String Concaténation de 1 à 9

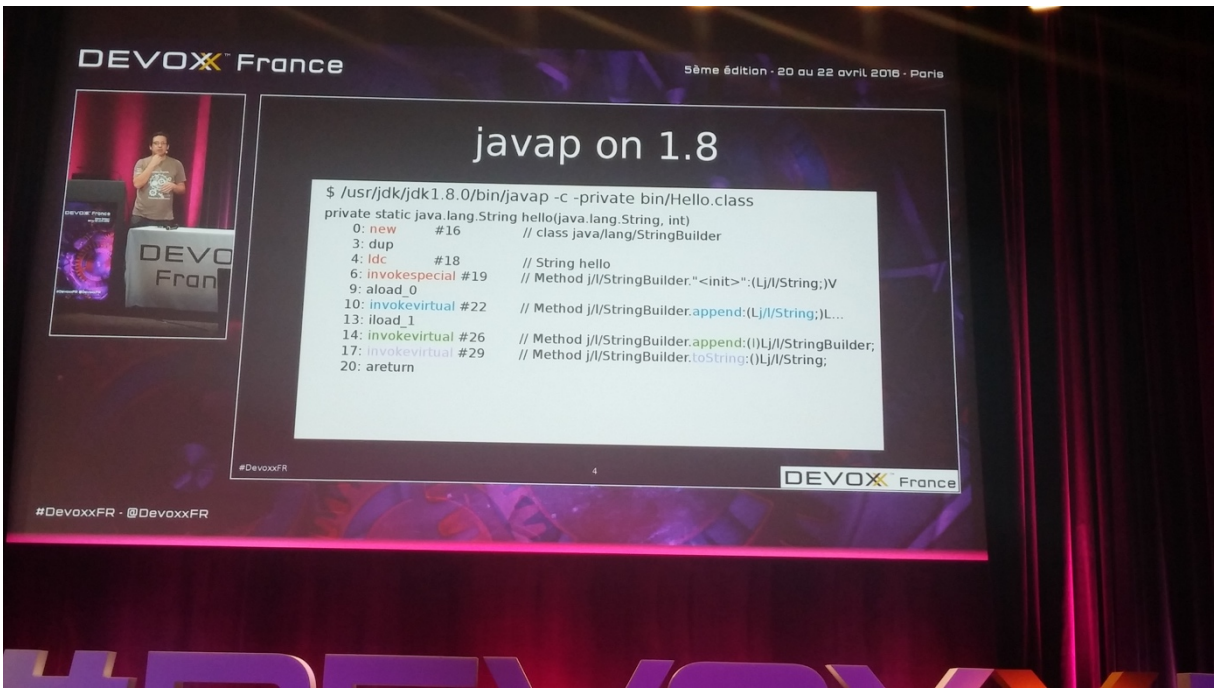
Speaker : Rémi Forax (Université Paris Est Marne la vallée)
Format : Conférence
Date : 21 avril 2016
Slides : https://speakerdeck.com/forax/string-concatenation-de-1-a-9

Développeur sur OpenJDK, ASM, Tatoo (compilateur de compilateur) et Dragon (projet surlequel Rémi fait travailler ses él !ves)

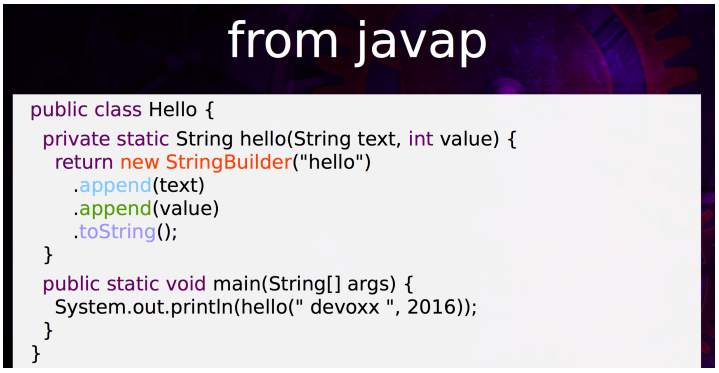
JCP Expert :

- Invokedynamic
- Lambda (JSR 335 + java.util/java.util.stream)
- Jigsaw (JSR 376)

Rémy utilise javap 1.8 pour regarder le bytecode :

