Notes de préparation de l'examen de DEV2

Nathan Furnal

13 juin 2021

Table des matières

1 List

Avant tout, une liste est une **interface**, c'est-à-dire un contrat qui décrit un comportement. Une interface est un type référence, similaire aux classes. Elle contient uniquement des constantes, des signatures de méthodes, des méthodes par défaut, des méthodes statiques et potentiellement des types imbriqués, par exemple une énumération. Seules les méthodes statiques ou par défaut peuvent contenir du code dans leur corps. Aussi, on utilise pas une interface à proprement parler, on doit l'implémenter. Cette discussion est détaillée à la section sur l'implémentation.

Plus précisément, une liste est une interface qui permet d'implémenter des séquences (des collections ordonnées), ces implémentations utilisées dans le cours sont notamment `ArrayList` et `LinkedList`. Attention, quand on parle de collection ordonnée, on indique la possibilité d'indexer où chaque incrément d'index représente la valeur suivante contenue dans la liste. C'est complètement différent d'avoir une séquence **triée**.

En clair, une liste est une collection d'éléments **ordonnés** accessibles par leur indice. La taille de la liste s'adapte à son contenu, les éléments ne sont pas nécessairement différents. La liste est ordonnée et pas nécessairement triée et on peut ajouter ou supprimer des éléments de la liste. Une liste est **générique**, elle prend le type de ses éléments entre chevrons comme ceci : `List<String> list`.

1.1 `ArrayList`

La classe `java.util.ArrayList` est une classe qui implémente `List`. De cette manière :

```java
import java.util.ArrayList;
List<String> nombrils = new ArrayList<>();
nombrils.add("Vicky");
nombrils.add("Jenny");
nombrils.add(1, "Karine");
System.out.println(nombrils); // ["Vicky", "Karine", "Jenny"]
```

Grâce à l'implémentation de `List`, on est assuré que `ArrayList` respecte les méthodes liées aux listes. Par exemple `.get()` qui donne l'élément à l'indice fourni en paramètre : `liste.get(1)`.

Ensuite, une liste est `Iterable`, c'est-à-dire qu'elle peut être parcourue, élément par élément. Donc, une classe qui implémente `Iterable` peut être parcourue et se trouver dans un *foreach*. Quand on l'implémente, on doit assurer que certains contrats sont remplis : savoir définir un élément suivant par exemple ou bien accéder à un élément par son index.

2 Assignation

Pour comprendre l'assignation, il faut d'abord savoir ce que sont les **expressions** et les **instructions**.

expression De manière générale, une expression est une construction faite de variables, opérateurs et invocation de méthodes, qui sont construites suivant la syntaxe du langage. Une expression a un type et une valeur.

instruction Une instruction ou *statement* contrôle la séquence d'exécution des programmes. Une instruction est exécutée pour son effet et n'a pas de valeur.

expression-instruction L'expression-instruction est une forme particulière de certaines expressions. Plus précisément, certaines expressions peuvent devenir des instructions grâce à l'ajout du symbole `;` en fin d'expression.

L'**assignation** est avant tout une expression, elle a un **type** et elle a une **valeur**. On en fait une instruction avec le symbole `;`.

Par exemple :

```java
// i = 1 est l'expression
// i = 1; est l'instruction, elle englobe l'expression et possède le ;
i = 1;
```

Voici quelques autres exemples :

```java
élément = 1
éléments[i] = j
éléments[i] = j;
i = j = k = l = 0;
i = (j = i+j) + 1;
f(i=1,j=0);
```

Il existe d'autres **opérateurs d'assignation**. Ce sont ceux qui effectue une opération via l'opérateur et qui assigne la nouvelle valeur à la variable. Par exemple, `= *= += -= %=`.

Dans le cas suivant, il y a une expression d'assignation et une expression de somme. L'ensemble devient aussi une instruction quand on utilise le point-virgule.

```java
i += 1; // équivaut i = i + 1
```

Voici des exemples plus complexes :

```java
i = 2; // Valide
i = i = (i *= 2) + 1; // Valide on a une variable, à laquelle on assigne une valeur
(i + 1) -= 2; // Non valide !! On a pas de variable à gauche, mais une expression.
```

3 `break` et `continue`

Ici, on parle des instructions de **rupture**. Les exemples utilisent aussi la notion d'étiquette, expliquée à la page page 452 du manuel de spécification de Java 15.

Voici la structure générale de l'instruction `break`.

```
BreakStatement:
    break Identifier(opt);
```

Cela permet d'arrêter une instruction. Si l'instruction a une **étiquette** alors on arrête l'instruction étiquetée, sinon on arrête la boucle englobante. La boucle englobante est la boucle dans laquelle l'instruction se trouve, s'il y a plusieurs niveaux de boucles alors seule la boucle englobante est concernée. Dans ce cas, le seul moyen d'arrêter une boucle de niveau supérieur, c'est de l'arrêter via son étiquette.

Voici la structure générale de l'instruction `continue`.

```
ContinueStatement:
    continue Identifier(opt);
```

Cela permet de passer **directement** à l'itération suivante, il n'a donc de sens que dans une itération, une boucle. Si l'instruction a une étiquette on recommence la boucle étiquetée, sinon on recommence la première instruction **répétitive englobante**.

Voici un exemple où l'impression des variables paires est omise puisqu'à chaque fois on `continue` plutôt que d'imprimer.

```java
for (int i = 0; i < 10; i++){
    if(i%2 == 0) continue;
    System.out.prinln(i);
}
```

Voici un autre exemple plus complexe où on fait référence à l'étiquette d'une boucle pour continuer l'itération. Ce cas est intéressant parce qu'en utilisant l'étiquette, on peut directement intervenir dans la boucle extérieur, là où un `continue` classique n'aura touché que la boucle englobante `bclj`.

```java
bcli : for(int i = 0; i < 10; i++){
    bclj : for(int j = 0; i < 10; j++){
        if((i*j)%2 == 0) continue bcli;
        System.out.prinln(j);
    }
    System.out.prinln(i);
}
```

4 `Collections`

La classe `Collections` est une classe utilitaire permettant de travailler avec les collections via des méthodes statiques. Elle permet de trier, trouver les extrema ou encore inverser des listes.

Voici un exemple :

```java
import java.util.ArrayList;
import java.util.Collections;
var arr = ArrayList<Integer>();
arr.add(3);
arr.add(2);
arr.add(1);
arr.add(11);
Collections.max(arr); // 11
```

```
Collections.min(arr); // 1
Collections.sort(arr); // [1, 2, 3, 11]
Collections.shuffle(arr); // [11, 1, 3, 2]
```

Attention, on ne peut pas utiliser `Collections` avec n'importe quel type de classe. En effet, si les interfaces attendues par `Collections` ne sont pas implémentées, certaines méthodes ne fonctionneront pas.

Enfin, la documentation de Java fournit plus d'exemples et d'explications sur l'utilisation de la classe `Collections`.

5 Constructeur

Constructeur Un constructeur est, en programmation orientée objet, une fonction particulière appelée lors de l'instanciation. Elle permet d'allouer la mémoire nécessaire à l'objet et d'initialiser ses attributs.

Quand on instancie un objet, on fait **référence** à sa définition de classe et on lui attribue un valeur en mémoire. Chaque nouvel objet a une nouvelle place en mémoire. Les membres de l'instance sont définis par le **constructeur** de classe qui crée attributs et méthodes au moment de l'instanciation. Donc, instancier un objet c'est lui réserver de l'espace en mémoire et initialiser ses attributs, gérés par le **constructeur**. En pratique, invoquer le mot-clé `new` avec un nom de classe, fera appel au constructeur et au nom de classe avec des paramètres. Sans ce mot-clé, on bloque de l'espace en mémoire pour un objet d'un certain type mais sans faire appel au constructeur, par exemple avec la commande `Video v1;`.

Il ne faut pas confondre `null` et `""`. Le premier indique qu'il n'y a pas d'objet, c'est une référence à rien. Alors que le second est une référence vers la chaîne de caractères `""`.

Il est de la responsabilité du développeur de créer des constructeurs robustes qui permet de créer des objets dans un état valide. Par exemple en ajoutant des conditions en début de construction de classe.

N.B : Une classe peut avoir plusieurs constructeurs, c'est une manière d'attribuer des valeurs par défaut au constructeur. Cependant, chaque constructeur unique doit avoir un nombre **différent** de paramètres. Voir la section sur la surcharge.

6 Encapsulation

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

En pratique, tous les attributs d'une classe sont privés, on passe par des mutateurs (*setters*) publics pour modifier les attributs de l'objet. Dans ce cas, on a établi un contrôle car les *setters* peuvent vérifier et contrôler les mutations appliquées aux attributs.

Voici un exemple où on permet d'accéder à une donnée et on ne permet pas de la modifier, on a jamais un accès direct à l'attribut. Ici on ne peut accéder à l'attribut que par des méthodes dédiées, si ces méthodes n'existaient pas, on ne pourrait pas faire de modification.

```
public class Employee {
    private final int salary;

    public Employee(int s){
        this.salary = s;
    }

    public int getSalary(){
        return this.salary;
    }

    public static void main(String[] args){
        Employee e = new Employee(2300);
        System.out.println("Le salaire est : " + e.getSalary());
    }
}
```

7 enum

Une **énumération** est un ensemble fixe et généralement petit de valeurs sémantiquement liées. Par exemples les saisons, les couleurs d'un jeu de carte, les jours de la semaine…

C'est un type à part entière dont les instances sont décrites dans la classe. Elles portent un nom et sont constantes et il est impossible d'en créer d'autres par la suite car le constructeur privé.

Voici un exemple avec la syntaxe Java :

```
public enum Saison {
    PRINTEMPS, ÉTÉ, AUTOMNE, HIVER;
}
```

L'énumération est comme une classe (plus particulièrement c'est une classe avec un comportement spécifique) dont les valeurs possibles sont des membres statiques. Plus spécifiquement, les valeurs de l'énumération sont les seules instances qu'on permet d'exister.

