Chapitre 4
Les expressions lambda

1. Introduction

Les expressions lambda ont été longuement attendues dans l'écosystème Java. Elles sont apparues avec Java 8. Java a ainsi rattrapé son retard sur certains langages concurrents comme le C#. Mais que sont les expressions lambda ? Ce sont tout simplement des fonctions que l'on peut passer en paramètre d'une autre fonction. Cela permet de simplifier le code dans certaines situations là où le développement objet offre des solutions verbeuses à base de classes anonymes notamment.

2. Fonctionnement

2.1 Les interfaces fonctionnelles

Le fonctionnement des expressions lambda s'appuie sur le fonctionnement des interfaces. En effet, un paramètre de méthode attendant une expression lambda est tout simplement un paramètre du type d'une interface. **Cette interface** doit cependant respecter une règle importante : elle **ne doit définir la signature que d'une seule méthode abstraite**.

Dans ce cas de figure, l'interface devient une interface fonctionnelle. Il est toujours possible d'avoir des méthodes complémentaires si elles ne sont pas abstraites (méthodes par défaut, méthodes statiques).

Pour exemple, voici le code simplifié de l'interface `java.util.Comparator<T>`. Son rôle est de proposer un mécanisme de comparaison d'objets d'un même type :

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

Une interface fonctionnelle est une interface comme les autres dans sa structure. La différence est la présence de l'annotation `@FunctionalInterface` qui est d'ailleurs optionnelle, mais qui permet au compilateur de faire les vérifications de cohérence nécessaire. Tant que vous n'avez pas exactement une et une seule méthode abstraite dans l'interface, le compilateur indiquera le message suivant :

Java propose un ensemble d'interfaces fonctionnelles répondant à la grande majorité des situations. Il ne sera donc pas forcément fréquent d'en créer soi-même. Elles sont disponibles dans le package `java.util.function`. Une section dédiée permettra d'explorer le contenu de ce package.

Un exemple concret d'utilisation des interfaces fonctionnelles est la méthode `sort` de l'interface `List<T>`. Cette méthode permet de trier les éléments de la liste selon un critère de comparaison fourni en paramètre de la méthode sous la forme d'une implémentation de l'interface `Comparator<T>` qui, je le rappelle, permet de comparer deux objets du même type.

Voici la signature de la méthode `sort` :

```
void java.util.List.sort(Comparator<? super String> c)
```

La méthode `sort` attend un paramètre de type `Comparator<T>`. Comme c'est une interface fonctionnelle, il est possible de passer en paramètre :

– Une instance de classe implémentant cette interface :

```
List<String> liste = new ArrayList<String>(
                    List.of("bonjour", "tout", "le", "monde"));

liste.sort(new Comparator<String>() {

           @Override
           public int compare(String s1, String s2) {
               return s1.compareTo(s2);
           }
       });
```

Pour plus d'informations sur les classes anonymes, veuillez vous référer à la section éponyme dans le chapitre Programmation objet.

– Une expression lambda (ou méthode anonyme) respectant la signature de l'unique méthode abstraite de cette interface :

```
liste.sort((s1,s2)->s1.compareTo(s2));
```

Pour plus d'informations sur les règles syntaxiques, veuillez vous référer à la section suivante, Les méthodes anonymes.

– Une référence vers une variable du type de l'interface :

```
Comparator<String> comparateur = (s1, s2)->s1.compareTo(s2);
liste.sort(comparateur);
```

– Une référence vers une méthode existante respectant la signature de l'unique méthode abstraite de cette interface :

```
private static int compare(String s1, String s2)
{
   return s1.compareTo(s2);
}
```

Pour passer en paramètre une référence vers cette méthode, l'écriture est la suivante :

```
liste.sort(MaClasse::compare);
```

Pour plus d'informations sur les règles syntaxiques, veuillez vous référer à la section Les références de méthodes un peu plus loin dans ce chapitre.

Ces premiers exemples montrent qu'une expression lambda offre une syntaxe beaucoup plus concise qu'une classe anonyme. La référence de méthode permet en plus de factoriser le traitement dans une méthode réutilisable.

