Les collections, les streams et les fonctions lambda en Java

Somaire

- Introduction aux Collections en Java
- Introduction à la programmation fonctionnelle et aux lambdas en Java
- Comprendre les interfaces fonctionnelles en Java
- Introduction aux streams en Java
- Approfondissement des streams en Java
- Cas pratiques – Les lambdas et les streams en action

Introduction aux Collections en Java

1. Définition
2. Types de Collections
3. Iterator
4. Enumeration
5. Différences entre Iterator et Enumeration

1. Définition

Les Collections en Java sont des structures de données similaires aux tableaux, mais avec des fonctionnalités plus avancées.

Elles peuvent contenir des éléments dynamiques et de différents types. Les Collections en Java sont fournies principalement via le Java Collections Framework.

2. Types de Collections

Il existe plusieurs types de collections, y compris `List`, `Set` et `Map`. Chacun de ces types a ses propres caractéristiques et utilisations.

1. **`List`** : Une collection ordonnée (ou séquentielle) qui peut contenir des éléments duplicats. Exemples : `ArrayList`, `LinkedList`
2. **`Set`** : Une collection qui ne peut pas contenir des éléments duplicats. Exemples : `HashSet`, `LinkedHashSet`, `TreeSet`
3. **`Map`** : Une collection de paires clé/valeur. Elle ne peut pas contenir de clés dupliquées et chaque clé est associée à une valeur. Exemple : `HashMap`, `LinkedHashMap`, `TreeMap`

3. Iterator

Une interface qui fournit des méthodes pour itérer sur n'importe quelle collection.

```
Iterator iterator = list.iterator();
while(iterator.hasNext()){
    System.out.println(iterator.next());
}
```

4. Enumeration

Une interface qui a des fonctionnalités similaires à l'interface Iterator, mais elle est plus ancienne et moins sûre. Son utilisation est donc généralement déconseillée.

```
Enumeration enumeration = vector.elements();
while(enumeration.hasMoreElements()){
    System.out.println(enumeration.nextElement());
}
```

5. Différence entre Iterators et Enumerations -1/2-

1. `Iterator` est une interface plus moderne qui a été introduite avec la Collection Framework dans Java 1.2, tandis que `Enumeration` est une interface plus ancienne, utilisée dans les anciennes classes comme `Vector` et `Hashtable`.
2. **Méthodes de suppression** : `Iterator` a une méthode `remove()` qui permet de supprimer l'élément actuel de la collection sous-jacente. `Enumeration` ne fournit pas cette fonctionnalité.
3. **Nom des méthodes** : Les méthodes dans `Iterator` sont `hasNext()` et `next()`, tandis que dans `Enumeration` elles sont `hasMoreElements()` et `nextElement()`.

5. Différence entre Iterators et Enumerations -2/2-

1. **Sécurité des threads** : Les `Enumerations` sont conçues pour être utilisées avec des classes thread-safe plus anciennes comme `Vector` et `Hashtable`, donc utiliser `Enumeration` peut être plus sûr si vous travaillez dans un environnement multithread.
2. **Performance** : `Iterator` est plus rapide que `Enumeration` car `Enumeration` nécessite un verrou pour la synchronisation.

En général, on recommande d'utiliser `Iterator` pour de nouveaux développements, car il a plus de fonctionnalités et est plus conforme au reste du Java Collections Framework.

Introduction à la programmation fonctionnelle et aux lambdas en Java

1. Qu'est-ce que la programmation fonctionnelle?
2. Introduction aux expressions lambda en Java
3. Pourquoi utiliser des expressions lambda?
4. Exemples pratiques d'expressions lambda

1. Qu’est-ce que la programmation fonctionnelle?

La programmation fonctionnelle est un style de programmation qui traite le calcul comme l’évaluation de fonctions mathématiques et évite les données d’état changeantes et les données mutables.

C’est une approche déclarative de la programmation, ce qui signifie que vous décrivez à quoi doit ressembler le résultat, et non comment y parvenir.

