

Bonnes pratiques avec les Lambda

L'évolution de Java

Java 8 : Tournant majeur

Introduction des
lambdas et API
Stream

Programmation fonctionnelle

Accessible aux
millions de
développeurs Java

Fonctions de première classe

Syntaxe concise,
moins de boilerplate



Avantages clés

Code concis

Expressivité et lisibilité améliorées

Parallélisation

Facilite le traitement concurrent

Composition

Fonctions réutilisables et composables

Immutabilité

Code plus robuste et prévisible

Interfaces fonctionnelles

Une seule méthode abstraite - fondation des lambdas



Predicate<T>

Test de condition : $T \rightarrow \text{boolean}$



Function<T,R>

Transformation : $T \rightarrow R$



Consumer<T>

Action sans retour : $T \rightarrow \text{void}$



Supplier<T>

Fournisseur : $() \rightarrow T$



Pattern : Lambdas simples

01

Identifier le cas d'usage

Déterminer si lambda appropriée

02

Privilégier la concision

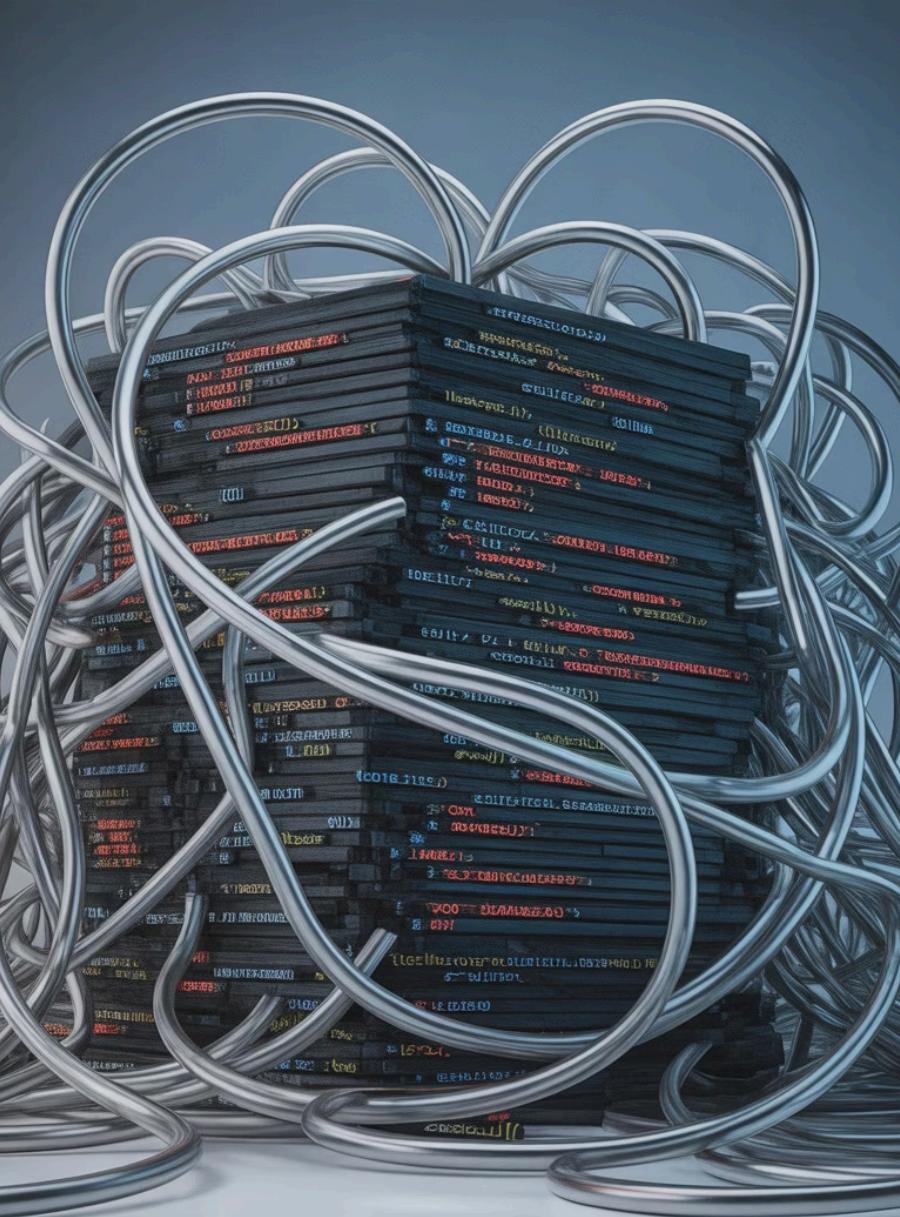
Lambdas courtes, une responsabilité

03

Syntaxe adaptée

Forme la plus lisible selon contexte

Règle d'or : Une lambda bien conçue tient sur une ligne et exprime clairement son intention