2.2 Les méthodes anonymes

Il existe différentes syntaxes d'écriture des méthodes anonymes en fonction de différents critères :

– Le nombre de paramètres,
– Le type de méthode : fonction ou procédure,
– Le nombre d'instructions à écrire dans la méthode anonyme.

2.2.1 Syntaxe générale

Une expression lambda a la forme générale suivante :

```
paramètre -> corps
```

Une expression lambda doit déclarer les paramètres avant de déclarer le corps de la méthode. Ces deux blocs sont forcément séparés par une flèche `->` opérateur *arrow*).

2.2.2 Déclaration des paramètres

S'il n'y a aucun paramètre, la déclaration s'écrit de la manière suivante :

```
() -> corps
```

La paire de parenthèses indique l'absence de paramètre.

S'il y a un ou plusieurs paramètres, la déclaration peut s'écrire de la manière suivante :

```
(TypeParametre nomParametre, ...) -> corps
```

Cela se fait donc exactement comme pour une méthode classique.

Les méthodes anonymes offrent cependant quelques facilités :

– Il n'est pas obligatoire de définir le type des paramètres. Le compilateur met en œuvre l'inférence de type pour deviner le type des paramètres. Il est donc possible de simplifier l'écriture précédente en ôtant les types :

```
(nomParametre, ...) -> corps
```

– S'il y a un unique paramètre, il est même possible d'ôter les parenthèses pour la déclaration des paramètres :

```
nomParametre -> corps
```

Dans ce cas là, il n'est pas possible de définir explicitement le type du paramètre.

2.2.3 Déclaration du corps

Après la déclaration des paramètres, il est nécessaire de déclarer le corps de la méthode anonyme. Pour cela, il est possible d'utiliser la même organisation qu'une méthode classique en utilisant les accolades (`{}`) comme délimiteurs :

```
... -> {
   instruction;
   instruction;
   [return ...;]
}
```

Tout comme pour les paramètres, les méthodes anonymes offrent quelques facilités pour l'écriture du corps de la méthode.

– Si la méthode ne contient qu'une seule instruction (ce qui est souvent le cas dans une méthode anonyme), alors il est possible d'enlever les accolades et le point-virgule !

```
... -> l'unique instruction sans point-virgule
```

– Si la méthode ne contient qu'une seule instruction dont le résultat doit être retourné, il n'est pas utile d'utiliser le mot-clé `return` :

```
... -> l'unique instruction retournant une valeur sans return
et point-virgule
```

2.2.4 Utilisation des variables "externes"

Une expression lambda peut accéder à des variables "externes", c'est-à-dire des variables déclarées en dehors de l'expression lambda. Il faut simplement que la variable soit déclarée `final` ou qu'elle soit effectivement `final`. Une variable est effectivement `final` si elle n'est pas déclarée `final` mais qu'elle est utilisée comme si elle était `final` dans l'expression lambda (c'est-à-dire qu'elle n'est utilisée qu'en lecture).

2.3 Les références de méthodes

La seconde manière d'exploiter les interfaces fonctionnelles est d'utiliser les références de méthodes. Il est possible d'utiliser en lieu et place d'une expression lambda une référence vers une méthode dont la signature respecte la signature de l'unique méthode abstraite de l'interface fonctionnelle. La manière de référencer peut évoluer en fonction de différentes situations décrites dans les sections suivantes.

2.3.1 Méthode d'instance

Si la méthode à référencer est une méthode d'instance, la syntaxe à utiliser est la suivante :

```
nomDeLaVariable::nomDeLaMethode
```

Si la méthode à référencer est sur l'instance en cours de manipulation, la syntaxe devient la suivante :

```
this::nomDeLaMathode
```

Les deux-points (`::`) permettent de séparer le nom de la méthode de son "propriétaire". Ils permettent aussi d'indiquer au compilateur que ce n'est pas un appel de méthode que l'on souhaite réaliser, mais simplement une référence. Notez bien aussi qu'aucun paramètre n'est fourni, les parenthèses sont absentes.