Aussi, une énumération est un type à part entière et elle peut avoir des attributs et des méthodes. Elle est convertie automatiquement vers une chaîne lors de l'impression (via `toString`), elle peut apparaître dans un `switch` et fournit un tableau des valeurs de l'énumération via la méthode `values()`.

Avant, Java utilisait des constantes numériques pour simuler la notion d'énumération via des attributs finaux de classe.

7.1 Notions avancées

Nous allons maintenant nous concentrer sur un exemple plus complexe d'énumération avec des méthodes un constructeur. Le constructeur a une visibilité privée car il sera utilisé lors des créations des valeurs de l'énumération mais on ne l'expose pas vers l'utilisateur.

Donc, on peut voir dans l'implémentation suivante qu'on peut implémenter des attributs, des méthodes statiques ou pas.

```
import java.time.LocalDate;
public enum Season {
    SPRING(21, 3),
    SUMMER(22, 6),
    FALL(23, 9),
    WINTER(21, 12);

    private LocalDate begin;

    private Season(int day, int month){
        var now = LocalDate.now();
        this.begin = LocalDate.of(now.getYear(), month, day);
    }

    public LocalDate getBegin(){
        return this.begin;
    }

    public static Period when(Season s){
        var now = LocalDate.now();
        return now.until(s.begin);
    }

    public Season next(){
        return Season.values()[(this.ordinal() + 1) % 4];
    }
}
```

8 `equals`

La méthode `equals` indique si un objet est égal à un autre, dans le sens défini par le développeur. Il est conseillé de la réécrire dans les sous-classes car son implémentation par défaut dans `Object` n'est pas toujours indiquée, elle est intimement liée à la méthode `hashCode`.

Comme mentionné au-dessus, la méthode `equals` est implémentée dans `Object` dont tous les objets Java héritent. La documentation indique aussi plus précisément ce qui est attendu de cette méthode, dans les cas qui ne sont pas `null`.

- Elle est réflexive : `x.equals(x)` doit toujours retourner `true`.
- Elle est symétrique : `x.equals(y)` retourne `true` uniquement si l'expression `y.equals(x)` retourne `true` elle aussi.
- Elle est transitive : Pour x, y et z non `null`, si x est égal à y et que y est égal à z, alors x est égal à z.
- Elle est cohérente : Pourvu qu'aucune information ne change entre les appels de méthodes, un même appel dans les mêmes conditions doit toujours retourner la même valeur. Donc, si `x.equals(y)` retourne `false` et qu'on l'appelle plusieurs fois, dans les mêmes conditions, alors cet appel doit continuer à retourner `false`.
- Un objet qui n'est pas `null` doit toujours retourner `false` quand il est comparé à `null`.

L'implémentation par défaut de cette méthode utilise l'égalité de référence, c'est-à-dire `x == y`. Souvent, ce n'est pas le comportement qui nous intéresse. On veut par exemple comparer l'état ou la valeur d'objets pour qualifier leur égalité. D'où la nécessité de `@Override` cette méthode.

9 Expression régulière

Les **expressions régulières** ou *regex* permettent de vérifier qu'une chaîne de caractères correspond à un certain schéma ou *pattern*. On utilisera donc la classe `Pattern` ainsi que la classe `String` qui propose une méthode `matches`. La documentation de la classe `Pattern` reprend le fonctionnement des expressions régulières en Java.

Voici un petite cartographie des expressions et un exemple en Java. Attention, certains caractères n'ont pas le même comportement suivant qu'il se trouvent dans une capture ou un groupe, il vaut donc mieux vérifier au préalable.

```
X   ---> match le caractère X
X|Y ---> match X ou Y
XY  ---> match XY
\d  ---> Un chiffre entre 0 et 9 donc [0-9]
\D  ---> Négation de \d donc [^0-9]
\w  ---> Un caractère alphanumérique [a-zA-Z_0-9]
\W  ---> La négation de \w donc [^\w]
.   ---> match n'importe quel caractère
+   ---> à coté d'une lettre, classe ou capture, dénote qu'il faut au moins une fois
?   ---> à côté d'une lettre, classe ou capture, dénote qu'il faut une fois ou pas du tout.
$   ---> oblige apparition en fin. donc [abc]d$ doit terminer par d.
[abc]  ---> classe simple, match a ou b ou c
[^abc] ---> négation, match une lettre qui n'est pas a ou b ou c
[A-Z]  ---> range, match une lettre dans l'intervalle A,B,C...Z
(abc)  ---> capture de groupe, match seulement abc
[abc]{2} ---> a, b ou c exactement deux fois donc ab, ac, aa, bb, bc, ca,...
^N[OI] ---> commencer obligatoirement par N et puis O ou I donc NO ou NI
^(HELL)O? ---> Commence par HELL et puis termine ou pas par O donc HELL ou HELLO
```

Voici leur usage en Java mais attention, on ne doit pas oublier que pour utiliser un backslash, il faut l'échapper avec un autre \ devant.

```java
import java.util.regex.Pattern;

public class RegexExample{
    public static void main(String[] args){
        String s = "g12345";
        s.matches("g\\d{5}"); // true
        // Plus efficace si on compile l'expression avant de l'utiliser
        Pattern pat = Pattern.compile("g\\d{5}");
        pat.matcher(s).matches(); // true
    }
}
```

Attelons-nous maintenant à trouver une expression régulière qui convient pour tous les mnémoniques de cours (pas les labos). Il faut donc :

— Obligatoirement commencer par 3 lettres majuscules alphanumériques.
— Optionnellement plus de lettres pour les options.
— Ces lettres sont suivies et terminent par un nombre entre 1 et 6 (les quadrimestres possibles).

```java
import java.util.regex.Pattern;

public class PatternMatching {
    public static void main(String[] args){
        Pattern cours = Pattern.compile("^[A-Z]{3}G?I?R?[1-6]$");
        String[] tab = {"WEBG2", "DEV3", "DONGIR5", "WEBD1", "MIC2", "STA7", "don3",
        ↪  "IMGI4", "SYSRG2"};
        for(String s : tab)
            System.out.print(cours.matcher(s).matches() + " ");
        }
    }
```

```
true true true false true false false true false
```

10 extends

Un autre grand concept de l'orienté-objet est l'héritage. L'héritage est la possibilité pour une classe "enfant" de réutiliser les membres de sa classe "parent". On l'utilise grâce au mot-clé `extends`. En Java, on ne peut hériter que d'une seule classe et on peut implémenter plusieurs interfaces. Mais rien n'empêche d'avoir plusieurs "générations", c'est-à-dire une classe enfant qui est classe parent d'une autre classe.

Lorsqu'une classe **hérite** d'une autre, elle possède les mêmes attributs que son parent et on peut en ajouter. Elle possède aussi les mêmes méthodes, on peut en ajouter ou les **réécrire** et le mot-clé `@Override` indique qu'on réécrit une méthode même s'il n'est pas nécessaire. Attention, les visibilités restent de mise (`protected`) et si les attributs sont privés, on ne peut pas y accéder.

Si on reprend la classe `Point`, on peut créer une classe enfant `ColoredPoint`.

On doit définir son constructeur, mais on peut utiliser le mot-clé `super` pour réutiliser les attributs de la classe parent.

Voici la classe `Point`, simplifiée.

```java
public class Point {
    private double x;
    private double y;

    public Point(double x, double y){
        this.x = x;
        this.y = y;
    }
    public void display(){
        System.out.println("(" + this.x + ", " + this.y + ")");
    }
}
```

Suivie d'une classe `ColoredPoint` qui hérite de `Point`.

```java
import java.awt.Color;
public class ColoredPoint extends Point {
    private Color color;
    public ColoredPoint(double x, double y, Color c){
        super(x, y); // Utilise le constructeur de Point
        this.color = c;
    }
    public ColoredPoint(double x, double y){
        this(x, y, Color.BLACK);
```

```java
    }

    @Override
    public void display(){
        System.out.println("(" + super.getX() + ", " + getY() + ") - " + color);
    }
}
```

Ce code est abordé à nouveau la section sur les mots-clés `this` et `super`.

11 Fichier texte/binaire

En général, l'information est codée en **binaire** ou en **textuel**. Le binaire est une représentation mémoire et le texte est une suite de caractères.

Pour passer de l'un à l'autre, on utilise une table qui permet la traduction entre le textuel et le binaire. Les tables classiques sont la table `ASCII` ou `UTF-8`. Comme elle est très limitée au niveau des caractères disponibles, de nouvelles tables ont été créées, comme l'`UTF-8` ou même `UTF-16`.

11.1 Binaire vs. texte

Comme expliqué plus haut, on a par exemple le caractère `A` :

`A => 65 => |0100|0001| => 0x41 (on vérifie dans la table ASCII)`

Mais on peut se poser la question de comment représenter des nombres sous forme de caractères.

Par exemple, pour le nombre 42, en sachant que les entiers sont représentés sur 4 *bytes*.

1. 42 → binaire

00000000	000000000	000000000	00101010

2. 42 → en texte peut être vu comme le caractère 4 et 2, si on suit la table `ASCII`, alors on a.

0011	0100	0011	0010

On peut voir que les représentations sont très différentes et que si la valeur sous forme de caractères est interprétée en tant qu'entier, on aura absolument pas 42.

Si on écrit `42` dans un fichier texte, on peut l'ouvrir et voir apparaître `42`. Mais si on l'ouvre en tant que fichier binaire, il va convertir le décimal `42` en hexadécimal qui correspond à `0x2a`. Ensuite, il vérifie la valeur dans la table `ASCII` ou `UTF-8`. Dans ce cas, c'est le symbole étoile `*`. On peut voir que la même information n'est pas comprise de la même manière suivant l'extension. Mais attention, il n'y a pas que la table `ASCII`, le début de la table `UTF-8` est le même que celle de la table `ASCII` pour assurer la rétro-compatibilité.

11.2 UTF-8

Aujourd'hui la plupart des fichiers, pages web, etc. Sont codés en `UTF-8`. Pour ne pas consommer trop de mémoire, on va limiter le nombre d'octets utilisés pour les caractères, suivant leur code. Voici la table qui reprend les nombres d'octets utilisés.

