Chapitre 1

Introduction

1.1. Pourquoi un autre langage ? Pourquoi Scala ?

Il existe suffisamment de langages de programmation. Les développements logiciels actuels peuvent très bien se faire en utilisant des langages qui ont fait leur preuve (Java, C++, etc.). Et donc, pourquoi prendre le temps d'apprendre un autre formalisme ? Est-ce que le temps et l'effort nécessaires en valent le coup ? Mais plus encore, pourquoi Scala plutôt qu'un autre langage ?

Qu'est-ce qu'un bon langage de programmation ?

Probablement, le meilleur langage c'est avant tout celui que vous maîtrisez le mieux. Celui qui ne vous limite pas en terme d'expressivité et qui vous permet d'écrire moins de code, d'être exécuté rapidement (ce qui est plus une réalité d'implémentation que de langage propre). Le meilleur langage doit vous permettre de réutiliser du code déjà existant. Simple à apprendre (donc qui ne s'écarte pas trop de ce que vous connaissez déjà). Il ne doit pas vous imposer une manière de voir mais s'adapter à la vôtre.

De nombreux langages de programmation ont vu le jour depuis le début de l'informatique : chaque langage apporte sa propre manière de formuler un problème et sa solution.

Le plus rapide est probablement le langage machine, mais rapide reste subjectif. L'exécution est sans doute optimale en temps mais le développement est délicat, dépendant de la machine d'exécution et la mise au point peut être longue et difficile. Les connaissances sur l'implémentation interviennent de manière importante dans les choix algorithmiques.

D'autres langages (de plus haut niveau) masquent cette dépendance avec la machine et permettent des abstractions qui offrent une expressivité plus concise, au prix quelques fois d'une phase de compilation longue ou d'une exécution moins rapide. La gestion par exemple d'une mémoire virtuelle et

d'inférences de types ; offre des garanties au développement, que ce soit au moment de la compilation ou de l'exécution.

Le meilleur langage est bien un compromis entre la difficulté de l'apprendre, entre la sécurité d'exécution ou l'expressivité offerte du langage, entre les abstractions possibles et les réalités techniques d'une implémentation, entre votre problème à résoudre, votre solution à implémenter et le formalisme du langage (Impératif ? Fonctionnel ? Déclaratif ?)

De manière générale, un logiciel est formé de composants qui n'ont pas la même finalité et n'implémentent pas les solutions aux mêmes problèmes. Certaines solutions s'expriment aisément de manière fonctionnelle, d'autres avec un formalisme impératif, d'autres encore s'expriment et s'organisent en suivant un modèle objet, etc.

Scala est un langage de développement multi-paradigme qui s'inspire de nombreux autres langages. Il apporte les possibilités de Java (et de ses librairies) et vous offre l'ouverture à la programmation fonctionnelle dans un cadre orienté objet cohérent. Son implémentation courante utilise la machine virtuelle Java et son apprentissage peut se faire de manière progressive. Vous choisissez les implémentations (Collection séquentielle ? Accélération et parallélisme ?) et la syntaxe est proche de celle de Java. Les possibilités fonctionnelles associées à des mécanismes d'inférence de type vous permettent de simplifier et d'accélérer le cycle de développement.

1.2. Contexte

La création du langage de programmation Java a ouvert des possibilités importantes en développement applicatifs Web et Internet. Conçu pour exploiter une machine virtuelle (*Java Virtual Machine* ou JVM) capable d'exécuter du code sur quasiment toutes les architectures d'ordinateurs, Java propose un système de ramasse-miettes (ou *garbage collector*) capable de décharger le développeur de la majorité des problèmes de gestion mémoire. Le succès de Java est certainement autant lié à ces caractéristiques qu'à ses librairies ou à sa syntaxe relativement proche de C++.

Scala est un langage de programmation développé à l'origine par Martin Odersky (Ecole Polytechnique de Lausanne ou EPFL). S'affranchissant du cycle de développement de Java, M.Oderksy et son équipe imaginent et implémentent un langage de programmation s'inspirant de plusieurs paradigmes

développés dans des langages importants comme Eiffel, Erlang, Haskell, Java, Lisp, Pizza, ML, OCaml, Scheme, Oz, Smalltalk. Scala est un langage interprété par la machine virtuelle Java. Il offre des possibilités importantes d'interopérabilité avec Java. Il est donc possible de réaliser des projets complets dans lesquels du code Java et du code Scala interagissent sur la même plateforme. Scala peut ainsi exploiter l'importante librairie Java.

