# TD 11 : Java fonctionnel

**Objectifs pédagogiques :**

- ***lambda expressions***
- **flux (*stream*) et collections**

## 11.1 Tris et comparateurs

Nous considérons (comme au TD 2) une classe `Etudiant` qui représente un étudiant par son nom, son prénom (des chaînes de caractères) et son numéro d'étudiant (un entier). La classe a donc le constructeur et les méthodes de signature suivante :

```
public Etudiant(String nom, String prenom, int numeroEtudiant);
public String getNom();
public String getPrenom();
public int getNumeroEtudiant();
public String toString();
```

L'implémentation de cette classe ayant été vue dans un TD précédent et étant maintenant acquise, nous ne la demandons pas ici.

### 11.1.1 Tri par nom ou par numéro d'étudiant

Nous souhaitons trier des listes d'étudiants, `List<Etudiant>`, soit par nom, soit par numéro d'étudiant. Une solution pour cela est de construire, pour chaque critère de tri, un comparateur obéissant à l'interface `Comparator` (rappelée dans l'encadré ci-dessous) puis d'appeler la méthode `sort` de tri générique de listes avec ce comparateur en argument.

**Question 1.** Définissez dans la classe `Etudiant` des attributs `parNom` et `parNumeroEtudiant` de type `Comparator`, correspondant respectivement à l'ordre alphabétique des noms (vous ignorerez les prénoms) et à l'ordre croissant des numéros d'étudiant. Vous initialiserez ces attributs avec des *lambda expressions* (voir les encadrés ci-dessous pour des rappels de cours). Ces attributs sont-ils `static` ? `final` ? Quelle est leur visibilité ?

**Interface `Comparator<T>`.** L'interface `java.util.Comparator<T>` est une interface fonctionnelle proposant une méthode abstraite `compare` :

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

Elle prend en argument deux objets de classe `T` et renvoie un entier strictement positif si `o1` est strictement plus grand que `o2`, nul si `o1` et `o2` sont égaux, et strictement négatif si `o1` est strictement plus petit que `o2`.
L'interface `List<T>` possède une méthode de tri paramétrée par un tel `Comparator` :

```
void sort(Comparator<? super T> c);
```

Le type `Comparator<? super T>` indique que le comparateur utilisé doit être capable de comparer des objets de classe `T` ou d'une classe parent de `T`. Ainsi un `Comparator<Object>` sera utilisable pour trier une liste `List<String>`, car le comparateur compare des objets de type `? = Object`, parent de `T = String`.

***Lambda expressions.*** Rappelons les syntaxes possibles des *lambda expressions* :

- `(T1 arg1,..., TN argN) -> { code }`
- `(T1 arg1,..., TN argN) -> expression`
  équivalent à `(T1 arg1,..., TN argN) -> { return expression; }`
- les types `T1`, ..., `TN` peuvent être omis la plupart du temps, car le compilateur les infère, ce qui donne : `(arg1,...,argN) -> { code }` ou `(arg1,...,argN) -> expression`
- sans argument : `() -> { code }`
- avec un unique argument, sans type : `arg -> { code }` ou `arg -> expression` (les parenthèses sont alors optionnelles)

Le type d'une *lambda expression* est compatible avec toute interface fonctionnelle, c'est-à-dire une interface ayant une unique méthode abstraite, du moment que la méthode possède `N` arguments et que le type de ses arguments est compatible avec ceux de `arg1` à `argN`.

**Question 2.** Donnez la ligne de code utilisée par un client pour trier dans l'ordre alphabétique des noms une liste d'étudiants `List<Etudiant> list`. Donnez également la ligne utilisée pour trier dans l'ordre croissant des numéros d'étudiant. Discutez la différence entre fournir des objets `Comparator` et implémenter l'interface `Comparable`.

### 11.1.2 Factorisation des comparateurs

Nos comparateurs fonctionnent tous de manière similaire : ils extraient un attribut (chaîne ou entier) de chaque objet `Etudiant`, puis ils utilisent la comparaison naturelle sur ces attributs. Dans cette question, nous allons factoriser ces comparateurs en définissant un comparateur générique paramétré par un *getter*. Un tel *getter* peut s'écrire comme une *lambda expression* `(Etudiant e) -> e.getNom()` ou, de manière plus concise, `Etudiant::getNom` (voir encadré ci-dessous).

**Références de méthodes comme *lambda expressions*.** Il est possible d'utiliser une référence à une méthode d'une classe là où une *lambda expression* est attendue. Diverses syntaxes sont possibles, selon que la méthode est statique, est un constructeur, est non statique avec un objet cible fixé ou est non statique avec un objet cible passé en argument. Ainsi :

- `Classe::methode`
  correspond à `(T1 arg1, ...) -> Classe.methode(arg1, ...)` pour une méthode statique ;
- `Classe::new`
  correspond à `(T1 arg1, ...) -> new Classe(arg1, ...)` pour un constructeur ;
- `obj::methode`
  correspond à `(T1 arg1, ...) -> obj.methode(arg1, ...)` pour une méthode non statique ;
- `Classe::methode`
  correspond à `(Classe obj, T1 arg1, ...) -> obj.methode(arg1,...)` pour une méthode non statique.

**Question 3.** Donnez le type des *getters* de `Etudiant::getNom`, `Etudiant::getPrenom`, `Etudiant::getNumeroEtudiant` sous forme d'interface fonctionnelle de la bibliothèque standard (voir encadré ci-dessous).

