Introduction programmation Java

Cours 8

Stéphane Airiau

Université Paris-Dauphine

Quelques détails vus en TD

Tests Junit

- chaque test est vu "presque comme" un nouveau `main`
  - a priori, par défault on ne contrôle pas l'ordre dans lequel les tests sont effectués
    on peut ajouter des instructions pour induire un ordre
    ⇨ il faut écrire les tests indépendemment les uns des autres
  - Les variables d'instance peuvent être recréées
- pas une bonne idée de ma part de vous demander d'écrire des méthodes dont le résultat est un affichage
  ⇨pourquoi est-ce difficile à tester ?

méthode `remove` dans une boucle `for`

Dans le TD, on demande d'implémenter une méthode qui enlève un élément dans une liste.
On pourrait écrire

```
List<String> ls = new LinkedList<>();
ls.add("Eenie");
ls.add("Meeney");
ls.add("Miney");
ls.add("Moe");
for(String w : ls){
    if(w.equals("Miny"))
        ls.remove(w);
}
```

et l'exécution donne

```
[Eenie, Meenie, Moe]
```

Pourtant, je dis que cela ne devrait pas marcher ! Pourquoi ?

méthode `remove` dans une boucle `for`

Il y a deux raisons pour lesquelles ce code n'est pas bien :

1- performance

2- cela devrait provoquer une erreur !

but nodoby's perfect !

`https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4902078`

méthode `remove` dans une boucle `for`

```
List<String> ls = new LinkedList<>();
ls.add("Eenie");
ls.add("Meeney");
ls.add("Miney");
ls.add("Moe");
Iterator<String> it = ls.iterator();
while(it.hasNext()){
    String w = it.next();
     if(w.equals("Miny"))
         it.remove();
}
```

equals

```
public class Box{
    static int count = 0;
    private int id;
    private int hgX, hgY, bdX, bdY;

    public Box(int a, int b, int c, int d){
        id= ++count;
        hgX=a; hgY=b; bdX=c; bdY=d;
    }

    public static void main(String[] args){
        List<Box> ls = new LinkedList<>();
        ls.add(new Box(0,100,200,50));
        ls.add(new Box(20,60,60,30));
        Box b = new Box(20,60,60,30);
        System.out.println(ls.contains(b));
    }
}
```

```
False
```

On peut lire le code des classes `Java`.

- soit sur le site de `openjdk`
- on peut "lier" le code source sous éclipse
  ⇨quand on a une erreur, on peut suivre le problème jusque dans le code source.

`http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/ArrayList.java`

- regardez par exemple la méthode `indexOf()`.
- par curiosité, on peut aussi regarder comment est géré la taille des tableaux et comment ils sont augmentés

Retour sur les streams

Remplacez des itérations

- en itérant, on précise l'ordre avec lequel on réalise des opérations
- avec les streams, on utilise des fonctions qui indique le traitement, on laisse `Java` choisir la meilleure manière de réaliser les opérations
  principe "what, not how"
- les opérations sur les streams sont "fainéantes" (lazy) : elles ne sont exécutée que quand on a vraiment besoin de les exécuter !

Utilisation avec les `Stream`

Supposons qu'on a une collection de chaînes de caractères et que l'on veut compter les mots de plus de 12 caractères.

```
1 int count =0;
2 for (String m: words)
3     if (m.length()>12)
4         count++;
```

L'idée avec les `streams` sera d'écrire le code suivant :

```
long count = words.stream().filter(w-> w.length > 12).count();
```

En lisant, on comprends exactement ce qui se passe.
Désormais, `Java` peut optimiser l'exécution de ce code

1. création d'un stream
2. opérations intermédiaires transformant le stream initial (en d'autres, pourrait utiliser plusieurs étapes)
3. opération terminale pour produire le résultat.

Nous allons voir plus en détail ces trois phases.

Création de `streams`

- A partir d'une collection : appel de la méthode `stream()`
- A partir d'un tableau :
  - utiliser la méthode de classe `of` de la classe `Stream` :

```
Stream<String> words = Stream.of(line.split(","));
// split découpe une chaîne de caractères et retourne un tableau de String
```

  - appel de la méthode de classe `stream(array, from, to)` de la classe `Arrays`

```
Stream<String> words = Arrays.stream(line.split(","), 3, 7);
```

- on peut créer un stream vide `Stream.empty();`
- on peut créer des stream infini
  - avec la méthode `generate` :

```
Stream<String> echos = Stream.generate( () -> "Echo”);
```

```
Stream<Double> randDoubles =
        Stream.generate(Math::random);
```

  - avec la méthode `iterate`

```
Stream<Integer> intSeq = Stream.iterate(0, n -> n.add(1);)
```

Création de `streams` (suite)

On peut aussi avoir une séquence finie avec `iterate` en ajoutant un test d'arrêt.

autres exemples de de création

- la méthode d'instance `tokens()` de la classe `Scanner` retourne un `Stream<String>`.
- la méthode de classe `lines(Path p)` de la classe `Files` retourne également un `Stream<String>`.

Transformation de `streams` : filtre

un filtre permet de récupérer un nouveau stream dont les éléments ont passé un test.