Quelques fois, une solution algorithmique peut s'exprimer plus simplement dans un langage de programmation plutôt qu'un autre. Le langage Scala propose d'associer de manière cohérente programmation à objets et programmation fonctionnelle. Différentes manières d'exprimer différents types de résolution de problèmes peuvent ainsi cohabiter au sein d'un programme Scala.

Le nom Scala vient de l'italien *scala* ou escalier, mais également de la combinaison des termes anglais *scalable* (capable de passer à l'échelle ou de s'adapter à la taille du problème à résoudre) et *language*. Ce langage peut être utilisé pour réaliser des scripts rapidement (l'aspect fonctionnel permettant d'écrire du code concis), du prototypage (l'utilisation des librairies Java permet d'utiliser du code existant) ou pour la réalisation de projets importants.

L'idée du *passage à l'échelle* ou du coté *adaptation* du langage se trouve également dans son expressivité. Scala offre de nombreuses librairies en plus de celles de Java. De plus, et c'est sans doute là le point le plus important, Scala vous donne la possibilité de créer vous même vos propres structures de contrôle, si vous en avez besoin.

La philosophie générale qui sous-tend certains choix de *design* de Scala est de permettre au programmeur de pouvoir fabriquer son langage et de choisir une syntaxe adaptée aux problèmes. Les outils fournis permettent de sculpter les formes syntaxiques et de s'approcher de manières de faire qui ressemblent à des prédéfinis dans d'autres langages. Scala possède une granularité syntaxique plus fine que dans ces autres langages et des contractions syntaxiques qui simplifient l'écriture du code. C'est sans doute déroutant à première vue et cela exige une certaine prudence. Rien n'empêche la réalisation de dialectes de Scala lisibles de vous seul.

Bien maîtrisée, cette caractéristique assez unique peut également donner l'illusion d'une extension du langage. Par exemple, une librairie rajoute des classes de communication acteurs d'une manière suffisamment naturelle dans le langage pour donner l'illusion de prédéfinis du langage (comme nous le verrons dans le chapitre 13 de ce livre).

Vous n'avez pas besoin d'attendre une nouvelle version du compilateur

pour rajouter une capacité d'expression nouvelle à votre langage. Vous avez les outils pour le faire vous-même.

Vous pouvez utiliser les cotés fonctionnels de Scala pour simplifier ou rendre plus génériques certaines parties de vos applications, ou bien utiliser un style impératif, profiter au besoin des mécanismes de reconnaissance et filtrage de motifs (*pattern matching*) pour simplifier l'analyse de message ou encore fabriquer votre propre version de contrôle de bloc pour simplifier l'écriture de votre code.

Ce langage a plusieurs niveaux. Il est, pour cette raison, délicat à maîtriser. Un développeur de librairie n'a en effet pas les mêmes besoins d'expressivité qu'un développeur d'applicatifs qui souhaite réutiliser du code Java.

Les langages fonctionnels (comme Lisp ou Caml) ne sont pas toujours aussi populaires que les langages impératifs (Java, C, C++). Les développeurs n'envisagent pas toujours de manière naturelle les solutions fonctionnelles à un problème. Scala vous donne la possibilité de commencer avec votre manière d'appréhender un problème, puis, au fur et à mesure de l'apprentissage du langage, de découvrir d'autres manières de faire, à en reconnaître les avantages et à les adopter (ou pas).

Scala ne propose pas énormément de concepts nouveaux. Mais la manière de faire est cohérente et souvent élégante sinon pratique et pragmatique. L'utilisation de `traits`, les possibilités d'extension d'expression, les aspects fonctionnels permettent souvent de faire plus en écrivant moins de code mais de plus grande qualité (fiabilité et performance).

De nombreuses entreprises (*Twitter*, *LinkedIn*, *Xerox*, *Sony*, *Siemens, EDF*) ont adopté Scala pour ces raisons.