**Interfaces fonctionnelles de Java 8.** Le paquetage `java.util.function` fournit un ensemble d'interfaces fonctionnelles génériques très utiles pour donner rapidement un type à une *lambda expression*. Par exemple :

- `Function<T,R>` a une méthode abstraite `R apply(T)` ;
- `UnaryOperator<T>` correspond à `Function<T,T>`, avec une méthode abstraite `T apply(T)` ;
- `BinaryOperator<T>` a une méthode abstraite `T apply(T,T)` ;
- `Supplier<T>` a une méthode abstraite `T get()` ;
- `Consumer<T>` a une méthode abstraite `void accept(T)` ;
- `Predicate<T>` a une méthode abstraite `boolean test(T)`.

**Question 4.** Programmez une méthode `makeComparator` qui prend en paramètre un *getter* d'attribut chaîne (comme `getNom`), et qui retourne un `Comparator`. Montrez comment retrouver le comparateur `parNom` et comment créer le comparateur `parPrenom` avec `makeComparator`. Est-il possible d'appliquer votre `makeComparator` pour construire des comparateurs sur d'autres classes qu'`Etudiant` ?

**Question 5.** Montrez comment étendre `makeComparator` pour qu'il puisse prendre en argument un *getter* renvoyant une valeur de type arbitraire (et plus uniquement une chaîne). Montrez comment retrouver `parNumeroEtudiant` avec `makeComparator`.

### 11.1.3 Comparateur lexicographique

Étant donnés deux ensembles ordonnés, l'ordre lexicographique permet d'ordonner des paires d'éléments : la paire $(a, b)$ est plus petite que la paire $(a', b')$ si $a$ est strictement plus petit que $a'$, ou si $a = a'$ et $b$ est plus petit que $b'$. Dans cette question, nous allons construire un combinateur lexicographique, c'est-à-dire une méthode prenant deux `Comparator` en argument et renvoyant un `Comparator` correspondant à l'ordre lexicographique.

**Question 6.** Proposez une telle méthode `makeLexicographique`. Utilisez ce combinateur pour trier par nom complet, c'est-à-dire par nom d'abord, puis par prénom pour ordonner les étudiants de même nom.

## 11.2 Flux (*Streams*)

### 11.2.1 Filtrage de flux et opérations élémentaires

Reprenons une liste d'étudiants `List<Etudiant>`. Nous souhaitons construire une nouvelle liste composée des numéros d'étudiant des étudiants dont le nom commence par une lettre entre A et L ; la liste étant triée dans l'ordre alphabétique des noms et des prénoms.

**Question 7.** Proposez une méthode pour créer cette nouvelle liste en utilisant les flux `java.util.stream` (voir encadré en fin de sujet).

### 11.2.2 Réduction

Étant donné un flux de collections, nous souhaitons calculer la somme des tailles des éléments du flux. Une approche prometteuse semble être la méthode :

```
public static<T> int totalSize(Stream<Collection<T>> stream) {
  int sum = 0;
  stream.forEach(e -> { sum += e.size(); });
  return sum;
}
```

**Question 8.** Cette méthode ne compile pas. Expliquez pourquoi.

**Question 9.** Proposez une implémentation correcte utilisant `map` et `reduce` (voir encadré en fin de sujet).

### 11.2.3 Flux utilisateur

Dans cette question, nous cherchons à générer nous-mêmes nos flux (infinis) d'entiers, grâce aux méthodes `generate` et `iterate` (voir encadré ci-dessous).

**Question 10.** Proposez une méthode `cstStream(int cst)` qui crée un flux infini d'entiers renvoyant toujours la valeur `cst`.

**Question 11.** Proposez une méthode `seqStream(int first)` qui crée un flux infini d'entiers renvoyant la séquence de valeurs `first`, `first+1`, `first+2`, etc.

**API `Stream`.** La classe `java.util.stream.Stream<T>` permet de manipuler des flux d'éléments de classe `T`. Quelques opérations importantes :

- `map(Function<T,U>)` transforme un flux de `T` en un flux de `U` en appliquant à chaque élément une fonction passée en argument ;
- `filter(Predicate<T>)` garde uniquement les éléments satisfaisant un prédicat passé en argument ;
- `sorted(Comparator<T>)` trie les éléments du flux vis-à-vis d'un comparateur passé en argument.

Un flux n'est pas une collection, mais il est possible de convertir un flux en collection et *vice versa* :

- La méthode `stream()` de `Collection` retourne un flux correspondant à une collection.
- La méthode `collect(Collector<T>)` de `Stream` retourne une nouvelle collection contenant les éléments du flux (dans l'ordre). L'argument de type `Collector<T>` précise la nature de la collection. La classe `Collectors` contient des méthodes statiques permettant d'obtenir les collecteurs les plus utiles, dont `toList()` pour obtenir une `List<T>`.

Par ailleurs, un flux est une structure *fonctionnelle* : chaque application d'une méthode retourne un nouveau flux et invalide le flux original, qui ne doit plus être utilisé.
D'autres méthodes existent pour créer des flux ou bien en extraire du contenu, notamment :

- la méthode statique `generate(Supplier<T> s)` crée le flux des éléments obtenus par les appels successifs à la méthode `get()` de `s` ;
- la méthode statique `iterate(T seed, UnaryOperator<T> f)` crée le flux des éléments obtenus à partir de `seed` par application répétée de l'opérateur `f` ;
- `reduce(T identity, BinaryOperator<T> acc)` produit un unique élément de type `T` en accumulant les éléments du flux : elle part de `identity` et, pour chaque élément du flux, calcule un nouveau résultat en appliquant l'opérateur binaire `acc` entre le résultat précédent et cet élément du flux, jusqu'à épuisement du flux.

Le paquetage `java.util.stream` comporte également des classes de flux spécialisées pour les types primitifs : `IntStream`, `DoubleStream`, etc. qui ne rentrent pas dans le moule générique `Stream<T>`.