**Questions conceptuelles**

1. **Quelles sont les principales fonctionnalités introduites dans Java 8 ?**
2. **Qu'est-ce qu'une interface fonctionnelle ?**
3. **Expliquez ce qu'est un Stream et ses méthodes clés.**
4. **Quelle est la différence entre map() et flatMap() ?**
5. **Qu'est-ce que Optional en Java 8 ?**
6. **Comment fonctionnent les méthodes default dans les interfaces ?**
7. **Quel est le rôle des Collectors ?**
8. **En quoi l'API Date et Heure de Java 8 diffère-t-elle de java.util.Date ?**
9. **Qu'est-ce qu'une référence de méthode en Java 8 ?**
10. **Qu'est-ce que parallelStream() en Java 8 ?**
11. **Quelle est la différence entre Stream.findFirst() et Stream.findAny() ?**
12. **Quels sont les différents types de Stream en Java 8 ?**
13. **Peut-on utiliser plusieurs filtres dans un seul Stream ?**
14. **Expliquez la méthode reduce() dans les Streams avec un exemple.**
15. **Comment Java 8 gère-t-il les méthodes par défaut en cas d'héritage multiple ?**
16. **Quelles sont les bonnes pratiques pour utiliser les Streams en Java 8 ?**

**Problèmes de codage**

1. **Afficher une liste en utilisant les expressions lambda.**
2. **Filtrer les nombres pairs d'une liste en utilisant les Streams.**
3. **Trouver la valeur maximale dans une liste en utilisant les Streams.**
4. **Convertir une liste de chaînes en majuscules.**
5. **Regrouper des chaînes par leur longueur en utilisant groupingBy().**
6. **Calculer la somme des nombres d'une liste en utilisant reduce().**
7. **Compter le nombre d'occurrences de chaque mot dans une liste.**
8. **Concaténer des chaînes en utilisant joining().**
9. **Trier des employés par salaire.**
10. **Trouver le premier caractère non répété dans une chaîne.**
11. **Supprimer les doublons d'une liste en utilisant Stream.distinct().**
12. **Trouver tous les éléments commençant par "A" dans une liste.**
13. **Trier une liste de chaînes alphabétiquement et en ordre inverse.**
14. **Aplatir une liste de listes en utilisant flatMap().**
15. **Séparer les nombres pairs et impairs en utilisant partitioningBy().**
16. **Trouver le deuxième plus grand nombre dans une liste.**
17. **Compter la fréquence des caractères dans une chaîne en utilisant les Streams.**
18. **Générer un flux infini de nombres pairs et le limiter à 10 éléments.**
19. **Vérifier si tous les éléments d'une liste sont supérieurs à une valeur donnée.**
20. **Calculer la moyenne d'une liste de nombres.**
21. **Générer la suite de Fibonacci en utilisant Stream.iterate().**
22. **Regrouper les employés par département en utilisant Collectors.groupingBy().**
23. **Compter les occurrences de chaque mot dans une phrase.**
24. **Trouver le mot le plus long dans une liste.**
25. **Fusionner deux listes en une seule en utilisant flatMap().**
26. **Trouver le premier élément d'un Stream supérieur à 10.**
27. **Trouver la valeur minimale dans une liste en utilisant Streams.**
28. **Utiliser Stream.generate() pour créer une liste de nombres aléatoires.**
29. **Trouver les éléments dupliqués dans une liste en utilisant les Streams.**
30. **Diviser une liste en nombres premiers et non premiers.**
31. **Utiliser Stream.flatMap() pour traiter des collections imbriquées.**
32. **Calculer la factorielle d'un nombre en utilisant les Streams.**
33. **Utiliser Stream.skip() et Stream.limit() pour extraire des sous-listes.**
34. **Utiliser Collectors.teeing() pour effectuer deux opérations sur un Stream.**
35. **Trouver tous les mots palindromes dans une liste.**

**Questions conceptuelles**

**1. Quelles sont les principales fonctionnalités introduites dans Java 8 ?**

* **Expressions lambda** : Permettent la programmation fonctionnelle en écrivant des fonctions en ligne.
* **API Stream** : Permet de traiter des collections en utilisant un style fonctionnel.
* **Interfaces fonctionnelles** : Interfaces avec une seule méthode abstraite (ex. Predicate, Function).
* **Optional** : Aide à éviter les NullPointerException.
* **Méthodes par défaut** : Permettent d'ajouter des implémentations par défaut aux interfaces.
* **API Date et Heure** : Améliore la gestion des dates et heures avec LocalDate, LocalTime, LocalDateTime.
* **Références de méthodes** : Simplifient la syntaxe pour appeler des méthodes existantes.