1.3. Les caractéristiques de Scala

1.3.1. Un langage orienté objet

Chaque valeur manipulée par Scala est une instance (donc un objet). Chaque type est représenté par une classe. Les comportements des objets sont décrits par des `trait`*s*. Les mécanismes d'héritage et de polymorphisme de Java sont présents en Scala. L'héritage multiple en tant que tel n'est pas possible mais un mécanisme de composition basé sur des *mixins* est envisageable.

Le langage possède un mécanisme de création de classes singletons (ou objets compagnons) utilisé pour décrire les méthodes de classe de Java.

1.3.2. Un langage fonctionnel

En Scala, les fonctions peuvent également être représentées par des instances de classe. Il est donc possible de passer des fonctions en argument d'autres fonctions (fonctions d'ordre supérieur). Les fonctions peuvent être imbriquées, composées. Il est possible de créer de nouvelles fonctions en renseignant partiellement les champs paramêtres. Scala permet la *curryfication* et la *clôture* ou fermeture fonctionnelle. Scala reconnaît et permet de définir des types algébriques à l'aide d'un mécanisme de filtrage par motif (*pattern matching*) et d'une définition étendue de classes (`case classes`). Ces caractéristiques associées au possibilités d'extension du langage permettent de créer des idiomes dédiés (comme par exemple des tournures syntaxiques pour gérer XML ou un analyseur grammatical).

1.3.3. Un langage statiquement (et fortement) typé

Scala possède un ensemble complet de définition de types et un mécanisme d'inférence qui permet éventuellement d'omettre un type quand l'expression utilisée est explicite. Associer un type à chaque expression Scala est une assurance supplémentaire de détecter de nombreuses erreurs au moment de la compilation (plutôt que pendant l'exécution). Le système d'inférence de types avancé de Scala permet non seulement de vérifier les choix des types indiqués par le développeur, mais aussi de compléter de manière automatique le code par des déclarations implicites (pour les types les plus courants et adaptés au problème à résoudre). Les classes génériques, annotations de variance et covariance, les types composés permettent d'exprimer des contraintes complexes sur les possibilités d'héritage ou de passage de paramètres.

1.3.4. Un langage extensible

Scala permet de définir des types algébriques et contient les constructions syntaxiques qui les intégrent syntaxiquement dans le langage. La majorité des accès (infixes, préfixes, etc.) possède une syntaxe de déclaration de méthode. Il est par exemple possible de définir le comportement de l'utilisation d'un appel syntaxique de fonction (représenté par des parenthèses qui encadrent des paramètres/arguments) par la méthode `apply`. Le passage de fonctions anonymes en arguments de fonctions (associé à la *curryfication*) permet également de définir des nouvelles structures de contrôle. Cet ensemble de caractéristiques fait de Scala un langage bien adapté à la définition de dialectes dédiés.

1.4. Présentation de l'ouvrage

Cet ouvrage se propose d'accompagner un développeur familier avec Java dans une découverte graduelle de Scala, pas à pas. L'objectif est d'aider le lecteur à monter suffisamment de marches sur l'escalier Scala pour lui permettre de continuer seul, à l'aide des ressources Internet ou des livres de référence proposés en bibliographie.

Scala est un langage multi-paradigme très riche en constructions et exceptions et l'ascension ne peut se faire d'une traite. Pour cette raison et pour éviter dans la mesure du possible la nécessité de retours en arrière dans la lecture, de nombreux concepts sont présentés plusieurs fois de manières différentes dans des contextes différents.

Certains rappels sont proposés en cours de route, au risque d'alourdir un peu le propos. Si les ouvrages d'apprentissage pouvaient être parcourus d'une manière linéaire sans pause, les notions n'auraient besoin d'être présentées qu'une seule et unique fois. Dans la majorité des cas, segmenter l'apprentissage avec des délais d'une journée à une semaine, revient à imposer au lecteur de constants retours en arrière (*c'était quoi déjà ?*). Cet ouvrage assume une redondance d'explications pour éviter au maximum les conséquences d'une interruption de lecture. Il s'agit là d'un choix délibéré que vous ne manquerez pas de noter.

Il est difficile d'éviter les références en avant. Hiérarchiser l'apprentissage d'un langage comme Scala est délicat car la cohérence du langage donne lieu à une interdépendance marquée entre les structures des données et les aspects fonctionnels.