Anti-pattern : Lambdas complexes

Problème

Logique imbriquée, conditions multiples, difficile à déboguer

Solution

Extraire dans méthodes privées nommées

Bénéfice

Testabilité, lisibilité, maintenabilité

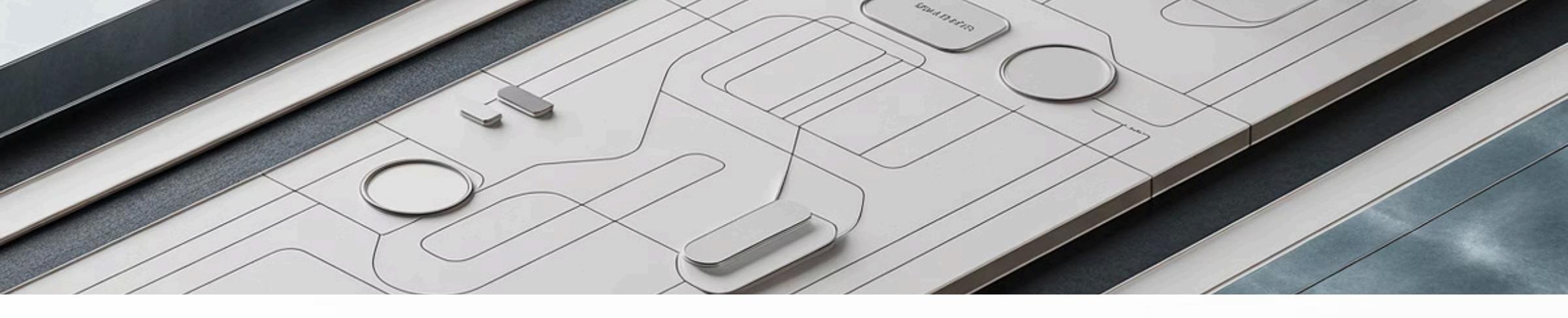
Pattern : Composition de fonctions

Concept puissant

Créer transformations complexes en combinant fonctions simples

- `andThen()` : applique fonction courante puis suivante
- `compose()` : applique paramètre puis fonction courante
- Réutilisabilité et testabilité maximales





Pattern : Prédicats composés

1

Prédicats simples

Conditions atomiques réutilisables

2

Composition logique

Combiner avec and(), or(), negate()

3

Prédicats complexes

Filtres sophistiqués maintenables



Anti-pattern : Effets de bord

Immutabilité

= Prévisibilité

Danger

Modification variables externes, état partagé, comportement non déterministe

Solution

Utiliser collect(), reduce(), opérations pures

Pattern : Utilisation des Streams



Évaluation paresseuse : optimisation automatique du pipeline



Anti-pattern : Abus des Streams

Quand éviter

- Collections <10 éléments
- Opérations nécessitant index
- Besoins de break/continue
- Logique très simple

Signaux d'alarme

- Streams imbriqués complexes
- >5 opérations intermédiaires
- Code moins lisible
- Difficulté à déboguer

Pattern : Method References



Méthode statique

`Integer::parseInt`



Instance liée

`object::toString`



Instance non liée

`String::length`



Constructeur

`ArrayList::new`

Pattern : Optional

01

Création

of(), ofNullable(), empty()

02

Transformation

map(), flatMap()

