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 4
Les bases de données avec Java EE

1. Modélisation de la base de données avec UML 2

Comme dans tout projet de développement d'application, il est primordial de bien penser et de bien concevoir en amont une base de données adaptée au besoin.

En effet, une base de données bien modélisée permet à la fois d'optimiser les requêtes SQL et le code d'un point de vue de la performance et de la maintenabilité.

1.1 Cahier des charges

L'objectif est de modéliser une base de données objet à l'aide d'UML 2.

Tout d'abord, nous allons lister les principales fonctionnalités attendues par les utilisateurs. Il va en découler naturellement les différentes tables à créer, implémenter, les relations entre elles et les différentes autres contraintes.

L'application web Java Devis Pro BTP permet à un artisan de gérer son portefeuille client, d'éditer des devis et des factures correspondant aux travaux qu'il va réaliser ou qu'il a réalisés.

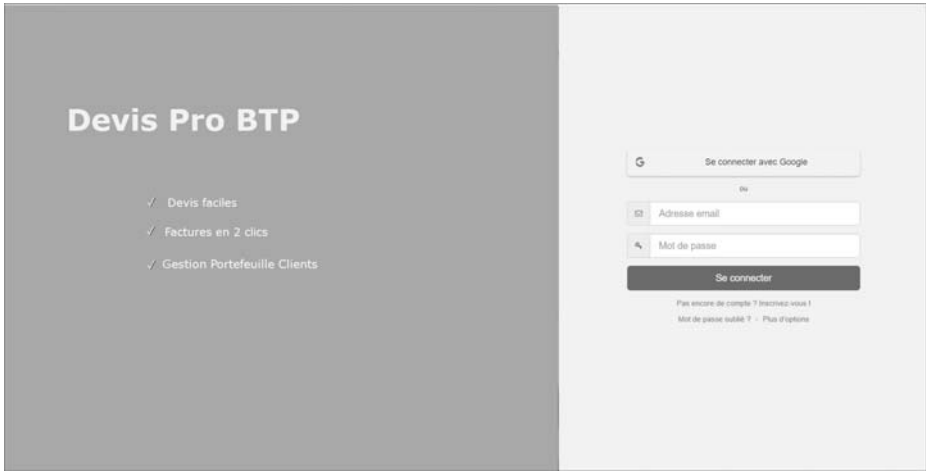
Voici ci-dessous les principales fonctionnalités de l'application :

- créer un devis,
- consulter un devis,
- modifier un devis,
- supprimer un devis,
- envoyer le devis à un client,
- ajouter un nouveau client,
- consulter une fiche client,
- rechercher un client à partir de son nom,
- mettre à jour les informations relatives à un client,
- supprimer une fiche client,
- créer une facture,
- consulter une facture,
- modifier une facture,
- supprimer une facture,
- rechercher une facture,
- gérer son profil artisan,
- se connecter,
- se déconnecter,
- changer son mot de passe,
- gérer le mot de passe et l'identifiant perdus.

1.2 Interface utilisateur

Afin de modéliser au mieux le besoin utilisateur, il est intéressant de s'appuyer dès à présent sur quelques prototypes ou maquettes des futures interfaces utilisateur.

Ci-dessous se trouve la maquette correspondant à l'interface qui permet à l'artisan de s'authentifier à l'application web de gestion pro des devis. Grâce à elle, il peut, par exemple créer un devis des travaux à réaliser chez un client.



Il est possible de s'authentifier de deux façons auprès de l'application.

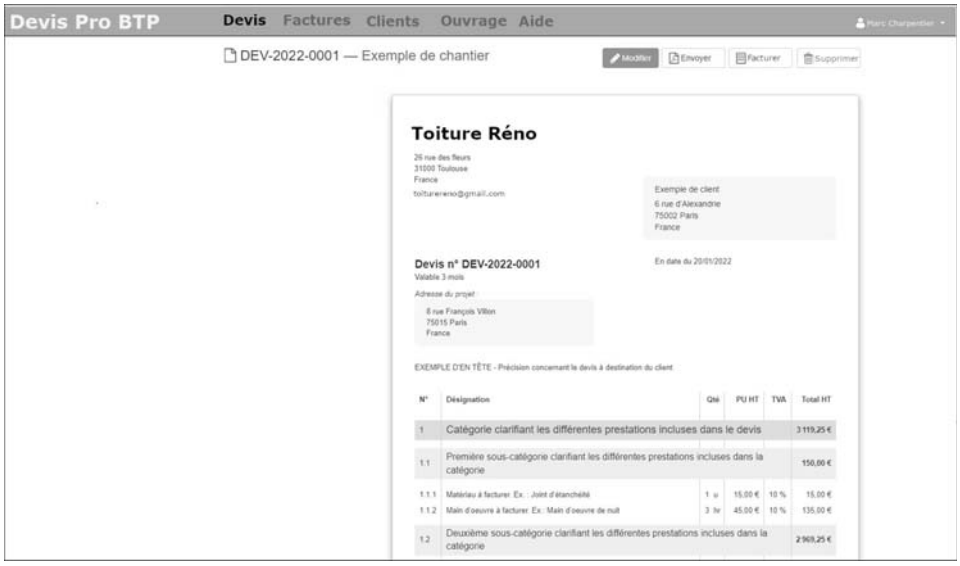
La première possibilité est de se connecter à partir de son adresse Gmail, si l'artisan en dispose une. La deuxième possibilité est de s'authentifier directement auprès de l'application à partir de son e-mail stocké dans la base de données ainsi que du mot de passe stocké de façon encryptée. Par exemple, on pourra créer une table nommée **Artisan** dans le but de stocker ces informations de connexion.

