opérateurs arithmétiques* : $+$, $-$, $*$, $/$, $\%$, $++$, $--$
— *Les opérateur de comparaison* : $<$, $>$, $<=$, $>=$, $==$, $!=$
— *Les opérateurs binaires* : $<<$, $>>$, $\&$, $|$, $\hat{}$
— *Les opérateurs booléens* : $\&\&$, $||$, $!$

— *Les opérateurs sur les pointeurs* : ∗, &
— *L'opérateur sur les tableaux* : []
— *Les opérateurs de flux* : <<, >>

Tout ces opérateurs fonctionnent de base sur leurs types prédéfinis. Mais il est possible de les surcharger afin de les utiliser sur les objets (sauf les opérateurs sur les pointeurs). Pour cela il suffit de rajouter la méthode `public fun operator<opérateur>(a: <type>, b: <type>): <type>`.

3 Exemples

Maintenant voici quelques exemple de programme en Fil un peu plus complexes.

3.1 Factoriel

Un simple programme qui renvoie 7!

```
package exemple

import fil.ios.iostream

fun fact(n : int): int (
    if (n == 0)
        1
    else
        n * fact(n - 1)
)

fun main(argc : int, argv : char): int {
    sout << fact(7) << endl

    return 0
}
```

3.2 Zoo

Un programme qui simule un zoo avec des animaux

```
// zoo.fil
package exemple.zoo
```

```
import String from fil.types
import ArrayList from fil.containers
import fil.ios.iostream

export abstract class Animal(public val nom : String,
                             public val cri : String) {
    public fun crier() {
        sout << nom << " va crier ! " << cri << endl
    }
}

class Zoo() {
    private val animaux : ArrayList<Animal*> = ArrayList<Animal*>()

    public fun ajout(a : Animal*) (
        animaux.add(a)
    )

    public fun crier() {
        sout << "Tout les animaux du zoo vont crier !" << endl
        for a in animaux
            a.crier()
    }
}

// main.fil
package exemple.zoo

import exemple.zoo.zoo

class Canard(nom : String): Animal {
    constructor {
        super(nom, "Coin coin !")
    }
}

class Singe(nom : String): Animal {
    constructor {
        super(nom, "Ouhouhouh !")
    }
```

```
}

fun main(argc : int, argv : char**) {
    val zoo = Zoo()
    zoo.ajout(new Canard("Jean"))
    zoo.ajout(new Singe("Eude"))

    zoo.crier()

    return 0
}
```