Retour d'expérience sur Java 8

Speaker : Jean-Michel Doudoux
Format : Conférence
Date : 21 avril 2016

Java 8 c'est plus que les Date, les lambdas et les Streams.
2 ans se sont écoulées depuis sa sortie.
Pour Jean-Michel, Java 8 est la plus ambitieuse version de Java.

CTO d'Oxiane Luxembourg
Java Champions
Auteur de 2 didacticiels
www.jmdoudoux.fr

Best Practices

C'est quoi une best practice ?

- Empiriques et subjectives
- Contextuelles
- Mouvantes : rééavulation périodique. Hardware et JVM évoluent.
- Concernes plusieurs facteurs : maintenabilité, performance, style de code
- Ce ne sont pas des règles rigides. On ne doit pas l'appliquer bêtement.

Optional

Classe qui encapsule une valeur ou l'absence de valeur.
L'utilisation d'Optional rend le code plus robuste (au détriment de la performance).
Limitations : classe finale et pas sérialisable
Sujet le plus controversé alors qu'il ne s'agit que d'une classe. Raison : souvent assimilé à d'autres langages

Utilisation comme valeur de retour :

- Nécessaire dans certaines circonstances
- Cas d'utilisation officiel (cf. Brian Goetz)
- A éviter dans les getters de bean

Utilisation dans les paramètres :

- Pas recommandé car pollue la signature
- Plus complexe pour l'appelant.
- Contournement : possibilité d'utilisé la surchage de méthode pour éviter de passer des paramètres null.

Utilisation comme variable d'instance :

- A éviter
- Support par certains frameworks

Utilisation comme variable locale :

- Jamais

Bonne pratiques :

- Passer par une fabriques of(), ofNullable(), empty()
- Essayer de limiter le caractère optionnel d'une valeur
- Pour les primitifs : OptionalInt, OptionLong …
- Attention à l'utilisation de la méthode get() => NoSuchElementException si pas de valeur. Utilisation de orElse() pour passer une valeur par défaut.
- Eviter d'utiliser Optional avec une collection ou un tableau => utiliser une collection ou un tableau vide avec isEmpty() ou length()

Parallel arrays

- Méthodes Arrays.parallelXXXX
- Exemple sur un tableau de 20 millions alimentés avec des nombres aléatoires
    - Méthode setAll() de Java8 permet d'initier un tableau avec une lambda => plus compact pour des perfs équivalentes
    - Parallélisation moins performance car classe Random thread-safe. Utiliser ThreadLocalRandom de Java 9.

Date & Time

- Enfin une API riche et complexe
- Thread-safe car classes immuables

Bonnes pratiques :

- Ne plus utiliser Date/Calendar ni Joda Time
- Bien choisir le type à utiliser selon les données temporelles requises

- Lors de la déclaration de variables, ne pas utiliser les interfaces (ex : Temporal), mais directement les classes (ex : DateTime)
- Utilisation des TemporalAdjuster
- Classe Clock
    - Facilite les tests automatisés
    - Par défaut, renvoie la date/heure système
    - Pour les tests, injecter une instance obtenue par Clock.fixed(). Permet de fixer le temps pour avoir des tests reproductibles

Lambda

Fonction anonyme : fournit une implémentation pour une interface fonctionnelle (SAM)
Java.util.function Function, Predicate, Consumer, Supplier
Nouvel opérateur **->**
Inférences de type : types facultatifs dans les lambdas
Référence de méthodes, opérateur **::**
Une lambda a accès aux variables effectivement finales. Plus besoin d'utiliser le mot clé final.

Bonnes pratiques :

- Utiliser les lambdas à la place des classes anonymes internes
- Privilégier
    - L'inférence de types (le compilateur s'en occupe)
    - L'utilisation des interfaces fonctionnelles fournies par le JDK
- Annoter ses interfaces fonctionnelles avec @FunctionalInterface => utilisé par le compilateur et la javadoc
- Si l'expression est l'invocation d'une méthode, utiliser les références de méthodes
- Garder les expressions lambdas les plus simples possibles
    - Eviter des blocs de code
- Les checked exception sont difficilement intégrable
- Les exceptions levées par une expression doivent être déclarées dans l'interface fonctionnelle.

API Stream

- Exécution de traitement sur une séquence d'éléments
    - Obtenus d'une source finie ou infinie
    - Exécution d'un pipeline d'opérations
    - Exécution séquentielle ou en //
- Requiert de réfléchir en fonctionnel

Bonnes pratiques :

- Attention à l'ordre des opérations intermédiaires
    - Ex : filter() + sorted() vs sorted() + filter() => performances
- Ne pas abuser des Streams
- Bien adapter pour les collections, moins pour les map
- Limiter l'utilisation du forEach (raisonnement impératif et pas fonctionnel)
- Déboguer un Stream
    - Plutôt difficile

- Utiliser la méthode peek()
- Ou utiliser une référence de méthode + point d'arrêt

Avec des données primitives, utilisez par DoubleStream, IntStream … il y'a gros impacts sur les perfs à cause de l'auto-boxing

- Utilisation des Stream infinis
    - Penser à utiliser des limites (ex : limit())
    - Parfois c'est plus subtil que çà
- Les Streams parallèles
    - Facilité de mise en œuvre qui ne rime pas forcément avec performance
    - Utilise le framework Fork/Join et son pool par défaut
    - Les opérations intermédiaires de type stateful vont dégrader un peu les perfs
    - Attention au Spliterator
        - Certaines sources de données (ex : LinkedList et I/O) sont peu performantes
    - Attention aux Collectors
        - Performance (ex : grouping())
        - Utilisation d'objets ayant un support de la concurrence => pas de HashMap
    - Attention au surcoût de la parallélisation
        - Doit être compensé par le volume de données à traiter