À la première connexion, l'artisan peut créer son compte en cliquant sur **Pas encore de compte ? Inscrivez-vous** s'il ne souhaite pas s'authentifier à partir d'une adresse Gmail.

S'il a oublié son mot de passe, en cliquant sur **Mot de passe oublié**, il peut le réinitialiser.

Une fois connecté, l'artisan a accès à l'interface principale de l'application Devis Pro BTP et aux fonctionnalités essentielles comme la création d'un nouveau devis, la création d'une nouvelle facture, l'ajout de nouveaux clients, la gestion et le suivi des paiements des factures, l'enrichissement de la liste des ouvrages liés à son activité.

Voici, ci-dessous, la maquette correspondant à l'interface principale de l'application web Devis Pro BTP :



Le menu principal comporte les items suivants : **Devis**, **Facture**, **Client**, **Ouvrage** et **Aide** pour accéder à chacune des interfaces web.

En haut en droite, l'artisan étant authentifié, son nom et prénom apparaissent, ainsi que l'icône permettant d'accéder à la déconnexion et à la gestion des informations liées à son profil.

1.3 Schéma UML2 : base de données devisprobtp

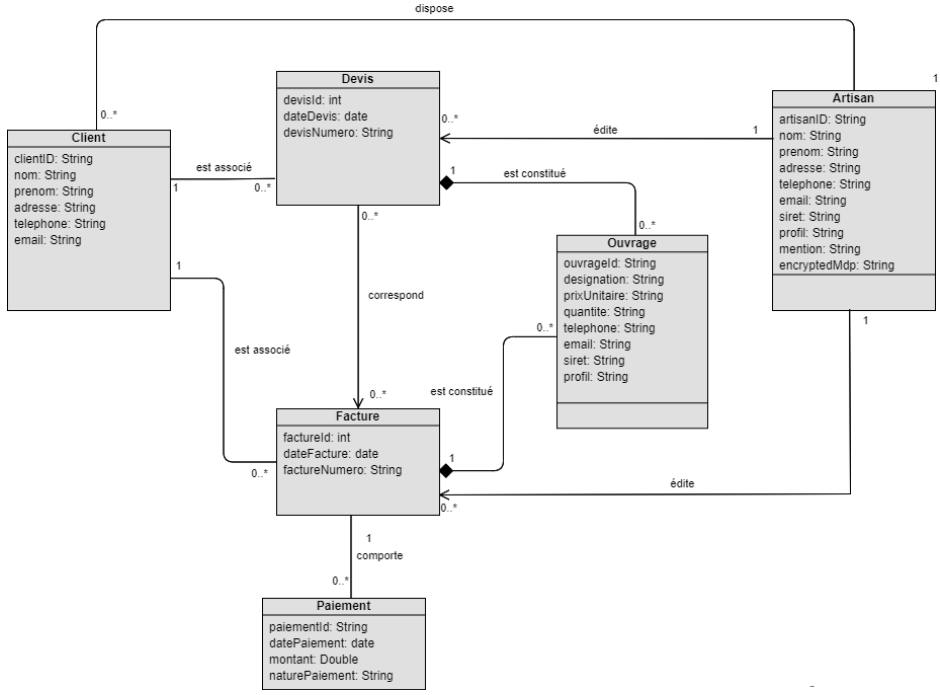
1.3.1 Schéma global de la base de données devis pro BTP

Voici, ci-dessous, la modélisation de la base de données avec UML 2 représentant les relations entre les différentes tables de l'application Devis Pro BTP :

– La table **Artisan** contient les données personnelles de l'artisan (nom, prénom, e-mail, adresse, mot de passe, Siret, profil…).

– La table **Devis** contient le `devisId` identifiant unique auto-incrémenté, `dateDevis` la date d'émission du devis et `devisNumero`.

– La table **Ouvrage** permet de retrouver les désignations des travaux liés à un devis ou à une facture.



1.3.2 Informations relatives à la table Artisan

D'un point de vue relations et cardinalités entre la table **Artisan** et les tables **Client**, **Facture** et **Devis**, nous pouvons noter que :

– Un artisan possède 0 ou plusieurs clients.

