Développement Logiciel Java

12/02/2025

Les interfaces fonctionnelles

https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/function/package-summary.html

- Function: prend un ou deux paramètres et retourne une valeur
- Consumer: prend un ou deux paramètres mais ne renvoie rien
- Predicate: prend un ou deux paramètres et renvoie un booléen
- Supplier: ne prend pas de paramètres et renvoie un objet.

Toutes ces interfaces fonctionnelles permettent d'utiliser des fonctions lambdas dans une grande variété de situations.

Applications aux collections

- L'interface iterable définit une méthode forEach:

```
default void forEach(Consumer<? super T> action)
```

- T est le type associé à notre itérable
- `<? super T>` représente "n'importe quel supertype de T"
- Par exemple si notre iterable est défini: `Iterable<User> collection` le consumer peut traiter n'importe quel parent de User ou n'importe laquelle de ses interfaces implémentées.

```
collection.forEach(user -> user.setName("example"));
```

L’API Stream

- l’API java.util.stream contient des classes utilitaires pour manipuler des données en séquences.
- La classe utilitaire *Arrays* contient une méthode *.stream()* pour transformer facilement un tableau en stream
- La classe *Stream* contient une méthode *.of()* pour créer rapidement un Stream à partir d’une liste de paramètres
- Enfin, toutes les collections possèdent une méthode *.stream()* pour obtenir un stream à partir d’une collection.

L’API Stream

Les opérations sur les streams peuvent être:

- Intermédiaires (elles renvoient un stream)
- Terminales (elles renvoient une valeur ou un objet)

Les opérations sur les streams ne changent pas les données sources.

```
long count = collection.stream().distinct().count();
```

Javadoc

Méthode de filtrage

Permet d'obtenir tous les éléments qui répondent à un critère.

```
collection.stream().filter(e -> "Jones".equals(e.getName()));
```

Version "classique"

```
var filtered = new ArrayList<>();
for (User user : collection) {
   if ("Jones".equals(user.getName())) {
       filtered.add(user);
   }
}
```

Méthode de *Mapping*

Permet d'obtenir une collection à partir de la collection initiale, où chaque objet a été transformé. Par exemple, obtenir une liste de noms à partir d'une liste d'users:

```
collection.stream().map(user -> user.getName());
```

Remplace

```
var mapped = new ArrayList<>();
for (User user : collection) {
   mapped.add(user.getName());
}
```

Methodes de *Matching*

Méthodes terminales (renvoie un booléen):

- anyMatch() renvoie true si au moins un élément satisfait la condition (false si le stream est vide)
- allMatch() renvoie true si tous les éléments satisfont la condition (true si le stream est vide)
- noneMatch() renvoie true si aucun élément ne satisfait la condition.

```
collection.stream().anyMatch(e -> "Jones".equals(e.getName()));
```

Remplace:

```
var found = false;
for (User user : collection) {
   if (user.getName().equals("Jones")) {
       found = true;
   }
}
```

Méthode de *Réduction*

Réduit la séquence d'éléments à une valeur grâce à la méthode *reduce()*:

```
collection.stream()
      .map(user -> user.getAge()) // Transforme la liste des users en liste d'âges
      .reduce(0, (a, b) ->  a + b); // additionne les âges
```

Ou:

```
collection.stream().mapToInt(user -> user.getAge()).sum();
```

Remplace:

```
var sum = 0;
for (User user : collection) {
   sum += user.getAge();
}
```

Méthodes de Collection

Méthode terminale permettant d'obtenir un objet ou une valeur.

```
collection.stream()
      .filter(e -> "Jones".equals(e.getName()))
      .collect(Collectors.toList());
```

Récupère les éléments filtrés sous forme de liste

```
collection.stream()
      .map(user -> user.getName())
      .collect(Collectors.joining(", "));
```

Concatène tous les noms en les séparant par “, “

Les références de méthodes

Il existe une dernière façon de simplifier les fonctions lambdas quand elles contiennent une seule ligne.

```
user -> user.getName()
```

Est équivalent à:

```
User::getName
```

Par exemple, la somme des âges évoquée précédemment devient:

```
collection.stream().mapToInt(User::getAge).sum()
```

Les références de méthodes

Il est possible de faire référence à:

- Une méthode statique
- Une méthode d’instance d’un objet particulier
- Une méthode d’instance d’un type particulier
- Un constructeur

Les références de méthodes

Méthode statique:

```
Stream.of("1","2", "3").map(Long::valueOf)
```

Fait appel à la méthode statique *Long.valueOf*.

Equivalent à:

```
Stream.of("1","2", "3").map(val -> Long.valueOf(val));
```

Les références de méthodes

Méthode d'instance d'un objet spécifique:

```
class MyUserTransformer {
    public String transform(User input){
        return input.getName() + " - " + input.getAge();
    }
}
MyUserTransformer myUserTransformer = new MyUserTransformer();
collection.stream().map(myUserTransformer::transform);
```

Equivalent à:

```
collection.stream().map(user -> myUserTransformer.transform(user));
```

Les références de méthodes

Méthode d'instance d'un type particulier:

```
collection.stream().map(User::getAddress);
```

Est équivalent à la fonction lambda:

```
collection.stream().map(user -> user.getAddress());
```

Java est capable de déterminer que puisque l'argument de la lambda est de type *User*. Et que la méthode *getAddress* n'est pas *static*, la référence *User::getAddress* se traduit donc par l'appel de la méthode d'instance *getAddress()* sur chacun des éléments de la boucle.

L’API Optional

Certaines méthodes terminales de l’API Stream renvoient des types *Optional<T>*.

Le type Optional est utilisé lorsqu’il n’est pas certain que l’opération renvoie une valeur non null.

Il contient les méthodes *isEmpty()* et *isPresent()* pour vérifier que l’*optional* est vide / n’est pas vide.

Il contient aussi des méthodes permettant de gérer automatiquement le cas vide:

- .or() permet d’appeler une lambda pour retourner une valeur par défaut si l’*optional* est vide;
- .orElse() permet de fournir une valeur par défaut si l’*optional* est vide.
- orElseThrow() permet de lever une exception si l’optional est vide.

TP

- Créer une classe Student qui contient les champs suivants:

```
List<Long> maths;
List<Long> french;
List<Long> history;
Long mathAverage;
Long frenchAverage;
Long historyAverage;
Long totalAverage;
String name;
```

Dans le constructeur les listes doivent être initialisées avec chacune 10 notes comprises entre 0 et 20.
`Math.floor(Math.random()*20)`

- Dans le *main*, créez une dizaine d'utilisateurs dans un arrayList avec un nom et un âge.
- Utilisez la méthode forEach pour calculer pour chaque utilisateur sa moyenne en maths, sa moyenne en français et sa moyenne en histoire, puis sa moyenne globale des trois notes.
- Utilisez les streams pour vérifier que personne n'a moins de 5 de moyenne et qu'au moins un étudiant à plus de 10.
- Trouvez l'étudiant qui a la meilleure moyenne (indices: utilisez *stream.max()* et explorez les implémentations de *java.util.Comparator*)
- Calculez la moyenne de la classe (indice: si vous transformez un *Stream* en *LongStream* vous disposez alors d'une méthode *average*) .
- Créez une Map avec en clé le prénom et en valeur la moyenne de Maths.