Définition du nombre d'octets utilisés dans le codage (uniquement les séquences valides)

Caractères codés	Représentation binaire UTF-8	Premier octet valide (hexadécimal)	Signification
U+0000 à U+007F	0xxxxxxx	00 à 7F	1 octet, codant 7 bits
U+0080 à U+07FF	110xxxxx 10xxxxxx	C2 à DF	2 octets, codant 11 bits
U+0800 à U+0FFF	1110 0000 10 1xxxxx 10xxxxxx	E0 (le 2e octet est restreint de A0 à BF)	3 octets, codant 16 bits
U+1000 à U+1FFF	1110 0001 10xxxxxx 10xxxxxx	E1	
U+2000 à U+3FFF	1110 001x 10xxxxxx 10xxxxxx	E2 à E3	
U+4000 à U+7FFF	1110 01xx 10xxxxxx 10xxxxxx	E4 à E7	
U+8000 à U+BFFF	1110 10xx 10xxxxxx 10xxxxxx	E8 à EB	
U+C000 à U+CFFF	1110 1100 10xxxxxx 10xxxxxx	EC	
U+D000 à U+D7FF	1110 1101 10 0xxxxx 10xxxxxx	ED (le 2e octet est restreint de 80 à 9F)	
U+E000 à U+FFFF	1110 111x 10xxxxxx 10xxxxxx	EE à EF	
U+10000 à U+1FFFF	1111 0000 10 01xxxx 10xxxxxx 10xxxxxx	F0 (le 2e octet est restreint de 90 à BF)	4 octets, codant 21 bits
U+20000 à U+3FFFF	1111 0000 10 1xxxxx 10xxxxxx 10xxxxxx		
U+40000 à U+7FFFF	1111 0001 10xxxxxx 10xxxxxx 10xxxxxx	F1	
U+80000 à U+FFFFF	1111 01xx 10xxxxxx 10xxxxxx 10xxxxxx	F2 à F3	
U+100000 à U+10FFFF	1111 0100 10 00xxxx 10xxxxxx 10xxxxxx	F4 (le 2e octet est restreint de 80 à 8F)	

Cette table va nous permettre de passer du binaire à l'unicode et de le convertir avec les bits de début du standard `UTF-8`. Une fois la conversion faite, on a une valeur correspondante en hexadécimal.

Par exemple dans le cas du symbole "œ" qui vaut `0153` en unicode, on le converti en binaire. On utilise la conversion `UTF-8` en bloquant les bits de débutant suivant la table. On reporte les bits nécessaires vers la gauche et on a une nouvelle représentation binaire. Une fois convertie, on a le code hexadécimal.

œ	
Code unicode	0153
En binaire	00000001 01 010011
Représentation binaire UTF-8	**110**00101 **10**010011
Représentation hexa UTF-8	c5 93

Voici un autre exemple avec le symbole `€` dont l'unicode est `u20AC`. Si on le met dans un fichier texte et qu'on utilise `hexdump`. On obtient sa valeur hexadécimale dans la table `UTF-8`.

```
hexdump -C euro.txt

# 00000000  e2 82 ac                                          |...|
# 00000003
```

Convertissons l'unicode en binaire et puis en UTF8 pour confirmer cette traduction.

2	0	A	C
0010	0000	1010	1100

Ajoutons le codage UTF8 précisé par la table, on est obligé de suivre les bits donnés par la table et de compléter avec le binaire du dessus jusqu'à épuisement. Ici, les trois bits de poids forts de 2 dans `20AC`, ne sont pas pris en compte car on a rempli les emplacements nécessaires avant d'arriver à eux.

Code UTF8	**1110001** x	**10** xxxxxx	**10** xxxxxx
Bits à répartir	0	000010	101100
Représentation UTF8	11100010	10000010	10101100
Hex UTF8	e2	82	ac

Ce qui donne bien l'hexadécimal calculé en haut.

12 Filtrer (fonctionnel)

À partir de Java 8, on peut utiliser la programmation fonctionnelle et chaîner des opérations à l'aide de fonctions dites *lambda*, noté λ.

Une fonction λ est une notation raccourcie pour une instance d'une classe anonyme, implémentant une interface ne possédant qu'une seule méthode. Beaucoup de méthodes qui peuvent recevoir des fonctions de l'interface fonctionnelle existaient avant que les fonctions λ soient ajoutées à Java.

Pour comprendre comment filtrer à l'aide de l'interface fonctionnelle, on doit aussi parler de *streams*. On peut voir un *stream* comme une séquence d'éléments qu'on manipule. Dans un *stream* on peut utiliser une **map** qui applique une fonction sur chaque élément du *stream* ou un **filter** qui sépare les éléments qu'on désire conserver ou pas. En fin de *stream*, on collecte les éléments qui ont été modifiés. Les opérations qui modifient un *stream* et qu'on peut chaîner autant de fois qu'on le désire sont dites non-terminales. Tandis que l'opération unique de fin, qui collecte ou bien modifie une dernière fois le *stream* pour en faire un autre objet, est dite terminale.

Dans le cas présent, `filter` est une opération non-terminale et si on veut la liste de retour, on doit la collecter avec une opération terminale.

```java
public class FilterExample {
    public static void main(String[] args){
        List<Integer> arr = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6, 7, 8));
        var out = arr.stream().filter(x -> x % 2 == 0).collect(Collectors.toList());
        System.out.println(out); // [2, 4, 6, 8] (une liste)
    }
};
```

13 `for`

Le `for` est un mot-clé qui permet de définir une boucle dont on connaît le nombre d'itérations. Il y a deux variantes, la classique qui incrémente simplement un compteur jusqu'à une certaine limite et le *foreach* qui fait apparaître les valeurs contenues dans un *itérable* qui est parcouru.

Voici deux exemples de `for` classique et *foreach* respectivement.

```java
// Imprime les valeurs de 0 à 9
for(int i = 0; i < 10; i++){
    System.out.println(i);
}

int[] arr = {1, 2, 3, 4, 5};

// Imprime chaque valeur du tableau `arr`
for(int val : arr){
    System.out.println(val);
}
```

Voici la grammaire du `for`, comme décrite dans le manuel de spécification Java 15 à la section 14.14.1. On retrouve le compteur `i`, l'expression permettant de savoir si on est à la fin du compteur et la manière de mettre le compteur à jour. Ensuite, chaque partie `[ForInit]`, `[Expression]` et `[ForUpdate]` sont décrites avec la grammaire légale.

```
BasicForStatement:
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) Statement

BasicForStatementNoShortIf:
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) StatementNoShortIf

ForInit:
  StatementExpressionList
  LocalVariableDeclaration

ForUpdate:
  StatementExpressionList

StatementExpressionList:
  StatementExpression {, StatementExpression}
```

ForInit Cette partie qui a pour but d'initialiser la boucle est facultative (pas d'initialisation si elle est absente). La grammaire nous apprend qu'elle peut être :

- Une déclaration de variable locale. C'est à cette catégorie qu'appartient l'exemple courant. On peut aussi imaginer qu'on déclare plusieurs variables.
- Une liste d'expression instructions. C'est à cette catégorie qu'appartiennent des exemples comme `i=0` (liste d'un seul élément) ou `i=0,j=n`. Mais on pourrait aussi imaginer des exemples plus tordus comme `brol()` ou `i++`.

Expression La grammaire nous apprend qu'il s'agit d'une expression quelconque. La sémantique ajoute qu'elle doit être de type `boolean` ou `Boolean`. Si elle est omise, on considère que le test est vrai, ce qui induit une boucle infinie à moins d'un `break` ou d'un `return` dans le corps de la boucle.

ForUpdate Cette partie a pour but de mettre à jour les indices avant de tester à nouveau la condition. La grammaire nous apprend qu'on peut y trouver une liste d'expressions instructions (comme le second cas de la partie `ForInit`). Les mêmes exemples sont donc valides. Si elle est omise, pas de mise à jour.

Enfin, quand on a une seule instruction dans un `for`, on a pas besoin de mettre d'accolades mais on doit le faire quand il y en a plusieurs. C'est dû au fait que les accolades permettent de traiter une série d'instructions, comme une instruction unique, via un **block**.

14 `foreach`

Nous revenons à la discussion du `for` précédent, pour parler maintenant du *enhanced* `for` aussi connu sous le nom de *foreach*. Le *foreach* n'utilise pas de compteur ou d'initialisation. Voici sa grammaire :

```
EnhancedForStatement:
for ( {VariableModifier} LocalVariableType VariableDeclaratorId
    : Expression )
  Statement

EnhancedForStatementNoShortIf:
for ( {VariableModifier} LocalVariableType VariableDeclaratorId
    : Expression )
  StatementNoShortIf

VariableModifier:
  Annotation
  final

LocalVariableType:
  UnannType
  var

VariableDeclaratorId:
  Identifier [Dims]

Dims:
  {Annotation} [ ] {{Annotation} [ ]}
```

On a donc dans l'entête, une *modifier* optionnel qui est une annotation ou le mot-clé `final`. Suivi du type de la variable et du nom local qu'elle portera dans la boucle. Cette variable conserve le même nom à travers toute l'exécution de la boucle mais elle représente une valeur différente de l'expression à chaque fois que la boucle recommence.

Enfin, l'expression qu'on trouve après les deux points doit être un itérable ou un *array* pour que le code puisse compiler. De manière générale, on doit pouvoir itérer sur l'expression pour que ce type de `for` fonctionne.

Le commentaire sur l'accolade ou pas dans le corps de la boucle est le même que pour le `for` classique. Soit on ne met pas d'accolades lorsqu'il y a une seule instruction, soit on en met pour signifier un bloc qui peut contenir une série d'instructions.

15 Grammaire

La grammaire d'un langage est l'ensemble des règles qui définissent l'usage correct du langage. C'est-à-dire l'ensemble des mots admissibles sur un alphabet donné.

