Programmation Fonctionnelle en Java : Les Bases

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 Objectifs d'apprentissage 1
2 Introduction à la Programmation Fonctionnelle en Java 2
  2.1 Comparaison 2
  2.2 Avantages de la version fonctionnelle 4
3 Les Interfaces Fonctionnelles 4
4 Les Lambdas Expressions 5
5 Intêret des Lambdas Expressions 6
6 Les Records (Java 14+) 6
7 Référence vers des méthodes `::` 7
8 Un exemple concret : Le tri sans lambdas 8
  8.1 Exemple: Trier en implantant `Comparator` 8
  8.2 Exemple: Trier avec une `classe anonyme` 9
  8.3 Exemple: Trier avec une `lambda` 10
  8.4 Exemple: Avec l'interface `Comparator` 10
9 Les interfaces fonctionnelles usuelles 11
  9.1 Function`<T,R>` (apply) 11
  9.2 Supplier`<T>` (get) 12
  9.3 Consumer`<T>` (accept) 14
  9.4 Predicate`<T>` (test) 14
10 Applications à la liste de personnes 15

1 Objectifs d'apprentissage

À l'issue de cette leçon, vous serez capable de :

- **OOC-FUNC-01** — Comparer des écritures impérative et fonctionnelle d'un même traitement et justifier les avantages de la programmation fonctionnelle en Java.

- **OOC-FUNC-02** — Reconnaître une interface fonctionnelle, y compris les types primitifs spécialisés, et rédiger le lambda ou la référence de méthode (`::`) correspondant.
- **OOC-FUNC-03** — Décrire les records comme porteurs de données immuables et écrire des déclarations adaptées à un traitement fonctionnel.
- **OOC-FUNC-04** — Réécrire en style fonctionnel des opérations sur une liste en mobilisant `Function`, `Supplier`, `Consumer`, `Predicate` et `Comparator`.

```
//|echo: false
//|output: false
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

final static Logger logger = LoggerFactory.getLogger("notebook");
```

2 Introduction à la Programmation Fonctionnelle en Java

La programmation fonctionnelle, un paradigme qui considère les calculs comme l'évaluation de fonctions mathématiques, offre une nouvelle perspective sur le développement logiciel. En mettant l'accent sur les expressions, les fonctions pures (sans effets de bord) et l'immutabilité, elle favorise un code plus déclaratif et moins sujet aux erreurs. Java, avec des fonctionnalités comme les expressions lambda et les streams, permet d'adopter une approche plus fonctionnelle, simplifiant ainsi la manipulation de collections et améliorant la lisibilité du code. De plus, la nature immuable des données en programmation fonctionnelle facilite grandement la parallélisation des traitements.

2.1 Comparaison

```
// Structure de données
class Course {
    String name;
    double grade;

    Course(String name, double grade) {
        this.name = name;
        this.grade = grade;
    }
}
```

```
// Données d'exemple

class Student {
```

```
    String name;
    List<Course> courses;

    Student(String name, List<Course> courses) {
        this.name = name;
        this.courses = courses;
    }
}
```

```
List<Student> students = Arrays.asList(
    new Student("Alice", Arrays.asList(
        new Course("Math", 18.5),
        new Course("Physics", 17.0)
    )),
    new Student("Bob", Arrays.asList(
        new Course("Math", 12.0),
        new Course("Physics", 13.5)
    )),
    new Student("Charlie", Arrays.asList(
        new Course("Math", 15.0),
        new Course("Physics", 16.0)
    ))
);
```

```
// Version fonctionnelle : Trouver les noms des étudiants ayant une moyenne > 15 en Math
List<String> highPerformers = students.stream()
    .filter(student -> student.courses.stream()
        .filter(course -> course.name.equals("Math"))
        .mapToDouble(course -> course.grade)
        .average()
        .orElse(0.0) > 15.0)
    .map(student -> student.name)
    .collect(Collectors.toList());
highPerformers;
```

```
[Alice]
```

```java
// Version impérative équivalente
List<String> highPerformersImperative = new ArrayList<>();
for (Student student : students) {
    double mathSum = 0.0;
    int mathCount = 0;
    for (Course course : student.courses) {
        if (course.name.equals("Math")) {
            mathSum += course.grade;
            mathCount++;
        }
    }
    if (mathCount > 0 && (mathSum / mathCount) > 15.0) {
        highPerformersImperative.add(student.name);
    }
}
highPerformersImperative
```

```
[Alice]
```

