Valhalla, to the hell and back

Date : 21 avril 2022
Speaker : Rémi Forax (enseignant chercheur à l'Université Gustave Eiffel)
Format : conférence (45mn)

Introduction

Disclaimer : le **projet Valhalla** est encore en **draft** et peut donc encore bouger d'ici sa release finale. Ce nom emprunté aux Vikings a été choisi par **John Rose**, architecte de la JVM et leader du projet.
Ce projet a commencé en 2014. Le 4ième prototype est sorti. Les prototypes sortent étrangement en août (Driven Conference Development).
Les machines d'aujourd'hui ne fonctionnent plus tout à fait comme celles des années 90.
L'**objectif du projet Valhalla** est d'essayer de **réaligner Java avec les CPU d'aujourd'hui**.

Objectifs :

- **Avoir gratuitement de l'abstraction** (pas d'allocation pour les objets intermédiaires)
- **Améliorer la densité d'information** : Java utilise des références qui nécessitent plusieurs sauts. On veut désormais essayer de manipuler les valeurs directement.

Abstraction for Free

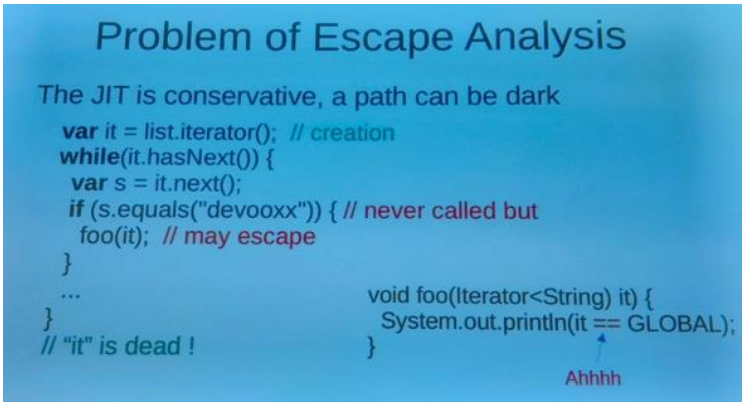
Les objets temporaires devraient être gratuits :

- Optional<T> : possible qu'on paie le cout de l'objet qui encapsule la valeur. Utiliser un Optional sur une propriété pose problème car il n'est pas nécessairement sur la pile (certains IDE le signale).
- True builder : logger.warning().on(console).message("Hello Devoxx")

Tous les petits objets : Month, Complex, LocalDate …

Problem of the Escape Analysis

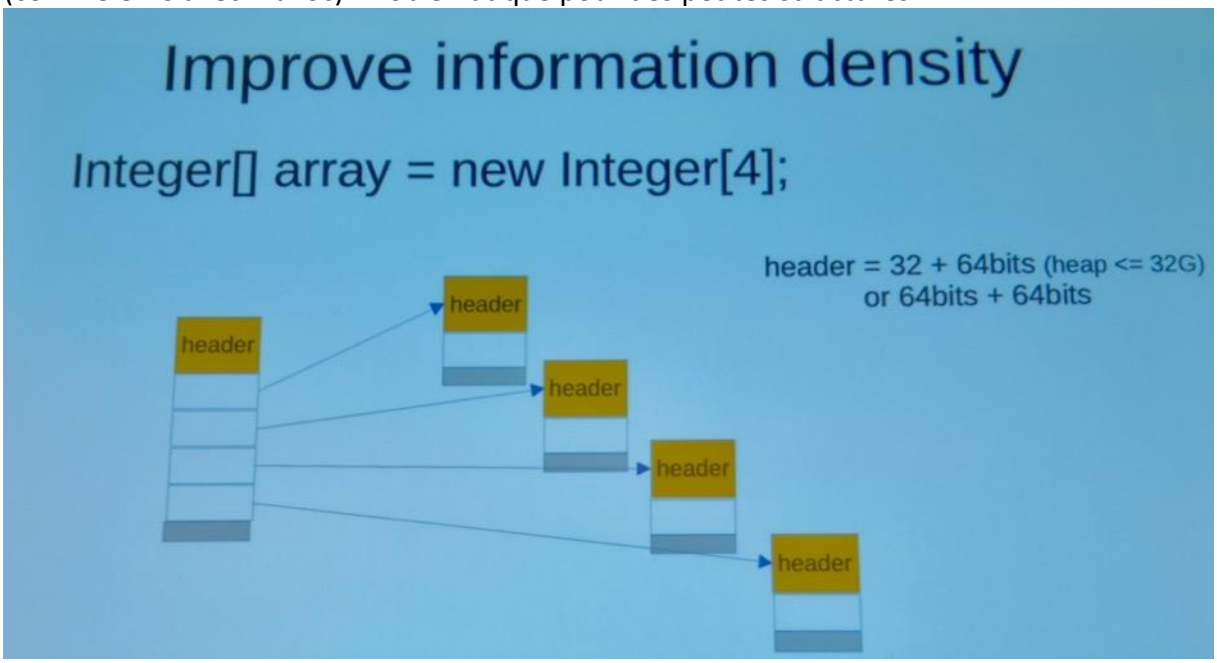
La JVM dispose déjà de l'algo **Escape Analysis**. Le JIT regarde la création d'un objet et son dernier usage. Il sait donc quand le détruire. C'est ce qui se passer lors de l'utilisation d'un itérateur.