**Plus de details :**

Une **interface fonctionnelle** est une interface qui contient **exactement une méthode abstraite**.

💡 **Caractéristiques** :  
✅ Peut contenir **plusieurs méthodes par défaut** (default) et **méthodes statiques**.  
✅ Peut être annotée avec @FunctionalInterface (facultatif mais recommandé).  
✅ Permet l'utilisation d'**expressions lambda** et de **références de méthodes**.

@FunctionalInterface

interface Calculateur {

int operation(int a, int b);

}

➡ Ici, Calculateur est une interface fonctionnelle car elle contient **une seule méthode abstraite** : operation(int a, int b).

**Utilisation avec une Expression Lambda**

Une **expression lambda** est une manière compacte de fournir une implémentation d'une interface fonctionnelle.

Exemple :

Calculateur addition = (a, b) -> a + b;

System.out.println(addition.operation(5, 3)); // Résultat : 8

Explication :

* (a, b) -> a + b est une **expression lambda** qui implémente la méthode operation().
* Elle remplace **la création d'une classe anonyme** qui implémenterait Calculateur.

**Interfaces Fonctionnelles Prédéfinies en Java 8**

Java 8 fournit plusieurs interfaces fonctionnelles dans le package **java.util.function**.

| **Interface** | **Signature** | **Description** |
| --- | --- | --- |
| Predicate<T> | boolean test(T t) | Vérifie si une condition est vraie. |
| Function<T, R> | R apply(T t) | Transforme un objet T en R. |
| Consumer<T> | void accept(T t) | Exécute une action sur T. |
| Supplier<T> | T get() | Fournit une valeur de type T. |
| UnaryOperator<T> | T apply(T t) | Transformation sans changement de type. |

**Predicate<T> → Filtrage**

Utilisé pour **tester une condition** (retourne true ou false).

Predicate<Integer> estPair = n -> n % 2 == 0;

System.out.println(estPair.test(10)); // Résultat : true

System.out.println(estPair.test(11)); // Résultat : false

**Function<T, R> → Transformation**

Transforme un objet d'un type T en un autre type R.

Function<String, Integer> longueur = s -> s.length();

System.out.println(longueur.apply("Java 8")); // Résultat : 6

**Consumer<T> → Exécuter une action**

Exécute une opération sur un objet de type T, **mais ne retourne rien**.

Consumer<String> afficher = s -> System.out.println("Bonjour " + s);

afficher.accept("Alice"); // Résultat : Bonjour Alice

**Supplier<T> → Fournir une valeur**

Fournit une **valeur sans paramètre d’entrée**.

Supplier<Double> random = () -> Math.random();

System.out.println(random.get()); // Résultat : un nombre aléatoire

**UnaryOperator<T> → Transformation sans changement de type**

Applique une transformation sur un élément du même type.

UnaryOperator<Integer> carre = n -> n \* n;

System.out.println(carre.apply(4)); // Résultat : 16

**Références de Méthodes avec Interfaces Fonctionnelles**

Une **référence de méthode** est une alternative aux **expressions lambda**.

🔹 **Exemple avec Consumer<T>** :

Consumer<String> print = System.out::println;

print.accept("Hello World"); // Résultat : Hello World

Explication :

* System.out::println est une **référence de méthode** équivalente à (s) -> System.out.println(s).

**Exemple Complet avec Plusieurs Interfaces Fonctionnelles**

* Voici un exemple qui combine **plusieurs interfaces fonctionnelles** en Java 8.

import java.util.function.\*;

public class Java8InterfacesFonctionnelles {

public static void main(String[] args) {

// Predicate : Vérifie si un nombre est positif

Predicate<Integer> estPositif = n -> n > 0;

System.out.println(estPositif.test(5)); // true

System.out.println(estPositif.test(-3)); // false

// Function : Convertir Celsius en Fahrenheit

Function<Double, Double> celsiusToFahrenheit = c -> (c \* 9/5) + 32;

System.out.println(celsiusToFahrenheit.apply(0.0)); // 32.0

// Consumer : Afficher un message

Consumer<String> afficherMessage = message -> System.out.println("Message : " + message);

afficherMessage.accept("Hello Java 8");