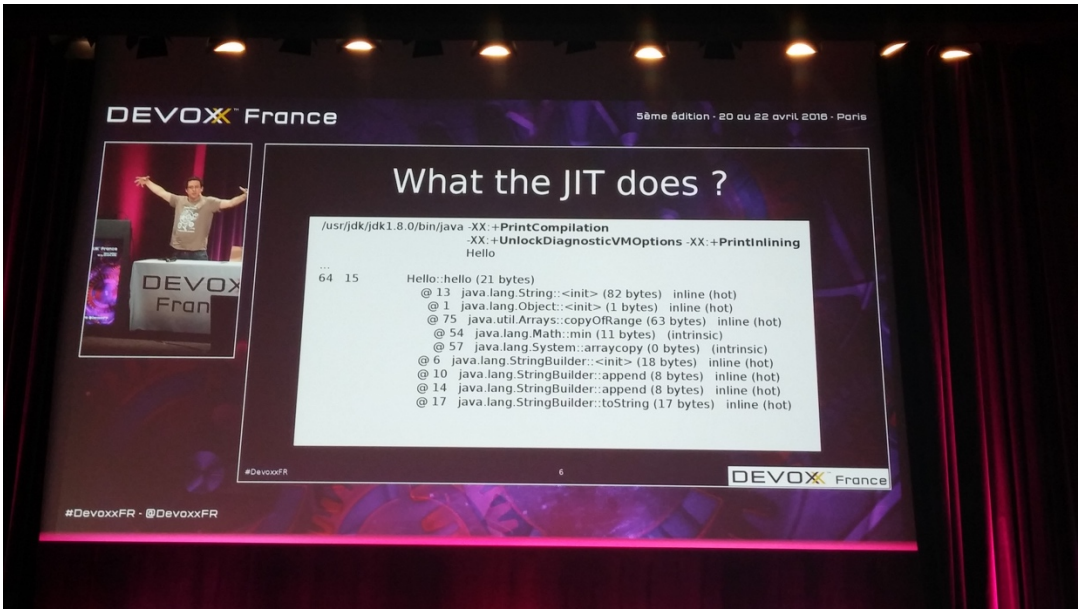


On remaque le new du StringBuilder(), des append() et le toString()
Le compilateur est malin et utilise lui même un StringBuilder. C'est le cas depuis la version 1.0 de Java.



Le bytecode est facile à lire mais çà ne sert à rien.
En cas d'optimisation, la JVM transforme le bytecode en code assembleur.

On peut demander à la JVM d'afficher ce que le JIT compile.
Dans notre exemple, tout le code est compilé. JIT s'arrête lorsque le code à compiler est trop gros.



Le mot clé **inline** indique qu'il a remis le code dans la méthode appelante.
Le mot clé **instrinsic** fait croire qu'une méthode est appelée alors que la JVM appelle directement le code assembleur (ex : System::arraycopy). Cela reImplémentation assembleur sur toutes les plateformes. Techniquement, ce n'est pas réellement du code assembleur mais un langage intermédiaire de code assembleur généré.

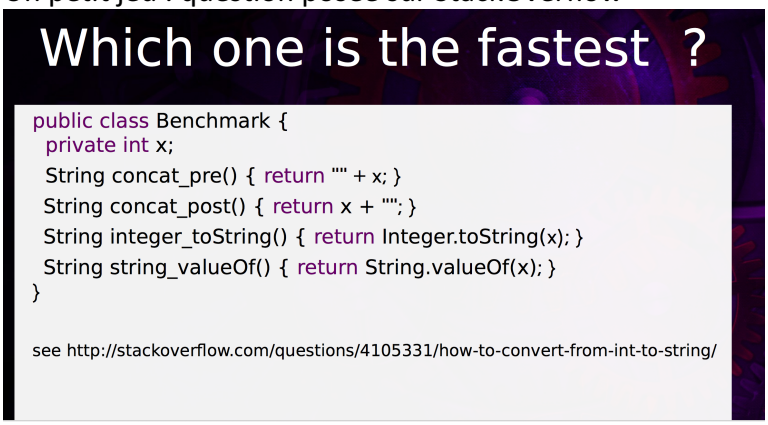
On peut également demander le code assembleur généré : illisible car trop de lignes.

Option par défaut –XX :OptimizeStringConcat
Très performant. Par défaut depuis Java 6

L'opération de concaténation est décomposée en plusieurs étapes :

1. Calcul de la taille de la chaîne
    a. 5 pour « hello »
    b. s.array.arraylength pour « devoxx »
    c. Integer.stringsize(i) pour 2016
2. Création d'un tableau de la bonne taille
3. Ecriture dans le tableau
    a. 2016 à l'envers
    b. hello avec 5 MOV assembleur pour gagner en perf

Un petit jeu : question posée sur StackOverflow



Mesures réalisées par Aleksey Shipilev (créateur de JMH)

Lorsque Java crée un tableau, il met des 0 dedans.
Le JIT connaît la taille du tableau il ne met pas de 0.