2.2 Avantages de la version fonctionnelle

1. **Lisibilité et Intent**
   - ✓ **Fonctionnel:** La chaîne d'opérations reflète directement l'intention du développeur
   - × **Impératif:** Multiples boucles et conditions imbriquées
2. **Concision et Maintenabilité**
   - ✓ **Fonctionnel:** Une seule expression fluide
   - × **Impératif:** Variables temporaires, compteurs, accumulateurs
3. **Immutabilité et Sûreté**
   - ✓ **Fonctionnel:** Pas d'effets de bord, thread-safe par défaut
   - × **Impératif:** État mutable partagé, risques de race conditions
4. **Composabilité et Réutilisation**
   - ✓ **Fonctionnel:** Fonctions comme briques de base combinables
5. **Parallélisation et Performance**
   - ✓ **Fonctionnel:** .parallel() suffit pour paralléliser
   - × **Impératif:** Synchronisation manuelle complexe
6. **Debugging et Testing**
   - ✓ **Fonctionnel:** Fonctions pures facilement testables
   - × **Impératif:** Tests complexes dus aux états mutables

3 Les Interfaces Fonctionnelles

Définition :

Une interface fonctionnelle en Java est une interface qui possède exactement une méthode abstraite. Ce concept, introduit dans Java 8, est étroitement lié à la programmation fonctionnelle et permet de traiter les fonctions comme des valeurs de première classe. Les interfaces fonctionnelles sont largement utilisées pour représenter des comportements tels que des fonctions, des actions ou des prédicats.

Une interface fonctionnelle en Java est une interface qui ne possède qu'une seule méthode abstraite. Elle sert de type cible pour les expressions lambda et les références de méthode, permettant ainsi d'écrire du code plus concis et expressif.

Principales interfaces fonctionnelles :

- Interfaces génériques :
  - Function<T, R> : Applique une fonction à un argument de type T et retourne un résultat de type R.
  - Consumer : Consomme un argument de type T (effectue une action sans retour).
  - Predicate : Représente une condition à vérifier sur un élément de type T.
  - Supplier : Fournit un élément de type T.
- Interfaces pour les types primitifs :
  - IntFunction, IntConsumer, IntPredicate, IntSupplier : Variantes des interfaces génériques pour les entiers.
  - LongFunction, LongConsumer, LongPredicate, LongSupplier : Variantes pour les nombres à virgule flottante de type long.
  - DoubleFunction, DoubleConsumer, DoublePredicate, DoubleSupplier : Variantes pour les nombres à virgule flottante de type double.
  - BooleanSupplier : Fournit une valeur booléenne.

Elles peuvent être vues comme des instances de classes anonymes n'ayant qu'une seule méthode dont les types de retour et des paramètres sont inférés.

4 Les Lambdas Expressions

Définition: Une lambda expression est une fonction anonyme qui peut être assignée à une variable. C'est une manière concise d'écrire une instance d'une interface fonctionnelle.

Elles peuvent être vues comme des instances de classes anonymes n'ayant qu'une seule méthode dont les types de retour et des paramètres sont inférés.

- **Syntaxe:**

```
(paramètres) → corps
```

  Une lambda expression est composée :

- d'un ou plusieurs noms de paramètres (entre parenthèses), éventuellement typés
- du symbole → (flèche lambda)
- d'un corps de fonction. Si le corps se réduit à une seule expression, le `return` et les accolades sont facultatifs.

