Pattern Matching

Speaker : Rémi Forax (Maitre de conférence / Développeur OpenJDK / Expert JCP)
Format : Conférence
Date : 7 avril 2017

Dans OpenJDK, il existe le projet **Valhalla** permettant d'ajouter 2 fonctionnalités : value types et generics

Qu'est-ce qui ne va pas avec les objets aujourd'hui ?
Pour chaque objet, on a systématiquement un en-tête de 64 bits (pour le synchronized, le hashcode, le GC).
Le problème commence lorsqu'on a des tableaux d'objets. Cela ajoute beaucoup de références et diminue le GC.
Le modèle des objets ne colle plus à l'architecture des applications actuelles.

Value types

Syntaxes inventées dans les slides

Les value types doivent être des struct du language C.
Le code ressemble à une classe mais se comporte comme une valeur à l'exécution. Pas d'identité. Pas d'adresse mémoire. Doivent être immutables. En Java, on ne peut pas récupérer l'adresse de quelque chose sur la pile. C'est ce qui permet à Java d'être plus performant que le C.

L'introduction de cette feature va impacter les autres fonctionnalités.
Un ArrayList<complex> serait différent d'un ArrayList<Object> car il faudrait spécialiser le code pour ne plus avoir de références vers des objets.
Le bytecode n'est pas prévu pour ajouter des Value types. La spécialisation du code sera fait par le JIT.

Projet Panama

On veut que Java puisse à parler avec du C sans JNI. Relativement simple : demander au JIT de générer le code.

Autre évolution : laisser au développeur la possibilité d'accéder aux nouvelles instructions SIMD/AVX. Nécessité de définir un type primitif sur 128, 256 ou 512 bits. Le nombre de changement est limité au JDK.

Plan :

- Java 10
    - Panama
    - Value types dans la JVM mais pas dans Java : permet aux autres langages de la plateforme de faire des retours

- Java 11
    - Valahla complet

Projet Amber

Ajouter un certain nombre de fonctionnalités

Local Variable Type Inference : le compilateur devra trouver le type de la variable locale. Exemple : var a = 3 ; Ne supportera pas les lambda et les références method/

Enhanced Enum : possibilité de paramétrer des énums. Pose soucis car chaque variable de l'énumération peut avoir un type différent.

Lambda leftovers :

- Réutilisation du caractère _ (déprécié en Java 9 puis interdit en Java 10) pour les unused parameters
- Réutilisation du nom de paramètre
- Meilleure inférence de type

La grosse feature : Pattern Matching

Fonctionnalité demandée depuis 1997 (via un switch sur les types)
Nécessité d'introduire une interface fermée (hiéarchie fermée) : liste des sous-types prédéfinies. Permet de retirer le default dans le switch.
Introduction d'une Data Class :
data Point(int x, int y) ;
Nécessaire pour les Value types.
Le pattern matching casse l'encapsumation. C'est l'une des raisons conceptuelles pour laquelle le pattern matching n'a pas été introduit en Java.
Introduction d'un déconstructeur pour sortir les valeurs d'une classe data (équivalent du *unapply* de Scala).
Implémentation possible à base de if isntanceof … else => pas génial
On va plutôt utiliser InvokedDynamic pour générer les if else à mesure que les valeurs arrivent à l'exécution. Le code va s'adapter dynamiquement aux variables qui vont être passées. Utilisation d'un automate qui n'utilise pas de instanceof. Au pire des cas, ce ne sera pas plus lent que Scala. Lorsqu'il y'a beaucoup de if (ex : 20 aine de classes), on peut les remplacer par une table de hachage au détriment du inlining => quel que soit le nombre de type à matcher, le temps de résolution est constant.
Le unapply de Scala alloue 2 objets => très lent.