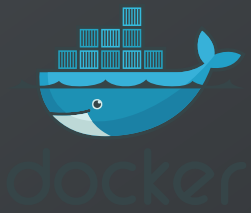
Nouveautés Java

Grégory BÉVAN

Consultant & Formateur @Zenika



Quelles nouveautés ?

Lambda functions

Stream API

Date Time API

Et encore…

λ

La programmation fonctionnelle

Origines la programmation fonctionnelle

Paradigme de programmation s’appuyant sur la théorie du calcul Lambda

Inventé par Alonzo Church en 1930 et rationalisé par Haskell Curry

L’idée de base est que tout est fonction

Premiers langages fonctionnels

Lisp (1958)
APL (1964)
Scheme (1970)
ML (1973)
FP (1977)
Erlang (1987)
Haskell(1990)

Les langages modernes (mixtes)

Python (1990)
Ruby (1995)
Javascript (1995)
F# (2002)
Scala (2003)
Clojure (2007)
Rust (2010)
Kotlin (2016)

Et Java alors ?

Des contraintes lourdes

Langage orienté objet

La rétrocompatibilité de Java

=> Comment résister à la pression des langages alternatifs (Groovy, Scala, Ceylon)

Profiter des classes anonymes

```
Runnable task = new Runnable() {
    @Override
    public void run() {
            System.out.println("Run a new thread");
    }
};

ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(task);
executor.shutdown();
```

Caractéristiques d'une classe anonyme

Créée à partir d'une interface ou d'une super classe

Implémentation fournie à l'instanciation

Peut être passée en paramètre d'une méthode

Lourdeur syntaxique

Avec Java 8

```
Runnable task = () -> System.out.println("Run a new thread with lambdas");

ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(task);
executor.shutdown();
```

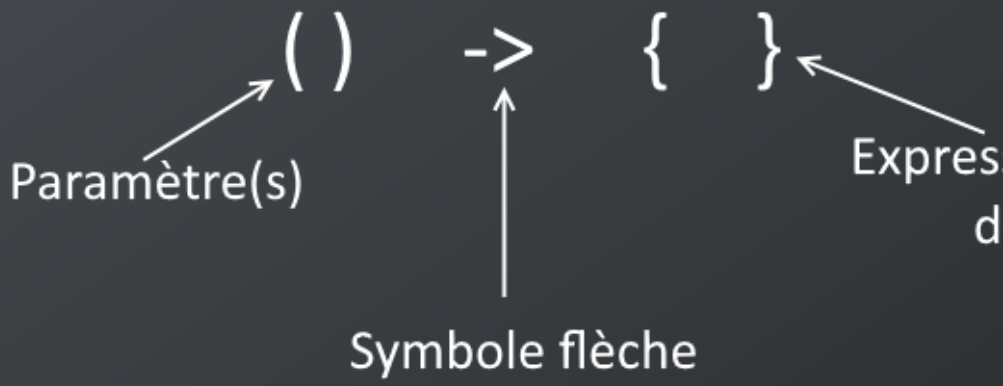
Expression lambda

Bloc de code (≈ closure)

Exécution différé dans le temps

Accède aux variables final du scope

La syntaxe



Différentes notations

Pas de paramètre / Expression simple

```
Runnable runnable = () -> System.out.println("New thread");
```

Avec paramètre typé / Bloc d'instruction

```
Function<String,String> uppercase = (String s) -> {return s.toUpperCase();};
```

Paramètre sans type (déduit par le compilateur) / Return implicite

```
Comparator<String> stringComparator = (s1, s2) -> s1.length() - s2.length();
```

Paramètre unique sans parenthèse

```
Function<String, String> lowercase = s -> s.toLowerCase();
```

Exemple de comparateur

```
Comparator<String> stringLengthComparator = (s1, s2) -> {
    int compareTo = Integer.compare(s1.length(), s2.length());
    System.out.printf("Compare \"%s\" et \"%s\" : %s \n", s1, s2, compareTo);
    return compareTo;
};
```

Les interfaces fonctionnelles

Interface ne contenant une unique méthode

Optionnellement annotée @FunctionalInterface

De nombreuses interfaces existantes dans le JDK (Comparator, Runnable, Listener…)

Exemple d'interface fonctionnelle

```java
@FunctionalInterface
public interface Operator {

    int apply(int a, int b);

}
```

Interfaces fonctionnelles génériques

Contenues dans le package *java.util.function*

Supplier<T>	Fournit un élément de type T	`T get()`
Consumer<T>	Consomme un élément de type T	`void accept(T t)`
Function<T,R>	Transforme un élément de type T en élément de type R	`R apply(T t)`
Predicate<T>	Vérifie une condition sur un élément de type T	`boolean test(T t)`

...

