Retours sur

Jean-Michel Doudoux @jmdoudoux

Java 8

StampedLock Adder Suppression Perm gen

Type Annotations Compact Profiles Nashorn

Lambda Stream Date & Time

Default Method Interface fonctionnelle Base64

Accumulator Method references Parallel array

Préambule

Probablement la mise à jour

- la plus importante
- la plus impactante

Deux ans depuis la release de Java 8

Intéressant de faire une rétrospective

Sur ce qui fonctionne bien … ou moins bien

Une forte adoption

Mais tout le monde ne l'utilise pas encore

Sondage

Depuis combien de temps utilisez vous Java 8 ?

Plus de 24 mois

Moins de 24 mois

Moins de 18 mois

Moins de 12 mois

Moins de 6 mois

Pas encore utilisé

Jean-Michel Doudoux

CTO chez Oxiane Luxembourg

http://www.jmdoudoux.fr

@jmdoudoux

Java Champions

Auteur de 2 didacticiels
Diffusés sous licence GNU FDL
- Développons en Java (3400 pages)
- Développons en Java avec Eclipse

Co-fondateur du LOR'nJUG, membre du AJUG

Roadmap

- Best practices
- Optional
- Parallel arrays
- Date & Time
- Lambda
- L’API Stream
- Impacts sur notre façon de coder


Best practices

Best practices

Une méthode ou une technique qui a montré de meilleurs résultats que ceux obtenus avec d'autres moyens

Sont empiriques et subjectives

- Issues de l’expérience et son partage

Sont contextuelles

Sont mouvantes

- Réévaluations périodiques

Best practices

Différents critères d'appréciation

Concernent plusieurs facteurs

Maintenabilité, performances, style de code, ...

Ne sont pas des règles

Ne doivent pas devenir des règles
à appliquer de manière rigide dans toutes les situations

Disclaimer

Une partie de ce talk concerne des best practices

Ou peut être assimilé comme tel

Ce sont mes opinions exprimées dans ce talk

Pour vous permettre :

D’obtenir des informations

De les évaluer

Et (peut être) de les mettre en œuvre


Optional

Définition et limitations

Classe qui encapsule :

- une valeur
- ou l’absence de valeur

L'utilisation d'Optional rend le code plus fiable

mais cela a un coût (très léger) en performance

Classe value-based final

N'est pas Serializable

Utilisation

Sûrement le sujet le plus controversé

Souvent assimilé (à tort) comme certaines fonctionnalités dans d’autres langages

Différents cas d'utilisation :

- Partout
- Dans les API public pour les valeurs de retour et les paramètres
- Dans les API public pour les valeurs de retour
- Dans des cas spécifiques
- Ne jamais utiliser

L'utilisation comme valeur de retour

Nécessaire dans certaines circonstances

- Evite la définition d’une valeur représentant l’absence de valeur

Le cas d'utilisation officiel

Généralement la durée de vie d'un objet retourné est courte

Eviter dans les getters de beans

- Pour limiter les instances de type Optional créées
- Support plus ou moins facile avec certains frameworks

L'utilisation dans les paramètres

N'est pas recommandé

Pollue la signature de la méthode

```
public MonMessage(String titre, String contenu, Optional<Attachment> attachment) {
    // ...
}
```

Plus complexe pour l'appelant

```
MonMessage m1 = new MonMessage("titre", "contenu", Optional.empty());
Attachment attachment = new Attachment();
MonMessage m2 = new MonMessage("titre", "contenu", Optional.ofNullable(attachment));
```

L'utilisation dans les paramètres

Préférer l'utilisation de la surcharge de la méthode

```
public MonMessage(String titre, String contenu) {
    this(titre, contenu, null);
}

public MonMessage(String titre, String contenu, Attachment attachment) {
    // ...
    if (attachment != null) {
    // ...
}
```

L'utilisation comme variable d’instance

Eviter de déclarer des variables d’instances de type Optional

Optional n’est pas Serializable

Attention au support par certains frameworks

L'utilisation comme variable locale

Ne pas déclarer de variable local de type Optional

Utiliser null pour indiquer l’absence de valeur

- Pour un objet
- Facile à gérer dans le scope d’une méthode

Bonnes pratiques

Optional est une classe

Ne JAMAIS définir une instance null

Toujours utiliser une des fabriques

- of()
- ofNullable()
- empty()