Lorsqu'on remplace 2 des String par un Object, le JIT n'arrive plus à optimiser.
Problème récurrent à tous les compilateurs.

Comment trouver les patterns dans le code Java ?
On prend un certains nombre d'applications et on regarde comment les développeurs ont développés. Ces applications sont les serveurs d'applications des années 90.
D'après Rémy, « Plus on code comme les autres, plus le code ira vite »

Concrètement, l'appel polymorphe text.toString() avec text en tant qu'Object, le OptimizeStringConcat ne sait pas le gérer.

Les append suivants ne sont pas optimisés car code trop volumineux :

1. long
2. double
3. value ++

Comment résoudre ce problème ? C'est ce qui a été fait dans Java 9.

Pour limiter les effets de bord, on pourrait faire les toString avant les append.
En réalité, cela ne fonctionne pas bien. Si le premier append plante, on aura quand même fait le toString(). Sans effet de bord, sans problème.

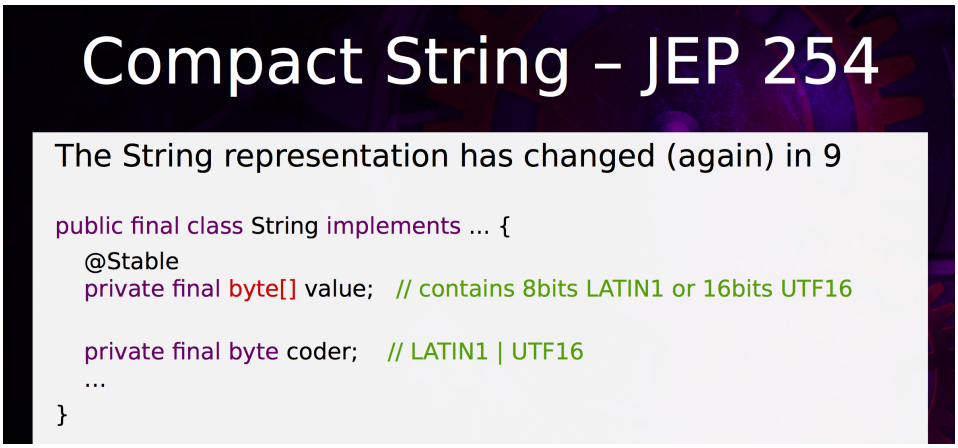
Autre idée : ajout d'une méthode synthétique $$concat$$ générée par le compilateur.
Ne marche pas bien car on rajoute du code dans les .class. Facebook atteint les limites.
Limitation connue de Java. Qui plus est : les $$concat$$ apparaissent dans la stacktrace

Solution : utiliser invokeddynamic
On va faire appel à du code qui n'existe pas encore et qu'on va générer à la volée.
Cette optimisation est faite dans le JDK, écrite en Java, au niveau du compilateur.

Plusieurs stragégies :

Apparté : une version d'ASM existe à l'intérieure du JDK.

Au final, le code assembleur fait 6 ou 7 lignes de plus. Pourquoi ?
A cause de la JEP 254. Les chaines ne sont plus codées avec des chars mais avec des tableaux de bytes. Il y'a un paramètre supplémentaire qui indique l'encodage (LATIN, UTF16)
Enorme travail de réimplémenter String car il y'a des méthodes (ex : equals) qui sont codés en assembleur. Travail de 15 personnes.



La plupart des grosses applications utilisent énormément de String.
Depuis Java 8, des efforts ont été effectués pour améliorer les performances.
Le GC G1 sait dé-dupliquer les String.

Lorsqu'on fait de la concaténation entre 2 chaînes, il faut se demander si les 2 chaînes ont le même encodeur (8 bits vs 16 bits). Si c'est différent, il faut changer l'encodeur. C'est le code assembleur supplémentaire.

Pour bénéficier de cette amélioration introduite dans la JEP 280, il faudra recompiler le code source.
Autre inconvénient : les langages de la JVM autres que Java (ex : Scala) devront utiliser ce modèle.

Risques de régression.

Règle : gérer les StringBuilder à la main que pour les boucles, sinon +