03

Filtrage

filter() pour conditions

04

Extraction

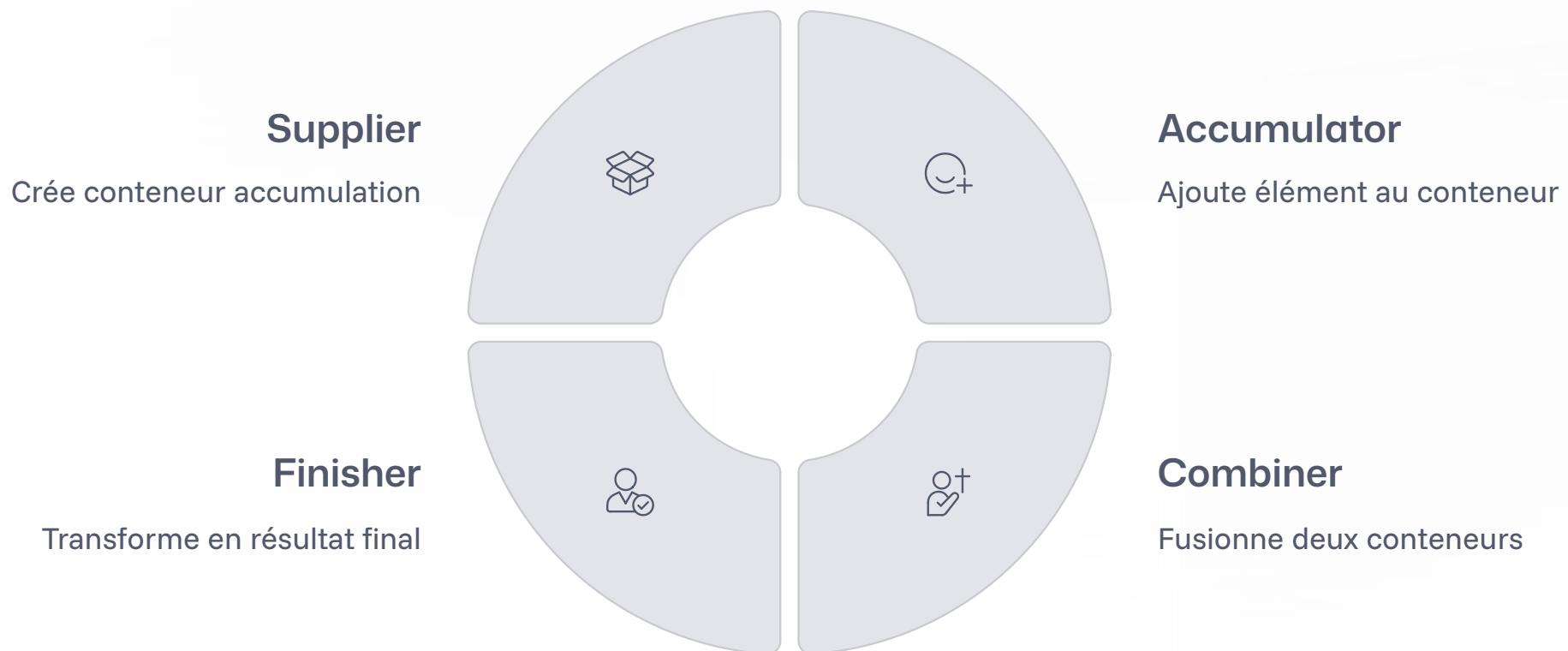
orElse(), orElseGet(), orElseThrow()



Anti-pattern : Mauvais usage d'Optional

- “ Ne jamais utiliser Optional comme champ de classe - augmente mémoire et complique sérialisation ”
- “ Éviter isPresent() + get() - manque l'intérêt d'Optional ”
- “ Ne pas passer Optional en paramètre - force création inutile ”

Pattern : Collectors personnalisés



Pattern : Gestion des exceptions

Défi

Interfaces fonctionnelles standard
ne déclarent pas exceptions
vérifiées

Besoin de wrappers élégants

Solutions

- Wrappers génériques
- Interfaces personnalisées
- Retour Optional en cas d'erreur
- Centralisation gestion erreurs



Anti-pattern : Ignorer les exceptions

Danger : Erreurs silencieuses

1

Avaler exceptions

Try-catch vides masquent erreurs

2

Logger sans action

Retour valeur défaut sans contexte

3

Solution

Gestion appropriée avec contexte

Pattern : Lazy evaluation

Définition

Encapsuler calcul dans Supplier

Transmission

Passer Supplier sans évaluer

Évaluation

Appeler get() uniquement si nécessaire

- ❑ **Performance :** orElseGet() vs orElse() - différence majeure pour calculs coûteux

Pattern : FlatMap

map()