Pour la JVM, le code non exécuté n'est pas vu. Dans l'exemple ci-dessus, cet algo ne fonctionne pas car ***it*** est utilisé pour tester une égalité de référence.

Améliorer la densité d'information

Exemple d'un tableau de 4 Integer : nécessite beaucoup d'en-têtes. Un objet vide prend de la mémoire. Sur certaines architectures, les doubles doivent être alignés en mémoire (comme en C avec malloc). Problématique pour des petites structures.



L'idée qui guide Valhalla : on utilise des classes mais derrière ce sont des primitives qui sont utilisées, ceci en conservant la compatibilité (référence Python 2 vs Python 3)

On ne manipule plus les objets par leur référence et on utilise directement leur valeur.
Limitations : listes chainées, graphes, arbres …

Questions auxquelles les concepteurs de Valhalla ont dû répondre :

- Ces objets sont-ils mutables ?
- Est-ce que ces champs héritent de Object
- Peuvent-ils être null ?
- Quelle est leur valeur par défaut ?
- Est-ce qu'ils sont « tearable » ? Exemple du long en 2 écritures sur archi 64 bits.

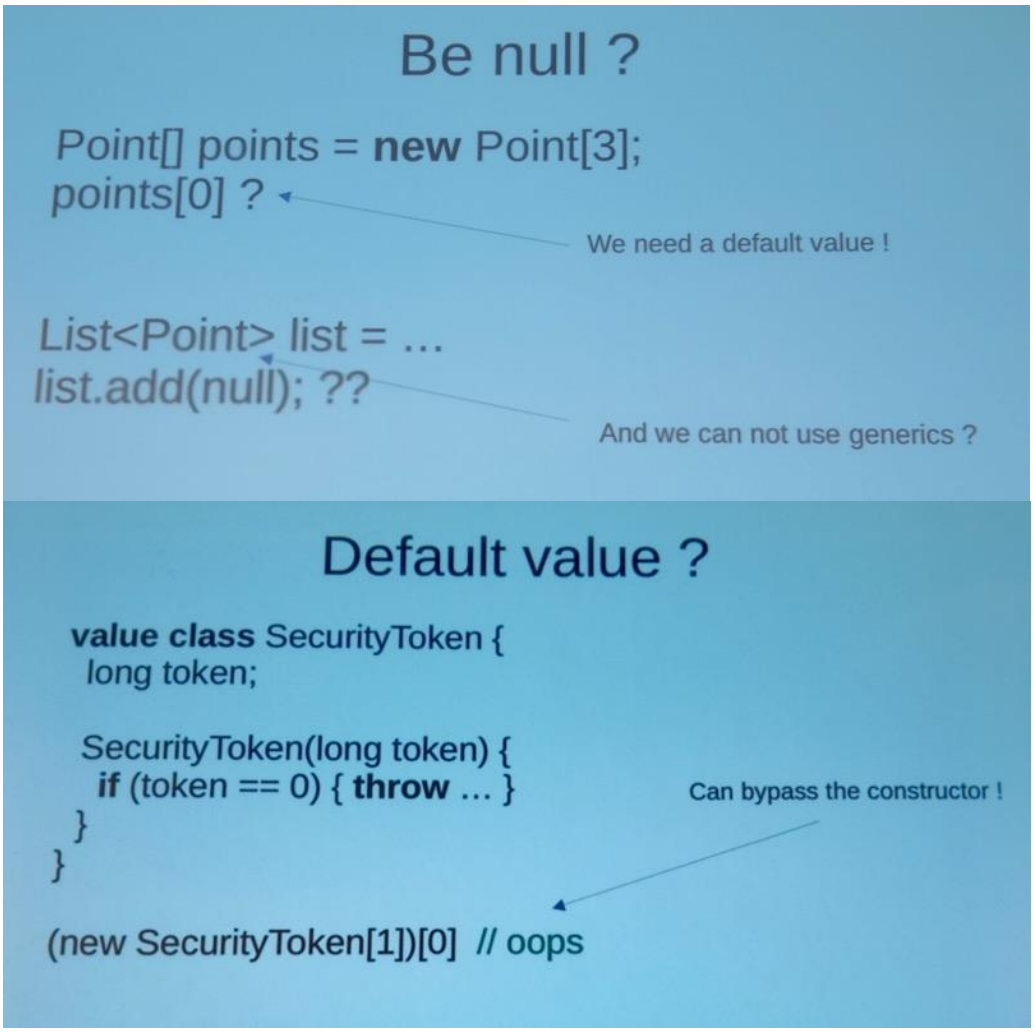
Mutable ?