– Un artisan édite 0 ou plusieurs devis.

– Un artisan édite 0 ou plusieurs factures.

1.3.3 Informations relatives à la table Client

D'un point de vue relation et cardinalités entre la table **Client** et les autres tables **Artisan**, **Facture** et **Devis**, on peut noter que :

– 0 ou plusieurs clients correspondent à un artisan.

– 0 ou plusieurs devis sont édités par un artisan.

– 0 ou plusieurs factures sont éditées par un artisan.

1.3.4 Informations relatives à la table Devis

D'un point de vue relation et cardinalités entre la table **Devis** et les tables **Facture**, **Ouvrage**, **Artisan** et **Client**, nous pouvons noter que :

– 0 ou plusieurs devis sont édités par un artisan donné.

– 0 ou plusieurs devis correspondent à un client donné. Un client peut souhaiter obtenir plusieurs devis comparatifs, par exemple avec des matériaux de construction différents ou des choix techniques différents, qui auront un impact sur le coût final des travaux.

– 0 ou plusieurs devis correspondent à 0 ou plusieurs factures. Ainsi, l'artisan aura la possibilité de transformer facilement un devis accepté par le client en facture finale.

– Un devis donné est constitué de 0 ou plusieurs ouvrages ou réalisations effectués par l'artisan (plusieurs lignes dans le devis).

1.3.5 Informations relatives à la table Facture

D'un point de vue relation et cardinalités entre la table **Facture** et les tables **Paiement**, **Client**, **Artisan**, **Ouvrage** et **Devis**, nous pouvons noter que :

– 0 ou plusieurs factures correspondent à un artisan donné.

– Une facture donnée correspond à 0 ou plusieurs paiements. En effet, un client peut régler en plusieurs fois, verser un acompte au préalable...

– 0 ou plusieurs factures sont associées à un client donné.

– Une facture donnée est constituée de 0 ou plusieurs ouvrages (liste de travaux).

– 0 ou plusieurs factures correspondent à 0 ou plusieurs devis.

1.4 Script SQL base de données devisprobtp

Pour créer la base de données devisprobtp, avant de pouvoir exécuter le script SQL qui permettra de créer les différentes tables de l'application :

- Lancez une console DOS ou un terminal sous Linux suivant votre système d'exploitation.
- Connectez-vous avec l'utilisateur postgres sur linux `sudo -i -u postgres` sous Windows `psql -U postgres`.
- Créez un compte spécifique pour la base de données devisprobtp avec la commande suivante :
  `create user admin WITH PASSWORD ' devisprobtp ';`
- Créez la base de données devisprobtp avec la commande SQL suivante : `create database devisprobtp`.
- Donnez tous les droits à l'utilisateur admin qui vient d'être créé sur la base de données devisprobtp : `GRANT ALL PRIVILEGES ON DATABASE devisprobtp to admin;`.
- Saisissez `\q` pour quitter.
- Reconnectez-vous en tant qu'utilisateur postgres sur linux `sudo -i -u postgres` sous windows `psql -U postgres`.
- Sélectionnez la base de données devisprobtp nouvellement créée `\c devisprobtp` (pour voir la liste de toutes les bases de données faites `\l`).

Voici, ci-dessous, le script SQL qui permet de créer les différentes tables pour la base de données devisprobtp :

```
/* Table Artisan */
create table artisan(
     artisanID serial PRIMARY KEY not null,
     nom varchar(60),
     prenom varchar(60),
     adresse varchar(100),
     telephone varchar(10),
     email varchar(100),
     siret varchar(14),
     profil varchar(60),
     encryptedMdp varchar(20)
```

```
);
/* Table Client */
create table client(
     clientId serial primary key not null,
     nom varchar(60),
     prenom varchar(60),
     addresse varchar(100),
     email varchar(100),
     artisanId integer,
     CONSTRAINT artisan_client_fk FOREIGN KEY (artisanId)
REFERENCES artisan (artisanId));

/* Table Facture */
create table facture(
     factureId serial primary key not null,
     dateFacture date,
     factureNumero varchar(100),
     artisanId integer,
     clientId integer,
     CONSTRAINT artisan_facture_fk FOREIGN KEY (artisanId)
REFERENCES artisan (artisanId),
     CONSTRAINT client_facture_fk FOREIGN KEY (clientId)
REFERENCES client (clientId)
);

/* Table Devis*/
create table devis(
     devisId serial primary key not null,
     dateDevis date,
     devisNumero varchar(100),
     artisanId integer,
     clientId integer,
     CONSTRAINT artisan_devis_fk FOREIGN KEY (artisanId)
REFERENCES artisan (artisanId),
     CONSTRAINT client_devis_fk FOREIGN KEY (clientId) REFERENCES
client (clientId)
);

/* Table Paiement */
create table paiement(
     paiementId serial PRIMARY KEY not null,
     datePaiement varchar(50),
     montant double,
     naturePaiement varchar(50),
```