// Supplier : Retourner un mot de passe aléatoire

Supplier<String> genererMotDePasse = () -> "Java8-" + (int) (Math.random() \* 1000);

System.out.println(genererMotDePasse.get());

// UnaryOperator : Doubler une valeur

UnaryOperator<Integer> doubler = x -> x \* 2;

System.out.println(doubler.apply(10)); // 20

}

}

**Pourquoi Utiliser les Interfaces Fonctionnelles ?**

✅ **Code plus lisible** : Moins de code répétitif grâce aux **expressions lambda**.  
✅ **Programmation fonctionnelle** : Approche plus efficace pour traiter des données.  
✅ **Optimisation des performances** : Utilisation des **Streams** et du **parallélisme**.  
✅ **Réduction du code boilerplate** : Évite les classes anonymes inutiles.

**Conclusion**

Les **interfaces fonctionnelles** sont essentielles en **Java 8** car elles permettent de **simplifier le code**, d'introduire des **expressions lambda** et d'améliorer la **programmation fonctionnelle**.

**2. Qu'est-ce qu'une interface fonctionnelle ?**

* Une interface fonctionnelle est une interface qui contient **exactement une méthode abstraite**.
* Elle supporte les **expressions lambda** et les références de méthodes.
* Exemples :
  + Runnable → void run().
  + Predicate<T> → boolean test(T t).
  + Function<T, R> → R apply(T t).

**3. Expliquez ce qu'est un Stream et ses méthodes clés.**

* Un **Stream** représente une séquence d'éléments pouvant être traités en pipeline.
* **Opérations intermédiaires** (renvoient un Stream) :
  + filter() → Filtre les éléments selon une condition.
  + map() → Transforme les éléments.
  + sorted() → Trie les éléments.
* **Opérations terminales** (consomment le Stream) :
  + collect() → Convertit en collection.
  + forEach() → Exécute une action sur chaque élément.
  + reduce() → Agrège les éléments.

**4. Quelle est la différence entre map() et flatMap() ?**

* map() transforme chaque élément, mais peut produire un **Stream de Streams**.
* flatMap() transforme et **aplatit** les structures imbriquées en un seul Stream.

**Exemple :**

java

CopierModifier

List<List<Integer>> nested = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3, 4));

List<Integer> flatList = nested.stream().flatMap(List::stream).collect(Collectors.toList());

// Résultat : [1, 2, 3, 4]

**5. Qu'est-ce que Optional en Java 8 ?**

* Optional est une classe permettant d'éviter les NullPointerException.
* Méthodes utiles :
  + of(value) → Crée un Optional avec une valeur non nulle.
  + empty() → Crée un Optional vide.
  + ifPresent(action) → Exécute une action si une valeur est présente.

**6. Comment fonctionnent les méthodes default dans les interfaces ?**

* Elles permettent d'ajouter une implémentation par défaut dans une interface sans casser les classes existantes.

**Exemple :**

java

interface MonInterface {

default void afficher() {

System.out.println("Méthode par défaut");

}

}

**7. Quel est le rôle des Collectors ?**

* Collectors est une classe utilitaire permettant de réduire des Streams.
* Exemples courants :
  + toList(), toSet() → Convertit en liste ou ensemble.
  + joining() → Concatène des chaînes.
  + groupingBy() → Regroupe les éléments par clé.
  + partitioningBy() → Sépare en deux groupes selon un prédicat.

**8. En quoi l'API Date et Heure de Java 8 diffère-t-elle de java.util.Date ?**

* Les nouvelles classes (LocalDate, LocalTime, LocalDateTime) sont **immuables et thread-safe**.
* Utilisation de DateTimeFormatter pour le formatage.
* Gestion des fuseaux horaires avec ZonedDateTime.

**9. Qu'est-ce qu'une référence de méthode en Java 8 ?**

* C'est une **abréviation des expressions lambda**.
* Types :
  + Méthodes statiques → Class::methode.
  + Méthodes d’instance → instance::methode.
  + Constructeurs → ClassName::new.

**Exemple :**

java

CopierModifier

List<String> noms = Arrays.asList("Alice", "Bob");

noms.forEach(System.out::println); // Référence à println

**10. Qu'est-ce que parallelStream() en Java 8 ?**

* parallelStream() permet d'exécuter un Stream en **parallèle** pour améliorer les performances.