15.1 Principe

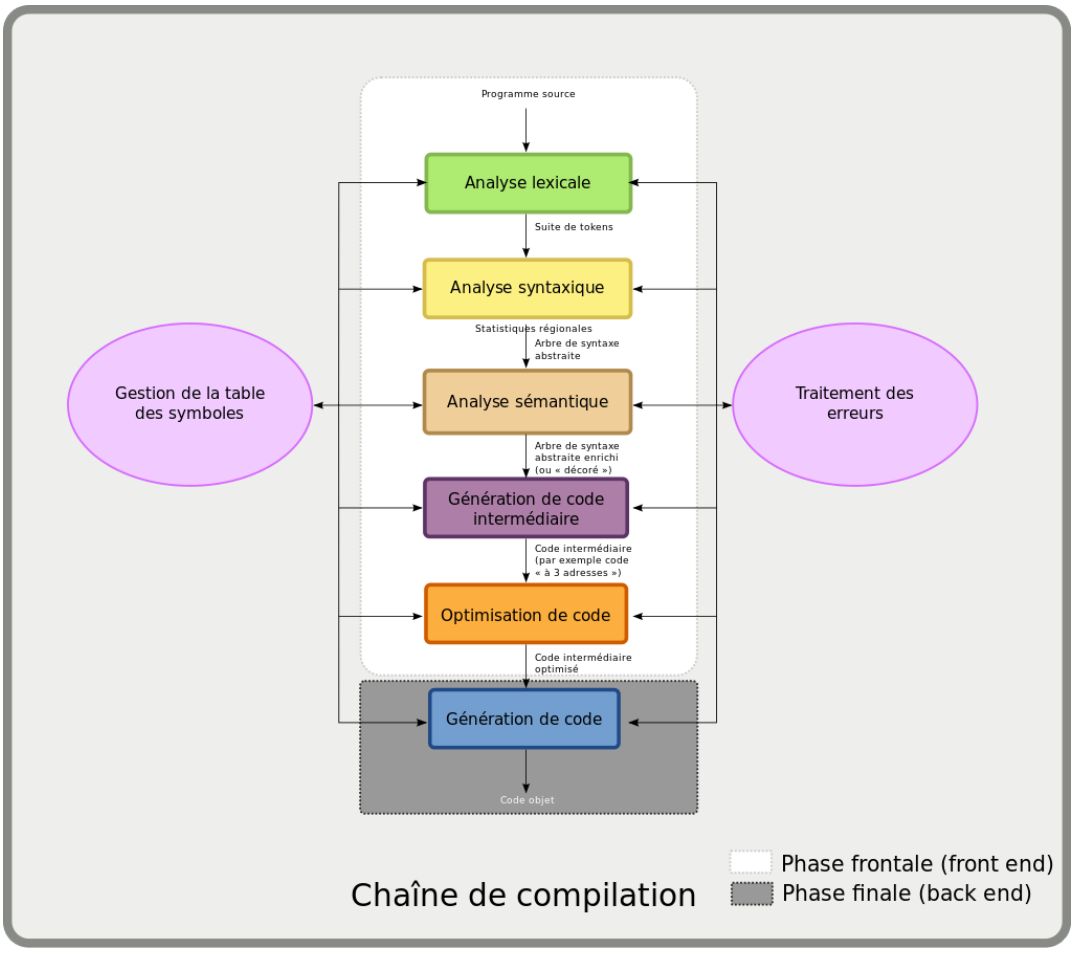
La nécessité de pouvoir écrire un compilateur, la nécessité de pouvoir décrire le langage imposent que les règles soient clairement décrites. C'est le rôle de la grammaire.

15.2 Fonctionnement

Une grammaire est une description finie de l'infinité des programmes.

- Un mot (*token*) doit être légal, c'est la **grammaire lexicale**.
- Une séquence de mots doit être légale, c'est la **grammaire syntaxique**.
- Le tout doit avoir un sens, c'est la **sémantique**.

Voici comment le code est généré en coulisses :



Le fonctionnement de la grammaire passent par trois étapes qui permettent de compiler code, les règles de grammaire sont définies dans le manuel de Java.

1. symbole de départ
2. règles de productions (*productions*)
3. symboles terminaux (*token*)

Un code est correct s'il peut être produit par la grammaire.

Voici un exemple où on caractérise un nombre décimal naturel et ce qui est valable comme nombre décimal naturel. Cet exemple définit deux règles, celle de `Nombre` et celle de `Chiffre`. `Nombre` est un symbole non terminal car il va encore être transformé, jusqu'à obtenir un symbole terminal.

```
Nombre :
Chiffre
Chiffre Nombre

Chiffre : one of 0 1 2 3 4 5 6 7 8 9
```

Suivant ces règles, on lit un nombre comme un chiffre ou comme un chiffre et un nombre. Dans ce cas, n'importe quel nombre devient ou chiffre ou une séquence de chiffre et finalement une valeur entre 0 et 9. De cette manière, on peut confirmer ou pas ce qui est valable comme représentation. Le compilateur Java permet par exemple de valider que `12` est une valeur entière correcte en effectuant un raisonnement similaire.

15.3 Lexicale

La grammaire lexicale (*lexical grammar*) définit les mots qui sont corrects en Java. Elle nous permet de passer des **caractères** aux mots.

— Les symboles terminaux sont les **caractères** qu'on prend au sein des caractères du standard unicode (*characters of Unicode character set*)
— Les règles de production forment les mots (*tokens*), éléments d'entrée (*input elements*) de la grammaire syntaxique.

Avant d'être interprétés, certains symboles sont considérés comme valables ou pas, par exemple on a des commentaires, des espaces et des *tokens*. Seuls les *tokens* sont utilisés dans la suite. Parmi les tokens on a les mots-clés

comme `public`, `class`, `byte`, `const`, `for`, `try`, ... On a aussi les **séparateurs** comme `(){};`, `,`, `[]`, ... Enfin il y a les **opérateurs** comme `= > < + - ? >> ++`, ...

En plus de ça, on a aussi les `literal`, pour les littéraux booléen on a `true false`. Les autres littéraux sont définis en détail dans le manuel de spécification de Java.

Enfin, on a les `Identifier`, ce sont les noms de variables, les chiffres, les lettres, etc. Mais ils ne peuvent pas être des mots-clés ou des littéraux.

15.4 Syntaxe

La grammaire syntaxique (*syntactic grammar*), définit comment les mots, le lexique, peuvent être mis ensemble pour obtenir du code Java correct. Elle permet de passer des *tokens* du langage vers un programme.

- Les symboles terminaux sont les *tokens*.
- Les règles de production permettent de définir ce qu'est un programme syntaxiquement correct.

Pour bien comprendre la grammaire syntaxique, on doit relever deux éléments importants, la notion d'**expression** et celle d'**instruction** ainsi qu'un mélange des deux, l'expression-instruction.

- Une **expression** a un type et une valeur. Un littéral est une expression comme : 12 (littéral entier décimal), 07 (littéral entier octal), `null` (littéral `null`), `true` (littéral booléen), `"Hello"` (littéral de type `String`).
- Les calculs sont des expressions : `12+3` est de type `int` et de valeur 15.
- Les appels de méthode sont des expressions.
- L'instanciation de classe est une expression.
- Accéder à un attribut est une expression.
- Accéder à un élément de tableau est une expression.
- Un littéral est une expression.

De manière générale, une expression est une construction faite de variables, opérateurs et invocation de méthodes, qui sont construites suivant la syntaxe du langage, évaluées de la gauche vers la droite. Une expression a une **valeur**.

16 `if`

Le `if` est un mot-clé marquant la condition en Java. On en trouve deux grandes formes, la condition de type `si` et celle et de type `si...sinon`. En clair, on permet l'exécution conditionnelle d'une expression ou un choix parmi deux expressions avec une seule des deux étant effectivement exécutée.

Ces différences dans l'exécution mènent en fait à trois possibilités, données par la grammaire :

```
IfThenStatement:
if ( Expression ) Statement

IfThenElseStatement:
if ( Expression ) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
if ( Expression ) StatementNoShortIf else StatementNoShortIf
```

Avec une `Expression` qui doit être de type `boolean` ou `Boolean`.

La grammaire définit plusieurs expressions comme `StatementNoShortIf` ou le simple `Statement` ou encore `IfThenElseStatementNoShortIf`, leur utilisation est expliquée ici-bas.

16.1 `if-then`

Un `if-then` est exécuté d'abord en évaluant l'`Expression`, si le résultat est un `Boolean` alors il y a *unboxing*. Si l'évaluation de l'expression s'arrête, la condition et l'expression `if-then` s'arrête elle aussi.

Sinon, l'exécution continue en faisant un choix basé sur le résultat :

- Si la valeur de l'expression est `true` alors, l'instruction est exécutée. La condition se termine si et seulement si l'instruction s'exécute correctement.
- Si la valeur de l'expression est `false`, aucune action supplémentaire est prise et le `if-then` continue sans problème.

16.2 `if-then-else`

Une instruction `if-then-else` doit d'abord vérifier l'`Expression` comme dans le cas précédent et avec les mêmes limitations déjà mentionnées.

Si tout fonctionne, l'exécution continue en faisant un choix basé sur la valeur de l'expression.

- Si la valeur est `true` alors la première instruction (`Statement`) est exécutée (celle avant le `else`). L'instruction `if-then-else` se termine normalement si et seulement si l'exécution de cette instruction se termine sans encombres.
- Si la valeur est `false`, alors la seconde instruction (celle après le `else`) est exécutée. L'instruction `if-then-else` se termine normalement si et seulement si l'exécution de cette instruction se termine sans encombres.

