Chapitre 6
Les procédures et fonctions

1. Présentation

Les algorithmes réalisés avant ce chapitre utilisent ce qui se nomme le paradigme de programmation impérative. Un paradigme de programmation est une façon de programmer et il propose un certain nombre d'outils le permettant. Le paradigme de programmation impérative donne des ordres au travers des instructions qui se succèdent et propose différentes structures de contrôle permettant de sauter ou de refaire certaines parties en fonction de conditions.

Nous pouvons constater que cette manière de procéder a quelques limites. Les algorithmes deviennent de plus en plus longs au fur et à mesure que leur complexité augmente et il est de plus en plus difficile de s'y retrouver. De plus, il peut y avoir du code qui est dupliqué, car il est nécessaire de l'exécuter plusieurs fois à différents endroits de notre algorithme.

Dans ce chapitre, nous allons découvrir un deuxième paradigme de programmation. Il se nomme le paradigme de programmation procédurale. Il ne vient pas remplacer le précédent, mais il le complète.

Afin de pallier les limitations précédemment citées, le paradigme de programmation procédurale introduit une nouvelle notion : il s'agit des sous-algorithmes. Un algorithme peut faire appel à un sous-algorithme pour réaliser un travail. Prenons une comparaison pour bien comprendre. Un chef de service a un certain nombre d'objectifs à réaliser, il va demander à ses subalternes de réaliser des tâches et lui va coordonner le travail. Dans cette métaphore, le chef de service correspond à l'algorithme (nommé algorithme principal) et les subalternes correspondent aux sous-algorithmes.

L'utilisation de sous-algorithmes va permettre de répartir le code entre l'algorithme principal et des sous-algorithmes. Cela va structurer le code et le rendre plus lisible, mais également éviter la duplication du code.

Il existe deux types de sous-algorithmes : les procédures et les fonctions. Une procédure est un sous-algorithme contenant une suite d'instructions. Une fonction est similaire, à la différence qu'elle retourne, à la fin de son traitement, une valeur à l'algorithme qui y a fait appel. Pour reprendre la métaphore précédente, une procédure correspondrait par exemple à la tâche de traiter le courrier reçu. La seule chose qui est attendue en retour, c'est de savoir que la tâche est terminée. Une fonction pourrait être par exemple la tâche de calculer le montant des dernières commandes passées. Dans ce cas-là, il y a bien une valeur qui est attendue à la fin de la tâche.

Sans le savoir, vous utilisez déjà des procédures et des fonctions : `écrire()` est une procédure permettant l'affichage d'éléments sur la console et `saisir()` et `aléa()` sont des fonctions retournant une valeur, soit saisie par l'utilisateur, soit choisie aléatoirement.

2. La déclaration d'un sous-algorithme

La déclaration d'un sous-algorithme est très similaire à celle d'un algorithme.

2.1 Déclaration d'une procédure

Afin de déclarer une procédure, il faut utiliser la syntaxe suivante :

```
Procédure nom_de_la_procédure(liste_des_paramètres)
#Déclarations

Début
   #Instructions

Fin
```

Pour une procédure, seule la première ligne change. Tout d'abord, le mot-clé `Algo` est remplacé par le mot-clé `Procédure`. Ensuite, après le nom de la procédure, il y a un couple de parenthèses dans lesquelles il est possible de passer des paramètres. Les paramètres sont les informations nécessaires à la procédure pour qu'elle puisse réaliser sa tâche. La procédure `écrire()` prend par exemple en paramètre le texte à afficher sur la console. La liste de paramètres est une suite de couples `nom : type` séparés par des virgules. Les paramètres sont comme des variables qui sont initialisées lors de l'appel de la procédure.

Cette première ligne s'appelle la signature de la procédure.

Exemple de déclaration d'une procédure :

```
Procédure afficheNfois(t : texte, n : entier)
Variable i : entier
Début
 Pour i <- 1 à n
   écrire(t)
 FPour
Fin
```

Dans cet exemple, la procédure prend en paramètres deux valeurs : d'une part `t`, de type texte, le message à afficher, et d'autre part `n`, de type entier, le nombre de fois qu'il faut afficher ce message.