5 Intêret des Lambdas Expressions

- **Pourquoi utiliser les expressions lambda ?**
  - **Concision:** Écriture de code plus compacte et lisible.
  - **Programmation fonctionnelle:** S'intègrent parfaitement avec les concepts de la programmation fonctionnelle (map, filter, reduce, etc.).
  - **Streams:** Utilisées avec les streams pour effectuer des opérations sur des collections.
- **Lien avec les interfaces fonctionnelles:** Une expression lambda est toujours associée à une interface fonctionnelle. Le compilateur infère le type de l'interface en fonction du contexte.

```
//| echo: true
//| output: true
import java.util.function.*;
DoubleBinaryOperator add = (x, y) -> x + y;  // Additionne deux nombres
DoubleBinaryOperator mul = (x, y) -> x * y;  // Additionne deux nombres

void apply(DoubleBinaryOperator operator, double x, double y) {
    logger.info("Résultat: " + operator.applyAsDouble(x, y));
}

apply(add, 2.0, 3.0);
apply(mul, 2.0, 3.0);
apply((x, y) -> x * x + y * y, 2.0, 3.0);
```

```
18:36:25.889 [IJava-executor-1] INFO  notebook -- Résultat: 5.0
18:36:25.906 [IJava-executor-1] INFO  notebook -- Résultat: 6.0
18:36:25.924 [IJava-executor-1] INFO  notebook -- Résultat: 13.0
```

6 Les Records (Java 14+)

Depuis l'introduction de Java 14, les développeurs disposent d'un nouveau mécanisme pour définir des classes dont le but premier est de contenir des données : les records. Un record est une classe spéciale, conçue pour représenter des valeurs immuables de manière concise et expressive.

Contrairement aux classes traditionnelles qui peuvent encapsuler à la fois des données et des comportements, les records se concentrent sur l'encapsulation de données. Ils sont particulièrement adaptés pour modéliser des entités qui ne nécessitent pas de méthodes complexes au-delà des méthodes d'accès aux données (getters).

Pourquoi utiliser des records ?

- **Concision:** La syntaxe des records est plus concise que celle des classes classiques, ce qui améliore la lisibilité du code.
- **Immuabilité:** L'immuabilité par défaut des records favorise la création de code plus sûr et plus prévisible.
- **Génération automatique:** Le compilateur génère automatiquement des méthodes essentielles comme `equals()`, `hashCode()` et `toString()`, basées sur les champs du record.
- **Lisibilité:** La structure simple des records facilite la compréhension du code, en particulier lorsqu'il s'agit de représenter des données de manière structurée.

```
//| echo: true
//| output: true
public record Person(String nom, int age) {}

//Immutable List of Records
List<Person> personnes = List.of(
    new Person("Alice", 30),
    new Person("Bob", 25)
    // ...
);
```

7 Référence vers des méthodes `::`

L'opérateur `::` permet de récupérer une référence vers un constructeur, une méthode statique ou d'instance existante que l'on pourra manipuler à travers une interface fonctionnelle.

On peut ainsi définir une méthode qui ajoute des Personnes à une liste qui est créée à partir d'un Supplier qui fournit une liste de Personne en paramètre.

```
//| echo: true
//| output: true
// Define two Supplier<List> lambdas:
Supplier<List> listFactory1 = ArrayList::new;  // Creates a Supplier that provides
a new ArrayList
```

```
Supplier<List> listFactory2 = LinkedList::new;   // Creates a Supplier that
provides a new LinkedList

// Function to initialize a List of Person objects:
public List<Person> initPersonList(Supplier<List> listFactory) {
  // Get a new List instance from the provided Supplier
  List<Person> personList = listFactory.get();

  // Add some Person objects to the list
  personList.add(new Person("Alice", 30));
  personList.add(new Person("Bob", 25));

  // Return the populated list
  return personList;
}

// Demonstrate usage with different Suppliers:
logger.info("List using ArrayList: {}", initPersonList(listFactory1));
logger.info("List using LinkedList: {}", initPersonList(listFactory2));
```

```
18:36:35.432 [IJava-executor-1] INFO   notebook -- List using ArrayList:
[Person[nom=Alice, age=30], Person[nom=Bob, age=25]]
18:36:35.452 [IJava-executor-1] INFO   notebook -- List using LinkedList:
[Person[nom=Alice, age=30], Person[nom=Bob, age=25]]
```