16.3 Commentaire sur les instructions dans la condition

On voit apparaître dans la grammaire de la condition, les instructions légales, notamment `Statement` et `StatementNoShortIf`. Elles sont définies dans la grammaire de la manière suivante :

```
Statement:
  StatementWithoutTrailingSubstatement
  LabeledStatement
  IfThenStatement
  IfThenElseStatement
  WhileStatement
  ForStatement

StatementNoShortIf:
  StatementWithoutTrailingSubstatement
  LabeledStatementNoShortIf
  IfThenElseStatementNoShortIf
  WhileStatementNoShortIf
  ForStatementNoShortIf

StatementWithoutTrailingSubstatement:
  Block
  EmptyStatement
  ExpressionStatement
  AssertStatement
  SwitchStatement
  DoStatement
  BreakStatement
  ContinueStatement
  ReturnStatement
  SynchronizedStatement
  ThrowStatement
  TryStatement
  YieldStatement
```

On retrouve bien les possibilités d'instructions "simples" sans avoir besoin d'utiliser les accolades ainsi que des instructions ne permettant pas un conditionnel "court", via l'utilisation de *block* et donc de plusieurs instructions dans le corps du *block*.

17 `implements`

Le mot-clé `implements` dénote l'utilisation d'une interface. Plus précisément, on matérialise une interface dans une classe. Habituellement on implémente une ou plusieurs interfaces via une classe concrète mais une classe abstraite peut elle aussi implémenter une interface.

Un code qui définit un **contrat** s'appelle une **interface**. Une interface spécifie le comportement que les classes doivent implémenter. Donc, une interface ne spécifiera que des signatures de méthode, sans code. Ce sont donc aux classes qui **implémentent** une interface de respecter le contrat et d'implémenter ces méthodes. Il faut donc définir dans la classe le code de **toutes** les méthodes spécifiées dans l'interface.

Voici ce que ça donne en Java, on doit implémenter au moins les méthodes abstraites de l'interface.

```java
public interface MyInterface {
    void foo(int i);
    boolean isBar(char c);
}

public class Myclass implements MyInterface {
    public void foo(int i){
        // do something interesting
    }

    public boolean isBar(char c){
        return true; // or something useful
    }

    // Possible to add other methods
}
```

Une interface définit un type de données, on peut donc déclarer une objet de ce type mais on ne peut pas instancier un objet par l'interface.

```java
MyClass o = new MyClass();          // OK
MyClass o = new MyInterface();      // Non
MyInterface o = new MyClass();      // OK
MyInterface o = new MyInterface();  // Non
```

Il faut noter qu'une méthode d'interface est considérée comme implicitement publique. De ce fait on a pas besoin de préciser sa visibilité à moins d'en vouloir une différente.

Précédemment, j'ai écrit que les interfaces ne font que définir les signatures des méthodes. C'est vrai mais avec un variation possible : on peut utiliser le mot clé `default` pour créer des méthodes avec un comportement par défaut. De plus, là où on doit absolument définir les méthodes abstraites dans la classe qui implémente l'interface, on ne doit pas nécessairement le faire pour les méthodes par défaut.

```java
public interface MyInterface {
    void foo(int i); // Méthode abstraite classique

    default void bar(int i){
        System.out.println("Hey my value is " + i); // Méthode par défaut
    }

    // Ignore this
    public static void main(String[] args){}
}
```

Voici maintenant une implémentation de l'interface avec une méthode abstraite et une méthode par défaut, les outputs du code sont écrits après la définition de la classe.

```java
public class MyClass implements MyInterface {
    @Override
    public void foo(int i){
        System.out.println(i);
    }
    public static void main(String[] args){
        MyClass mc = new MyClass();
        mc.foo(11);
        System.out.println();
        mc.bar(12);
    }
}
```

11

Hey my value is 12

18 `import`

Pour importer une méthode d'un paquet ou bien une autre classe d'un même paquet sans devoir réécrire l'entièreté du nom de la classe, on peut utiliser `import`.

De plus, on peut importer les membres statiques directement pour ne pas devoir citer la classe qu'on utilise à chaque fois.

par exemple :

```
Math.sqrt(4);
```

Peut être utilisé comme ça :

```
import static java.lang.Math.*;
sqrt(4);
```

La différence principale avec un `import` classique, c'est que l'approche statique permet d'accéder à tous les membres statiques, méthodes et attributs compris ; sans devoir explicitement donner le nom de la classe. Cela peut faciliter la lecture du code mais peut aussi créer de la confusion sur la provenance des membres. À utiliser avec précaution.

La notion d'import est intimement liée à celle de visibilité.

19 Itérer (fonctionnel)

L'itération au sens premier signifie répéter une étape. C'est typiquement ce qu'on fait dans les boucles. L'itération fonctionnelle nous mène plus loin car elle demande qu'un objet soit itérable, c'est-à-dire qu'il puisse être l'objet d'une boucle foreach. Dans notre cas, les objets qui sont concernés sont les listes, bien qu'on puisse aussi utiliser des tableaux avec des *streams*.

Prenons néanmoins le cas d'une liste sur laquelle on décide d'itérer, à l'aide d'une méthode des *streams* nommée `forEach`. Attention, c'est une opération **terminale** et qui de plus ne modifie pas la liste sur laquelle elle opère car les *streams* créent des copies en coulisse. Voir la discussion dans la section sur le filtre fonctionnel.

```
import java.util.List;
import java.util.ArrayList;

public class MyList {
    public static void main(String[] args){
        List<Integer> list = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6, 7));
        System.out.println("Les élements de la liste sont : ");
        list.stream().forEach(x -> System.out.printf("%2d", x));
    }
}
```

Les élements de la liste sont : 1 2 3 4 5 6 7

20 `Object`

La classe parent de toutes les classes est `Object`. Toutes les classes hérite de cette classe et donc toutes ont :

- `String toString()` pour la représentation textuelle. On doit souvent la réécrire pour avoir une représentation qui a du sens car par défaut elle retourne type et le *hash* de l'objet.
- `boolean equals(Object o)` pour l'égalité sémantique et permet de **définir** la notion d'égalité. Elle doit être réécrite pour permettre une comparaison entre deux objets. Par défaut, elle utilise == qui teste une égalité de **référence** (donc `false` pour des objets différents, même s'ils sont de même type). De ce fait, il faut ré-implémenter la notion d'égalité pour cette méthode, en général en utilisant les attributs de classe. C'est au programmeur de décider ce qui représente une égalité et donc on utilisera `getClass()` ou `instanceof` suivant qu'on veut comparer sur base d'une classe, un parent, des valeurs, une combinaison de ces facteurs…

— `int hashCode()` pour créer un *hash* associé à l'objet. Cette méthode crée le *hash* associé à un objet. On la réécrit pour obtenir le comportement voulu : deux objets égaux au sens de `equals` doivent avoir le même *hash* et si des objets ont des *hash* différents alors ils sont supposés différents. Attention, des objets différents n'ont pas forcément un *hash* différent même si c'est mieux.
— `Object clone()` qui retourne une copie de l'objet. Cette méthode peut poser problème et on préféra l'utilisation d'un constructeur par copie ou d'une méthode statique.

21 Objects

La classe `Objects` est une classe utilitaire qui contient des méthodes statiques pour travailler sur les objets de manière générale.

On y retrouve :

`static boolean equals(Object o1, Object o2)`

```
Object.equals(o1, o2);
// Remplace
o1 == 02 || o1 != null && o1.equals(o2);
```

Il y a aussi `static boolean deepEquals(Objecto1, Object o2)` qui exécute la même opération mais pour les tableaux.

Puis, on a `static T requireNonNull(T t)` :

```
this.bar = Objects.requireNonNull(bar);
// Remplace
if(bar == null){
    throw new NullPointerException("bar is null");
}
this.bar = bar;
```

On a aussi une méthode qui permet de créer un *hash* : `static int hash(Object... values)`

```
@Override
public int hashCode(){
    return Objects.hash(x,y);
}
```

22 Polymorphisme

Le **polymorphisme** est un principe important de l'orienté objet. Ce que ce principe dit est que : une instance et une variable peuvent être de types différents. Lors de l'appel d'une méthode, c'est la bonne méthode qui sera appelée.

Voici un exemple :

```
public static void display (List<String> liste){
    for(int i = 0; i < liste.size(); i++){
        System.out.println(i + ": " + liste.get(i));
    }
}
```

Et une fois qu'on déclare une variable avec le type `List` et qu'on instancie une `ArrayList` ou une `LinkedList`, on se rend compte que comme ces deux classes implémentent `List`, elles utiliseront leur propre comportement pour exécuter les méthodes. Donc, en créant une seule méthode qui reçoit une `List`, on utilise le comportement propre de chaque implémentation de cette interface ; puisque chaque classe implémente `.get()`. La méthode `display` utilisera donc la méthode implémentée de chaque classe bien que son paramètre initial soit une interface.

```java
List<String> l1 = new ArrayList<>();
List<String> l2 = new LinkedList<>();
display(l1); // fonctionne
display(l2); // fonctionne
```

23 Post-incrémentation

En Java, il existe plusieurs manières d'incrémenter, notamment, avec une variable entière `i`.

- `i++` ou post-incrémentation.
- `++i` ou pré-incrémentation.

Dans le premier cas, celui de la post-incrémentation, la valeur de `i` n'est mise à jour qu'après l'évaluation de l'expression-instruction ++.

```java
int i = 3; // Initialisation et assignation
System.out.println(i++); // Imprime 3
System.out.println(i); // Imprime 4
```

Voici un exemple plus complexe, le premier membre de la somme vaut $i = 3$ puis 4 et la pré-incrémentation élève la variable à 5. On a donc un résultat de 8. On ne considère que la valeur post-incrémentée de `i` seulement lorsqu'elle est utilisée dans une autre expression-instruction.

```java
int i = 3;
int j = (i++) + (++i); // j vaut 8 et i vaut 5 !!
```

24 `static`

Le mot clé `static` exprime qu'un membre fait référence à la classe et non à une instance de la classe. Le membre est partagé par **toutes** les instances de la classe.

Dans le cas de classes utilitaires, comme les fonctions mathématiques, les représentations en `String` ou les `Scanner`, on a pas besoin d'instancier de classe pour utiliser leurs méthodes. C'est parce qu'elles sont `static`.