Référence de méthode

Permet d’appeler une méthode existante

Sucre syntaxique

Utilisation d’un nouvel opérateur **::**

Référence de méthode de classe

L'expression lambda suivante

```
Arrays.sort(new String[]{"Grégory", "Antoine", "Eric"},
            (nom1, nom2) -> nom1.compareTo(nom2));
```

se simplifie en

```
Arrays.sort(new String[]{"Grégory", "Antoine", "Eric"}, String::compareTo);
```

en faisant référence à la méthode

```
public int compareTo(String anotherString)
```

Référence de méthode d'objet

La classe suivante possède une méthode *compareTo*

```
class ReverseComparator {
        public int compareTo(String a, String b) {
            return b.compareTo(a);
        }
}
```

qui permet d'écrire le code suivant

```
ReverseComparator customStringComparator = new ReverseComparator();
Arrays.sort(new String[]{"Grégory", "Antoine", "Eric"},
            (nom1, nom2) -> customStringComparator.compareTo(nom1, nom2));
```

et que nous simplifions comme ceci

```
Arrays.sort(new String[]{"Grégory", "Antoine", "Eric"},
              customStringComparator::compareTo);
```

Référence de méthode statique

A partir de la méthode statque suivante

```
public class StaticMethodReference {
  public static int compareLength(String a, String b) {
      return Integer.compare(a.length(), b.length());
  }
}
```

il est possible d'écrire

```
Arrays.sort(new String[]{"Grégory", "Antoine", "Eric"},
              StaticMethodReference::compareLength);
```

Référence de constructeur

En prenant l'interface fonctionnelle suivante

```
@FunctionalInterface
public interface TeamBuilder {
    Team createTeam(String[] teamFirstName);
}
```

et la classe définissant une équipe

```
public class Team {
    public List<String> firstNames;

    public Team(final String[] firstNames) {
        this.firstNames = Arrays.asList(firstNames);
    }
}
```

Référence de constructeur

Il est possible d'écrire

```
TeamBuilder teamBuilder = (names) -> new Team(names);
Team team = teamBuilder.createTeam(
                            new String[]{"Antoine", "Guillaume", "Fabien"});
```

ou de préférence en utilisant une référence de constructeur

```
TeamBuilder teamBuilder = Team::new;
```

Portée des variables

Une expression lambda peut accéder :

- aux paramètres de l'expression
- aux variables définies dans le bloc d'instruction
- aux variables englobantes si elles sont finales
- aux variables effectivement finales (non déclarées finales)

Illustration de la portée des variables

```
int counter = 0;
int[] counterArray = new int[1];
List<String> list = new ArrayList<>();

Runnable runnable = () -> {
    System.out.printf("Counter : %s \n", counter); // Counter peut être lue
    // counter++; // Counter ne peut  pas être modifiée
    counterArray[0]++; // Pas thread safe
    list.add(String.valueOf(counterArray[0]));
    System.out.printf("Counter array : %s \n", counterArray[0]);
};
```

Méthode par défaut

Nouveau type de méthode dans les interfaces

Permet de fournir une implémentation par défaut

Facilite l'enrichissement d'interface

Déclaration avec le mot clé default

Exemples de méthode par défaut

Nouvelle méthode *sort* de l'interface *List*

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

Méthode *andThen* dans l'interface fonctionnelle *Function*

```
default <V> Function<T, V> andThen(Function<R,V> after) {
    Objects.requireNonNull(after);
    return (T t) -> after.apply(apply(t));
}
```

Méthode statique

Désormais autorisée dans les interfaces

Permet de revoir l'esprit de contrat lié à une interface

Possibilité de regroupée interface et classe utilitaire

Déclaration avec le mot clé static

Exemples de méthode statique

Nouvelle méthode *comparingByKey* de l'interface *Map*

```
public static <K extends Comparable<? super K>, V>
    Comparator<Map.Entry<K,V>> comparingByKey() {
        return (Comparator<Map.Entry<K, V>> & Serializable)
            (c1, c2) -> c1.getKey().compareTo(c2.getKey());
}
```