Stream<T> → Stream<R>

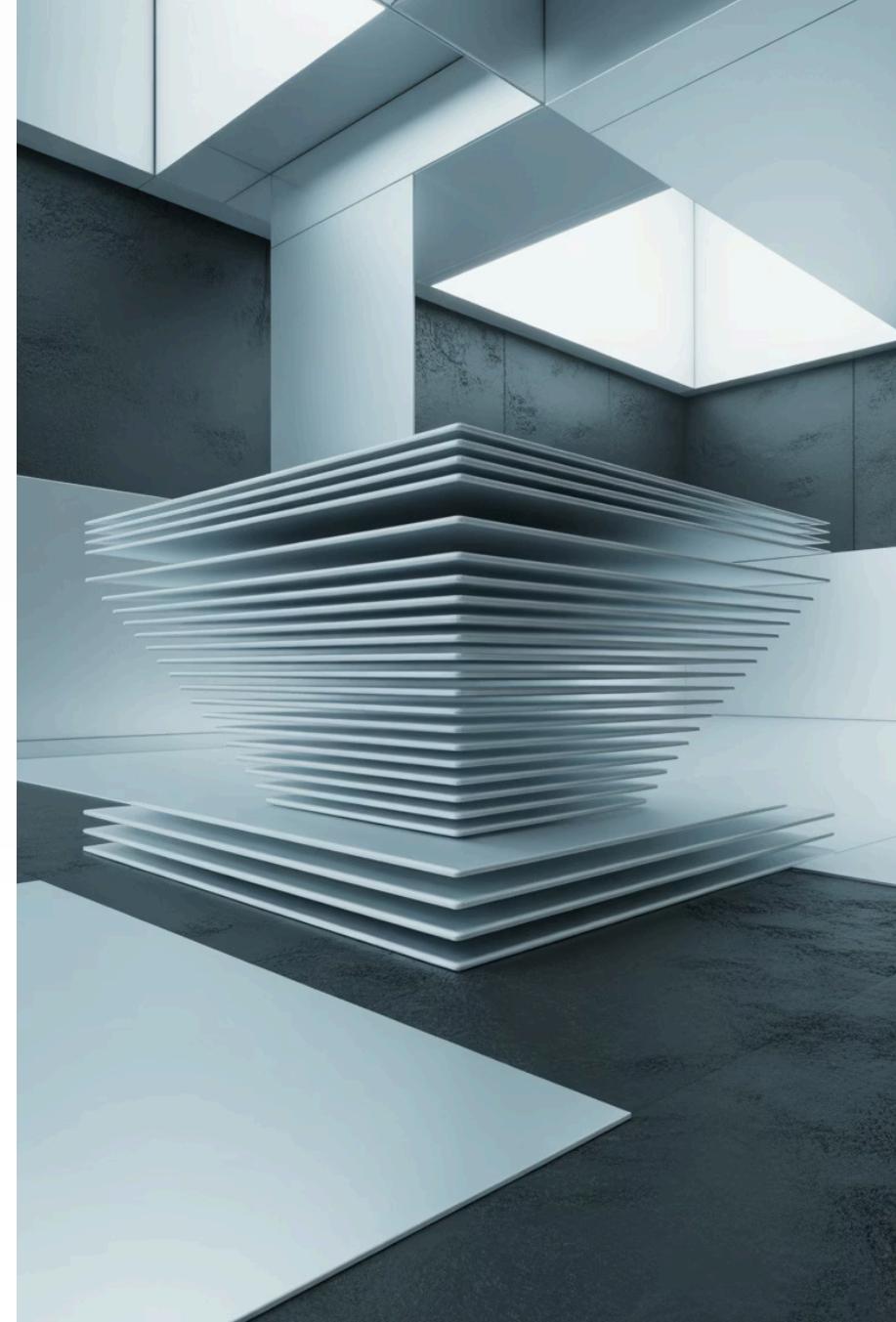
Transformation 1:1

Essentiel pour structures imbriquées et chaînage Optional

flatMap()