**Exemple :**

java

CopierModifier

List<Integer> nombres = Arrays.asList(1, 2, 3);

nombres.parallelStream().map(n -> n \* 2).forEach(System.out::println);

**11. Quelle est la différence entre Stream.findFirst() et Stream.findAny() ?**

* findFirst() → Retourne **le premier élément** du Stream (utile pour les Streams ordonnés).
* findAny() → Retourne **un élément quelconque**, utile pour les Streams parallèles.

**12. Quels sont les différents types de Stream en Java 8 ?**

* **Stream séquentiel** → Traite les éléments un par un sur un seul thread.
* **Stream parallèle** → Traite les éléments simultanément sur plusieurs threads.

**13. Peut-on utiliser plusieurs filtres dans un seul Stream ?**

Oui, il est possible de chaîner plusieurs filter().

**Exemple :**

java

CopierModifier

List<Integer> nombres = Arrays.asList(1, 2, 3, 4, 5);

nombres.stream()

.filter(n -> n > 2)

.filter(n -> n % 2 == 0)

.forEach(System.out::println); // Résultat : 4

**14. Expliquez la méthode reduce() dans les Streams avec un exemple.**

* reduce() est utilisée pour agréger les éléments d'un Stream.

**Exemple : Calcul de la somme**

java

CopierModifier

List<Integer> nombres = Arrays.asList(1, 2, 3, 4);

int somme = nombres.stream().reduce(0, Integer::sum);

System.out.println(somme); // Résultat : 10

**15. Comment Java 8 gère-t-il les méthodes par défaut en cas d'héritage multiple ?**

* Si deux interfaces ont la même méthode par défaut, la classe doit **surcharger** la méthode pour résoudre le conflit.

**Exemple :**

java

interface A { default void afficher() { System.out.println("A"); } }

interface B { default void afficher() { System.out.println("B"); } }

class C implements A, B {

public void afficher() { A.super.afficher(); } // Résout le conflit

}

**16. Quelles sont les bonnes pratiques pour utiliser les Streams en Java 8 ?**

* **Ne pas utiliser de Stream pour de petites collections** (les boucles sont plus performantes).
* **Utiliser parallelStream() uniquement pour les grandes collections**.
* **Privilégier les références de méthodes** pour la lisibilité.
* **Utiliser les opérations terminales** (collect(), reduce()) pour éviter les Streams inutilisés.

CHAPITRE

1. Afficher une liste en utilisant les expressions lambda

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(name -> System.out.println(name));

}

}

**Sortie**

Alice

Bob

Charlie

2. Filtrer les nombres pairs d'une liste en utilisant les Streams

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

List<Integer> evens = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

System.out.println(evens); // [2, 4, 6]

}

}

3. Trouver la valeur maximale dans une liste en utilisant les Streams

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 20, 30, 40);

int max = numbers.stream()

.max(Integer::compare)

.orElse(0);

System.out.println(max); // 40

}

}

4. Convertir une liste de chaînes en majuscules

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<String> names = Arrays.asList("alice", "bob");

List<String> upperNames = names.stream()

.map(String::toUpperCase)

.collect(Collectors.toList());

System.out.println(upperNames); // [ALICE, BOB]

}

}

5. Regrouper des chaînes par leur longueur en utilisant groupingBy()

import java.util.\*;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

Map<Integer, List<String>> grouped = names.stream()

.collect(Collectors.groupingBy(String::length));

System.out.println(grouped); // {3=[Bob], 5=[Alice], 7=[Charlie]}

}

}

6. Calculer la somme des nombres d'une liste en utilisant reduce()

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

int sum = numbers.stream()

.reduce(0, Integer::sum);

System.out.println(sum); // 10

}

}

7. Compter le nombre d'occurrences de chaque mot dans une liste

import java.util.\*;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<String> words = Arrays.asList("apple", "banana", "apple");

Map<String, Long> wordCount = words.stream()

.collect(Collectors.groupingBy(w -> w, Collectors.counting()));

System.out.println(wordCount); // {apple=2, banana=1}

}

}

8. Concaténer des chaînes en utilisant joining()

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<String> words = Arrays.asList("Java", "est", "puissant");

String sentence = words.stream()

.collect(Collectors.joining(" "));