Le test doit donc être une fonction qui retourne un booléen
⇨ doit donc suivre l'interface fonctionnelle `Predicate<T>` qui est une fonction $T \rightarrow \{True, False\}$

```
long count = words.stream().filter(w-> w.length > 12).count();
```

Transformation de `streams` : `map`

L'opérateur `map` permet de *transformer* le stream.

```
Stream<String> lowerCase =
          words.stream().map(String::toLowerCase);
```

La plupart du temps, il n'existera pas une référence méthode adéquate, on pourra donc utiliser une expression λ.

```
Stream<String> firstLetters =
          word.stream().map( s -> s.substring(0,1));
```

Transformation de `streams` : `flatmap`

Si la fonction retourne un stream, on pourrait avoir comme résultat un stream de stream. Si on veut simplement un seul stream, on peut alors utiliser l'opérateur `flatmap`

transformation de stream

La méthode `limit(int n)` retourne un nouveau stream qui se termine après au plus `n` éléments

```
Stream<Double> randDoubles =
        Stream.generate(Math::random).limit(100);
```

La méthode `skip(int n)` fait l'opposé : elle retourne un stream sans les premiers `n` éléments.

La méthode `takeWhile(predicate)` prend tout élément tant que le prédicat est vrai, et s'arrête ensuite.

```
Stream.of(1,3,5,6,8,6,2,18)
.takeWhile(no -> no<=5).forEach(System.out::println);
```

La méthode `dropWhile(predicate)` fait le contraire : elle ne prend pas les éléments tant que le prédicat est vrai, et retourne donc le stream partant du moment ou le prédicat devient faux.

```
Stream.of(1,3,5,6,8,6,2,18).
dropWhile(no -> no<=5).forEach(System.out::println);
```

Autres transformations de streams

La méthode de classe `concat` de la classe Stream concatène deux streams. Evidemment, il vaudrait mieux que le premier stream ne soit pas infini !

La méthode `distinct` retourne un stream qui n'a pas de doublons.

```
Stream.of("boo", "bee", "boo", "boo”, "bee”, "boo").distinct();
```

va être un stream avec seulement "bee" et "boo".

La méthode `sorted` permet de trier un stream (mieux vaut qu'il contienne des objets d'une classe qui implémente `Comparable`!)

La méthode `peek` retourne le même stream, mais applique une fonction à chaque élément

```
Integer[] powersOfTwo =
          Stream.iterate(1.0, n ->2*n)
              .peek(e -> System.out.println("treating "+e))
              .limit(20).toArray();
```

Réduction de streams

Réduction simple : `count`, `min`, `max`
Point délicat : que se passe-t-il quand le stream est vide ?
⇨ certaines opération de réduction retourne une valeur de type `Optional<T>`.

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
```

findFirst retourne la première valeur dans une collection non vide

```
Optional<String> startsWithW =
        words.filter(s -> s.startsWith("W")).findFirst();
```

Il existe de la même manière `findAny` si on ne se focalise pas vraiment sur le premier.
Si on veut juste savoir s'il y a au moins un élément, `anyMatch` sera alors judicieux.

```
boolean b = words.anyMatch(s -> s.startsWith("W"));
```

Il existe également des méthodes `allMatch` et `NoneMatch` qui vérifient si tous ou aucun élément satisfait un prédicat.

Le type optionel

L'idée est de ne pas utiliser qu'une méthode retourne `null` quand il n'y a pas de résultat (provoque un `NullPointerException` pas toujours simple à corriger)

- donner une valeur alternative

```
String result = optionalString.orElse("");
```

- jeter une exception (on verra cela bientôt)

```
String result =
      optionalString.orElseThrow(IllegalStateException::new);
```

- lancer un calcul alternatif avec `orElseGet()`.

```
String result = optionalString.orElseGet(
          () -> if (Math.random()>0.5)
                      return "bonjour";
                  else return "bye");
```

`ifPresent` prend comme paramètre une fonction : si la valeur optionelle existe, elle est passée à cette fonction.

```
optionalValue.ifPresent(v -> results.add(v));
```

Obtenir le résultat

ex affichage :

```
stream.forEach(System.out::println);
```

ex : obtenir un tableau :

```
String[] result = stream.toArray(String[]::new);
```

Notez que le type de `stream.toArray()` est `Object[]` car on ne peut pas faire un tableau avec des génériques.
On peut avoir le bon type en passant le constructeur en paramètre.

L'interface `Collector` est utilisée pour collecter les éléments dans une autre structure. La classe `Collectors` propose des méthodes pour des structures classiques.

```
List<String> result = stream.collect(Collectors.toList());
```

```
Set<String> result = stream.collect(Collectors.toSet());
```

```
TreeSet<String> result =
      stream.collect(Collectors.toCollection(TreeSet::new));
```

Obtenir le résultat (suite)

concatenation de chaînes de caractères :

```
String result = stream.collect(Collectors.joining());
```

concatenation de chaînes de caractères avec ajout d'un délimiteur :

```
String result = stream.collect(Collectors.joining(", "));
```

On peut utiliser une méthode d'aggrégation (somme, compter, moyenne, min, max) :

```
IntSummaryStatistics summary = stream.collect(
                Collectors.summarizingInt(String::length)));
double avg = summary.getAverage();
double max = summary.getMax();
```

Obtenir le résultat : reduce

```
Optional<Integer> sum =
      values.stream().reduce((agg, next) -> agg +next);
```

est équivalent à la somme.
On retrouve ici des choses similaires à la programmation fonctionnelle.