

17/05/2022

Version 1

JAVA 8

JÉRÉMY PERROUULT

A decorative wavy line in light blue and white, flowing vertically along the left edge of the slide.

INTRODUCTION

INTRODUCTION

INTRODUCTION

- Oracle JDK
 - Licence commerciale, payant
- OpenJDK
 - Licence OpenSource, gratuit
 - Téléchargeables sur <https://openjdk.java.net/>
- Depuis Java9, la stratégie de nommage des versions a changé
 - Par exemple, Java 8 pouvait s'appeler « 8 » ou « 1.8 »
 - Ce n'est plus le cas, Java 9 s'appelle « 9 »

INTRODUCTION

- A partir de Java9, une release tous les 6 mois
- Pour vérifier les nouveautés, comparer entre versions
 - <https://javaalmanac.io/>

INTRODUCTION

- Java8 a apporté le plus de changements
 - Interfaces
 - Comportements
 - Interfaces fonctionnelles
 - Expressions lambda
 - Références de méthodes
 - Variables finales
 - Annotations répétées
 - **Stream**
 - **Optional**
 - DateTime API
 - Nouveautés Asynchrone
- C'est une version **LTS** (Long Time Support)
 - Supportée jusqu'en Mai 2026
 - Les suivantes sont Java 11 et Java 17



INTERFACES

COMPORTEMENT DANS INTERFACES

INTERFACES

- Une interface peut avoir du comportement défini
 - Grâce au mot-clé « default »
 - C'est l'inverse d'une méthode « abstract » dans une classe Abstraite

```
public interface IFaceDemo {  
    public default void demo() {  
        System.out.println("Démonstration");  
    }  
}
```

- Une interface peut aussi avoir du comportement statique défini

```
public interface IFaceDemo {  
    public static void demoStatic() {  
        System.out.println("Démonstration statique !");  
    }  
}
```

A decorative wavy line in light blue and white, resembling a stylized wave or a ribbon, runs vertically along the left side of the slide.

INTERFACES FONCTIONNELLES

STOCKER UNE MÉTHODE

INTERFACES FONCTIONNELLES

- Une interface fonctionnelle
 - Doit avoir exactement une méthode abstraite
 - Qui peut prendre autant de paramètres que nécessaire
 - Peut avoir une ou plusieurs méthodes par défaut / statique
 - Peut être annotée de **@FunctionalInterface**
 - Recommandé puisque cette annotation permet de vérifier que l'interface respecte les contraintes

```
@FunctionalInterface
public interface IFaceFonctionnelle {
    public void fnc();

    public default void demo() {
        System.out.println("Démonstration");
    }
}
```

INTERFACES FONCTIONNELLES

- Les interfaces fonctionnelles sont utilisées pour stocker une référence à une méthode
 - Peut être une référence à une méthode existante, ou un constructeur
 - En utilisant le nom de la classe puis « :: » le nom de la méthode
 - `String::toUpperCase`
 - En utilisant le nom de l'instance puis « :: » le nom de la méthode
 - `fruit::toUpperCase`
 - En utilisant le nom de la classe puis « ::new »
 - `String::new`
 - Peut être une référence à une méthode anonyme, une expression lambda
- Et appeler cette méthode stockée « plus tard » (principe de callback)

INTERFACES FONCTIONNELLES

```
public class App {  
    public static void main(String[] args) {  
        IFaceFonctionnelle demo = new Demo();  
        demo.demo();  
  
        IFaceFonctionnelle demofnc = App::methodeReferencee;  
        demofnc.fnc();  
  
        IFaceFonctionnelle demolambda = () -> System.out.println("Expression lambda !");  
        demolambda.fnc();  
    }  
  
    public static void methodeReferencee() {  
        System.out.println("Méthode référencée !");  
    }  
}
```

INTERFACES FONCTIONNELLES

- Java 8 nous met à disposition des interfaces fonctionnelles
 - **Consumer** 1 argument pas de valeur de retour
 - **Predicate** 1 argument valeur booléenne en retour
 - **Function** 1 argument valeur de retour
 - **UnaryOperator** 1 argument valeur de retour dérive de **Function**, types identiques
 - **Supplier** 0 argument valeur de retour

 - **BiConsumer** 2 arguments pas de valeur de retour
 - **BiPredicate** 2 arguments valeur booléenne en retour
 - **BiFunction** 2 arguments valeur de retour
 - **BinaryOperator** 2 arguments valeur de retour dérive de **BiFunction**, types identiques

 - **Comparable** 1 argument valeur de retour

 - **Runnable** 0 argument pas de valeur de retour
 - **Callable** 0 argument valeur de retour

