Interfaces Fonctionnelles en Java

Le package `java.util.function` en Java fournit une collection d'interfaces fonctionnelles conçues pour prendre en charge la programmation fonctionnelle, introduite dans Java 8. Ces interfaces ont chacune une seule méthode abstraite, ce qui les rend compatibles avec les expressions lambda et les références de méthode. Cette réponse explique comment utiliser certaines des interfaces fonctionnelles les plus courantes de ce package—`Function<T, R>`, `Predicate<T>`, `Consumer<T>` et `Supplier<T>`—avec des exemples pratiques.

Qu'est-ce que les Interfaces Fonctionnelles ?

Une interface fonctionnelle est une interface avec exactement une méthode abstraite. Le package `java.util.function` propose des interfaces fonctionnelles prédéfinies pour des tâches courantes, de sorte que vous n'avez pas besoin de créer les vôtres. Ces interfaces sont largement utilisées avec les expressions lambda, les références de méthode et l'API Stream pour écrire du code concis et expressif.

Voici comment utiliser les interfaces clés :

---

1. `Function<T, R>` : Transformation de l'Entrée en Sortie

L'interface `Function<T, R>` représente une fonction qui prend une entrée de type `T` et produit une sortie de type `R`. Sa méthode abstraite est `apply`.

Exemple : Obtenir la Longueur d'une Chaîne de Caractères

```java
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = s -> s.length();
        System.out.println(stringLength.apply("Hello")); // Affiche : 5
    }
}
```

- **Explication** : L'expression lambda `s -> s.length()` définit une `Function` qui prend une `String` (`T`) et retourne un `Integer` (`R`). La méthode `apply` exécute cette logique.

---

2. `Predicate<T>` : Test d'une Condition

L'interface `Predicate<T>` représente une fonction à valeur booléenne qui prend une entrée de type `T`. Sa méthode abstraite est `test`.

Exemple : Vérifier si un Nombre est Pair

```
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println(isEven.test(4)); // Affiche : true
        System.out.println(isEven.test(5)); // Affiche : false
    }
}
```

- **Explication** : Le lambda `n -> n % 2 == 0` définit un `Predicate` qui retourne `true` si l'entrée est paire. La méthode `test` évalue cette condition.

---

3. `Consumer<T>` : Exécution d'une Action

L'interface `Consumer<T>` représente une opération qui prend une entrée de type `T` et ne retourne aucun résultat. Sa méthode abstraite est `accept`.

Exemple : Imprimer une Chaîne de Caractères

```
import java.util.function.Consumer;

public class Main {
    public static void main(String[] args) {
        Consumer<String> printer = s -> System.out.println(s);
        printer.accept("Hello, World!"); // Affiche : Hello, World!
    }
}
```

- **Explication** : Le lambda `s -> System.out.println(s)` définit un `Consumer` qui imprime son entrée. La méthode `accept` exécute l'action.

---

4. `Supplier<T>` : Génération d'un Résultat

L'interface `Supplier<T>` représente un fournisseur de résultats, ne prenant aucune entrée et retournant une valeur de type `T`. Sa méthode abstraite est `get`.

Exemple : Générer un Nombre Aléatoire

```
import java.util.function.Supplier;
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        Supplier<Integer> randomInt = () -> new Random().nextInt(100);
        System.out.println(randomInt.get()); // Affiche un entier aléatoire entre 0 et 99
    }
}
```

- **Explication** : Le lambda `() -> new Random().nextInt(100)` définit un `Supplier` qui génère un entier aléatoire. La méthode `get` récupère la valeur.

Utilisation des Interfaces Fonctionnelles avec les Streams

Ces interfaces brillent dans l'API Stream de Java, où elles permettent un traitement de données concis. Voici un exemple qui filtre, transforme et imprime une liste de chaînes de caractères :

Exemple : Traiter une Liste de Chaînes de Caractères

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList("a", "bb", "ccc", "dddd");

        Predicate<String> longerThanTwo = s -> s.length() > 2;       // Filtre les chaînes de caractères plus
        Function<String, String> toUpperCase = s -> s.toUpperCase(); // Convertit en majuscules
```

```
        Consumer<String> printer = s -> System.out.println(s);       // Imprime chaque résultat

        strings.stream()
               .filter(longerThanTwo)    // Garde "ccc" et "dddd"
               .map(toUpperCase)         // Convertit en "CCC" et "DDDD"
               .forEach(printer);        // Affiche : CCC, DDDD (sur des lignes séparées)
    }
}
```

- **Explication** :
  - `filter` utilise un `Predicate` pour garder les chaînes de caractères de longueur > 2.
  - `map` utilise une `Function` pour transformer les chaînes de caractères en majuscules.
  - `forEach` utilise un `Consumer` pour imprimer chaque résultat.

Utilisation des Références de Méthode Vous pouvez rendre cela encore plus court avec des références de méthode :

```
strings.stream()
       .filter(s -> s.length() > 2)
       .map(String::toUpperCase)      // Référence de méthode pour Function
       .forEach(System.out::println); // Référence de méthode pour Consumer
```

Composition des Interfaces Fonctionnelles

Certaines interfaces permettent la composition pour des opérations plus complexes : - **Composition de Fonctions** : Utilisez `andThen` ou `compose`. `java Function<String, Integer> toLength = s -> s.length(); Function<Integer, String> toString = i -> "Length is " + i; Function<String, String> combined = toLength.andThen(toString); System.out.println(combined.apply("Hello")); // Affiche : Length is 5` - **Combinaison de Prédicats** : Utilisez `and`, `or`, ou `negate`. `java Predicate<String> isLong = s -> s.length() > 5; Predicate<String> startsWithA = s -> s.startsWith("A"); Predicate<String> isLongAndStartsWithA = isLong.and(startsWithA); System.out.println(isLongAndStartsWithA.test("Avocado")); // Affiche : true`

Résumé

Voici quand et comment utiliser ces interfaces : - `Function<T, R>` : Transformer une entrée en sortie (par exemple, `apply`). - `Predicate<T>` : Tester une condition (par exemple, `test`). - `Consumer<T>` : Exécuter une

action sur une entrée (par exemple, `accept`). - `Supplier<T>` : Générer une valeur sans entrée (par exemple, `get`). - **Avec les Streams** : Les combiner pour un traitement de données puissant. - **Composition** : Les enchaîner ou les combiner pour une logique complexe.

Vous pouvez implémenter ces interfaces en utilisant des expressions lambda (par exemple, `s -> s.length()`) ou des références de méthode (par exemple, `String::toUpperCase`). Elles permettent un style de programmation fonctionnelle en Java, rendant votre code plus concis, lisible et réutilisable, surtout avec l'API Stream.