Stream<T> → Stream<Stream<R>>
→ Stream<R>

Transformation 1:N avec
aplatissement





Pattern : Reduce

Forme 1

Fonction binaire → Optional

Forme 2

Identité + fonction → résultat garanti

Forme 3

Identité + accumulator + combiner

Propriétés requises : associativité et élément neutre

Anti-pattern : Variables mutables

État partagé = Bugs

Problème

Contournement avec tableaux ou
wrappers mutables

Risque

Comportement non déterministe,
bugs parallèles

Solution

Opérations réduction appropriées :
`count()`, `collect()`

Pattern : Streams parallèles

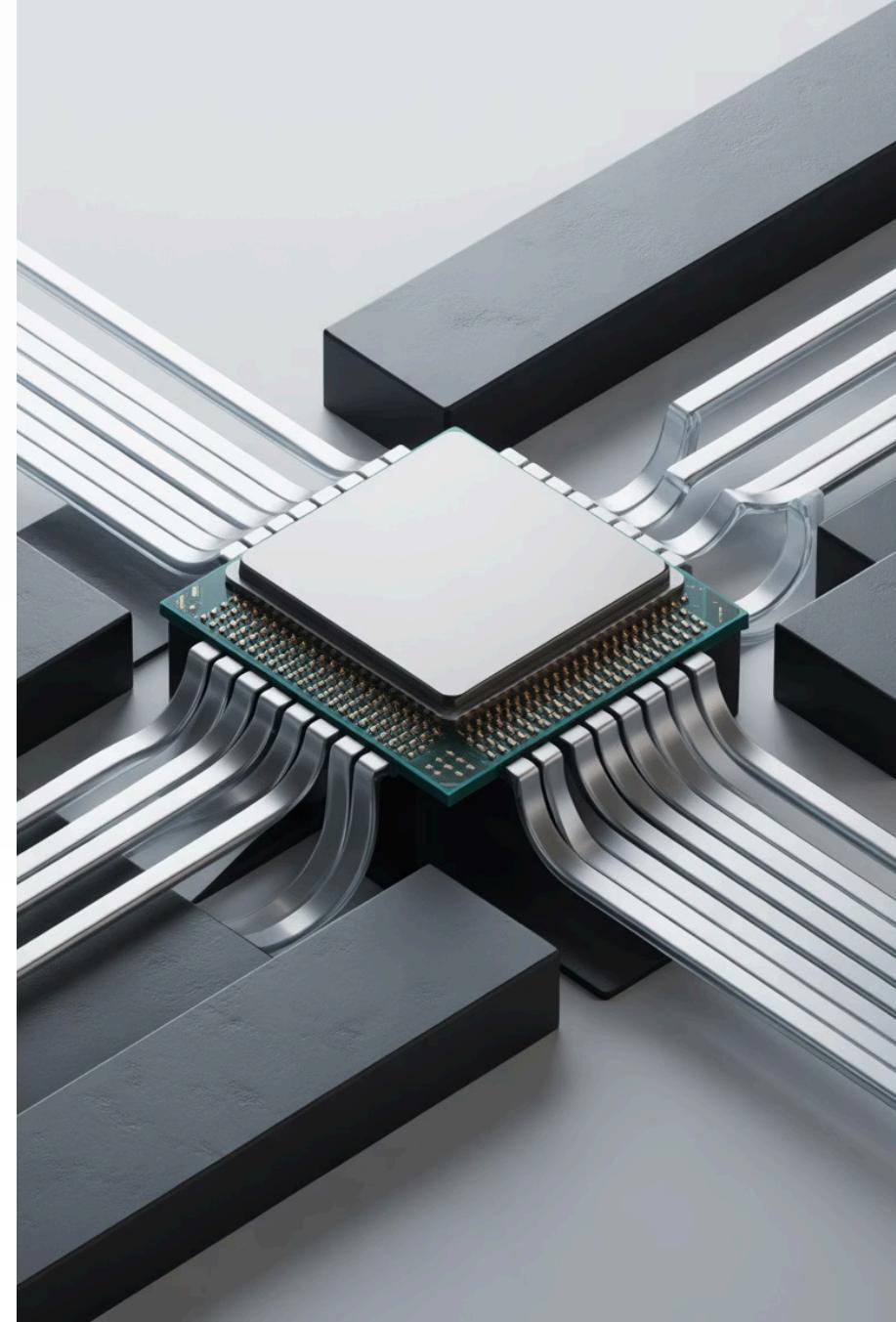
Quand utiliser

- Grande quantité données (>10000)
- Opérations coûteuses par élément
- Opérations sans état ni effets bord
- Toujours mesurer performances réelles