INTERFACES FONCTIONNELLES

- Ces interfaces fonctionnelles sont à utiliser selon le contexte
 - Certaines fonctionnalités de Java attendent des interfaces fonctionnelles précises
 - La méthode *forEach* des collections attend un **Consumer**
 - La méthode *sort* des collections attend un **Comparator**
 - Un nouveau **Thread** peut attendre un **Runnable**

INTERFACES FONCTIONNELLES

```
public class AppRunnable {  
    public static void main(String[] args) {  
        Runnable execution = AppRunnable::run;  
  
        System.out.println(Thread.currentThread().getName());  
        new Thread(execution).start();  
    }  
  
    public static void run() {  
        System.out.println(Thread.currentThread().getName());  
    }  
}
```



LAMBDA

EXPRESSIONS LAMBDA

AVANT JAVA 8

- Avant Java 8, on peut déclarer une instance d'interface anonyme
 - On peut utiliser ce mécanisme pour se rapprocher des interfaces fonctionnelles et des lambdas

```
public interface BeforePredicate<T> {  
    public boolean test(T i);  
}
```

```
BeforePredicate<Integer> lambda = new BeforePredicate<Integer>() {  
    @Override  
    public boolean test(Integer i) {  
        return i > 5;  
    }  
};
```

```
System.out.println(lambda.test(1));
```


EXPRESSIONS LAMBDA

- Une expression lambda est comme une « fonction fléchée » ou une méthode anonyme
 - Qui peut prendre plus ou moins d'arguments
 - Et retourner éventuellement une valeur

EXPRESSIONS LAMBDA

- Dans le cas où il n'y a aucun paramètre

```
Supplier<String> supplier = () -> {  
    return "valeur";  
};
```

```
System.out.println(supplier.get());
```

EXPRESSIONS LAMBDA

- Dans le cas où il y a un paramètre
 - Les parenthèses sont optionnelles

```
Function<Integer, Integer> function = (arg) ->
{
    return arg * 2;
};
```

```
System.out.println(function.apply(5));
```

```
Function<Integer, Integer> function = arg -> {
    return arg * 2;
};
```

```
System.out.println(function.apply(5));
```

EXPRESSIONS LAMBDA

- Dans le cas où il y a plus d'un paramètre

```
BinaryOperator<Integer> binaryOperator = (a, b) -> {  
    return a * b;  
};  
  
System.out.println(binaryOperator.apply(5, 10));
```

EXPRESSIONS LAMBDA

- Si le comportement est composé que d'une seule instruction
 - On peut omettre les accolades et le mot-clé « return » est interdit
 - Puisqu'il est en fait implicite et sera utilisé si nécessaire

```
BinaryOperator<Integer> binaryOperator = (a, b) -> a * b;  
System.out.println(binaryOperator.apply(5, 10));
```

- S'écrit de la même syntaxe, valeur de retour ou non

```
Consumer<String> consumer = arg -> System.out.println(arg);  
consumer.accept("Démo");
```

- Et il est possible d'utiliser une référence à une méthode, puisqu'une Lambda en est une

```
Consumer<String> consumer = System.out::println;  
consumer.accept("Démo");
```

EXPRESSIONS LAMBDA

- Même exemple qu'avant Java 8, en utilisant les lambdas cette fois-ci

```
Predicate<Integer> lambda = i -> i > 5;  
System.out.println(lambda.test(1));
```



STREAMS

MANIPULER LES COLLECTIONS

STREAMS

- Provient de l'interface **Stream<T>**
- Encapsule une collection, sans en modifier son contenu
 - Les traitements créeront une nouvelle **Stream**, ou de nouveaux objets
- Possibilité de traitement des éléments de façon séquentielle, ou parallèle

```
List<String> lettres = Arrays.asList("a", "b", "c", "d");  
  
lettres.stream(); // Permet de créer une Stream<String> séquentielle  
lettres.stream().parallel(); // Permet de créer une Stream<String> parallèle  
lettres.parallelStream(); // Permet de créer une Stream<String> parallèle  
lettres.parallelStream().sequential(); // Permet de créer une Stream<String> séquentielle
```


STREAMS

- Quelques classes utilitaires
 - **IntStream**
 - **LongStream**
 - **DoubleStream**
 - ...