En Java, une procédure se déclare de la manière suivante :

```
public static void nom_de_la_procédure(liste_des_paramètres) {
 // traitements

}
```

Les mots-clés `public static void` sont expliqués plus loin dans cet ouvrage. Pour le moment, il faut retenir qu'il est nécessaire de les mettre pour déclarer une procédure. Vous avez sûrement remarqué que ces mots-clés vous sont familiers et que vous les utilisez pour coder votre algorithme principal :

```
public static void main(String[] args) {
   ...
}
```

Il s'agit en effet d'une procédure, c'est même celle qui se nomme la procédure principale. Elle se nomme forcément `main()` et prend en paramètre un tableau de chaînes de caractères.

Voici l'exemple de procédure précédent codé en Java :

```
public static void afficheNfois(String t, int n) {
   for (int i = 0; i < n; i++) {
         System.out.println(t);
   }
}
```

2.2 Déclaration d'une fonction

La déclaration d'une fonction est très proche de la déclaration d'une procédure. Voici la syntaxe de déclaration d'une fonction :

```
Fonction nom_de_la_fonction(listeParametres) Retourne type_retour
#Déclarations

Début
   #Instructions

   Retourner valeur
Fin
```

Tout d'abord, le mot-clé `Procédure` a bien entendu laissé place au mot-clé `Fonction`. Ensuite, après la liste des paramètres, un nouveau mot-clé fait son apparition : `Retourne`. Il est suivi du type de la valeur qui sera retournée par la fonction. La signature de la fonction permet ainsi de visualiser à la fois ce qui est attendu en paramètre, mais le type de valeur qui est retournée par celle-ci. Enfin, la dernière différence est juste avant le mot-clé `Fin`. Le mot-clé `Retourner` est suivi de la valeur qu'il faut retourner à l'algorithme ayant fait appel à cette fonction.

Exemple de déclaration d'une fonction :

```
Fonction puissance(a : réel, n : entier) Retourne réel
Variable i : entier
Variable p : réel <- 1
Début
   Pour i <- 1 à n
         p <- p * a
   FPour
   Retourner p
Fin
```

La fonction `puissance` prend en paramètres deux valeurs : d'une part la valeur que l'on souhaite élever à une puissance donnée et d'autre part l'exposant. Cette fonction calcule donc a^n, c'est-à-dire a à la puissance n.

Pour mémoire $a^n = a \times a \times \ldots \times a$ n fois.

Deux variables sont déclarées : la première, `i`, est la variable servant pour la boucle `Pour` et la seconde, `p`, est utilisée pour calculer la puissance. Initialement, cette variable vaut 1. Après la première itération de la boucle, elle vaut `1 × a`, c'est-à-dire `a`. À chaque itération, sa valeur est multipliée par `a`. Ainsi, après `n` itérations, cette variable vaut bien a^n. Au final, l'instruction `Retourner p` permet de retourner cette valeur à l'algorithme ayant fait appel à cette fonction.

En Java, la syntaxe de déclaration d'une fonction est la suivante :

```
public static type_retour
       nom_de_la_fonction(liste_des_paramètres) {
 // traitements

 return valeur;
}
```

Il n'y a pas de mot-clé particulier pour indiquer qu'il s'agit d'une fonction, mais le mot-clé `void` a été remplacé par le type de retour de la fonction. Une procédure est en fait en Java un cas particulier de fonction qui ne renvoie rien (du vide !). Le mot-clé `return` permet de mettre fin à la fonction et de retourner la valeur souhaitée.

Voici la fonction puissance codée en Java :

```
public static double puissance(double a, int n) {
   double p = 1;
   for(int i=0; i<n; i++)
         p *= a;
   return p;
}
```

Cette fonction déclare qu'elle retourne un type `double`, c'est-à-dire un nombre réel. Elle prend en paramètres deux valeurs : un nombre réel et un nombre entier. À la fin de la fonction, le résultat de calcul de la puissance est stocké dans la variable p et c'est cette valeur qui est retournée.

3. L'appel à un sous-algorithme

3.1 L'appel à une procédure

Pour faire appel à une procédure, il faut simplement écrire le nom de la procédure suivie, entre parenthèses, des valeurs permettant d'initialiser les paramètres dans l'ordre dans lequel ceux-ci sont attendus.

Exemple :

```
Algo Punition
# demande à l'utilisateur le texte à copier et le nombre de fois
# qu'il faut le copier
Variable  phrase : texte
Variable  nbFois : entier
Début
 phrase <- saisir("Quelle phrase avez-vous à copier comme punition ?")
 nbFois <- saisir("Combien de fois avez-vous eu à la copier ?")
 afficheNfois(phrase, nbFois)
Fin
```

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`.