Soit une méthode peut être **statique**, soit un attribut peut être **statique**. Une méthode statique n'a pas accès aux attributs de la classe. Dans le cas non `static`, on applique une méthode sur une instance de classe et on accède par exemple à des attributs de l'instance, qu'on modifie. Exemple :

```java
Video v = new Video("Omar Sy", "Lupin");
v.liker(); // Méthode non statique qui modifie un attribut de l'instance
double value = Math.sqrt(4); // Méthode statique d'une classe non instanciée
```

Pour les attributs, la situation est un peu différente. Un attribut statique sera partagé par toutes les instances de classe, il n'y a qu'un seul exemplaire de cet attribut. Par exemple, `Math.PI` est la valeur de π, on a pas besoin d'instancier la classe `Math` et on est assuré que `Math.PI` aura toujours la même valeur peu importe quand et où cet attribut est appelé.

Le mot-clé `static` a aussi un rôle à jouer lors des imports, voir la section correspondante.

25 Surcharge / redéfinition

La surcharge est une possibilité offerte par certains langages de programmation de définir plusieurs fonctions ou méthodes de même nom, mais qui diffèrent par le nombre ou le type des paramètres effectifs.

La surcharge est particulièrement utile pour définir plusieurs constructeurs dans une classe ou bien permettre des comportements différents en fonction de types ou nombre des paramètres.

Par exemple :

```java
public class Overloading{
    public static float div(int num, int denom){
        return (float) num / denom;
    }

    public static String div(String num, String denom){
        return num + "/" + denom;
    }

    public static void main(String[] args){
        float res1 = Overloading.div(1, 2);
        String res2 = Overloading.div("1", "2");
        System.out.println("Le résultat de " + res2 + " est : " + res1);
    }
}
```

Le résultat de 1/2 est : 0.5

La redéfinition d'une méthode quant à elle concerne surtout l'implémentation des interfaces ou bien l'héritage. En effet, on peut signaler au compilateur qu'on redéfinit une méthode via `@Override`, comme vu dans la section sur l'héritage.

26 `switch`

Le mot-clé `switch` permet d'exécuter des instructions ou des expressions en fonction de la valeur d'une expression. Cette dernière doit être un `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, `String` ou bien une `enum`.

Voici sa grammaire :

```
SwitchStatement:
  switch ( Expression ) SwitchBlock
```

Le corps d'un instruction `switch` ou d'une expression `switch` sont appelés *switch block*. Voici leur grammaire :

```
SwitchBlock:
  { SwitchRule {SwitchRule} }
  { {SwitchBlockStatementGroup} {SwitchLabel :} }

SwitchRule:
  SwitchLabel -> Expression ;
  SwitchLabel -> Block
  SwitchLabel -> ThrowStatement

SwitchBlockStatementGroup:
  SwitchLabel : {SwitchLabel :} BlockStatements

SwitchLabel:
  case CaseConstant {, CaseConstant}
  default

CaseConstant:
  ConditionalExpression
```

Un *switch block* peut-être, ce qu'on appelle :

- *switch rules*, qui utilisent la flèche → pour insérer une *switch rule expression*, un *switch rule block* ou *switch rule* `throw` *statement*.
- *switch labeled statement groups* qui utilise les deux points : pour insérer un *switch labeled block statements*.

Dans les deux cas, on commence les possibilités d'un `switch` par un *label* `case` ou `default`. Dans le cas du `case`, chaque cas doit être une expression constante ou bien une valeur d'énumération, ces expressions doivent être homogènes, c'est-à-dire avoir le même type au sein du même bloc ou bien faire partie des valeurs de la même énumération. Les cas d'un même *switch block* sont considérés faire partie d'un tout, de ce fait, deux mêmes cas ne peuvent pas apparaître dans le bloc.

Pour voir si la valeur d'un cas parmi les possibilités du `switch` correspond, Java utilise l'opérateur d'égalité == à part pour les `String` où la méthode `.equals()` est utilisée. Enfin, si aucun cas ne correspond mais qu'il existe un cas par défaut, il est utilisé. De plus, ce cas par défaut doit être unique.

Les détails précis de l'exécution du code du `switch` se trouvent dans le manuel Java.

27 Tableau 1D

Un tableau est une structure de données de **taille fixe** et **homogène**, en Java. Une fois sa taille décidée, on ne peut plus la changer et toutes les valeurs doivent être du même type. Aussi, un tableau est un **type référence**, C'est-à-dire qu'en mémoire, le tableau ne possède pas les valeurs qu'il contient. En fait, il a une zone mémoire qui contient une référence (une adresse vers une autre zone mémoire) vers les valeurs du tableau.

Un tableau à une dimension peut s'instancier par les manières suivantes :

```
int[] tab = {1, 2, 3, 4}; // On donne directement ses valeurs
int[] tab = new int[4]; // On donne sa taille et son type
```

28 Tableau 2D

En Java, il n'y a pas de manière spécifique de stocker des tableaux en deux dimensions. Comme un tableau peut contenir n'importe quel type homogène, on peut facilement avoir des tableaux de tableaux.

On voit ici-bas qu'un tableau en deux dimensions contient trois tableaux, chacun contenant des valeurs entières. Ces tableaux ne doivent pas nécessairement être de même taille.

En mémoire, chaque case du tableau contient une référence vers un tableau et chaque tableau à une dimension contient une référence vers ses valeurs. Les valeurs ne sont pas contenues directement par les tableaux, peu importe la structure. Dû à ce choix de représentation, les valeurs peuvent être à des endroits complètement différents en mémoire.

```
int[][] tab = {{1, 2, 3}, {2, 2, 1}, {1, 1, 1, 5}};
```

Du coup, les accès classiques aux attributs et aux méthodes des tableaux fonctionnent de la même manière en deux dimensions, pour chaque tableau qui compose le "grand" tableau extérieur.

```
int[][] tab = {{1, 2, 3}, {1, 1, 1}};
tab.length; // Nombre de lignes
tab[0].length; // Nombre de colonnes
```

28.1 Création en donnant des valeurs

On peut créer un tableau en donnant des valeurs ou des tailles comme mentionné au-dessus.

De manière générique, on crée un tableau avec son **type**, ses dimensions et on peut le remplir ou pas. Si on choisit de le remplir, on utilise les accolades pour initialiser le tableau et on le remplit de valeurs du type du tableau.

```
int[][] tab = new int[4][2]; // 4 lignes et 2 colonnes, non rempli
int[][] tab = new int[][] {{1, 2, 3}, {3, 1, 1}}; // 2 lignes, 3 colonnes
int[][] tab = {{1, 1, 1}, {2, 2, 2}, {3, 3, 3}}; // 3 lignes et 3 colonnes, rempli
```

N'importe quel type peut peupler un tableau :

```
String[][] stringArray = new String[1][1];
stringArray[0][0] = "Hello";
```

28.2 Création en donnant des tailles

On peut créer un tableau en explicitant sa taille, cette taille peut être nulle et doit être positive. Le type est accompagné de crochets qui désignent à chaque fois une taille, chaque nouveaux crochets désignent une nouvelle dimension. Si les valeurs ne sont pas spécifiées, le tableau se remplit de 0.

On peut créer un tableau étape par étape avec une notation différente, si on crée la première dimension sans définir la deuxième, on peut avoir des tableaux non définis qui seront `null`, `null` "est" rien et donc pas une référence.

```
int[][] tab = new int[2][]; // Deux lignes qui sont en fait {null, null};
tab[0] = new int[3]; // La première ligne n'est plus null mais bien un tableau de 3
 ↪  éléments
```

28.3 Parcours

Pour parcourir un tableau, on peut utiliser la stratégie classique en accédant aux indices. Si le tableau est de plusieurs dimensions, on devra imbriquer les `for` pour aller des tableaux extérieurs aux tableaux intérieurs.

```
int[] tab = {1, 2, 3, 4, 5};
// Parcours à une dimension
for(int i = 0; i < tab.length; i++){
    System.out.println(tab[i]);
}

int[][] tab = {{1, 2, 3, 4, 5}, {1, 3, 4, 8, 9}};
// Parcours à deux dimensions
for(int i = 0; i < tab.length; i++){
    for(int j = 0, j < tab[0].length){
        System.out.println(tab[i][j]);
}
```

On peut aussi utiliser un `foreach` détaillé ici bas mais la grande **différence** c'est que l'accès par indice permet de modifier les valeurs du tableau. Le `foreach` crée une **copie** de la référence d'une valeur du tableau ! Du coup, on ne peut pas modifier un tableau de cette manière. Aussi, cette manière de faire ne permet pas de parcours colonne par colonne puisqu'on a pas accès aux indices.

Voici un exemple :

```
int[][] tab = {{1, 2, 3, 4}, {2, 1, 4, 5}};
// Foreach loop
for(int[] row : tab){
    for(int val : row){
        System.out.println(val);
    }
}

// Accéder aux colonnes
// On fait l'hypothèse que toutes les lignes ont le même nombre de colonnes
for(int j = 0; j < tab[0].length; j++){
    for(int i = 0; i < tab.length; i++){
        System.out.println(tab[i][j]);
}
```

Et un autre exemple avec les modifications en boucle classique et l'absence de modification avec le `foreach`.

```
int[] tab = {1, 2, 3, 4, 5};

for(int i = 0; i < tab.length; i++){
    tab[i] = 11;
}

// Le tableau vaut maintenant {11, 11, 11, 11, 11};
```

```java

// Rien ne se passe dans cette boucle
// Chaque elem est une copie de la références des éléments de tab, donc a rien modifié
for(int elem : tab){
    elem = 22;
}
```

Ce qui clôture cette section sur le parcours.

29 `this` et `super`

29.1 `this`

Le mot-clé `this` fait référence à l'objet sur lequel on est en train de travailler, c'est une référence à soi-même. On retrouve `this` dans plusieurs contextes :

- comme constructeur : `this();`
- comme attribut : `this.titre;`
- comme méthode : `this.liker();`

Suivant le langage, on doit utiliser le mot-clé `this` ou `self`. Bien que son usage ne soit pas obligatoire en Java, cela facilite la lecture et la compréhension, pour différencier les paramètres du constructeur de la valeur des attributs en tant que telle.