8 Un exemple concret : Le tri sans lambdas

Avec une liste d'instances de la classe Person(prenom, age), si la classe redéfini `equals` et `hashcode` et implante l'interface `Comparable` en fonction du prenom on peut trier « canoniquement » la liste par nom avec `Collections.sort(personnes);` (impossible ici avec un record).

8.1 Exemple: Trier en implantant `Comparator`

Étant donnée une liste d'objets `Person` (possédant les attributs "prénom" et "âge"), si la classe `Person` redéfinit les méthodes `equals` et `hashCode` et implémente l'interface `Comparable` en se basant sur le prénom, il est possible de trier cette liste par ordre alphabétique de prénoms en utilisant la méthode `Collections.sort(personnes)`. Cependant, cette approche n'est pas directement applicable aux records.

Pour trier des records ou par un autre critère par exemple l'âge, on peut implanter l'interface `Comparator`

```
//| echo: true
//| output: true
public class PersonComparatorByAge
  implements Comparator<Person> {
    public int compare(Person person1, Person person2) {
        return person1.age()-person2.age();
    }
}
```

et en fournir une instance à la méthode de classe `sort`.

```
//| echo: true
//| output: true
List<Person> listByAges = new ArrayList<Person>(personnes);
Collections.sort(listByAges, new PersonComparatorByAge());
listByAges;
```

```
[Person[nom=Bob, age=25], Person[nom=Alice, age=30]]
```

8.2 Exemple: Trier avec une `classe anonyme`

Si le comparateur ne doit être utilisé qu'une fois on peut utiliser une classe anonyme.

```
//| echo: true
//| output: true
//Tri par nom, puis age
List<Person> listByNomAge = new ArrayList<Person>(personnes);
Collections.sort(listByNomAge, new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
      return o1.nom()
        .equals(o2.nom())
           ?o1.age()-o2.age()
             :o1.nom().compareTo(o2.nom());
    }
});

listByNomAge;
```

```
[Person[nom=Alice, age=30], Person[nom=Bob, age=25]]
```

8.3 Exemple: Trier avec une `lambda`

```
//| echo: true
//| output: true

// Tri par nom en utilisant une lambda expression
List<Person> listByNames = new ArrayList<Person>(personnes);
Collections.sort(listByNames, (o1, o2) -> o1.nom().compareTo(o2.nom()));
logger.info("Tri par nom: {}",listByNames);

// Tri par nom en utilisant une clé de tri
List<Person> listByAges = new ArrayList<Person>(personnes);
listByNames.sort(Comparator.comparing(Person::nom));
logger.info("Tri par age: {}",listByNames);

// Tri par âge
List<Person> listByAges = new ArrayList<Person>(personnes);
listByAges.sort(Comparator.comparingInt(Person::age));
logger.info("{}",listByAges);
```

```
18:37:02.832 [IJava-executor-1] INFO  notebook -- Tri par nom: [Person[nom=Alice,
age=30], Person[nom=Bob, age=25]]
18:37:02.876 [IJava-executor-1] INFO  notebook -- Tri par age: [Person[nom=Alice,
age=30], Person[nom=Bob, age=25]]
18:37:02.913 [IJava-executor-1] INFO  notebook -- [Person[nom=Bob, age=25],
Person[nom=Alice, age=30]]
```