2. Introduction aux expressions lambda en Java

En Java, une expression lambda est une fonction anonyme; c'est-à-dire une fonction sans nom. Elle est utilisée pour fournir une implémentation d'une interface fonctionnelle. Une expression lambda est caractérisée par la syntaxe suivante:

```
(parameter) -> {body}
```

Par exemple, un bout de code lambda qui ajoute deux nombres pourrait ressembler à ceci:

```
(a, b) -> a + b
```

3. Pourquoi utiliser des expressions lambda?

Les expressions lambda permettent d'écrire du code plus concis et plus lisible. Elles sont particulièrement utiles pour la programmation fonctionnelle et sont souvent utilisées avec les streams en Java.

4. Exemples pratiques d'expressions lambda

```
// Sans paramètre:
() -> System.out.println("Hello World")

// Avec un seul paramètre:
n -> n * n

// Avec plusieurs paramètres:
(a, b) -> a * b
```

Comprendre les interfaces fonctionnelles en Java

1. Qu’est-ce qu’une interface fonctionnelle?
2. Création d’une interface fonctionnelle
3. Comment fonctionnent les interfaces fonctionnelles avec les lambdas
4. Types d’interfaces fonctionnelles: `Predicate`, `Function`, `Consumer`, `Supplier`
5. `Method References`
6. Exemples et exercices pratiques

1. Qu’est-ce qu’une interface fonctionnelle ?

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

Elle peut contenir un nombre quelconque de méthodes par défaut et de méthodes statiques.

Elles sont utilisées comme cible de type pour les expressions lambda et les références de méthode.

2. Création d'une interface fonctionnelle

```
@FunctionalInterface
interface Calculator {
    int calculate(int x, int y);
}
```

N.B. : *Pour garantir qu'une interface est bien une interface fonctionnelle, il est recommandé d'utiliser l'annotation `@FunctionalInterface`.*

```
/* Exemple d'instanciation d'une interface fonctionnelle
 * avec une expression lambda
 */
Calculator addition = (x, y) -> x + y;
System.out.println(addition.calculate(5, 3));  // Output : 8
```

3. Comment fonctionnent les interfaces fonctionnelles avec les lambdas

Les expressions lambda peuvent être utilisées pour fournir une implémentation à une méthode abstraite d'une interface fonctionnelle.

Par exemple, on peut créer une instance de `Runnable` (qui est une interface fonctionnelle) à l'aide d'une expression lambda:

```
Runnable runnable = () -> System.out.println("Hello, World!");
```

4. Types d’interfaces fonctionnelles

Il y a plusieurs interfaces fonctionnelles fournies dans la bibliothèque standard de Java.

`Predicate<T>`

- Prend un argument de type T et renvoie un booléen.
- Méthode : `test`
- Exemple :

```
Predicate<String> lengthIsGreaterThan5 = str -> str.length() > 5;
```

`Function<T, R>`

- Prend un argument de type T et renvoie un résultat de type R.
- Méthode : `apply`
- Exemple :

```
Function<String, Integer> stringLength = str -> str.length();
```

`Consumer<T>`

- Prend un argument de type T et ne renvoie rien.
- Méthode : `accept`
- Exemple :

```
Consumer<String> printString = str -> System.out.println(str);
```

`Supplier<T>`

- Ne prend aucun argument et renvoie un résultat de type T.
- Méthode : `get`
- Exemple :

```
Supplier<String> stringSupplier = () -> "Hello, World!";
```

5. Method References

Les `Method References` permettent de réutiliser une méthode existante et de la traiter comme une expression lambda. Ils sont souvent utilisés pour améliorer la lisibilité du code.

Il existe quatre types de références de méthodes :

- Référence à une méthode statique (`Integer::parseInt`).
- Référence à une méthode d'instance d'un objet particulier (`System.out::println`).
- Référence à une méthode d'instance d'un objet arbitraire d'un type particulier (`String::length`).
- Référence à un constructeur (`ArrayList::new`).

6. Exemples et exercices pratiques

1. Créez un `Predicate<String>` qui vérifie si une chaîne de caractères est non vide.
2. Créez une `Function<String, String>` qui convertit une chaîne en majuscules.
3. Créez un `Consumer<String>` qui imprime une chaîne de caractères sur la console.
4. Créez un `Supplier<Double>` qui retourne un nombre aléatoire.