Non, pas possible. On ne va pas changer la moitié de la valeur d'un Point(x,y)

Be null ?

Une liste de Point nécessite de pouvoir faire un add de null (API Collection)
Nécessité de réconcilier ce point.

Les valeurs par défaut passent outre le constructeur. Problématique déjà rencontrée lors de la désérialisation.



Inherit from Objet ?

Que veut dire == sans pointeur ?
Créer une *WeakRef()* sans pointeur ?
o.getClass() => nécessite le header

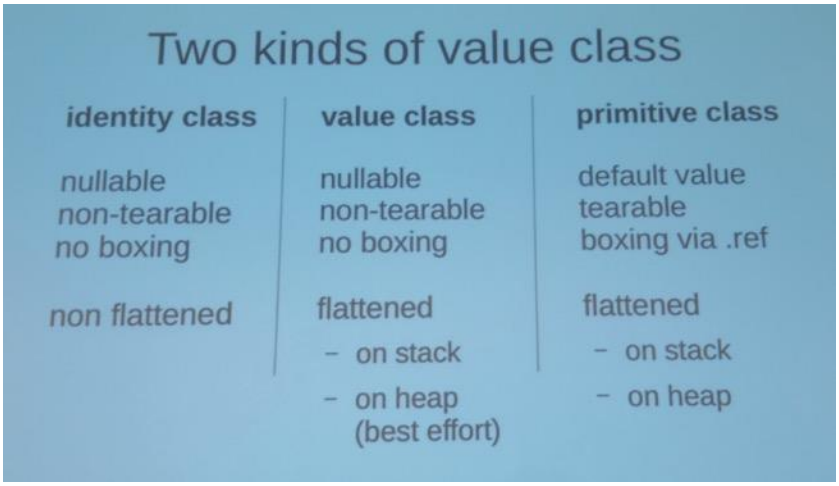
La solution

On triche en créant **2 types de value class** en plus de l'Identity Class

1. Identity class
    a. Les **classes déjà existantes** qu'on manipule par référence
    b. La valeur null est pratique
2. Value class
    a. Reste nullable
    b. Pas de boxing