Méthode *getZoneId* de la nouvelle interface *TimeClient*

```
static ZoneId getZoneId (String zoneString) {
    try {
        return ZoneId.of(zoneString);
    } catch (DateTimeException e) {
        System.err.println("Invalid time zone: " + zoneString +
            "; using default time zone instead.");
        return ZoneId.systemDefault();
    }
}
```

L’API Stream

Eviter la confusion

Java I/O utilise le terme Stream

InputStream, OutputStream…

La nouvelle API Stream n’a pas de lien avec cette notion

Pourquoi une nouvelle API

Faciliter le traitement d'ensemble de données

Gagner en lisibilité du code

Améliorer les performances des traitements

Caractéristiques d'un Stream

Pas une structure de stockage, mais une séquence d'éléments

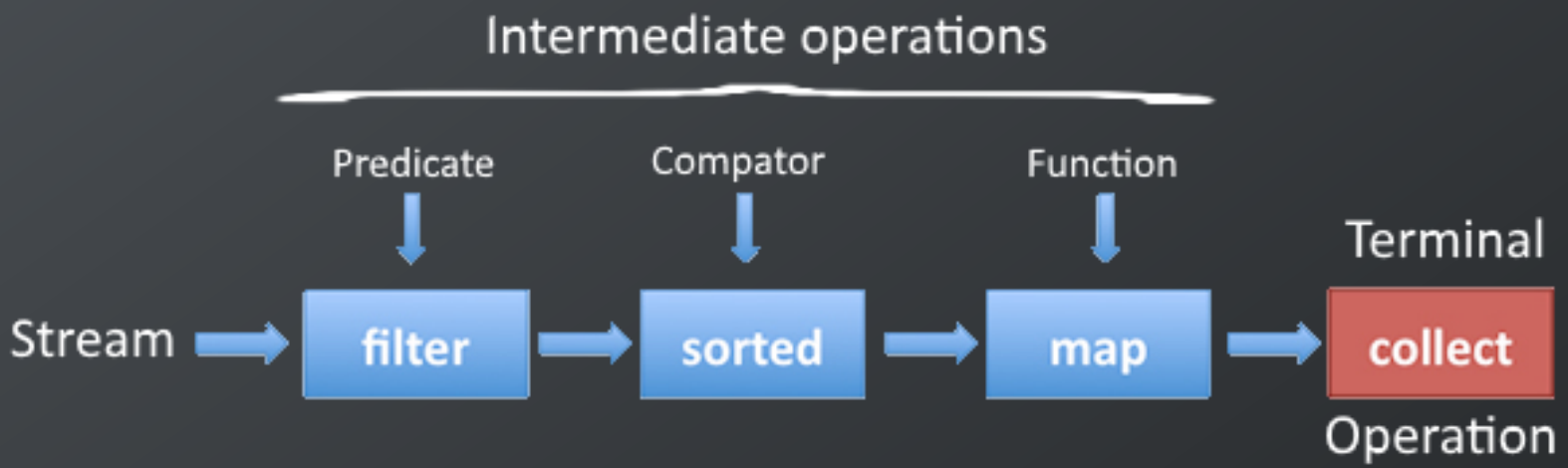
Ne peut être réutilisé

N'altère pas la source d'éléments

Pipeline d'opérations optimisées et parallélisables

Un premier exemple de Stream

Pipeline d'opérations



Création de Stream

A partir d'une collection

A partir d'un tableau

A partir de méthodes statiques

A partir de méthodes sur des classes du JDK

Opérations intermédiaires sans état

```
// Filtrer sous condition
filter(Predicate<? super T> predicate)
```

```
// Appliquer une transformation
map(Function<? super T,? extends R> mapper)
```

```
// Applique une transformation et linéarise le résultat
flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
```

```
// Réalise l'action et retourne un Stream identique
peek(Consumer<? super T> action)
```

```
// Limite la taille du Stream
limit(long maxSize)
```

```
// Skip les n derniers éléments du Stream
skip(long n)
```

Opérations intermédiaires avec état

```
// Supprime les éléments en doublon
distinct()
```

```
// Tri les éléments selon l'ordre naturel
sorted()
```

```
// Tri les éléments selon le comparateur fourni
sorted(Comparator<? super T> comparator)
```

Opérations terminales - Itérations

```java
// Réalise l'action sur chaque élément du Stream
void forEach(Consumer<? super T> action)
```

```java
// Réalise l'action sur chaque élément du Stream dans l'ordre du flux
// en traitement séquentiel ou parallèle
void forEachOrdered(Consumer<? super T> action)
```