Une fonction peut être quelques fois considérée comme une méthode et souvent exploitée comme une instance. Les commentaires qui décrivent les exemples proposent souvent des explications simplifiées mais suffisantes dans un premier temps. Des précisions sont apportées progressivement au fil des chapitres dédiés. Quelques choix de conceptions du langage (ou limitations) sont liés à l'implémentation du langage et ne présentent un interêt que pour les lecteurs déjà familiarisés avec le langage.

Ce livre n'a pas pour objectif de faire de vous un spécialiste Scala : certaines librairies évoluent encore, d'autres fonctionnalités sont d'usages extrèmement spécifiques (les continuations par exemple) et la référence définitive se trouve toujours dans la documentation en ligne de la dernière version. Vous ne trouverez pas d'exemples de programme complet et de projet. Le but de cet ouvrage est de décortiquer au maximum les notions importantes pour vous permettre

ensuite de profiter d'une bibliographie spécialisée proposée à la fin de l'ouvrage. Une attention particulière a été apportée à la progression pour vous aider à prendre de l'assurance pour la marche suivante. L'auteur a essayé de mettre dans cet ouvrage les réponses aux questions qu'il s'est posé (et que d'autres se posent) en cours de son propre apprentissage et la progression dont il aurait eu besoin pour aller plus vite. Le contenu et la manière de présenter sont donc absolument subjectifs. Un spécialiste de la programmation fonctionnel ou un expert en composants présenterait Scala d'une autre manière : ici, l'accent est mis sur l'exemple, le résultat et une explication pragmatique.

Les premiers chapitres illustrent l'utilisation de Scala comme langage de script, puis les analogies avec Java sont proposées et enfin les spécificités de Scala sont présentées au fil des chapitres en proposant d'abord les aspects subjectivement les plus immédiats en développement. Les derniers chapitres concernent des spécificités du langage réservés à des problèmes moins courants.

Le chapitre 2 (L'interpréteur Scala) est destiné au lecteur qui souhaite rapidement essayer du code Scala. Des exemples sont proposés et commentés très rapidement de manière volontairement superficielle. L'idée est de présenter quelques originalités de Scala avec quelques exemples, mais également de familiariser le lecteur avec l'interpréteur de script (qui peut servir de bloc-note de test ensuite).

Le chapitre 3 (Scala essentiel) présente un sous-ensemble minimal permettant de mener un projet simple en Scala.

Le chapitre 4 (Les tableaux) est une première introduction aux collections en utilisant l'exemple des tableaux. Ce chapitre présente les différentes syntaxes et quelques méthodes essentielles.

Le chapitres 5 (Les principales collections) présente les collections les plus usuelles.

Le chapitre 6 (Les collections Scala en détail) introduit la hiérarchie d'héritage des collections et explicite la majorité des méthodes utiles et pratiques.

Le chapitre 7 concerne les aspects programmation objet et présente les notions de classe, instance et d'héritage.

Le chapitre 8 (*Pattern matching*) illustre les capacités de filtrage de motifs de Scala.

Le chapitre 9 (Les fonctions) introduit la définition des fonctions et présente les mécanismes de composition et de passage en paramètre.

Le chapitre 10 (Les types paramétriques) décrit les types paramétriques Scala.

Le chapitre 11 XML présente l'intégration d'XML dans la syntaxe Scala.

Le chapitre 12 (Future et Promise) est une introduction à la librairie `scala.concurrent` qui offre des possibilités importantes de contrôle d'évaluation concurrente d'expressions.

Le chapitre 13 (Acteurs et Akka) présente les principes de base des acteurs Akka.

Le chapitre 14 (Les continuations délimitées) est une introduction aux continuations. Leur utilisation permet de modifier la séquence d'évaluation des expressions et de programmer des structures de contrôle dédiées.

Le chapitre 15 (Conclusion) propose une bibliographie à consulter pour continuer votre apprentissage du langage.

Chaque chapitre commence éventuellement par les items :

Présentation : introduction générale du chapitre et remise en contexte des notions présentées.

Prérequis : logiciels nécessaires et connaissances souhaitées.

Contenu : description rapide du chapitre sous la forme d'une série de mots-clés, de périphrases ou de noms de section.