4x

Gain potentiel

Sur quad-core



Anti-pattern : Parallélisation inappropriée

1

Petites collections

Overhead > gain pour <1000 éléments

2

Opérations bloquantes

I/O, DB, locks : incompatibles parallélisme

3

État partagé

Contention, conditions course

4

Opérations stateful

sorted(), distinct(), limit() coûteux

Pattern : GroupingBy

Transformer collections plates en structures hiérarchiques

Groupement simple

Par propriété unique

Avec comptage

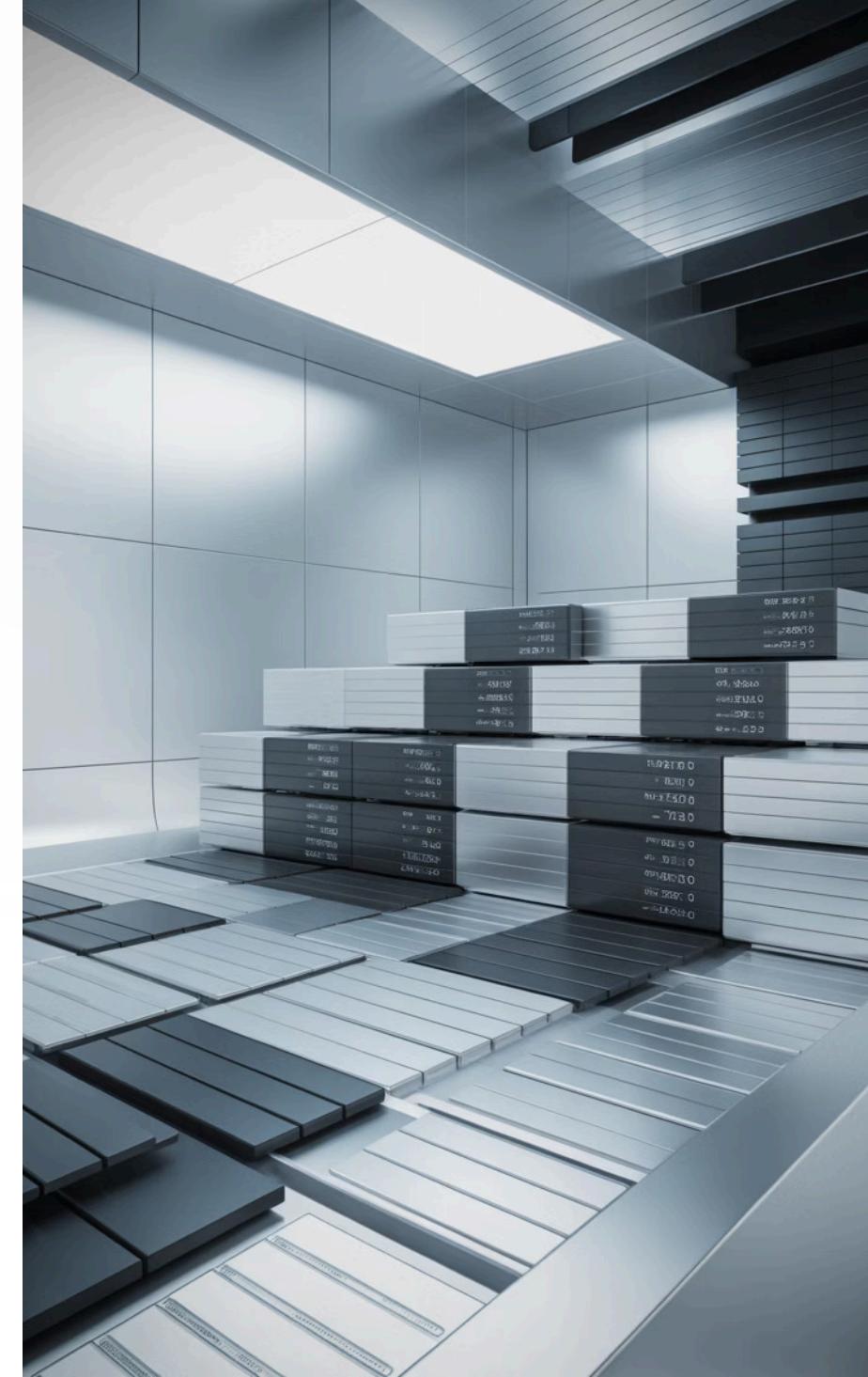
Collectors.counting()