Voici un exemple :

```java
public class Example {
    private String name;
    private int value;
    private boolean nice;

    public Example(String aName, int aValue, boolean niceness){
        // On assigne aux attributs les valeurs des paramètres
        // this avec la notation pointée
        this.name = aName;
        this.value = aValue;
        this.nice = niceness;
    }

    public Example(String aName){
        // Utiliser this() comme fonction !!!
        // Permet d'assigner des valeurs par défaut
        // Doit être la première ligne
        this(aName, 11, false);
    }

public static void main(String[] args){
    // construction classique
    Example monExemple = new Example("hello", 10, true);
    // construction avec moins de paramètres et des valeurs par défaut
    Example other = new Example("hi hi");
}
    }
```

29.2 `super`

`super`, à l'instar de `this` permet d'accéder aux membres du parent et donc :

- Dans un contexte d'attribut `super.foo`
- Dans un contexte de constructeur `super(x,y)`
- Dans un contexte de méthode `super.bar()`

Comme en général les attributs de classe sont privés, on va utiliser les *getters* pour accéder aux attributs parents, dans la classe enfant. Si on veut en plus préciser qu'on utilise une méthode de la classe parent, on peut utiliser `super`. Par exemple en écrivant la méthode suivante pour la classe `ColoredPoint`, on voit que les accesseurs sont utilisés via `super` ou pas, c'est au choix.

```
public void display(){
    System.out.println("(" + super.getX() + ", " + getY() + ") - " + color);
}
```

N.B : On utilise la même classe se basant sur la logique de la section sur l'héritage.

30 `throw` et `throws`

Tout d'abord, souvenons-nous de ce qu'est une exception. On les lance avec `throw new` quand on se rend compte de problèmes dans le code qu'on ne sait soit pas gérer ou pour montrer qu'il y a un problème. Par exemple :

```
throw new IllegalArgumentException("Ici un problème");
```

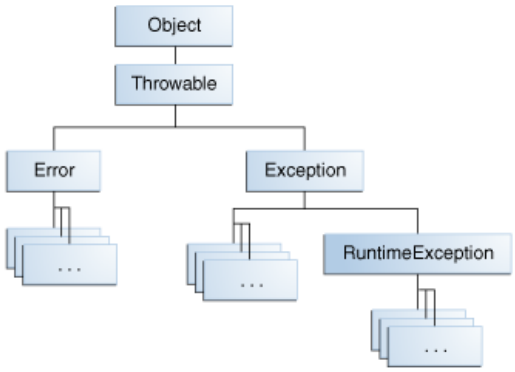
Une fois qu'une exception est lancée, on peut l'attraper pour gérer l'erreur, ça ne fonctionne pas pour toutes les exceptions, on doit préciser laquelle. Néanmoins, si on s'attend à lancer une exception particulière, on améliorer la gestion des erreurs de cette manière. Si on attrape une exception, on va directement dans le `catch` et puis le code continue son exécution.

```
try {
    // code pouvant lancer une exception
} catch (IllegalArgumentException e){
    // gestion de l'erreur
}
```

On utilise `catch` pour attraper une exception qu'on est capable de résoudre, pas pour éviter qu'il y en ait une. Autrement, il vaut mieux laisser l'erreur s'afficher à l'utilisateur.

30.1 Hiérarchie et exception contrôlée

Les exceptions ne sont pas toutes égales, elles suivent une hiérarchie présentée ici :



Dans cet arbre, on voit les objets `throwable` qui sont tous les objets "jetables". Il y a les `Error`, les exceptions qui ne **doivent pas** être gérées. On a les `Exception`, les exceptions qui **doivent** être gérées via les exceptions **contrôlées**. Finalement, on a les `RuntimeException`, les exceptions qui peuvent être gérées.

On a donc mentionné les exceptions **contrôlées**. Il faut préciser quand une exception contrôlée est lancée, via l'utilisation de `throws`. Quand une exception est contrôlée, on doit la déclarer dans l'entête dans la méthode.

```
public void myMethod() throws FileNotFoundException {
    //...
    throw new FileNotFoundException("Raison de l'exception");
    //...
}
```

Une exception contrôlée doit être gérée :

```
try {
    myMethod();
} catch (FileNotFoundException ex){
    // gérer l'exception
}
```

Ou être relancée si on appelle du code qui pourrait créer une exception contrôlée :

```
public void otherMethod() throws FileNotFoundException {
    myMethod();
}
```

De manière générale, on demande au programmeur d'être explicite sur les exceptions qui sont lancées et sur la manière de les gérer.

31 `toString`

La méthode `toString` retourne une chaîne de caractères destinée à représenter un objet de manière textuelle, apte à être lu par un être humain. Il est conseillé de la réécrire dans les sous-classes car son implémentation par défaut dans la classe `Object` n'imprime qu'un nom de classe et une chaîne hexadécimale.

C'est une des méthodes de base de la classe `Object`, décrite à cette section. Comme expliqué, elle permet la représentation textuelle des objets, pour être lue par des utilisateurs humains.

32 Trier

La classe `Collections` possède une méthode statique qui permet de trier une liste suivant un ordre par défaut. Dans l'ordre croissant pour les nombres et dans l'ordre alphabétique pour les chaînes de caractères. Le tri est possible seulement si l'interface comparable est implémentée.

On peut aussi définir notre propre manière de trier en implémentant l'interface `Comparable`. Cette dernière ne contient qu'une méthode :

```
int compareTo(T o) // T est un type auquel on compare
```

Cette méthode renvoie -1 si l'élément comparé est plus petit que celui reçu en paramètre, 0 s'ils sont égaux et 1 si l'élément est plus grand.

Si on définit nous-mêmes une classe et qu'on veut qu'elle soit comparable, qu'on puisse la trier, etc. Alors, il faut implémenter l'interface `Comparable` et donner une manière de comparaison.

Cet exemple datant de la section sur la classe `Collections` montre comment utiliser l'interface de comparaison basée sur l'âge. Dans les documents du cours, on implémente la comparaison avec une méthode de `String` déjà existante et on choisit d'utiliser le nom comme base de comparaison.

```
public int compareTo(Personne other){
    return this.name.compareTo(other.name);
}
```

En clair, une classe qui implémente l'interface `Comparable` devra définir une méthode `compareTo` qui définit *comment* comparer à d'autres objets, basé sur **un seul** élément.

32.1 Les tris et le fonctionnel avec `Comparator`

Il arrive qu'on veuille trier suivant un autre ordre que celui par défaut. Ou alors on voudrait bien comparer une fois certaine valeurs, sans avoir besoin d'implémenter une méthode dans une classe. Typiquement si on a pas accès au code de la classe et qu'on ne peut pas le modifier.

Dans ce cas, on peut utiliser l'interface `Comparator` qui elle aussi, permet de définir des choix de comparaisons. Elle sera utilisée avec `Collections.sort()` par exemple. Elle implémente la méthode `compare(T o1, T o2)`.

Le nom est différent et le nombre d'arguments aussi. Effectivement, on ne compare plus une instance de classe avec un autre objet, mais bien deux objets de même type sans avoir besoin d'implémenter la méthode dans une classe au préalable.

Voici un exemple :

```
import java.util.*;

List<String> mots = List.of("Ceci", "est", "une", "liste", "immuable", "de",
 ↪  "String");
List<String> mots = new ArrayList<>(mots); // liste mutable
```

On peut la trier du mot le plus court au mot le plus long, avec des opérateurs ternaires.

```
Collections.sort(mots,
                 (m1, m2) -> m1.length() < m2.length() ?
                 -1 : (m1.length() == m2.length() ? 0 : 1));
```

Comme on compare des longueurs et donc des entiers, on peut en fait utiliser une méthode qui existe déjà dans la classe `Integer`.

```
Collections.sort(mots,
                 (m1, m2) -> Integer.compare(m1.length(), m2.length());
```

Mieux encore, l'interface `Comparator` peut créer et retourner une fonction λ.

```
Collections.sort(mots,
           Comparator.comparing(m -> m.length()));

// Ou encore par référence de méthode

Collections.sort(mots,
                 Comparator.comparing(String::length));

// À l'envers !

Collections.sort(mots,
                 Comparator.comparing(String::length).reversed());
```

On peut tout aussi bien comparer sur **plusieurs** critères, en chaînant des fonctions λ. Dans le cas suivant, la méthode `thenComparing` permet de choisir l'ordre de la première fonction à moins qu'elle ne renvoie 0 (une égalité), alors on choisira l'ordre établi par la seconde fonction.

```
Collections.sort(mots,
                 Comparator.comparingInt(String::length)
                 .thenComparing(Comparator.naturalOrder()));
```

32.2 `Comparable` vs. `Comparator`

J'ai trouvé un tableau assez utile qui reprend les différences entre les deux interfaces :

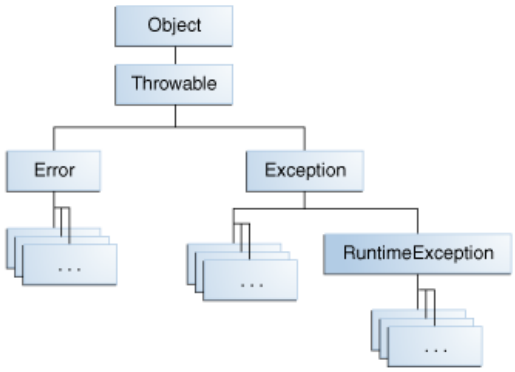
Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

FIGURE 1 – Différence entre Comparable et Comparator

33 try-catch

Cette section est à lire après la section sur les exceptions.

Les exceptions ne sont pas toutes égales, elles suivent une hiérarchie présentée ici :



Dans cet arbre, on voit les objets `throwable` qui sont tous les objets "jetables". Il y a les `Error`, les exceptions qui ne **doivent pas** être gérées. On a les `Exception`, les exceptions qui **doivent** être gérées via les exceptions **contrôlées**. Finalement, on a les `RuntimeException`, les exceptions qui peuvent être gérées.