8.4 Exemple: Avec l'interface `Comparator`

L'interface `Comparator` fournit de nombreuses méthodes de comparaisons fonctionnelles :

```
//| echo: true
//| output: true
// Sort list of Person objects in reverse order:
// 1. Primary sort by name (nom)
// 2. Secondary sort by age for same names
Collections.sort(listByNomAge, Comparator.comparing(Person::nom)
                                         .thenComparing(Person::age)  // Break
ties by age
                                        .reversed());                // Reverse
the entire ordering
// Return the sorted list (descending by name then age)
listByNomAge;
```

```
[Person[nom=Bob, age=25], Person[nom=Alice, age=30]]
```

9 Les interfaces fonctionnelles usuelles

Nous allons voir des exemples avec quatre interfaces courament utilisées : `Function`, `Predicate`, `Supplier` et `Consumer`.

9.1 Function<T,R> (apply)

L'interface Function applique un traitement sur son unique paramètre.

Le nom de l'unique méthode d'une `Function` est `apply`.

- Utilisation :
  - Transformer des données: Convertir des types, appliquer des règles métier.
  - Composer des fonctions: Enchaîner plusieurs transformations.
  - Passer des comportements en paramètre: Rendre du code plus flexible et réutilisable.
- `.andThen()` method
  - Takes another Function as parameter
  - Returns a new composed Function
  - Executes functions in order: first the original, then the parameter
- `.compose()` executes functions in reverse order:

```
//| echo: true
//| output: true
import java.util.function.Function;
// Email formatter functions
// 1. Basic Functions
Function<String, String> addDomain = name → name + "@company.com";
Function<String, String> makeValidEmail = String::toLowerCase;

// 2. Composition with andThen
Function<String, String> createEmail = makeValidEmail.andThen(addDomain);

// Test with different inputs
System.out.println("Simple: " + addDomain.apply("john.doe"));
System.out.println("Lowercase: " + makeValidEmail.apply("JOHN.DOE"));
System.out.println("andThen: " + createEmail.apply("John.Doe"));
System.out.println("compose:                                                    "
+ addDomain.compose(makeValidEmail).apply("JOHN.DOE"));
```

```
Simple: john.doe@company.com


Lowercase: john.doe
andThen: john.doe@company.com
compose: john.doe@company.com
```

On peut alors définir des méthodes qui prennent en paramètre une instance d'une interface fonctionnelle.

Ici une méthode qui parcourt la liste de personnes, applique la Function sur le prénom et l'affiche.

```
//| echo: true
//| output: true
public class AfficheurDePersonnes {
  public static void appliqueEtAfficher(Function<String,String> fonction) {
    for (Person p : personnes) System.out.println(fonction.apply(p.nom()));
  }
}

AfficheurDePersonnes.appliqueEtAfficher(createEmail);
```

```
alice@company.com
bob@company.com
```

On peut aussi directement utiliser une lambda de types compatibles (noter que les types de retour et du paramètre `s` sont inférés à partir des paramètres de l'interface fonctionnelle attendue).

```
//| echo: true
//| output: true
AfficheurDePersonnes.appliqueEtAfficher(s→s.toUpperCase());
```

```
ALICE
BOB
```

9.2 Supplier<T> (get)

Un Supplier est une fonction sans paramètre qui produit une valeur et possède donc un seul paramètre générique (le type de l'objet retourné). C'est une sorte de factory.

Ainsi `Supplier<String>` fournis des chaines de caratères, `Supplier<Integer>` des objets représentant des entiers, ...

Utilisation :

- Création d'objets : Générer des instances d'une classe.
- Fourniture de valeurs par défaut: Définir des valeurs initiales. Génération de données aléatoires: Produire des nombres aléatoires, des chaînes aléatoires.