Groupement imbriqué

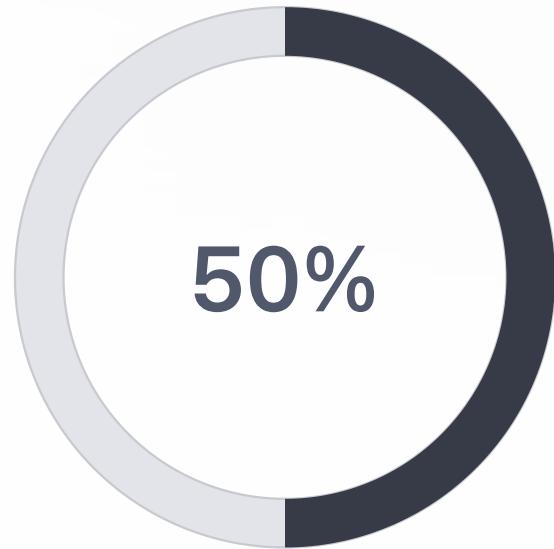
Multi-niveau hiérarchique

Avec statistiques

summarizingDouble()



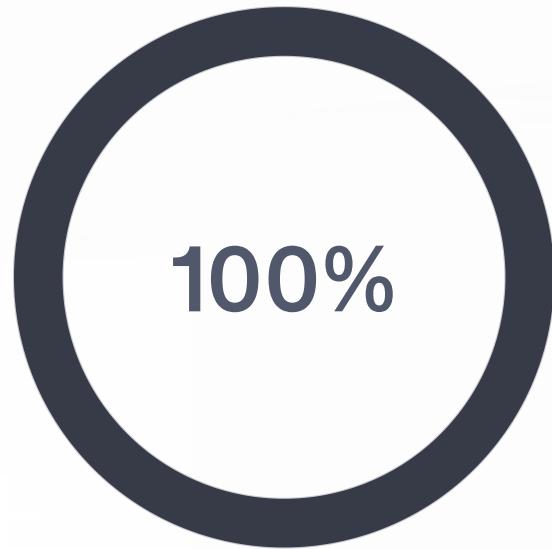
Pattern : PartitioningBy



Séparation binaire

Deux groupes : true/false

Résultat : Map<Boolean, List<T>> avec clés garanties



Efficacité

Plus rapide que groupingBy

Pattern : Teeing (Java 12+)