Bonnes pratiques

Essayer de limiter le caractère optionnel d’une valeur

Pour des valeurs primitives :

- OptionalInt, OptionalLong, OptionalDouble

Attention à l'utilisation de la méthode get()

- Lève une unchecked NoSuchElementException
- Si aucune valeur n'est encapsulée
- Utiliser orElse() si possible

Bonnes pratiques

Eviter d’utiliser Optional typé avec une collection ou un tableau

- Optional n’apporte aucune plus value
- Et complexifie le code
- Préférer une collection ou un tableau vide
  - isEmpty() ou length() pour tester la présence d’éléments

Parallel Arrays

Parallel arrays intro

- Méthodes Arrays.parallelXXX()

  parallelSetAll() : initialisation des éléments

  parallelSort() : tri des éléments

- Exécution des traitements en parallèle

  Utilise le framework Fork/Join

- Apporte un gain de performance plus ou moins important

  Essentiellement sur des tableaux de grande taille

Utilisation avant Java 8

- Initialiser un tableau de 20 millions d'entiers avec des valeurs aléatoires

```
int[] array = new int[20_000_000];
Random rnd = new Random();

for (int i = 0; i < array.length; i++) {
  array[i] = rnd.nextInt(100);
}
```

```
Benchmark                                  Mode  Cnt      Score     Error  Units
InitTableauBenchmark.boucleFor             avgt   20    271,341 ±   5,606  ms/op
```

Avec Java 8 : setAll()

```
int[] array = new int[20_000_000];
Random rnd = new Random();
Arrays.setAll(array, i -> rnd.nextInt(100));
```

Code plus compact

```
Benchmark                                  Mode  Cnt     Score     Error  Units
InitTableauBenchmark.boucleFor             avgt   20   271,341 ±   5,606  ms/op
InitTableauBenchmark.setAll                avgt   20   271,406 ±   6,411  ms/op
```

Performances similaires

Utilisation de Parallel arrays

La méthode parallelSetAll() initialise le tableau de manière parallélisée

```
int[] array = new int[20_000_000];
Random rnd = new Random();
Arrays.parallelSetAll(array, i -> rnd.nextInt(100));
```

Beaucoup plus performant puisque parallélisé … OU PAS

```
Benchmark                                  Mode  Cnt      Score      Error  Units
InitTableauBenchmark.setAll                avgt   20    271,406 ±    6,411  ms/op
InitTableauBenchmark.parallelSetAll        avgt   20   2172,501 ± 160,561  ms/op
```

Beaucoup plus lent

Réduction de la contention

La classe Random est thread-Safe mais elle induit la contention

Utilisation de la classe ThreadLocalRandom

```
int[] array = new int[20000000];
ThreadLocalRandom rnd = ThreadLocalRandom.current();
Arrays.parallelSetAll(array, i -> rnd.nextInt(100));
```

Les performances sont bien meilleures

Benchmark	Mode	Cnt	Score	Error	Units
InitTableauBenchmark.setAll	avgt	20	271,406	± 6,411	ms/op
InitTableauBenchmark.parallelSetAll	avgt	20	2172,501	± 160,561	ms/op
InitTableauBenchmark.parallelSetAllAvecTLR	avgt	20	94,822	± 3,724	ms/op

Moralité

Facile de paralléliser des traitements

- Simplement en préfixant la méthode par parallel

Mais attention aux performances

... Sans attentions particulières


Date & Time

Date & Time intro

(Enfin) Une API riche et complète

Comble de nombreux manques et lacunes de Date et Calendar

Gestion du temps humain et machine

Classes non mutables

Fluent interface

Bonnes pratiques

Utiliser cette API plutôt que Date/Calendar ou Joda Time

Bien choisir le type à utiliser selon les données temporelles requises

Déclaration de variables :

- Ne pas utiliser les interfaces

```
Temporal localDate = LocalDate.of(2016, 04, 20);
```

- Utiliser les types

```
LocalDate localDate = LocalDate.of(2016, 04, 20);
```

Bonnes pratiques

Utilisation des TemporalAdjuster

Injecter une instance de type Clock

Et utiliser les surcharges qui attendent un type Clock

- Pour faciliter les tests automatisés
- Par défaut, renvoie la date/heure système
- Pour les tests, injecter une instance obtenue par Clock.fixed()