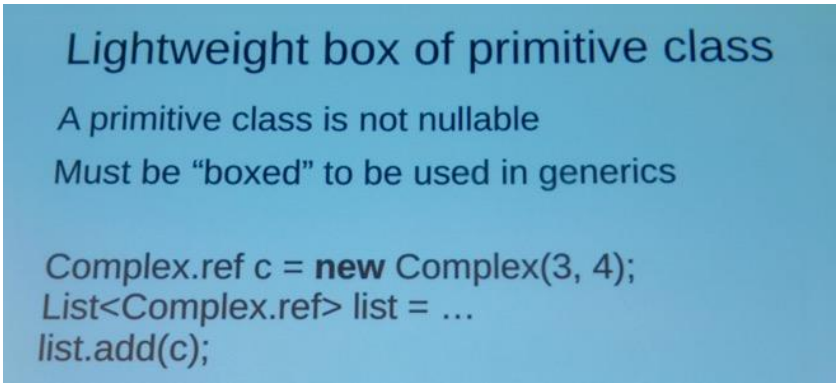
c. Sur la pile on ne paie pas le cout d'une allocation. Sur le tas, çà dépendra

3. Primitive class :
    a. Pas de valeur par défaut
    b. Tearable
    c. Boxing via ref



Primitive Class

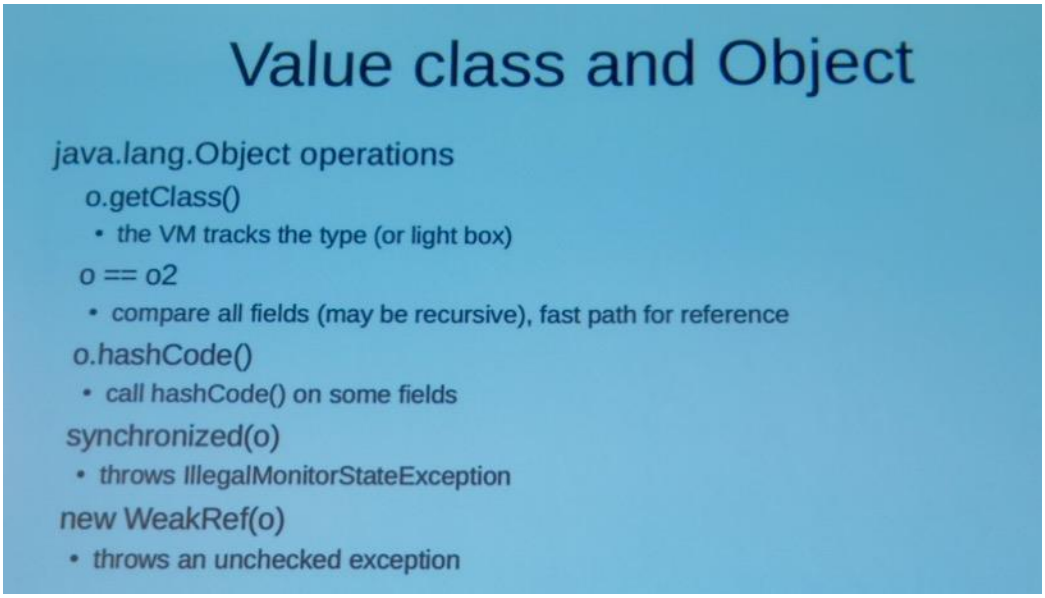
Ce type viole l'encapsulation. Le compilateur va en ajouter un pour le développeur.
L'utilisation des Primite Class avec les génériques passe par des ***.ref***
Screenshot. Contrairement au vraie boxing la VM peut remplacer Complex.ref par Complex.
Moins couteux.



Les classes existantes pourront être vue comme des value class. Exemple du Optional, LocalDate … fait magiquement par le JVM pour les dévs.
Conditions nécessaires : classe final, champs finaux, constructeur non public, pas de synchronized, pas de référence faible (WeakRef), et ne pas utiliser le ==

Value class and Object

On veut que la classe Object reste la classe parent de tous les objets.