2.3.2 Méthode de classe

Si la méthode à référencer est une méthode de classe (autrement dit statique), la syntaxe diffère quelque peu :

```
NomDeLaClasse::nomDeLaMethode
```

2.3.3 Constructeur

Si la méthode à référencer est un constructeur, la syntaxe est la suivante :

```
NomDeLaClasse::new
```

Pour un tableau, il faudra utiliser la syntaxe suivante :

```
NomDeLaClasse[]::new
```

Pour une classe générique, il faudra utiliser la syntaxe suivante :

```
NomDeLaClasse<UnType>::new
```

2.4 L'API java.util.function

2.4.1 Présentation de l'API

L'API `java.util.function` contient 43 interfaces fonctionnelles et donc autant de signatures de méthodes abstraites. La plupart de ces méthodes sont génériques et s'adaptent donc en fonction du contexte dans lequel vous souhaitez les utiliser. La javadoc est disponible à l'adresse suivante : https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/function/package-summary.html

Ces 43 interfaces respectent une convention de nommage. Le suffixe permet de déterminer le rôle général de l'interface :

– `XxxConsumer` : les interfaces `XxxConsumer` permettent de déterminer la signature d'une procédure. En fonction de l'interface, la procédure accepte un ou deux paramètres. Bien sûr, elle ne retourne rien. Ce type d'interface permet d'effectuer une action. La méthode abstraite de ces interfaces s'appelle toujours `accept`.

Chapitre 2
La conception orientée objet

1. Approche procédurale et décomposition fonctionnelle

Avant d'énoncer les bases de la programmation orientée objet, nous allons revoir l'approche procédurale à l'aide d'un exemple concret d'organisation de code.

Dans cet ouvrage reviennent souvent les termes "code", "coder", "codage" ; il ne s'agit ni d'alarmes ni de fonctions de cryptage de mot de passe. Le code ou code source est en fait le nom donné au contenu que le développeur entre à longueur de journée dans son éditeur de texte favori pour être ensuite converti (ou encore compilé) en flux d'instructions exécutables par l'ordinateur.

La programmation procédurale est un paradigme de programmation considérant les différents acteurs d'un système comme des objets pratiquement passifs qu'une procédure centrale utilisera pour une fonction donnée.

Prenons l'exemple de la distribution d'eau courante dans nos habitations et essayons d'en modéliser le principe dans une application très simple. L'analyse procédurale (tout comme l'analyse objet d'ailleurs) met en évidence une liste d'objets qui sont :

– le robinet de l'évier ;
– le réservoir du château d'eau ;
– un capteur de niveau d'eau avec contacteur dans le réservoir du château d'eau ;
– la pompe d'alimentation puisant l'eau dans la rivière.

Le code du programme "procédural" consisterait à créer un ensemble de variables représentant les paramètres de chaque composant puis à écrire une boucle de traitement de gestion centrale testant les valeurs lues et agissant en fonction du résultat des tests. On notera qu'il y a d'un côté les variables, de l'autre, les actions et au-dessus le gestionnaire tout puissant par qui tout transite.

2. La transition vers l'approche objet

La programmation orientée objet est un paradigme de programmation considérant les différents acteurs d'un système comme des objets actifs et en relation. L'approche objet est souvent très voisine de la réalité.

Dans notre exemple :

– L'utilisateur ouvre le robinet.
– Le robinet relâche la pression et l'eau s'écoule du réservoir du château d'eau à l'évier.
– Comme notre utilisateur n'est pas tout seul à consommer, le capteur/flotteur du réservoir arrive à un niveau qui déclenche la pompe de puisage dans la rivière.
– L'utilisateur referme le robinet.
– Alimenté par la pompe, le réservoir du château d'eau continue à se remplir jusqu'à ce que le capteur/flotteur atteigne le niveau suffisant qui arrêtera la pompe.

– Arrêt de la pompe.

Dans cette approche, vous constatez que les objets "interagissent" ; il n'existe pas de traitement central définissant dynamiquement le fonctionnement des objets. Il y a eu, en amont, une analyse fonctionnelle qui a conduit à la création des différents objets, à leur réalisation et à leur mise en relation.