Appliquer deux collectors sur même stream en une passe



Collector 1

Première agrégation

Collector 2

Deuxième agrégation

Fusion

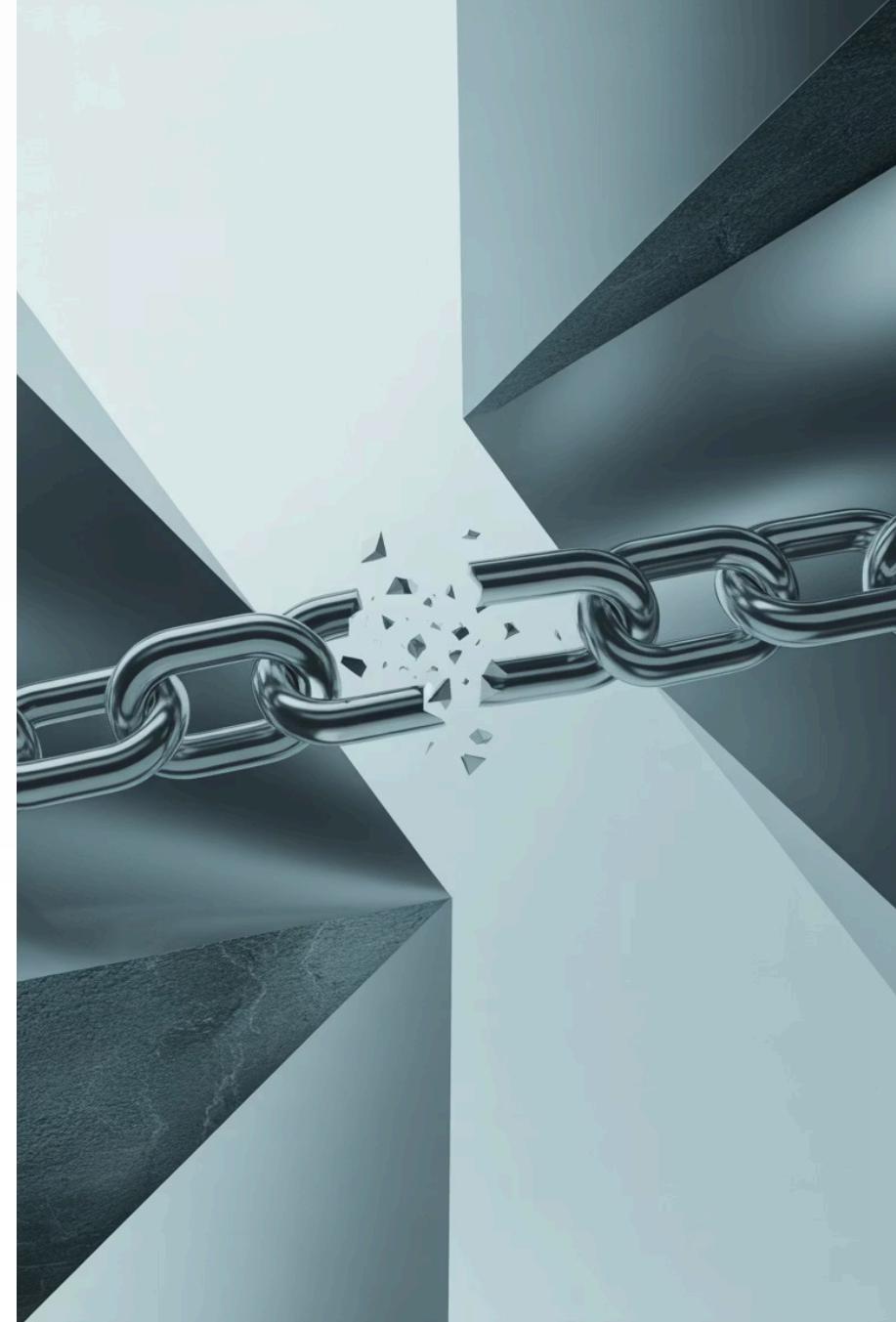
Combiner résultats



Anti-pattern : Chaînes map() excessives

5+ map() consécutifs : extraire dans méthode dédiée

Fusionner transformations simples en un seul map()



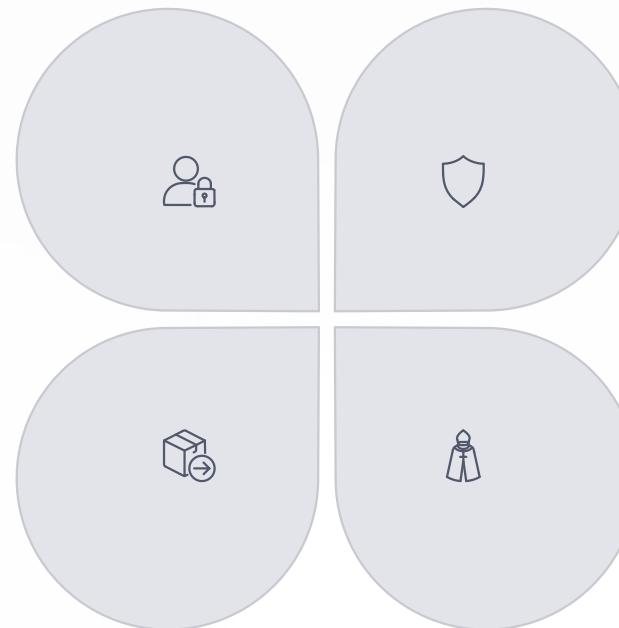
Pattern : Immutabilité

Thread-safety

Naturellement sans synchronisation

Composition

Pipelines robustes



Fiabilité

État ne change jamais

Transformation

Nouvelles instances à chaque modification

Records Java : immuables par défaut depuis Java 14

Checklist des bonnes pratiques

✓ Patterns à adopter

- Lambdas simples et courtes
- Method references
- Composition fonctions
- Optional pour absences
- Streams grandes collections
- Immutabilité objets
- Prédicats composable

ⓧ Anti-patterns à éviter

- Lambdas trop complexes
- Effets de bord
- Abus des streams
- Variables mutables
- Parallélisation inappropriée
- Ignorer exceptions
- Optional mal utilisé

Prochaines étapes

01

Consolidation

Réviser projets actuels, identifier opportunités

02

Expérimentation

Créer projets personnels, concepts avancés

03

Partage

Documenter apprentissages, échanger avec équipe

04

Évolution

Rester à jour, nouvelles versions Java

La programmation fonctionnelle complète l'orienté objet. Maîtriser les deux fait de vous un développeur plus complet.