System.out.println(sentence); // Java est puissant

}

}

9. Trier des employés par salaire

import java.util.\*;

class Employee {

String name;

int salary;

Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

@Override

public String toString() {

return name + ": " + salary;

}

}

public class Main {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", 5000),

new Employee("Bob", 3000),

new Employee("Charlie", 4000)

);

List<Employee> sorted = employees.stream()

.sorted(Comparator.comparingInt(e -> e.salary))

.collect(Collectors.toList());

System.out.println(sorted);

// [Bob: 3000, Charlie: 4000, Alice: 5000]

}

}

**10. Trouver le premier caractère non répété dans une chaîne**

import java.util.LinkedHashMap;

import java.util.Map;

import java.util.function.Function;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

String input = "swiss";

Character result = input.chars()

.mapToObj(c -> (char) c)

.collect(Collectors.groupingBy(Function.identity(), LinkedHashMap::new, Collectors.counting()))

.entrySet()

.stream()

.filter(entry -> entry.getValue() == 1)

.map(Map.Entry::getKey)

.findFirst()

.orElse(null);

System.out.println(result); // w

}

}

**Section 2 : Manipulation avancée des Streams et collections**

**11. Supprimer les doublons d'une liste en utilisant Stream.distinct()**

**Section 2 : Manipulation avancée des Streams et collections**

**11. Supprimer les doublons d'une liste en utilisant Stream.distinct()**

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5);

List<Integer> distinctNumbers = numbers.stream()

.distinct()

.collect(Collectors.toList());

System.out.println(distinctNumbers); // [1, 2, 3, 4, 5]

}

}

12. Trouver tous les éléments commençant par "A" dans une liste

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Annie", "Alex");

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("A"))

.collect(Collectors.toList());

System.out.println(filteredNames); // [Alice, Annie, Alex]

}

}

13. Trier une liste de chaînes alphabétiquement et en ordre inverse

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

List<String> names = Arrays.asList("Charlie", "Alice", "Bob");

// Tri en ordre croissant

List<String> sortedNames = names.stream()

.sorted()

.collect(Collectors.toList());

System.out.println(sortedNames); // [Alice, Bob, Charlie]

// Tri en ordre décroissant

List<String> reversedNames = names.stream()

.sorted(Comparator.reverseOrder())

.collect(Collectors.toList());

System.out.println(reversedNames); // [Charlie, Bob, Alice]

}

}

14. Aplatir une liste de listes en utilisant flatMap()

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<List<Integer>> nestedList = Arrays.asList(

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5),

Arrays.asList(6, 7, 8)

);

List<Integer> flatList = nestedList.stream()

.flatMap(List::stream)

.collect(Collectors.toList());

System.out.println(flatList); // [1, 2, 3, 4, 5, 6, 7, 8]

}

}

15. Séparer les nombres pairs et impairs en utilisant partitioningBy()

import java.util.Arrays;

import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

Map<Boolean, List<Integer>> partitioned = numbers.stream()

.collect(Collectors.partitioningBy(n -> n % 2 == 0));

System.out.println(partitioned); // {false=[1, 3, 5], true=[2, 4, 6]}

}

}

1. Trouver le deuxième plus grand nombre dans une liste

import java.util.Arrays;

import java.util.List;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);

int secondHighest = numbers.stream()

.sorted(Comparator.reverseOrder())

.skip(1) // Ignorer le plus grand élément

.findFirst()

.orElseThrow(() -> new RuntimeException("Aucun deuxième plus grand nombre trouvé"));

System.out.println(secondHighest); // 40

}

}

1. Compter la fréquence des caractères dans une chaîne en utilisant les Streams

import java.util.Map;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

String input = "java";

Map<Character, Long> frequency = input.chars()

.mapToObj(c -> (char) c)

.collect(Collectors.groupingBy(c -> c, Collectors.counting()));

System.out.println(frequency); // {a=2, j=1, v=1}

}

}

1. Générer un flux infini de nombres pairs et le limiter à 10 éléments

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 20, 30, 40);

boolean allGreater = numbers.stream().allMatch(n -> n > 5);

System.out.println(allGreater); // true

}

}

1. Calculer la moyenne d'une liste de nombres

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 20, 30, 40);

double average = numbers.stream()

.mapToInt(Integer::intValue)

.average()

.orElse(0.0);

