

Cheatsheet — Java fonctionnel : Streams, Lambdas, Optionals

Résumé compact pour TP et exercices

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

2026-10-14

Table des matières

1 Raccourcis essentiels	1
2 Lambdas & références de méthode	1
3 Stream API — motifs courants	1
4 Optionals — usage bref	2
5 Patterns utiles	2
6 Performance / bonnes pratiques	2
7 Commandes & snippets utiles	2
8 Références rapides	2

```
Function<String,Integer> len = s ->
s.length();
Function<String,Integer> len2 =
String::length;

@FunctionalInterface
interface Mapper<T,R> { R map(T t); }
Mapper<String,Integer> m =
String::length;
```

1 Raccourcis essentiels

- Entrée : `public static void main(String[] args) ou main(String... args)`
- Import
utiles : `java.util.*`, `java.util.stream.*`,
`java.util.function.*`

Quand écrire une interface fonctionnelle :

```
@FunctionalInterface
interface Fn<T,R> { R apply(T t); }
```

Privilégiez les interfaces standard
(Function, Predicate, Consumer, Supplier,
UnaryOperator, BinaryOperator).

2 Lambdas & références de méthode

Syntaxe :

```
(x, y) -> x + y // lambda
String::length // référence
de méthode
```

Exemples rapides :

3 Stream API — motifs courants

- Créer un stream : `collection.stream()` ou `Stream.of(...)`
- Opérations intermédiaires (paresseuses) :
`filter`, `map`, `distinct`, `sorted`
- Opérations terminales : `collect`, `forEach`,
`count`, `sum`, `reduce`

Exemples :

```
List<String> names = List.of("Alice", "Bob")

List<String> aNames = names.stream()
    .filter(s -> s.startsWith("A"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());

int total = names.stream().mapToInt(String
    .length::intValue)
    .sum();

Map<Boolean, List<String>>
partitioned = names.stream()
    .collect(Collectors.partitioningBy(s
    -> s.length() > 3));
```

Collectors utiles : `toList()`, `toSet()`, `toMap()`, `joining()`, `groupingBy()`, `partitioningBy()`, `toUnmodifiableList()`.

Conseil : éviter les effets de bord dans `map/filter` et préférer les collectors pour l'accumulation.

4 Optionals — usage bref

```
Optional<String> maybe =
Optional.ofNullable(findName());
maybe.ifPresent(System.out::println);
String v = maybe.orElse("n/a");
String v2 = maybe.orElseGet(() ->
computeDefault());
String v3 = maybe.orElseThrow(() ->
new IllegalStateException());
```

Utilisez `map` / `flatMap` sur `Optional` pour enchaîner sans `null`.

5 Patterns utiles

- Méthode principale courte + stream :

```
public static void main(String[]
args) {
    List.of("a", "bb", "ccc").stream()
```

```
.map(String::length)
    .forEach(System.out::println);
}
```

- Méthode utilitaire avec `Function` :

```
Function<String, Integer> len =
String::length;
List<Integer> lens = names.stream().map(len).collect(Collectors.toList());
```

6 Performance / bonnes pratiques

- Préférer `List.of(...)` et `Collectors.toUnmodifiableList()` si immuabilité souhaitée.
- `parallelStream()` : bénéfique pour opérations CPU-bound et grandes collections, attention aux effets de bord.
- Préférer primitives `mapToInt/mapToLong` pour agrégations numériques.

7 Commandes & snippets utiles

- Générer un stream à partir d'un tableau : `Arrays.stream(arr)`
- Convertir un tableau en stream : `list.stream().toArray(String[]::new)`

8 Références rapides

- `java.util.stream` — Stream API
- `java.util.function` — interfaces fonctionnelles
- `java.util.Optional` — gestion des absences

Petit rappel : garder les opérations de pipeline pures (sans effets secondaires) facilite le raisonnement et la parallélisation.