Lambda

Lambda intro

Fonction anonyme

fournit une implémentation pour une interface fonctionnelle (SAM)

Package java.util.function

Function, Predicate, Consumer, Supplier, …

Syntaxe : (paramètres) -> expression;
(paramètres) -> { corps };

Inférence de types

Références de méthodes, opérateur ::

Accès aux variables effectivement finale

Bonnes pratiques

Utiliser des expressions Lambda plutôt que des classes anonymes internes

Privilégier

- l'inférence de type des paramètres
- l'utilisation des interfaces fonctionnelles fournies dans le JDK

Annoter ses interfaces fonctionnelles avec @FunctionalInterface

Si l’expression est l’invocation d’une méthode

Utiliser les références de méthodes

Bonnes pratiques

Garder les expressions Lambda les plus simples possibles

- Eviter les bloc de code dans les expressions
- Utiliser les références de méthodes

Les checked exceptions

Il est difficile d'intégrer les checked exceptions dans les lambdas

Les exceptions levées par une expression

- Doivent être déclarées dans l’interface fonctionnelle

La plupart des interfaces fonctionnelles ne déclarent pas d'exceptions

Wrapper le code dans un try/catch

- Qui relève une exception de type RunTime


L’API Stream

Stream intro

Exécution de traitement sur une séquence d'éléments

- Obtenus d'une source finie ou infinie
- Exécution d'un pipeline d'opérations
- Exécution séquentiel ou parallèle

Fluent interface

- Opérations intermédiaires : lazy, renvoient un Stream
- Opérations terminales : déclenchent les traitements, ferment le Stream

Requiert de penser fonctionnel et pas impératif

- Ne pas penser boucles et état mutable

Bonnes pratiques

Attention à l'ordre des opérations intermédiaires

Ex : filter() + sorted() vs sorted() + filter()

Ne pas abuser des Streams

Parfois une boucle est plus lisible

Bien adapté pour les Collections, moins pour les Maps

Limiter l'utilisation de la méthode forEach()

Déboguer un Stream

Plutôt difficile

- majorité des traitements réalisés par l'API

Utiliser la méthode peek()

- peek(System.out::println)
- ou peek (e –> e) + point d'arrêt

Utiliser une référence de méthode + point d'arrêt

Avec des données primitives

Utiliser les classes dédiées : DoubleStream, IntStream, LongStream

Améliore (grandement) les performances

```
public void testStreamLong() {
    Long somme = Stream
        .iterate(0L, i -> i + 1L)
        .limit(20_000_000)
        .filter(i -> (i % 2) == 0)
        .map(i -> i + 1)
        .sorted()
        .reduce(0L, Long::sum);
}
```

```
public void testLongStream() {
    long somme = LongStream
        .iterate(0, i -> i + 1)
        .limit(20_000_000)
        .filter(i -> (i % 2) == 0)
        .map(i -> i + 1)
        .sorted()
        .sum();
}
```

```
Benchmark                          Mode  Cnt      Score      Error  Units
StreamBenchmark.testLongStream     avgt   10    233,061 ±   10,962  ms/op
StreamBenchmark.testStreamLong     avgt   10   4322,430 ± 258,794  ms/op
```

Les Stream infinis

Attention lors de l'utilisation de Stream infinis

```
IntStream.iterate(0, i -> i + 1).forEach(System.out::println);
```

Il faut introduire une condition d'arrêt

```
IntStream.iterate(0, i -> i + 1).limit(5).forEach(System.out::println);
```

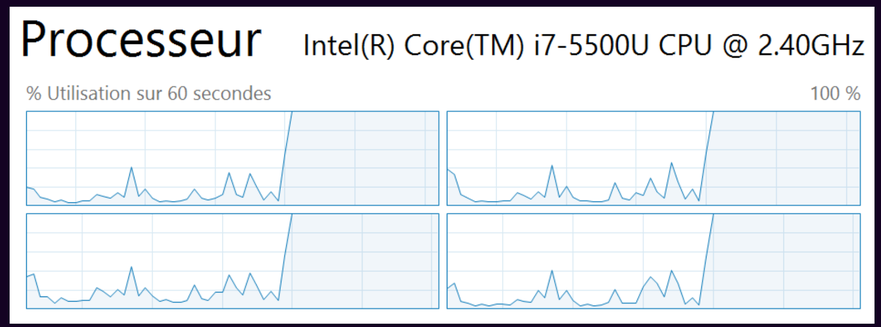
Les Stream infinis

Parfois les traitements infinis sont plus subtils

```
List<Double> vale  = Stre
    .generate(Math:.  m)
    .filter(v -> (v  1  && (v < 20))
    .limit(10).co ect(Col ctors.toList());
```