System.out.println(average); // 25.0

}

}

**Section 3 : Programmation fonctionnelle et manipulations avancées**

**21. Générer la suite de Fibonacci en utilisant Stream.iterate()**

import java.util.stream.Stream;

public class Main {

public static void main(String[] args) {

Stream.iterate(new int[]{0, 1}, f -> new int[]{f[1], f[0] + f[1]})

.limit(10)

.map(f -> f[0])

.forEach(System.out::print); // 01123581321

}

}

1. Regrouper les employés par département en utilisant Collectors.groupingBy()

import java.util.\*;

import java.util.stream.Collectors;

class Employee {

String name;

String department;

Employee(String name, String department) {

this.name = name;

this.department = department;

}

}

public class Main {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", "RH"),

new Employee("Bob", "IT"),

new Employee("Charlie", "RH"),

new Employee("David", "IT")

);

Map<String, List<Employee>> groupedByDepartment = employees.stream()

.collect(Collectors.groupingBy(emp -> emp.department));

groupedByDepartment.forEach((dept, emps) -> {

System.out.println(dept + ": " + emps.stream().map(e -> e.name).collect(Collectors.toList()));

});

// Sortie :

// RH: [Alice, Charlie]

// IT: [Bob, David]

}

}

1. Compter les occurrences de chaque mot dans une phrase

import java.util.\*;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

String sentence = "Java est puissant et Java est populaire";

Map<String, Long> wordCount = Arrays.stream(sentence.split(" "))

.collect(Collectors.groupingBy(word -> word, Collectors.counting()));

System.out.println(wordCount);

// Sortie : {Java=2, est=2, puissant=1, et=1, populaire=1}

}

}

1. Trouver le mot le plus long dans une liste

import java.util.Arrays;

import java.util.List;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

List<String> words = Arrays.asList("pomme", "banane", "cerise", "datte");

String longestWord = words.stream()

.max(Comparator.comparingInt(String::length))

.orElse(null);

System.out.println(longestWord); // banane

}

}

1. Fusionner deux listes en une seule en utilisant flatMap()

import java.util.\*;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class Main {

public static void main(String[] args) {

List<Integer> list1 = Arrays.asList(1, 2, 3);

List<Integer> list2 = Arrays.asList(4, 5, 6);

List<Integer> mergedList = Stream.of(list1, list2)

.flatMap(List::stream)

.collect(Collectors.toList());

System.out.println(mergedList); // [1, 2, 3, 4, 5, 6]

}

}

1. Trouver le premier élément d'un Stream supérieur à 10

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(5, 8, 12, 3, 20);

int first = numbers.stream()

.filter(n -> n > 10)

.findFirst()

.orElse(-1);

System.out.println(first); // 12

}

}

1. Trouver la valeur minimale dans une liste en utilisant Streams

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 20, 5, 15);

int min = numbers.stream()

.min(Integer::compareTo)

.orElseThrow(() -> new RuntimeException("Aucun minimum trouvé"));

System.out.println(min); // 5

}

}

1. Trouver la valeur minimale dans une liste en utilisant Streams

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 20, 5, 15);

int min = numbers.stream()

.min(Integer::compareTo)

.orElseThrow(() -> new RuntimeException("Aucun minimum trouvé"));

System.out.println(min); // 5

}

}

1. Utiliser Stream.generate() pour créer une liste de nombres aléatoires

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class Main {

public static void main(String[] args) {

List<Double> randomNumbers = Stream.generate(Math::random)

.limit(5)

.collect(Collectors.toList());

System.out.println(randomNumbers);

}

}

1. Trouver les éléments dupliqués dans une liste en utilisant les Streams

import java.util.\*;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 2, 3);

Set<Integer> duplicates = numbers.stream()

.filter(n -> Collections.frequency(numbers, n) > 1)

.collect(Collectors.toSet());

System.out.println(duplicates); // [2, 3]

}

}

1. Diviser une liste en nombres premiers et non premiers

import java.util.\*;

import java.util.stream.Collectors;

import java.util.stream.IntStream;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(2, 3, 4, 5, 6, 7, 8, 9, 10);

Map<Boolean, List<Integer>> partitioned = numbers.stream()

.collect(Collectors.partitioningBy(Main::isPrime));

System.out.println(partitioned);

}

static boolean isPrime(int num) {

if (num <= 1) return false;

return IntStream.rangeClosed(2, (int) Math.sqrt(num))

.noneMatch(n -> num % n == 0);

}

}