Le code du programme "objet" va suivre cette réalité en proposant autant d'objets que décrit précédemment mais en définissant entre ces objets des méthodes d'échange adéquates qui conduiront au fonctionnement escompté.

Remarque

Les concepts objets sont très proches de la réalité...

3. Les caractéristiques de la POO

3.1 L'objet, la classe et la référence

3.1.1 L'objet

L'objet est l'élément de base de la POO. L'objet est la réunion :

– d'une liste de variables d'états ;

– d'une liste de comportements ;

– d'une identification.

Les variables d'états changent durant la vie de l'objet. Prenons le cas d'un lecteur de musiques numériques. À l'achat de l'appareil, les états de cet objet pourraient être :

– mémoire libre = **100%** ;

– taux de charge de la batterie = **correct** ;

– aspect extérieur = **neuf**.

Au fur et à mesure de son utilisation, ses états vont être modifiés. Rapidement la mémoire libre va chuter, le taux de charge de la batterie variera et l'aspect extérieur va changer en fonction du soin apporté par l'utilisateur.

Les comportements de l'objet définissent ce que peut faire cet objet :

– Jouer de la musique
– Aller à la piste suivante
– Aller à la piste précédente
– Monter le son
– etc.

Une partie des comportements de l'objet est accessible depuis l'extérieur de cet objet : Bouton Play, Bouton Stop... et une autre partie est uniquement interne : lecture de la carte mémoire, décodage de la musique à partir du fichier. On parle "d'encapsulation" pour définir la limite entre les comportements accessibles de l'extérieur et les comportements internes.

L'identification d'un objet est une information, détachée de la liste des états, permettant de différencier cet objet particulier de ses congénères (c'est-à-dire d'autres objets qui ont le même type que lui). L'identification peut être un numéro de référence ou une chaîne de caractères unique construite à la création de l'objet ; elle peut également être l'adresse mémoire où l'objet est stocké. En réalité, sa forme dépend totalement de la problématique associée.

L'objet programmé peut être attaché à une entité bien réelle, comme pour notre lecteur numérique, mais il peut être également attaché à une entité totalement virtuelle comme un compte client, l'entrée d'un répertoire téléphonique... La finalité de l'objet informatique est de gérer l'entité physique ou de l'émuler.

Même si ce n'est pas dans sa nature de base, l'objet peut être rendu persistant, c'est-à-dire que ses états peuvent être enregistrés sur un support physique mémorisant l'information de façon intemporelle. À partir de cet enregistrement, l'objet pourra être reconstruit avec ses états quand le système le souhaitera. Le support typique capable d'une telle prouesse est la base de données.

3.1.2 La classe

La classe est le "moule" à partir duquel l'objet va être créé en mémoire. La classe décrit les états et les comportements communs d'un même type. Les valeurs de ces états seront contenues dans les objets issus de la classe.

Les comptes courants d'une même banque contiennent tous les mêmes paramètres (coordonnées du détenteur, solde, etc.) et ils ont tous les mêmes fonctions (créditer, débiter, nous rappeler à l'ordre si besoin, etc.). Ces définitions doivent être contenues dans une classe et, à chaque fois qu'un client ouvre un nouveau compte, cette classe servira de modèle à la création du nouvel objet compte.

Le présentoir de lecteurs de musiques numériques propose un même modèle en différentes couleurs, avec des tailles mémoires différentes et modulables, etc. Chaque appareil du présentoir est un objet qui a été fabriqué à partir des informations d'une seule classe. À la réalisation, les attributs de l'appareil ont été choisis en fonction de critères esthétiques et commerciaux décidés par des chefs produits très au fait des tendances d'achats des clients.

Une classe peut contenir beaucoup d'attributs. Ils peuvent être de type primitif – des entiers, des caractères... – mais également de type plus complexe. En effet, une classe peut contenir une ou plusieurs classes d'autres types. On parle alors de composition ou encore de "couplage fort". Dans ce cas la destruction de la classe principale entraîne, évidemment, la destruction des classes qu'elle contient. Par exemple, si une classe hôtel contient une liste de chambres, la destruction de l'hôtel entraîne la destruction de ses chambres.