STREAMS – FILTER

- Permet de filtrer les données
- Attend un **Predicate**

```
List<String> fruits = Arrays.asList("Orange", "Pomme", "Fraise", "Tagada");  
  
fruits.stream()  
    .filter(f -> f.contains("ra"))  
    .forEach(System.out::println);
```

STREAMS – MAP

- Permet de transformer la donnée
- Attend un **Function**

```
List<String> fruits = Arrays.asList("Orange", "Pomme", "Fraise", "Tagada");  
  
fruits.stream()  
    .map(f -> f.toUpperCase())  
    .forEach(System.out::println);
```

```
List<String> fruits = Arrays.asList("Orange", "Pomme", "Fraise", "Tagada");  
  
fruits.stream()  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

STREAMS – FLATMAP

- Permet d'applatir une **Stream** de **Stream**
- Attend un **Function**

```
List<List<String>> fruits = Arrays.asList(  
    Arrays.asList("Orange", "Pomme"),  
    Arrays.asList("Pomme", "Banane")  
);  
  
fruits.forEach(System.out::println);  
  
fruits.stream()  
    .flatMap(List::stream)  
    .forEach(System.out::println);
```

STREAMS – REDUCE

- Permet de transformer la donnée
- Attend un élément initial et un **BinaryOperator** (ou sans élément initial)

```
List<String> fruits = Arrays.asList("Orange", "Pomme", "Fraise", "Tagada");  
  
String reduce = fruits.stream()  
    .reduce("INITIAL", (accumulator, fruit) -> accumulator + " - " + fruit);  
  
System.out.println(reduce);
```

```
System.out.println(  
    fruits  
        .stream()  
        .reduce(String::concat)  
        .orElse("Pas de données")  
);
```

STREAMS – COLLECT

- Permet de collecter les données
 - Sous forme de liste par exemple
- Attend un **Collector**

```
List<String> fruitsFiltered = fruits.stream()  
    .filter(f -> f.contains("ra"))  
    .collect(Collectors.toList());
```

STREAMS – ..MATCH

- La méthode *allMatch*
 - Permet de vérifier si tous les éléments correspondent au **Predicate** donné
- La méthode *anyMatch*
 - Permet de vérifier si au moins un des éléments correspond au **Predicate** donné
- Ces méthodes retournent un booléen

STREAMS – FIND..

- D'autres fonctionnalités permettent d'extraire un élément
 - Le premier élément, avec *findFirst*
 - Le premier élément d'une **Stream** parallèle, avec *findAny*, pour ne pas altérer les performances
- Retournent des **Optional<T>**



OPTIONAL

ENCAPSULER UN ÉLÉMENT

OPTIONAL

- Classe qui permet d'encapsuler un élément
 - Evite la manipulation de référence **NULL**
 - Permet de retourner une valeur par défaut, ou une **Exception**

OPTIONAL

```
List<String> fruits = Arrays.asList("Orange", "Pomme", "Fraise", "Tagada");

Optional<String> optFruit = fruits.stream()
    .filter(f -> f.contains("ra"))
    .findAny();

if (optFruit.isPresent()) {
    System.out.println("Fruit trouvé : " + optFruit.get());
}
```

```
List<String> fruits = Arrays.asList("Orange", "Pomme", "Fraise", "Tagada");

Optional<String> optFruit = fruits.stream()
    .filter(f -> f.contains("sra"))
    .findAny();

System.out.println("Fruit trouvé : " + optFruit.orElse("Défaut"));
```

OPTIONAL

- Pour fabriquer un **Optional**
 - Utiliser les méthodes statiques *of*, *empty*, ou *ofNullable*

```
Optional<String> opt = Optional.empty();
```

```
Optional<String> opt = Optional.of("Démo");
```

- Retournera une **Exception**

```
Optional<String> opt = Optional.of(null);
```

- Ne retournera pas une **Exception**

```
Optional<String> opt = Optional.ofNullable(null);
```

```
Optional<String> opt = Optional.ofNullable("Démo");
```



DATE & HEURE

API DATETIME

DATETIME

- Selon les besoins, classes **LocalDate**, **LocalTime** ou **LocalDateTime**
 - Contrairement aux « anciennes » classes **Date** et **Calendar**
 - Elles sont Thread-safe et immuable

```
LocalDate date = LocalDate.now();  
LocalTime time = LocalTime.now();  
LocalDateTime datetime = LocalDateTime.now();
```

```
LocalDate date = LocalDate.of(2022, Month.MAY, 17);
```

DATETIME

- Il est possible de faire des opérations
 - Calculer des différences, ajouter des jours, des semaines, en retirer, etc.