Pour les primitifs, il existe des variantes dédiées de l'interface (donc sans paramètres) comme IntSupplier.

```
//| echo: true
//| output: true
import java.util.function.Supplier;
Random random = new Random();

// Different dice types using method reference
Supplier<Integer> D6 = () -> random.nextInt(6) + 1;
Supplier<Double> RandomTemperature = () -> random.nextDouble()*50 - 19;

System.out.println("D6 Roll: " + D6.get());
System.out.println("Random Temperature: " + RandomTemperature.get());

public class Statistique<T extends Number> {
    public static <T extends Number> double average(Supplier<T> supplier, int
iterations) {
        double sum = 0.0;
        for (int i = 0; i < iterations; i++) {
            sum += supplier.get().doubleValue();
        }
        return sum / iterations;
    }
}
final int NB = 10;
logger.info("Random    dices:    {}",    Statistique.average(D6,    NB)+"    (%d
rolls)".formatted(NB));
logger.info("Random  temperatures:  {}",  Statistique.average(RandomTemperature,
NB)+" (%d temperatures)".formatted(NB));
logger.info("Random 0-100: {}", Statistique.average(()->random.nextInt(100)+1,
NB)+" (%d numbers)".formatted(NB));
```

```
D6 Roll: 2
Random Temperature: -8.676348852325994
```

```
18:44:12.812 [IJava-executor-5] INFO  notebook -- Random dices: 4.8 (10 rolls)
18:44:12.823  [IJava-executor-5]  INFO    notebook  --  Random  temperatures:
11.28101560265375 (10 temperatures)
18:44:12.837 [IJava-executor-5] INFO  notebook -- Random 0-100: 62.3 (10 numbers)
```

9.3 Consumer<T> (accept)

L'interface `Consumer<T>` est une composante fondamentale de la programmation fonctionnelle en Java, introduite pour gérer les opérations qui acceptent une entrée sans produire de sortie. Cette interface fait partie du package java.util.function et est particulièrement utile pour les traitements terminaux dans les flux de données.

Son principe de base repose sur une seule méthode abstraite accept(T t) qui prend un paramètre de type T et ne retourne rien (void). Cette caractéristique en fait l'outil idéal pour les opérations qui produisent des effets

```
//| echo: true
//| output: true
// Create list of names
List<String> names = List.of("Alice", "Bob", "Charlie");

// Define two consumers
Consumer<String> upperCase = name -> System.out.print(name.toUpperCase());
Consumer<String> greet = name -> System.out.println(" -> Hello " + name);

// Chain consumers with andThen
Consumer<String> process = upperCase.andThen(greet);

// Process each name
// The forEach method provides a functional approach to iteration, making code
more concise and focusing on what to do with elements rather than how to iterate.
names.forEach(process);
```

```
ALICE -> Hello Alice
BOB -> Hello Bob
CHARLIE -> Hello Charlie
```

9.4 Predicate<T> (test)

L'interface fonctionnelle `Predicate<T>` représente un concept fondamental de la programmation fonctionnelle en Java : l'évaluation d'une condition. À travers sa méthode abstraite `test(T t)`, elle permet d'encapsuler une logique de test qui, appliquée à un objet de type `T`, retourne un résultat booléen. Cette interface s'avère particulièrement

puissante grâce à ses méthodes de composition par défaut (`and`, `or`, `negate`) qui permettent de combiner plusieurs prédicats pour créer des conditions complexes.

Les cas d'utilisation du Predicate sont nombreux et variés dans le développement Java moderne. On le retrouve principalement dans les opérations de filtrage de collections, où il permet de sélectionner précisément les éléments répondant à des critères spécifiques. Il joue également un rôle central dans la validation de données, permettant d'exprimer des règles métier de manière claire et composable. Enfin, son utilisation dans les structures conditionnelles apporte une flexibilité accrue en permettant de définir et manipuler des conditions comme des objets de première classe.

```
//| echo: false
import java.util.function.Predicate;
```

```
//| echo: true
//| output: true
// Name validation predicates
Predicate<String> startsWithCapital = name → name.matches("[A-Z][a-z]+");
Predicate<String> hasValidLength = name → name.length() ⩾ 2 && name.length()
⩽ 30;
Predicate<String> noSpecialChars = name → name.matches("[A-Za-z]+");

// Combine predicates
Predicate<String> isValidName = startsWithCapital
    .and(hasValidLength)
    .and(noSpecialChars);

// Test
System.out.println("Is 'John' valid? " + isValidName.test("John"));
System.out.println("Is 'j0hn' valid? " + isValidName.test("j0hn"));
```

```
Is 'John' valid? true
Is 'j0hn' valid? false
```