Dans un Stream parallèle, c'est la catastrophe

```
List<Double> vale  = Strea
    .generate(Math:.  m)
    .parallel().filt  > (v > 10) && (v < 20))
    .limit(10).co ect(Col ctors.toList());
```



Les Stream parallèles

Facilité de mise en œuvre

- Ne rime pas forcément avec meilleures performances

Utilise le framework Fork/Join

- Et son pool par défaut
- Ne pas utiliser d’opérations bloquantes ou trop longues

La source de données ne doit pas être modifiée

Les Stream parallèles

Attention à certaines opérations intermédiaires

- Opération stateful : sorted()
- Opération requérant un ordre : limit()

Les Stream parallèles

- Attention au Spliterator
  - Certaines sources offrent de mauvaises perf
  - Ex: LinkedList, sources effectuant des I/O, ...
- Attention aux Collectors
  - Performance
  - Utilisation d'objets ayant un support de la concurrence
- Le surcoût de la parallélisation
  - Doit être compensé par le volume de données à traiter


Impacts sur notre
façon de coder

Exemple avec JUnit 5

Exemple avec JUnit5 (actuellement en version Alpha)

Utilise des fonctionnalités de Java 8

- Notamment les Lambda

Propose de nombreuses fonctionnalités

- Focus sur celles utilisant Java 8

Exemple avec JUnit 5

Utilisation des Lambda

Dans les assertions

```
assertTrue(() -> "".isEmpty(), "string should be empty");

assertTrue(false, () -> "message évalué de manière lazy");
```

Les assertions groupées

```
Dimension dim = new Dimension(800, 600);
assertAll("Dimensions non conformes",
    () -> assertTrue(dim.getWidth() == 800, "width"),
    () -> assertTrue(dim.getHeight() == 600, "height"));
```

Exemple avec JUnit 5

Pour tester les exceptions levées

```
assertThrows(RuntimeException.class, () -> {
    throw new NullPointerException();
});
```

```
Throwable exception = expectThrows(RuntimeException.class, () -> {
    throw new NullPointerException("Ne peux pas etre null");
});
assertEquals("Ne peux pas etre null", exception.getMessage());
```

Pour les assomptions

```
assumeTrue("DEV".equals(System.getenv("ENV")), () -> "Arret des tests :
execution uniquement sur un poste de DEV");
```

Exemple avec JUnit 5

Définition de tests dans les méthodes par défaut

```
import static org.junit.gen5.api.Assertions.assertEquals;

import org.junit.gen5.api.Test;

public interface Testable {

  @Test
  default void monCasDeTest() {
    assertEquals(2, 1 + 1);
  }
}

public class MonTestable implements Testable {
}
```

Exemple avec JUnit 5

Utilisation des annotations répétées

Annotations @Tag, @ExtendWith, …

```
import org.junit.gen5.api.Tag;
import org.junit.gen5.api.Test;

@Tag("tag1")
@Tag("tag2")
public class MonTest {

  @Test
  void monTest() {
    // …
  }
```

Conclusion

Conclusion

Java 8, c'est aussi beaucoup d'autres fonctionnalités

- Améliorations de l'API Collection
- La suppression de la perm gen -> meta space dans Hotspot
- Nashorn
- JavaFX
- StampedLock, Adder, Accumulator, ConcurrentHashMap
- Java type annotations, annotations répétées
- Concaténation de chaînes
- Reflection : accès aux noms des paramètres
- Base64
- ...

Conclusion

Java 8 changent profondément

- La manière de coder certaines fonctionnalités

Attention à la facilité de paralléliser

Lorsque cela concerne les performances

- Il faut mesurer, même si cela est (très) difficile
  - avec JMH

Continuez à approfondir l'utilisation de Java 8

- avant l'arrivée de Java 9

Merci pour votre attention