```
LocalDate date = LocalDate.of(2022, Month.MAY, 17);
```

```
date = date.plusDays(3);  
date = date.minus(1, ChronoUnit.WEEKS);
```

```
date.isAfter(...)  
date.isBefore(...)
```

DATETIME

- Il est possible de faire intervenir les zones avec **ZoneId** et **ZonedDateTime**

```
ZoneId.getAvailableZoneIds().forEach(System.out::println);  
ZoneId zoneId = ZoneId.of("America/Guadeloupe");  
  
System.out.println(ZonedDateTime.now(zoneId));  
System.out.println(LocalDateTime.now(zoneId));  
System.out.println(LocalDateTime.now());
```


DATETIME

- Calculer des périodes et des durées est possible grâce à
 - **Period** pour les dates (jours, mois, années)
 - **Duration** pour les heures (heures, minutes, secondes, nanosecondes)

```
LocalDate now = LocalDate.now();  
LocalDate lastYear = now.minusYears(1);  
  
System.out.println(Period.between(now, lastYear));
```

```
LocalTime now = LocalTime.now();  
LocalTime lastHour = now.minusHours(1);  
  
System.out.println(Duration.between(now, lastHour));
```

DATETIME

- Pour le formatage, utilisation de **DateTimeFormatter**

```
LocalDateTime now = LocalDateTime.now();

System.out.println(now.format(DateTimeFormatter.ISO_DATE));
System.out.println(now.format(DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm")));

System.out.println(now.format(
    DateTimeFormatter
        .ofLocalizedDateTime(FormatStyle.SHORT)
        .withLocale(Locale.FRANCE)
    ));
```



ASYNCHRONE

API COMPLETABLEFUTURE

JAVA 7 – FORK & JOIN

- Java7 a apporté le pattern *Fork and Join*
 - Avec une classe **ForkJoinPool** pour gérer un pool de Threads
 - Et des classes abstraites pour exécuter des tâches
 - **ForkJoinTask** Pour la classe « principale »
 - **RecursiveTask** Tâche qui retourne une valeur
 - **RecursiveAction** Tâche qui ne retourne pas de valeur

```
ForkJoinPool pool = new ForkJoinPool(2);  
ForkJoinTask<String> task = new CustomTask();  
  
pool.submit(task);  
System.out.println(task.join());  
  
Thread.sleep(1000);
```

JAVA 7 – FORK & JOIN

- Plusieurs fonctionnalités des tâches
 - `join`
 - Récupère le résultat du traitement, l'attend s'il n'est pas encore disponible
 - En cas d'erreur, lève une **Unchecked Exception**
 - `get`
 - Récupère le résultat du traitement, l'attend s'il n'est pas encore disponible
 - A plus d'options que `join` (Timeout par exemple)
 - En cas d'erreur, lève une **Checked Exception**
- Seront reprises par **CompletableFuture**

JAVA 7 – FORK & JOIN

- Autre fonctionnalité des tâches
 - fork Permet d'exécuter, si applicable, le traitement dans un thread

```
ForkJoinTask<String> task = new CustomTask();  
  
task.fork();  
System.out.println("Autre traitement ici ...");  
System.out.println(task.join());
```

COMPLETABLEFUTURE

- **CompletableFuture** complète **Future** (implémente son interface)
- Permet l'exécution d'un traitement asynchrone

```
public static CompletableFuture<String> helloAsync() {  
    return CompletableFuture.supplyAsync(() -> {  
        try {  
            Thread.sleep(500);  
        }  
        catch (InterruptedException e) {  
            e.printStackTrace();  
            return null;  
        }  
  
        System.out.println(Thread.currentThread().getName());  
        return "Hello";  
    });  
}
```

COMPLETABLEFUTURE

- Plusieurs fonctionnalités
 - `supplyAsync` Traitement asynchrone encapsulé
 - `thenApply` Appliquer un nouveau traitement à la suite (chainables)
 - Attend **Function** Résultat du traitement précédent
 - `thenCompose` Appliquer un nouveau traitement asynchrone à la suite (chainables)
 - Attend **Function** Résultat du traitement précédent
 - `thenAccept` Appliquer un nouveau traitement à la suite (chainables)
 - Attend **Consumer** Résultat du traitement précédent
 - `thenRun` Appliquer un nouveau traitement à la suite (chainables)
 - Attend **Runnable** Pas d'argument passé
 - `handle` Appliquer un traitement en cas d'**Exception** levée pendant les traitements
 - Attend **BiFunction** Résultat du traitement précédent et **Exception** levée

COMPLETABLEFUTURE

```
CompletableFuture<String> completableFuture = helloAsync()  
    .thenApply(s -> s + " World")  
    .thenApply(s -> s + " !");  
  
System.out.println(completableFuture.get());
```

COMPLETABLEFUTURE

- Est capable de combiner plusieurs **CompletableFuture** avec la méthode `allOf`
 - Les traitements à suivre vont s'effectuer lorsque tous les **CompletableFuture** seront résolus