Introduction aux streams en Java

1. Qu'est-ce qu'un stream?
2. Différences entre les collections et les streams
3. Création de streams
4. Opérations sur les streams
5. Exemple d'application.
6. Exemples et exercices pratiques

1. Qu’est-ce qu’un stream?

Un Stream dans Java est une séquence d’éléments supportant différentes méthodes qui peuvent être pipelined pour produire le résultat souhaité. Les fonctionnalités des Java Streams sont :

- Ils ne stockent pas d’éléments. Ils sont calculés à la demande.
- Ils ne modifient pas la source de données d’origine.
- Ils peuvent être parcourus une seule fois.

2. Différences entre les collections et les streams

Alors que les collections sont des structures de données essentiellement destinées à stocker et accéder aux données, les streams sont des outils destinés à décrire des calculs sur les données.

Les streams offrent également l'avantage d'être potentiellement infinis et de permettre des opérations paresseuses (c'est-à-dire des opérations qui ne sont réalisées que lorsque c'est nécessaire).

3. Création de streams

Avec la méthode `Stream.of(T ... values)`:

```
Stream<String> stream = Stream.of("A", "B", "C");
```

Avec la méthode `Arrays.stream(T[] array)`:

```
String[] arr = new String[]{"A", "B", "C"};
Stream<String> stream = Arrays.stream(arr);
```

Avec la méthode `Collection.stream()`:

```
List<String> list = Arrays.asList("A", "B", "C");
Stream<String> stream = list.stream();
```

4. Opérations sur les streams

`filter(Predicate predicate)`: prend un prédicat (une fonction qui renvoie un booléen) comme argument et renvoie un stream qui inclut tous les éléments qui satisfont le prédicat.

`map(Function mapper)`: prend une fonction comme argument et renvoie un stream où les éléments sont le résultat de l'application de cette fonction aux éléments d'origine.

`reduce(BinaryOperator accumulator)`: prend une fonction qui accepte deux arguments et produit un résultat, et renvoie un Optional qui décrit le résultat de l'accumulation.

5. Exemple d’application

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Optional<Integer> sum = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .reduce((a, b) -> a + b);

System.out.println(sum.get());  // Affiche 220
```

. Exemples et exercices pratiques

1. Créez un stream à partir d'une collection de chaînes de caractères et utilisez la méthode filter pour ne sélectionner que les chaînes qui commencent par la lettre "A".
2. Créez un stream à partir d'un tableau d'entiers et utilisez la méthode map pour doubler chaque valeur.
3. Utilisez la méthode reduce pour obtenir la somme des nombres dans un stream.

Approfondissement des streams en Java

1. Collecteurs
2. Opérations d’agrégation
3. Les opérations en parallèle et leur utilisation
4. Exemples et exercices pratiques

1. Collecteurs

Les collecteurs sont utilisés pour combiner le résultat de l'opération de traitement d'un stream.

- `Collectors.toList()` : pour collecter tous les éléments dans une liste.

```
List<String> list = Stream.of("A", "B", "C")
                          .collect(Collectors.toList());
```

- `Collectors.toSet()` : pour collecter tous les éléments dans un ensemble (set).

```
Set<String> set = Stream.of("A", "B", "C")
                        .collect(Collectors.toSet());
```

- `Collectors.toMap()` : pour collecter les éléments dans une map.

```
Map<String, Integer> map = Stream.of("A", "B", "C")
                                 .collect(Collectors.toMap(Function.identity(), String::length));
```

2. Opérations d’agrégation

Les opérations d’agrégation sont des opérations terminales qui renvoient un certain type de résultat des streams.

count()

Compte le nombre d’éléments dans un stream.

```java
long count = Stream.of("A", "B", "C").count(); // 3
```

`min(Comparator)` et `max(Comparator)`

Trouvent le minimum et le maximum d'un stream selon un comparateur donné.

```
Optional<String> min = Stream.of("A", "B", "C")
                         .min(Comparator.naturalOrder()); // Optional[A]
```

`sum()`

- Calcule la somme des éléments d’un stream d’entiers.
- Disponible pour les streams spécialisés tels que IntStream, LongStream, etc.

