

# Dernière fonctionnalités utiles du JDK 17

Je n'ai ici gardé que les plus importants changements à utiliser.

## Depuis JDK 8

### Lambdas et interfaces fonctionnelles

Les lambdas qui sont des fonctions anonymes stockées dans des variables. Et les interfaces fonctionnelles sont des interfaces n'ayant qu'une seule méthode pour une lambda.

Grâce à cela, on peut avoir des types de fonctions plus précis et rendre le code plus sûr.

Voici un exemple d'interface fonctionnelle :

```
// Définition d'une interface fonctionnelle "MyFunction" ayant une fonction prenant en argument 2 int et en
retournant 1
@FunctionalInterface
interface MyFunction {
    int apply(int x, int y);
}

// On peut ensuite créer une lambda suivant cette interface
// Le type est donc MyFunction, les arguments sont x et y et le corps de la méthode est après la flèche
MyFunction add = (x, y) -> x + y;

// On peut ensuite utiliser notre fonction
System.out.println(add.apply(1,1)); // Ceci va afficher "2"
```

# Default methods in interfaces

Maintenant on peut écrire un code par défaut dans les interfaces (rendant donc l'implémentation de ces méthodes par défaut optionnelle).

```
public interface Vehicle {  
    // Ceci sont des méthodes d'interface normales, il est donc obligatoire de les écrire pour implémenter  
    l'interface  
    String getBrand();  
    String speedUp();  
    String slowDown();  
  
    // Ceci sont des méthodes ayant un code par défaut, il n'est donc pas obligatoire de les écrire pour  
    implémenter l'interface  
    default String turnAlarmOn() {  
        return "Turning the vehicle alarm on.";  
    }  
  
    default String turnAlarmOff() {  
        return "Turning the vehicle alarm off.";  
    }  
}
```

## Date and time API

Avant Java 8 il n'y avait pas d'API par défaut pour gérer le temps et les dates. Maintenant il y en a une. Vous pouvez avoir tous les détails dans [la Javadoc de java.time](#).

A partir de Java 8 on peut simplement récupérer la date et l'heure actuelle en faisant

```
LocalDateTime.now()
```

## Stream API

Pour expliquer plus en détails les avantages de la stream API, voici quelques exemples de code avec et sans stream.

Les streams API sont assez important et sont attendu à l'AI.

Vous pouvez apprendre à les utiliser en regardant [la Javadoc de Stream](#).

## Exemple 1

Sans stream api :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
for (int number: numbers) {
    if (number % 2 == 0) {
        System.out.println(number);
    }
}
```

Avec stream api :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

## Exemple 2

Sans stream api :

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
for (String name: names) {
    System.out.println(name.toUpperCase());
}
```

Avec stream api :

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream() // Génère le Stream
    .map(String::toUpperCase) // Passe tout en uppercase
    .forEach(System.out::println); // Print chaque élément dans la console
```

## Exemple 3

Imaginons que l'on a une liste de valeurs en nombre et en String et que l'on veut faire la moyenne tout en passant le premier élément :

Sans la stream API :

```

public static double getAverage(List<String> values) {
    double sum = 0.0; // Commence à 0
    final int counter = values.size(); // Compte le nombre d'items dans la liste
    if (list.isEmpty()) return 0; // Retourne 0 si la liste est vide pour empêcher une division par
0
    for (int i = 1; i < values.size(); i++) { // Fait une boucle passant le premier item
        String stringValue = value.get(i); // Récupère la valeur en String
        double numericValue = Double.parseDouble(stringValue); // La converti en nombre
        sum += numericValue; // L'ajoute à la somme
    }
    return sum / counter; // Fait le calcul de la somme
}

```

Avec la stream API :

```

public static double getAverage(List<String> values) {
    return values.stream() // Converti la liste en Stream
        .skip(1) // Passer le premier élément
        .mapToDouble(value -> Double.parseDouble(value)) // Tout convertir en double
        .average() // Calculer la moyenne de tout
        .orElse(0.0); // Si la liste est vide, va retourner 0
}

```

# Depuis JDK 9

## Listes immuables

On peut maintenant créer des listes (ou d'autres collections) immuables en Java avec `of` ou `copyOf` :

```
List<String> list = List.of("Hello", "World");
```

# Depuis JDK 10-11

# Inférence de type

Depuis Java 10 ou 11 on peut maintenant utiliser l'inférence de type, on est donc plus obligé de préciser le type d'une variable locale. On peut simplement utiliser le mot clé `var`

```
// Avant Java 10
String message = "Hello, World!";

// Possible après Java 10
var message = "Hello, World!";
```

## JDK 17

### Sealed classes

Les *sealed classes* permettent de limiter les classes qui peuvent hériter de la classe actuelle. Il n'est pas recommandé d'utiliser ceci pour la même raison qu'il n'est pas recommandé d'utiliser l'héritage.

```
// Seule les classes Circle et Square pourront hériter de Shape
public sealed class Shape permits Circle, Square {
}
```

### Switch expressions

Les switch peuvent maintenant fonctionner avec n'importe quel type (et pas uniquement int comme avant), de plus on peut en écrire plusieurs sur la même ligne :

```
int month = 2;
int daysInMonth = switch (month) {
    case 1, 3, 5, 7, 8, 10, 12 -> 31;
    case 4, 6, 9, 11 -> 30;
    case 2 -> 28;
    default -> throw new IllegalArgumentException("Invalid month: " + month);
}
```

```
};
```

# Records

Les records permettent de créer très simplement des data class pour représenter certains concepts et les stocker.

Voici comment représenter une Personne avec un nom et un age avant les Records :

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public int getAge() {  
        return this.age;  
    }  
}
```

Et voici comment représenter à peu près la même chose avec un Record

```
record Person(String name, int age) {}
```

# Text blocks

Depuis JDK 17 on peut aussi simplement faire des Strings de plusieurs lignes avec `"""`.

Voici comment faire un String de 3 lignes avant JDK 17 :

```
String message = "Welcome\n How are you?\n I hope you have a nice day";
```

Et voici comment faire la même chose depuis JDK 17 :

```
String message = ""  
    Welcome  
    How are you?  
    I hope you have a nice day  
    "";
```

---

Revision #2

Created 18 September 2023 16:41:29 by SnowCode

Updated 27 September 2023 11:31:32 by SnowCode