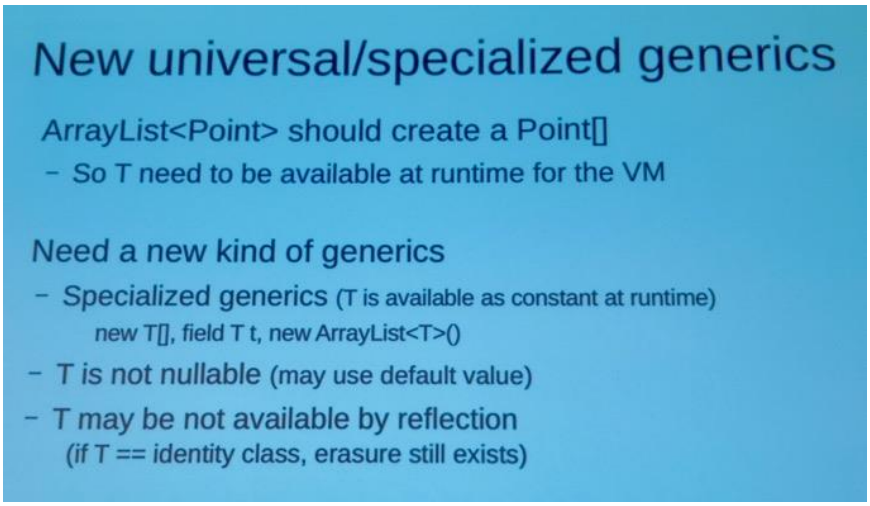
Rémi rappelle que l'opérateur == actuel n'est pas si rapide que cela.
Avec les Value Class, l'opérateur == nécessitera qu'on compare 2 les champs, peut être récursif.

Lorsqu'on crée une classe non mutable, à l'intérieur d'un constructeur, on … mute des champs. Le compilateur va recréer une méthode statique factory.
L'appel de méthode avec ***this*** en paramètre dans un constructeur ne sera plus autorisé.

L'héritage ne sera pas supporté, de même que les enum, record et les lambda (depuis Java 5 les concepteurs du langage évitent d'utiliser l'héritage). L'utilisation d'interface perdure.

L'appel à ***new Integer()*** sera déprécié. La JVM réécrira ce code.

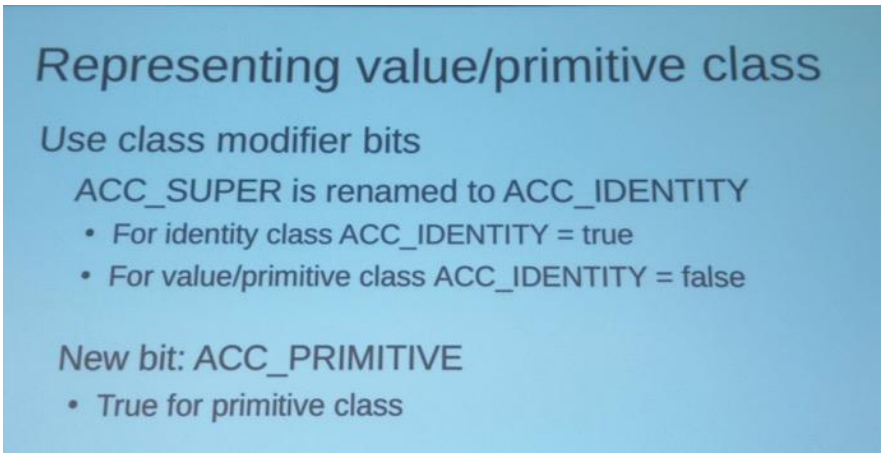
New universal/specialized generics



On perdrait l'erasure à la compilation sur les ArrayList<Point>
Pour les Value Type et Primive Type, le type du générique sera propagé des questions de

performance. Le <T> restera indisponible au runtime pour le développeur
A termes, on pourra faire des ArrayList<int>

Représentation dans le bytecode :



On utilise quelques bits de l'entête modifier
ACC_SUPER est renommé en ACC_IDENTITY
Ce flag permet d'appeler le super()
Pour les classes value/primitive, ACC_IDENTITY est positionné à false

La JVM doit savoir très tôt s'il s'agit de value class pour passer la valeur des champs. En Java, les classes sont chargées très tard. Nécessité de flagger ces nouvelles classes.

Pour les Value Class, le stockage par valeur sur le tas sera autorisé si sa taille est inférieure à 128 bits.

Lorsqu'on recompilera le code, le compilateur ajoutera un flag qui sera utilisé par la JVM pour reconnaitre le type de classes.

En conclusion

Le projet Valhalla commencé il y'a 8 ans n'est pas mort.
Les concepteurs ne sont malheureusement pas arrivés à un seul type de value class.
Les value class sont rétro-compatibles si pas de constructeur public, ce qui est le cas des classes récentes de la JVM.