10 Applications à la liste de personnes

Ces functions sont très utilises pour construire efficacement des traitements sur des séquences de données. Avant le cours, sur les flux de données (`Stream`). Voilà une illustration pour construire un traitement générique d'une liste d'élément: filtrage en fonction d'un condition passée en paramètre, transformation des éléments de la liste et application d'un traitement a!ux résultats.

```
//| echo: true
//| output: true
import java.util.function.Predicate;
public class Processor<T, R> {
    public void run(Iterable<T> iterable, Predicate<T> predicate,  Function<T,
R> function, Consumer<R> consumer) {
        iterable.forEach(t -> {
            if (predicate.test(t)) {
                R result = function.apply(t);
                consumer.accept(result);
            }
        });
    }
}
```

---

et utiliser une lambda expression pour le prédicat et une référence vers une fonction pour afficher le résultat

```
//| echo: true
//| output: true
List<Person> people = List.of(
   new Person("Alice", 25),
   new Person("Bob", 17),
   new Person("Charlie", 30),
   new Person("Alfred", 16)
);

// Multiple predicates
Predicate<Person> isAdult = p -> p.age() >= 18;
Predicate<Person> nameStartsWithA = p -> p.nom().startsWith("A");

// Multiple functions
Function<Person, String> formatName = p -> p.nom().toUpperCase();
Function<String, String> addTitle = p -> "Mr/Ms. " + p;

// Multiple consumers
Consumer<String> print = System.out::println;
Consumer<String> log = s -> logger.info(s);

// Process the data
Processor<Person, String> processor = new Processor<>();
processor.run(people,
   isAdult
```

```
      .and(nameStartsWithA),
   formatName
      .andThen(addTitle),
   print
      .andThen(log)
);
```

```
Mr/Ms. ALICE
18:46:39.465 [IJava-executor-6] INFO  notebook -- Mr/Ms. ALICE
```

La même classe générique `Processor` peut être utilisée pour filtrer des entiers.

```
//| echo: true
//| output: true
// Sample data
List<Integer> numbers = List.of(1, 15, 30, 45, 60, 75, 90);

// Multiple predicates
Predicate<Integer> isGreaterThan30 = n -> n > 30;
Predicate<Integer> isDivisibleBy15 = n -> n % 15 == 0;

// Multiple functions
Function<Integer, String> formatNumber = n -> String.format("Number: %d", n);
Function<String, String> addComment = s -> s + " (multiple of 15)";

// Multiple consumers
Consumer<String> print = System.out::println;
Consumer<String> log = s -> logger.info(s);

// Process the data
Processor<Integer, String> processor = new Processor<>();
processor.run(numbers,
    isGreaterThan30
        .and(isDivisibleBy15),
    formatNumber
        .andThen(addComment),
    print
        .andThen(log)
);
```

```
Number: 45 (multiple of 15)
18:46:52.825 [IJava-executor-6] INFO  notebook -- Number: 45 (multiple of 15)
```

```
Number: 60 (multiple of 15)
18:46:52.825 [IJava-executor-6] INFO  notebook -- Number: 60 (multiple of 15)
Number: 75 (multiple of 15)
18:46:52.825 [IJava-executor-6] INFO  notebook -- Number: 75 (multiple of 15)
Number: 90 (multiple of 15)
18:46:52.825 [IJava-executor-6] INFO  notebook -- Number: 90 (multiple of 15)
```

```

```