```java
int sum = IntStream.of(1, 2, 3, 4).sum(); // 10
```

`average()`

- Calcule la moyenne des éléments d’un stream d’entiers.
- Disponible pour les streams spécialisés tels que IntStream, LongStream, etc.

```
OptionalDouble average = IntStream.of(1, 2, 3, 4).average(); // OptionalDouble[2.5]
```

3. Les opérations en parallèle et leur utilisation

Java Streams offre également la possibilité d'exécuter des opérations en parallèle, ce qui peut améliorer les performances pour les grandes quantités de données. Vous pouvez utiliser la méthode `parallelStream()` au lieu de `stream()` pour obtenir un stream parallèle à partir d'une collection.

Notez cependant que toutes les tâches ne bénéficient pas d'une exécution en parallèle, et certaines peuvent même être plus lentes à cause des coûts de surcharge liés à la mise en place du parallélisme.

4. Exemples et exercices pratiques

1. Utilisez `Collectors.toList()` pour convertir un Stream en List.
2. Utilisez `Collectors.toSet()` pour convertir un Stream en Set.
3. Utilisez `Collectors.toMap()` pour convertir un Stream en Map.
4. Calculez le nombre d'éléments dans un Stream.
5. Trouvez l'élément minimum et maximum d'un Stream.
6. Calculez la somme et la moyenne des éléments d'un Stream d'entiers.
7. Essayez d'exécuter certaines tâches en parallèle avec `parallelStream()`.

Cas pratiques – Les lambdas et les streams en action

1. Application des concepts précédents dans des scénarios réels
2. Comment choisir entre les lambdas et les méthodes traditionnelles
3. Meilleures pratiques pour utiliser les lambdas et les streams
4. Exemples et exercices pratiques

1. Application des concepts précédents dans des scénarios réels

Les expressions lambdas et les streams sont des outils puissants que vous pouvez utiliser dans diverses situations.

Ils peuvent vous aider à simplifier le code en remplaçant les classes anonymes, à améliorer la lisibilité du code, et à écrire des opérations en chaîne efficaces sur des collections.

Exemple : Imaginez que vous ayez une liste de personnes et que vous vouliez obtenir une liste des noms de toutes les personnes qui ont plus de 18 ans. Avec les lambdas et les streams, vous pouvez accomplir cela en une seule ligne de code.

```
List<String> names = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(Person::getName)
    .collect(Collectors.toList());
```

2. Comment choisir entre les lambdas et les méthodes traditionnelles

Il est important de comprendre quand utiliser les lambdas et les streams et quand utiliser les méthodes traditionnelles.

En général, les lambdas et les streams sont plus appropriés lorsque vous effectuez des opérations sur des collections ou lorsque vous travaillez avec des opérations qui peuvent être exprimées de manière plus claire et plus concise avec des lambdas.

Cependant, les méthodes traditionnelles peuvent être plus appropriées lorsque vous travaillez avec des opérations plus complexes qui ne peuvent pas être facilement exprimées avec des lambdas ou des streams.

3. Meilleures pratiques pour utiliser les lambdas et les streams

Assurez-vous que les lambdas restent petites et lisibles. Si une lambda devient trop complexe, considérez l'utilisation d'une méthode séparée.

Évitez les effets de bord dans les lambdas. Les lambdas sont supposées être des fonctions pures.

Utilisez les streams pour les opérations sur les collections. Cependant, n'oubliez pas que les streams ne sont pas toujours la solution la plus performante.

Utilisez `parallelStream` avec précaution. Il peut améliorer les performances pour les grandes collections, mais il peut aussi introduire des problèmes d'ordre et de synchronisation.

4. Exemples et exercices pratiques

1. Utiliser un stream pour filtrer et transformer une liste d'objets.
2. Créer une liste d'entiers et utiliser un stream pour obtenir une liste des carrés des nombres impairs.
3. Utiliser une expression lambda dans une méthode `sort`.
4. Créer une liste de chaînes de caractères et utiliser une lambda pour trier les chaînes par leur longueur.

Ressources supplémentaires

- Oracle Java Collections Tutorial
- Java Collections Framework Cheat Sheet
- Java 8 Stream Tutorial
- Java Docs: Interface Function
- Java Docs: Class Stream