Opérations terminales - Réductions simples

```
// Retourne le nombre d'élément du Stream
count()
```

```
// Retourne un tableau d'élément du Stream
toArray()
```

```
// Retourne l'élément min selon le comparateur
min(Comparator<? super T> comparator)
// Idem mais pour l'élément max
max(Comparator<? super T> comparator)
```

Opérations terminales - find

Pour récupérer un unique élément du flux

```
// Retourne le premier élément du Stream s'il existe
Optional<T> findFirst()
```

```
// Retourne un élément du Stream (non déterministe)
Optional<T> findAny()
```

Optional

Wrapper d'objet pour éviter les NullPointerException

Exemple de déclaration

```
Optional<String> nonNull  = Optional.of("Non null");
Optional<String> empty    = Optional.empty();
Optional<String> inconnu  = Optional.ofNullable(nullOuPas);
```

Exemple d'utilisation

```
String s2 = inconnu.orElse("Par défaut");
// or
inconnu.ifPresent(System.out::println);
```

Operations terminales : match

Pour vérifier la correspondance d'éléments à une condition

```
// Vrai si tous les éléments du stream répondent à la condition
boolean allMatch (Predicate<T> predicate)
```

```
// Vrai si un ou plusieurs éléments du flux répondent à la condition
boolean anyMatch(Predicate<? super T> predicate)
```

```
// Vrai si aucun élément du flux ne répond à la condition
boolean noneMatch(Predicate<? super T> predicate)
```

Operations terminales : reduction

Retourne une valeur résultant d'une combinaison des éléments du Stream

Les plus connus sont : somme, moyenne, max, min

Exemple d'opération de réduction produisant la somme des éléments

```
Integer totalAgeReduce = roster.stream()
                              .map(Person::getAge)
                              .reduce(0, (a, b) -> a + b);
```

Exemple simplifié

```
Integer totalAge = roster.stream().mapToInt(Person::getAge).sum();
```

Operations terminales : collect

Récupérer les éléments du stream dans une nouvelle structure

Dans un tableau

```
Person[] men = roster.stream().filter(p -> p.getGender() == MALE)
                             .toArray(Person[]::new);
```

Dans une liste

```
List<Person> collect = roster.stream().filter(p -> p.getAge() < 18)
                                      .collect(Collectors.toList());
```

Operations terminales : collect

Dans un set

```
Set<Person> collect = roster.stream().filter(p -> p.getAge() < 18)
                                     .collect(Collectors.toSet());
```

Dans une map

```
Map<String, Person> rosterAsMap = roster.stream().collect(
                                Collectors.toMap(p->p.firstName.toLowerCase(),
                                Function.identity()))
```

Operations terminales : joining & grouping

Concaténation des éléments avec l'opération joining

```
String firstNames = roster.stream().map(p -> p.firstName)
                                   .collect(Collectors.joining(", "));
```

Regroupement des éléments pour constituer une map

```
Map<Gender, List<Person>> groupedByGender = roster.stream()
                                  .collect(Collectors.groupingBy(p -> p.gender));
```

Parallel stream

Création avec la méthode *parallelStream()* au lieu de *stream()*

Opérations réalisées sur plusieurs Thread

Opérations stateless car exécutées de façon aléatoires (ThreadSafe)

```
int[] shortWords = new int[12];
words.parallel().forEach(s -> {
    if (s.length() < 12) shortWords[s.length()]++; // Not thread safe!
});
System.out.println(Arrays.toString(shortWords));
```

L’API Date Time

Pourquoi une nouvelle API date

Combler les lacunes et incohérences de *java.util.Date* et *java.util.Calendar*

- Année débutant à 1900, Mois commençant à 0…
- Pas Thread Safe car Date et Calendar sont mutables
- Une objet Date contient les heures (DateTime?)
- API complexe, non intuitive

=> Joda Time a permis de compenser ces lacunes

3 principaux concepts

Créer un ensemble de classes immutables (Thread safe)

Application de DDD pour concevoir les classes

Notion de chronologie pour différencier les calendriers du monde

=> Le calendrier par défaut est nommé ISO-8601

Gestion des dates

LocalDate	Date sans fuseau horaire	*2017-06-19*
LocalTime	Heure sans fuseau horaire	*09:30:00*
LocalDateTime	Date et heure sans fuseau horaire	*2017-06-19T09:30:00*
ZonedDateTime	Date et heure avec fuseau horaire	*2007-12-03T10:15:30+01:00 Europe/Paris*