On a donc mentionné les exceptions **contrôlées**. Il faut préciser quand une exception contrôlée est lancée, via l'utilisation de `throws`. Quand une exception est contrôlée, on doit la déclarer dans l'entête dans la méthode.

```
public void myMethod() throws FileNotFoundException {
    //...
    throw new FileNotFoundException("Raison de l'exception");
    //...
}
```

Une exception contrôlée doit être gérée :

```
try {
    myMethod();
} catch (FileNotFoundException ex){
    // gérer l'exception
}
```

Ou être relancée si on appelle du code qui pourrait créer une exception contrôlée :

```
public void otherMethod() throws FileNotFoundException {
    myMethod();
}
```

De manière générale, on demande au programmeur d'être explicite sur les exceptions qui sont lancées et sur la manière de les gérer.

33.1 Créer sa propre exception

Une exception est avant tout une classe, on peut donc hériter de celle-ci pour créer ses propres exceptions.

```
public class MyException extends Exception {
    public MyException(String s){
        super(s);
    }
}
```

On voit que l'exception qu'on a créée est une classe enfant de `Exception` et qu'on a **hérité** de la classe mère par le mot-clé `extends`. Aussi, on a fait appel au constructeur de la classe mère en utilisant le mot-clé `super`.

33.2 Précisions sur l'utilisation de `catch`

Il ne faut pas oublier qu'une exception est un objet. On peut utiliser la méthode `.getMessage()` et `.printStackTrace()` pour obtenir le message lié à l'exception et l'ensemble de la trace.

Auparavant on avait montré des exceptions qui n'avaient qu'un seul `catch` mais elles peuvent en avoir plusieurs. Attention, l'**ordre** a de l'importance, on met donc l'exception la plus précise au-dessus et on descend vers le moins particulier.

```
try {
    // code
} catch (MyException e1){
    // gestion de l'exception e1
} catch (MyException e2) {
    // gestion de l'exception e2
}
```

Non seulement on peut avoir plusieurs `catch`, on peut attraper plusieurs exceptions pour lesquelles on effectue le même traitement.

```
try {
    // code
} catch (MyException | IOException e){
    // code pour n'importe laquelle des deux exceptions
}
```

Enfin, on a aussi le `try` avec ressources. Ils nécessitent des objets `closeable`, ce sont des objets qu'on ouvre et ferme comme des fichiers ou des *sockets*.

```
String path = "cheminDuFicher";
try (BufferedReader br = new BufferedReader(new FileReader(path))){
    return br.readLine();
} catch (NoSuchFileException e){
    // traitement de l'absence du fichier
}
```

34 `var`

L'**inférence de type** pour les déclarations de variables locales avec initialiseurs est la possibilité de ne **pas** répéter le type d'une variable locale lors de sa déclaration, à l'aide du mot-clé `var`.

```
import java.util.ArrayList;
var i = 3; // Infère int
var s = "Hello"; // Infère String
var arr = new ArrayList<Integer>(); // Infère ArrayList<Integer>
//...
```

Dans tous les cas, il faut bien respecter les conventions de nommage pour que les types des variables soient très clairs à l'usage. Aussi, on minimise la portée des variables le plus possible. C'est-à-dire qu'on utilise pas des variables à des endroits physiquement très éloignés dans le code, il vaut mieux conserver les actions près des déclarations. Donc, on utilise `var` quand on a une bonne information sur l'usage et le type de la variable.

On évitera de l'utiliser ou on fera en tout cas attention quand les types déclarés sont importants.

```
List<String> words = new ArrayList<String>(); // Fonctionne mais redondant
List<String> words = new ArrayList<>(); // OK
var words = new ArrayList<String>(); // OK mais attention de donner le type dans
 ↪  l'instanciation
var words = new ArrayList<>(); // Liste d'éléments Object car pas de type défini
```

35 Var args

La signature de la méthode suivante utilise ce qu'on appelle des `varargs`, un nombre non défini de paramètres repris dans un tableau et notée par les trois points après le type des potentiels paramètres.

```
public static Path get(String first, String... more)
```

C'est une formulation qui permet de recevoir un nombre non-spécifié d'arguments au préalable, sous forme de tableau.

Par exemple,

```
public void foo(String... args){
    System.out.println("Nb de params reçus : " + args.length);
    for(String arg : args){
        System.out.println(arg);
    }
}
```

Cette formulation est très pratique quand on ne connaît pas initialement le nombre de paramètres. Voici un autre exemple pour réaliser une somme.

```
public class SumExample{
    public static int sum(int... args){
        int sum = 0;
        for(int el : args){sum += el;}
        return sum;
    }
    public static void main(String[] args){
        int sum1 = sum(1, 2, 3, 4, 5); // 15
        int sum2 = sum(1, 2); // 3
        int sum3 = sum(2, 4, 6, 8, 10); // 30
        System.out.println("sum1 : " + sum1 + ", sum2 : " + sum2 + ", sum3 : " +
         ↪  sum3);
    }
}
```

sum1 : 15, sum2 : 3, sum3 : 30

36 Visibilité

Dans la plupart des langages de programmation, il existe une notion d'espace où un membre est accessible ou non aux autres méthodes. On parle de **visibilité** ou de portée.

En java, il existe :

`private` Membre accessible **uniquement** depuis la classe.

`package` Visible dans toutes les classes d'un même paquet, c'est le comportement par défaut.

`protected` Visibilité liée à l'héritage.

`public` Visible dans toutes les classes.

37 `while` et `do-while`

37.1 `while`

Le `while` est une instruction qui exécute une expression et une instruction de manière répétée jusqu'à ce que la valeur de l'expression soit `false`, elle est donc de type `boolean` ou `Boolean`. Voici la grammaire :

```
WhileStatement:
  while ( Expression ) Statement

WhileStatementNoShortIf:
  while ( Expression ) StatementNoShortIf
```

Une instruction `while` est exécutée en évaluant d'abord l'expression entre parenthèses et si cette expression évalue à `true` et que l'instruction dans le corps peut être poursuivie alors le code s'exécute et la boucle se répète en réévaluant l'expression entre parenthèses. Dans le cas du `false` la boucle est terminée.

On notera, les deux cas avec `Statement` et `StatementNoShortIf`. Dans le second cas, on doit utiliser les accolades pour dénoter un bloc.

37.1.1 Interruption du `while`

Voici les différents cas mettant fin à la boucle `while` et la gestion de ces interruptions.

- Si l'exécution de l'instruction dans le corps est interrompue par un `break` sans étiquette alors la boucle se termine normalement.
- Si l'exécution de l'instruction dans le corps est interrompue par un `continue` sans étiquette, alors l'entièreté de la boucle `while` est exécutée à nouveau.
- Si l'exécution est interrompue due à un `continue` avec étiquette et que la boucle courante porte l'étiquette, elle est exécutée à nouveau, sinon elle est interrompue.
- Si l'instruction s'interrompt pour quelque raison, la boucle aussi.

37.2 `do`

Le `do` (`while`) est une construction très proche de la boucle `while` à une exception près, elle exécute d'abord son corps une seule fois avant de vérifier la validité de l'expression booléenne et opérer de manière similaire à un `while`. Voici sa grammaire :

```
DoStatement:
  do Statement while ( Expression ) ;
```

Comme pour le `while`, l'expression entre parenthèses doit être de type `boolean` ou `Boolean`.

Une autre différence d'avec le `while` et que puisque l'instruction est exécutée une fois avant l'expression, elle doit pouvoir se terminer correctement pour entrer dans la boucle. Autrement, les considérations d'interruptions sont les mêmes que pour le `while`.

38 *Wrapper / boxing*

Une des contraintes des implémentations de l'interface de `List` est que ces implémentations ne peuvent contenir que des objets. De ce fait, on doit trouver une alternative pour les **types primitifs** qui ne peuvent pas initialement être dans une liste.

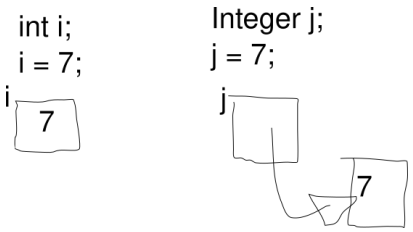
La solution passe par la notion de **wrapper**, des classes qui permettent l'usage de types primitifs comme s'ils étaient eux-mêmes des classes (des types référence). Voici leurs correspondances.

Type primitif	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Par contre, lorsqu'on insère des types primitifs dans une liste, la conversion entre le type primitif et son *wrapper* est automatique. On ne doit pas la préciser, comme le montre l'exemple suivant :

```
1 import java.util.ArrayList;
2 List<Integer> list = new ArrayList<>();
3 list.add(1); // Mettre le nombre 1 dans la liste
4 int number = list.get(0); // Assigner ce nombre à une variable de type int
5 Integer number2 = list.get(0); // Assigner ce nombre à une variable de type Integer
```

Voici une représentation mémoire, le passage de l'un à l'autre est entièrement géré par le compilateur Java qui attribue une place en mémoire ou une référence en mémoire suivant le type choisi.



On peut voir qu'il ne faut pas préciser explicitement les types de variables qu'on ajoute à la liste et que leurs conversions entre type primitif et référence se fait automatiquement. Cette mécanique de conversion porte le nom de *autoboxing* et *unboxing*, comme si on mettait ou enlevait le type primitif d'une boîte (son *wrapper*).

Les *wrappers* sont aussi des classes utilitaires, ils fournissent des méthodes statiques pour opérer sur les types primitifs ou bien les convertir comme :

```
1 String s = "11";
2 int val = 22;
3 Integer.parseInt(s); // 11, un int (type primitif)
4 Integer.valueOf(s); // 11, un Integer (type référence)
5 Integer.toBinaryString(val); // "10110", 22 en binaire, sous forme de String
```

39 Crédits

Cours de DEV2 2020-21 par Pierre Bettens et Marco Codutti à l'ESI, Bruxelles.