

---

# Java Avancé

## Cours 4 : Eléments de programmation orienté fonction

---

Arsène Lapostolet & Nada Nahle

# Concepts

---

# Définition

- **Fonction de première classe** : manipuler les fonction comme des variables
- **Référence de méthode** : variable / argument qui contient une méthode
- **Interface fonctionnelle** : interface Java qui type une référence de méthode
- **Lambda** : méthode anonyme déclarée en tant qu'expression

# Exemples

```

1 Runnable printHelloWorld = () -> System.out.println("Hello Java
   World");
2
3 Consumer<String> printHelloName = (String name) ->
   System.out.println("Hello, " + name);
4
5 Function<Integer,Integer> square = (Integer number) -> number
   * number;
6
7 BiFunction<Integer, Integer,Integer> plus = (leftOperand,
   rightOperand) -> leftOperand + rightOperand;
8

```

```
9 Calculator<Integer> calculator = new Calculator<Integer>();
10
11 calculator.addOperator(
12     "+",
13     (leftOperand, rightOperand) -> leftOperand + rightOperand
14 );
15
16 BiFunction<Integer, Integer, Integer> bigDecimalPlus =
17     Integer::add;
18 calculator.addOperator(
19     "+",
20     Integer::add
21 );
```



Des questions ?

# Traitement fonctionnels des collections

---

## Interface `Stream<T>`

**Stream<T>** : abstraction d'une séquence d'éléments. On peut faire des opérations fonctionnelles dessus.

*.stream()* permet de transformer n'importe quelle collection Java en `Stream<T>`



# Opérateurs de filtrage

---

# filter

Filtrer à partir d'un prédicat

```
1 List<Employee> employees = List.of(  
2     new Employee("Shepard", 28),  
3     new Employee("Liara", 106)  
4 );  
5  
6 Stream<Employee> seniors = employees  
7     .stream()  
8     .filter(employee -> employee.getAge() >= 50);  
9  
10 // Resultat : [ Employee {name = "Liara", age = 106} ]
```

Java

# limit

Récupérer un certain nombre d'éléments

```
1 List<Employee> employees = List.of(  
2     new Employee("Shepard", 28),  
3     new Employee("Liara", 106)  
4 );  
5  
6 Stream<Employee> seniors = employees  
7     .stream()  
8     .limit(1);
```

Java

# skip

## Sauter des éléments

```
1 List<Employee> employees = List.of(  
2     new Employee("Shepard", 28),  
3     new Employee("Liara", 106),  
4     new Employee("Tali", 23)  
5 );  
6  
7 Stream<Employee> skipTwoEmployees = users  
8     .stream()  
9     .skip(2);  
10 // Resultat : [ Employee {name = "Tali", age = 23} ]
```

Java

# Opérateurs de transformation

---

# map

Associer chaque élément à un nouvel élément

```
1 List<Employee> employees = List.of(  
2     new Employee("Shepard", 28),  
3     new Employee("Liara", 106),  
4     new Employee("Tali", 23)  
5 );  
6  
7 Stream<String> employeesNames = employees  
8     .stream()  
9     .map(employee -> employee.getName())  
10 // Resultat : [ "Shepard", "Liara", "Tali" ]
```

Java

# flatMap

Applatir des collections

```
1 List<Team> teams = List.of(  
2     new Team(  
3         "First Team",  
4         List.of(  
5             new Employee("Shepard", 28),  
6             new Employee("Liara", 106),  
7             new Employee("Tali", 23)  
8         });  
9     ),  
10    new Team(  
11
```

**Java**

```
11     "Second Team",
12     List.of(
13         new Employee("Garrus", 27),
14         new Employee("Kaidan", 34),
15         new Employee("Joker", 30)
16     );
17 ),
18 );
19
20 Stream<String> employeesName = teams
21     .stream()
22     .flatMap(team -> team.getMembers().stream())
23 // Retourne Stream<Employee> qui contient les 6 employés
```



# Opérateurs terminaux

---

# Collecter sous formation de collection

Enumère le Stream sous forme d'une collection :

- Une liste `toList()`
- Un ensemble `toSet()`
- Un dictionnaire `toMap()`

# findFirst

Récupérer le premier élément de la série (retourne un `Optional<T>`).

```
1 List<Employee> employees = List.of(
2     new Employee("Liara", 106),
3     new Employee("Tali", 23)
4 );
5 Employee tali = employees
6     .stream()
7     .filter(employee -> Objects.equals(employee.getName(),
8     "Tali"))
9     .findFirst()
10    .get();
```

Java

# allMatch et anyMatch

Vérifie si les éléments valident un prédicats :

- allMatch vérifie tous les éléments valide le prédicat
- anyMatch vérifie qu'au moins un élément valide le prédicat

```
1 List<Employee> employees = List.of(  
2     new Employee("Shepard", 28),  
3     new Employee("Liara", 106),  
4     new Employee("Tali", 23)  
5 );  
6
```

**Java**

```
7  boolean areAllEmployeesAdults = employees
8      .stream()
9      .allMatch(employee -> employee.getAge() > 18);
10
11 // Resultat : true
12
13 boolean isSomeOneOlderThan100 = employees
14     .stream()
15     .allMatch(employee => employee.getAge() > 100);
16
17 // Résultat : true
```

# Comptage

- count
- min
- max



Des questions ?

# **Autre outils fonctionnels**

---



# Pattern matching

Tester une expression, pour vérifier si elle a certaines caractéristiques.

```

1 public State PerformOperation(String command) {
2     return switch (command) {
3         case "SystemTest" -> runDiagnostics();
4         case "Start" -> startSystem();
5         case "Stop" -> stopSystem();
6         case "Reset" -> resetToReady();
7         default -> throw new IllegalArgumentException("Invalid
string value for command");
8     };
9 }

```

Java

# Record class

Classe qui représente des objets-valeur :

- Syntaxe plus concise
- `equals()`, `hashCode()` et `toString()` générés automatiquement
- Immutable

```
1 record Rectangle(double length, double width) {  
2  
3 }
```

Java