Une classe peut faire "référence" à une autre classe ; dans ce cas, le couplage est dit "faible" et les objets peuvent vivre indépendamment. Par exemple, votre PC est relié à votre imprimante. Si votre PC rend l'âme, votre imprimante fonctionnera certainement avec votre futur PC ! On parle alors d'association.

En plus de ses attributs, la classe contient également une série de "comportements", c'est-à-dire une série de méthodes avec leurs signatures et leurs implémentations attachées. Ces méthodes appartiennent aux objets et sont utilisées telles quelles.

On déclare la classe et son contenu dans un même fichier source à l'aide d'une syntaxe que l'on étudiera en détail. Les développeurs C++ apprécieront le fait qu'il n'existe plus, d'un côté, une partie définitions du contenu de la classe et, de l'autre, une partie implémentations. En effet, en Java (tout comme en C#), le fichier programme (d'extension .java) contient les définitions de tous les états avec éventuellement une valeur "de départ" et les définitions et implémentations de tous les comportements. Contrairement au C#, une classe Java de haut niveau – nous reviendrons sur cette notion plus loin – doit être définie dans un seul fichier source.

3.1.3 La référence

Les objets sont construits à partir de la classe, par un processus appelé l'instanciation, et donc, tout objet est une instance d'une classe. Chaque instance commence à un emplacement mémoire unique. L'adresse de cet emplacement mémoire connue sous le nom de pointeur par les développeurs C et C++ devient une référence pour les développeurs Java et C#.

Quand le développeur a besoin d'un objet pendant un traitement, il doit faire deux actions :

– déclarer et nommer une variable du type de la classe à utiliser ;

– instancier l'objet et enregistrer sa référence dans cette variable.

Une fois cette instanciation réalisée, le programme accédera aux propriétés et aux méthodes de l'objet par la variable contenant sa référence. Chaque instance est unique. Par contre, plusieurs variables peuvent "pointer" sur une même instance. C'est d'ailleurs quand plus aucune variable ne pointe sur une instance donnée que le ramasse-miettes enregistre cette instance comme devant être détruite.

3.2 L'encapsulation

L'encapsulation consiste à créer une sorte de boîte noire contenant en interne un mécanisme protégé et en externe un ensemble de commandes qui vont permettre de la manipuler. Ce jeu de commandes est fait de telle sorte qu'il sera impossible d'altérer le mécanisme protégé en cas de mauvaise utilisation. La boîte noire sera si opaque qu'il sera impossible à l'utilisateur d'intervenir en direct sur le mécanisme.

Vous l'aurez compris, la boîte noire n'est autre qu'un objet avec des méthodes publiques de "haut niveau" contrôlant avec rigueur les paramètres passés avant de les utiliser dans un traitement. En respectant le principe de l'encapsulation, vous ferez en sorte que jamais l'utilisateur de l'objet ne puisse accéder "en direct" à vos données. Grâce à ce contrôle, votre objet sera correctement utilisé, donc plus fiable, et sa mise au point sera plus facile. Dans le monde Java, les méthodes permettant d'accéder en lecture aux données sont des accesseurs et celles permettant d'accéder en écriture sont des mutateurs.

Java ne propose pas d'équivalence aux propriétés (*properties*) du langage C# qui permettent de garder le côté pratique de l'accès direct aux données de l'objet tout en respectant les principes de l'encapsulation.

3.3 L'héritage

Une autre caractéristique importante de la POO consiste à pouvoir créer une classe en partant d'une autre. Cela s'appelle l'héritage. Pour l'expliquer, imaginons que nous devions construire un système de gestion des différents types de collaborateurs d'une société. Une analyse rapide met en évidence une liste de propriétés communes à tous les postes. En effet, que le collaborateur soit ouvrier, cadre, intérimaire ou encore directeur, il a toujours un nom, un prénom et un numéro de sécurité sociale... On appelle cela la généralisation ; elle consiste à factoriser les éléments communs d'un ensemble de classes dans une classe plus générale appelée superclasse en Java et plutôt "classe de base" en C# et C++. La classe qui hérite de la superclasse est appelée sous-classe, classe héritière ou encore classe spécialisée.