LocalDate

Construction de date locale

```
LocalDate now = LocalDate.now();
LocalDate startingDate = LocalDate.of(2017, 06, 19);
LocalDate endingDate = LocalDate.parse("2017-06-19");
```

Méthodes pour manipuler les dates

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
DayOfWeek monday = LocalDate.parse("2017-06-19").getDayOfWeek();
boolean leapYear = LocalDate.now().isLeapYear();
boolean notBefore = LocalDate.now().isBefore(LocalDate.parse("2016-06-18"));
```

LocalTime

Construction d'heure locale

```
LocalTime now = LocalTime.now();
LocalTime startingTime = LocalTime.of(9, 30);
LocalTime endingTime = LocalTime.parse("09:30");
```

Méthodes pour manipuler les heures locales

```
LocalTime oneHourLater = LocalTime.now().plusHours(1);
int minutes = LocalTime.now().getMinute();
boolean isAfter = LocalTime.now().isAfter(LocalTime.of(11, 0);
LocalTime max = LocalTime.MAX;
```

LocalDateTime

Construction de date et heure locale

```
LocalDateTime now = LocalDateTime.now()
LocalDateTime startingDateTime = LocalDateTime.of(2017, Month.JUNE, 19, 9, 30);
LocalDateTime endingDateTime = LocalDateTime.parse("2017-06-19T11:00");
```

Méthodes pour manipuler les dates et heures locales

```
LocalDateTime nextSession = LocalDateTime.of(2017, 06, 19, 9, 30)
                                         .plusDays(1).plusMinutes(30);
LocalDateTime firstDayOfNextMonth = LocalDateTime.now().with(TemporalAdjusters
                                                  .firstDayOfNextMonth());
LocalDateTime min = LocalDateTime.MIN;
LocalDateTime max = LocalDateTime.MAX;
```

ZonedDateTime

Construction de date et heure avec fuseau horaire

```
ZonedDateTime now =  ZonedDateTime.now();
ZonedDateTime fromLocalDate = ZonedDateTime.of(LocalDateTime.now(),
                                               ZoneId.of("Europe/Paris"));
ZonedDateTime startingDate = ZonedDateTime.parse("2017-06-19T07:30:00Z");
```

Manipulation de date et heure avec fuseau horaire

```
ZonedDateTime utc = ZonedDateTime.now().withZoneSameInstant(ZoneId.of("UTC");
ZoneOffset offset = ZonedDateTime.now().getOffset();
ZoneId ourZoneId = ZonedDateTime.now().getZone();
```

Period & Duration

Gérer des périodes

```
LocalDate manufacturingDate = LocalDate.of(2017, Month.JANUARY, 1);
LocalDate expiryDate = LocalDate.of(2017, Month.JUNE, 20);

Period expiry = Period.between(manufacturingDate, expiryDate);
int numberOfDays = expiry.getDays();
```

Gérer des durées

```
LocalDateTime startingDate = LocalDateTime.of(2017, Month.JULY, 19, 9, 30);
LocalDateTime endingDate = LocalDateTime.of(2017, Month.JULY, 19, 11, 00);

Duration talkDuration = Duration.between(startingDate, endingDate);
long hours = ChronoUnit.HOURS.between(startingDate, endingDate);
long minutes = ChronoUnit.MINUTES.between(startingDate, endingDate);
```

Instant

Représente le temps machine

Adapté pour gérer les timestamps

Démarre en 1970-01-01T00:00:00Z (Unix Epoch)

```
Instant timestamp = Instant.now();
long numberOfSecondsSinceEpoch = now.getEpochSecond();
long hoursBeforeEnfOfMachineTime = now.until(Instant.MAX, ChronoUnit.HOURS);
```

Formater les dates et heures

`java.time.format.DateTimeFormatter` permet de convertir les dates :

- transformer un objet date en chaine de caractères : `format()`
- parser une chaine de caractères en date : `parse()`

```
LocalDate date = LocalDate.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy MM dd");
String text = date.format(formatter);
```

```
String input = "2016 03 17";
LocalDate parsedDate = LocalDate.parse(input, formatter);
```

Et encore...

En vrac

La classe `java.util.concurrent.CompletableFuture` : les promesses en Java

JavaFX : RDA en Java

Nashorn : le nouveau moteur d'exécution Javascript

Enrichisssement de type basique du JDK (String, Files, Math…)

Questions ?